# Generating Cohorts

Anthony G. Sena and Martijn J. Schuemie

2022-07-18

## Contents



# 1 Guide for generating cohorts using CohortGenerator

This guide will provide examples of using the CohortGenerator package to generate cohorts in R. In this vignette, we will walk through the process of downloading cohorts from ATLAS and look at the options available for generating the cohorts.

## 1.1 Basic Example

### 1.1.1 Downloading cohorts from ATLAS

We can create cohorts using ATLAS and download them for generation in R. To do this, we will use the `ROhdsiWebApi` package to download some cohort definitions to R:

```
# A list of cohort IDs for use in this vignette
cohortIds <- c(1778211,1778212,1778213)
# Get the SQL/JSON for the cohorts
cohortDefinitionSet <- ROhdsiWebApi::exportCohortDefinitionSet(baseUrl = baseUrl,
                                                               cohortIds = cohortIds)
```

The code above connects to ATLAS' WebAPI, retrieves all of the `cohortDefinitions` and then filters them based on the name to produce the `cohortsForGeneration` dataframe. `cohortsForGeneration` contains the list of cohort IDs to use for calling the function `ROhdsiWebApi::exportCohortDefinitionSet` to download the set of cohort definitions into `cohortDefinitionSet`. The cohort definition set data frame has the following columns:

```
names(cohortDefinitionSet)
```

```
#> [1] "atlasId"          "cohortId"         "cohortName"       "sql"
#> [5] "json"             "logicDescription" "generateStats"
```

Here is how these columns are used:

- **atlasId**: The ATLAS ID for the cohort. This provides linkage back to the ATLAS cohort definition
- **cohortId**: The cohortId will be the same as the atlasId upon export from ATLAS. We provide this column in case you'd like to alter the numbering scheme for your cohort definition set.
- **cohortName**: The name of the cohort in ATLAS.
- **sql**: The SQL used to construct the cohort.
- **json**: The Circe compliant JSON representation of the cohort definition
- **logicDescription**: The description of the cohort from ATLAS.

#### 1.1.2 Saving in a study package

The `cohortDefintionSet` contains all of the details about each cohort that we would like to use for generation. As a best practice, we recommend that you embed these cohort details into a study package. To do this, we've created a function to save the cohort definition set to the file system:

```
saveCohortDefinitionSet(cohortDefinitionSet = cohortDefinitionSet,
                        settingsFileName = file.path(packageRoot, "inst/settings/CohortsToCreate.csv"),
                        jsonFolder = file.path(packageRoot, "inst/cohorts"),
                        sqlFolder = file.path(packageRoot, "inst/sql/sql_server"))
```

By default, saving the cohort definition set will create the files under a folder called `inst` which is where resources for a study package will live. Under `inst` the following folders and files are created:

- **inst/settings/CohortsToCreate.csv**: This will hold the list of cohorts found in the cohortDefinitionSet.
- **inst/cohorts**: This folder will contain a .json file per cohort definition.
- **inst/sql/sql_server**: This folder will contain a .sql file per cohort definition.

Your study package can later re-construct the cohortDefinitionSet by reading in these resources using the `getCohortDefinitionSet` function.

```
cohortDefinitionSet <- getCohortDefinitionSet(settingsFileName = file.path(packageRoot, "inst/settings/
                                              jsonFolder = file.path(packageRoot, "inst/cohorts"),
                                              sqlFolder = file.path(packageRoot, "inst/sql/sql_server"))
```

#### 1.1.3 Generating Cohorts

Now that we have obtained the `cohortDefinitionSet` from ROhdsiWebApi, we're ready to generate our cohorts against our OMOP CDM. In this example, we will use the Eunomia data set as our CDM.

```
# Get the Eunomia connection details
connectionDetails <- Eunomia::getEunomiaConnectionDetails()

# First get the cohort table names to use for this generation task
cohortTableNames <- getCohortTableNames(cohortTable = "cg_example")

# Next create the tables on the database
createCohortTables(connectionDetails = connectionDetails,
                   cohortTableNames = cohortTableNames,
                   cohortDatabaseSchema = "main")

# Generate the cohort set
```

```
cohortsGenerated <- generateCohortSet(connectionDetails= connectionDetails,
                                      cdmDatabaseSchema = "main",
                                      cohortDatabaseSchema = "main",
                                      cohortTableNames = cohortTableNames,
                                      cohortDefinitionSet = cohortDefinitionSet)
```

The code above starts by obtaining the `connectionDetails` from Eunomia. This is where you'll likely want to substitute your own connection information. Next, we call `getCohortTableNames` to obtain a list of `cohortTableNames` that we'll use for the generation process. We then call `createCohortTables` to create the cohort tables on the database server in the `cohortDatabaseSchema`. Once these tables are created, we use the function `generateCohortSet` to generate the `cohortDefintionSet`. When calling `generateCohortSet`, we must specify the schema that holds our CDM (`cdmDatabaseSchema`), the location of the `cohortDatabaseSchema` and `cohortTableNames` where the cohort(s) will be generated. We will cover the other parameters available in the Advanced Options section.

