R API Client Demo Notebook

This is an R Markdown (http://rmarkdown.rstudio.com) Notebook. When you execute code within the notebook, the results appear beneath the code. To execute chunks, click the *Run* button within the chunk, or by placing your cursor inside it and press *Cmd+Shift+Enter*.

FEEDBACK REQUEST: Thank you for checking out our API demo. We would greatly appreciate your feedback. You can send an issue request (https://github.com/datacommonsorg/api-r/issues) and mark the issue using the **feedback** label!

DISCLAIMER: This notebook uses an experimental version of the Data Commons R API Client. The semantics and availability of this API may be subject to change without prior notice!

Using the R API Client to Analyse Statistics in Data Commons

This tutorial introduces the Data Commons open knowledge graph and discusses how to programmatically access its data through the R API Client.

If you already installed the client (https://github.com/datacommonsorg/api-r/blob/master/README.md#installing), you can skip this chunk and just run the next one. **If you still need to install the client**, uncomment and run the following chunk.

```
## SKIP THIS CHUNK IF YOU'VE ALREADY INSTALLED THIS CLIENT
# if(!require(devtools)) install.packages("devtools")
# library(devtools)
# devtools::install_github("datacommonsorg/api-r@v1.0.0-beta", subdir="datac
ommons")
```

Let’s load the client.

```
library(datacommons, quiet=TRUE)
```

```
## ── Attaching packages ───────────────────────────────────────────────────
────────── tidyverse 1.2.1 ──
```

```
## ✔ ggplot2 3.2.1     ✔ purrr   0.3.2
## ✔ tibble  2.1.3     ✔ dplyr   0.8.3
## ✔ tidyr   0.8.3     ✔ stringr 1.4.0
## ✔ readr   1.3.1     ✔ forcats 0.4.0
## ── Conflicts ──────────────────────────────────────────────────────────
──── tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
```

```
##
## Attaching package: 'jsonlite'
```

```
## The following object is masked from 'package:purrr':
##
##     flatten
```

Using the Data Commons API requires you to provision an API key on GCP, and enable Data Commons on your GCP project:

1. Follow the instructions on this page (http://docs.datacommons.org/api/setup.html) to create your API key and enable the Data Commons API for your GCP project.
2. Replace `Sys.getenv("API_KEY")` with your API key, such as "123456". Then run the following block.

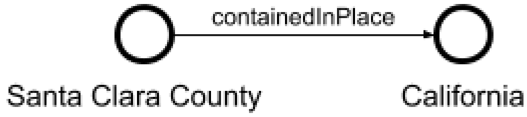
```
REPLACE_WITH_YOUR_API_KEY = Sys.getenv("API_KEY")

set_api_key(REPLACE_WITH_YOUR_API_KEY)
```

What is Data Commons?

Data Commons is an open knowledge graph of structured data. It contains statements about real world objects such as

- Santa Clara County (https://browser.datacommons.org/kg?dcid=geoId/06085) is contained in the State of California (https://browser.datacommons.org/kg?dcid=geoId/06)
- The latitude of Berkeley, CA (https://browser.datacommons.org/kg?dcid=geoId/0606000) is 37.8703
- The population of all persons in Maryland (https://browser.datacommons.org/kg?dcid=dc/o/6w1c9qk7hxjch) has a total count of 5,996,080.

In the graph, *entities* (https://en.wikipedia.org/wiki/Entity) like Santa Clara County (https://browser.datacommons.org/kg?dcid=geoId/06085) are represented by nodes. Every node is uniquely identified by its **`dcid`** (Data Commons Identifier) and has a **`type`** corresponding to what the node represents. For example, California (https://browser.datacommons.org/kg?dcid=geoId/06) is identified by the `dcid geoId/06` and is of type State (https://browser.datacommons.org/kg?dcid=State). *Relations* between entities are represented by directed edges between these nodes. For example, the statement "Santa Clara County is contained in the State of California" is represented in the graph as two nodes: "Santa Clara County" and "California" with an edge labeled "containedInPlace (https://schema.org/containedInPlace)" pointing from Santa Clara to California. This can be visualized by the following diagram.



A portion of the Data Commons graph

Here, we call the edge label, “containedInPlace”, the *property label* (or *property* for short) associated with the above relation. We may also refer to “California” as the *property value* associated with Santa Clara County along property “containedInPlace”.

Notice that the direction is important! One can say that “Santa Clara County” is containedInPlace of “California”, but “California” is certainly not contained in “Santa Clara County”! In general, how Data Commons models data is similar to the Schema.org (https://schema.org) Data Model as Data Commons leverages schema.org to provide a common set of types and properties. For a broader discussion on how data is modeled in Data Commons, one can refer to documentation on the Schema.org data model (https://schema.org/docs/datamodel.html).

The Data Commons Browser

Throughout this tutorial, we will be using the Data Commons browser (https://browser.datacommons.org). The browser provides a human readable way of navigating nodes within the knowledge graph. This is particularly useful for discovering what parameters to pass into the R API Client in order to correctly query for nodes in the graph.

The R API Client

The R API Client provides functions for users to programmatically access nodes in the Data Commons open knowledge graph. In this tutorial, we will be demonstrating how to use the API access nodes in the Data Commons graph and store their information in a tibble (the tidyverse data frame).

Using the API To Plot Unemployment Data

The Bureau of Labor Statistics (https://www.bls.gov) provides a monthly count for number of individuals who are employed at the State, County, and City level. This data is surfaced in the Data Commons; for example, one can find employment statistics associated with Santa Clara County here (https://browser.datacommons.org/kg?dcid=dc/p/y6xm2mny8mck1&db=). Our task for this tutorial will be to extract employment data associated with counties in California from Data Commons using the R API Client and view it in a tibble. We will focus on how functions such as

- `get_property_values`
- `get_places_in`
- `get_populations`
- `get_observations`

operate when using a tibble, as well as how statistical observations are modeled within the Data Commons graph. To see a full list of functions from the Data Commons R API Client, you can use R’s `ls` function:

```
ls(pos = "package:datacommons")
```

```
## [1] "get_observations"    "get_places_in"       "get_populations"
## [4] "get_property_labels" "get_property_values" "get_triples"
## [7] "query"               "set_api_key"
```

To begin, we will initialize a tibble with the `dcid` associated with California: geoId/06 (https://browser.datacommons.org/kg?dcid=geoId/06).

```
# Initialize the tibble
data = tibble('state' = 'geoId/06')

# View the frame
data
```

```
## # A tibble: 1 x 1
##   state
##   <chr>
## 1 geoId/06
```

Accessing Properties of a Node

For all properties, one can use **`get_property_values`** to get the associated values. We would like to know that the `dcid` we have in our data frame belongs to California by getting the name of the node identified by “geoId/06”.

To see the documentation for any function, you can bring up the R doc in the Help panel using the built-in `help` function or its `?