If we'd like to see the results of the generation process, we can use the `getCohortCounts` method to query the cohort table for a summary of the persons and events for each cohort:

```
getCohortCounts(connectionDetails = connectionDetails,
                cohortDatabaseSchema = "main",
                cohortTable = cohortTableNames$cohortTable)
```

```
#> Connecting using SQLite driver
#> Currently in a tryCatch or withCallingHandlers block, so unable to add global calling handlers. Paral
```

```
#> Counting cohorts took 0.0647 secs
```

```
#>   cohortId cohortEntries cohortSubjects
#> 1  1778211          1800           1800
#> 2  1778212           569            569
#> 3  1778213           266            266
```

## 1.2 Advanced Options

### 1.2.1 Cohort Statistics (Inclusion Rule Statistics)

Cohorts defined in ATLAS may define one or more inclusion criteria as part of the cohort's logic. As part of cohort generation, we may want to capture these cohort statistics for use in other packages. For example, CohortDiagnostics has functionality that allows for review of inclusion rule statistics to understand how these rules may materialize between data sources. Here we will review how to generate cohorts with inclusion rule statistics and how to export these results for use by downstream packages such as CohortDiagnostics. Building on our basic example, let's export the cohorts from WebAPI but this time indicate that we'd like to also include the code that `generatesStats`:

```
# Get the cohorts and include the code to generate inclusion rule stats
cohortDefinitionSet <- ROhdsiWebApi::exportCohortDefinitionSet(baseUrl = baseUrl,
                                                               cohortIds = cohortIds,
                                                               generateStats = TRUE)
```

Next we'll create the tables to store the cohort and the cohort statistics. Then we can generate the cohorts.

```
# First get the cohort table names to use for this generation task
cohortTableNames <- getCohortTableNames(cohortTable = "stats_example")

# Next create the tables on the database
createCohortTables(connectionDetails = connectionDetails,
                   cohortTableNames = cohortTableNames,
                   cohortDatabaseSchema = "main")

# We can then generate the cohorts the same way as before and it will use the
# cohort statstics tables to store the results
# Generate the cohort set
generateCohortSet(connectionDetails= connectionDetails,
                  cdmDatabaseSchema = "main",
                  cohortDatabaseSchema = "main",
                  cohortTableNames = cohortTableNames,
                  cohortDefinitionSet = cohortDefinitionSet)
```

At this stage, your cohorts are generated and any cohort statistics are available in the cohort statistics tables. The next step is to export the results to the file system which is done using the code below:

```
insertInclusionRuleNames(connectionDetails = connectionDetails,
                         cohortDefinitionSet = cohortDefinitionSet,
                         cohortDatabaseSchema = "main",
                         cohortInclusionTable = cohortTableNames$cohortInclusionTable)

exportCohortStatsTables(connectionDetails = connectionDetails,
                        cohortDatabaseSchema = "main",
                        cohortTableNames = cohortTableNames,
                        cohortStatisticsFolder = file.path(someFolder, "InclusionStats"))
```

The code above performs two steps. First, we insert the inclusion rule names from the Circe expressions in the `cohortDefinitionSet`. This is important since these names are not automatically inserted into the database when generating the cohorts. Second, we export the cohort statistics to the file system which will write comma separated value (CSV) files per cohort statistic table in the `InclusionStats` folder.

Once you have exported your cohort statistics, you can optionally drop the statistics tables by using the following command:

```
dropCohortStatsTables(connectionDetails = connectionDetails,
                      cohortDatabaseSchema = "main",
                      cohortTableNames = cohortTableNames)
```

### 1.2.2 Incremental Mode

CohortGenerator provides an `incremental` option for some of its functions. The purpose of this `incremental` setting is to allow for the code to attempt to skip an operation if it has already completed it. For example, in the context of cohort generation we may want to keep track of cohorts that we have already generated against a source and skip it if we know the cohort definition has not changed. To illustrate incremental mode and explain how it works, we'll continue along with our example from earlier.

```
# Create a set of tables for this example
cohortTableNames <- getCohortTableNames(cohortTable = "cohort")
```

```
createCohortTables(connectionDetails = connectionDetails,
                   cohortTableNames = cohortTableNames,
                   cohortDatabaseSchema = "main",
                   incremental = TRUE)
```

As expected, the code created the cohort tables as requested. Under the hood, since `incremental = TRUE` was set, the code did a check against the database to see if the tables already exist before creating them. To verify this, we can call the function again and check the results:

```
createCohortTables(connectionDetails = connectionDetails,
                   cohortTableNames = cohortTableNames,
                   cohortDatabaseSchema = "main",
                   incremental = TRUE)
```

```
#> Connecting using SQLite driver
#> Currently in a tryCatch or withCallingHandlers block, so unable to add global calling handlers. Paral

#> Table "cohort" already exists and in incremental mode, so not recreating it.

#> Currently in a tryCatch or withCallingHandlers block, so unable to add global calling handlers. Paral

#> Table "cohort_inclusion" already exists and in incremental mode, so not recreating it.

#> Currently in a tryCatch or withCallingHandlers block, so unable to add global calling handlers. Paral

#> Table "cohort_inclusion_result" already exists and in incremental mode, so not recreating it.

#> Currently in a tryCatch or withCallingHandlers block, so unable to add global calling handlers. Paral

#> Table "cohort_inclusion_stats" already exists and in incremental mode, so not recreating it.

#> Currently in a tryCatch or withCallingHandlers block, so unable to add global calling handlers. Paral

#> Table "cohort_summary_stats" already exists and in incremental mode, so not recreating it.

#> Currently in a tryCatch or withCallingHandlers block, so unable to add global calling handlers. Paral

#> Table "cohort_censor_stats" already exists and in incremental mode, so not recreating it.
```

The use of `incremental = TRUE` here allows for assurance that tables and results from previous runs are preserved. Next, we can generate our `cohortDefinitionSet` in incremental mode.

```
generateCohortSet(connectionDetails= connectionDetails,
                  cdmDatabaseSchema = "main",
                  cohortDatabaseSchema = "main",
                  cohortTableNames = cohortTableNames,
                  cohortDefinitionSet = cohortDefinitionSet,
                  incremental = TRUE,
                  incrementalFolder = file.path(someFolder, "RecordKeeping"))
```

Here we indicate that we are performing this operational incrementally by specifying `incremental = TRUE` and by specifying a folder for holding a record keeping file to track where cohorts are generated: `incrementalFolder = file.path(someFolder, "RecordKeeping")`. Once a cohort is generated in incremental mode, the cohort ID and a checksum of the cohort SQL are saved in the `incrementalFolder` in a file called `GeneratedCohorts.csv`. If we attempt to re-generate the same cohort set in incremental mode, `generateCohortSet` will inspect the SQL for each cohort in the `cohortDefinitionSet` and if the checksum of that cohort matches the checksum found in the `incrementalFolder` for the same cohort ID, the generation is skipped. To illustrate how this looks:

```
generateCohortSet(connectionDetails= connectionDetails,
                  cdmDatabaseSchema = "main",
                  cohortDatabaseSchema = "main",
                  cohortTableNames = cohortTableNames,
                  cohortDefinitionSet = cohortDefinitionSet,
                  incremental = TRUE,
                  incrementalFolder = file.path(someFolder, "RecordKeeping"))
```

```
#> Connecting using SQLite driver
#> Currently in a tryCatch or withCallingHandlers block, so unable to add global calling handlers. Paral
#>
#> Currently in a tryCatch or withCallingHandlers block, so unable to add global calling handlers. Paral

#> Skipping cohortId = '1778211' because it is unchanged from earlier run

#> Currently in a tryCatch or withCallingHandlers block, so unable to add global calling handlers. Paral

#> Skipping cohortId = '1778212' because it is unchanged from earlier run

#> Currently in a tryCatch or withCallingHandlers block, so unable to add global calling handlers. Paral

#> Skipping cohortId = '1778213' because it is unchanged from earlier run
#> Generating cohort set took 0.09 secs
```

#### 1.2.2.1 Potential Pitfalls of Incremental Mode

Incremental mode makes some assumptions to provide a more flexible way to generate cohorts. These assumptions come with some risks that we would like to highlight:

- **Record keeping files**: The incremental approach for cohort generation makes use of a record keeping file to check if a cohort has already been generated. It does **not** check the database to verify this so you need to take care and preserve the file system where you store the record keeping file. If the file is removed, CohortGenerator will simply repeat the cohort generation process.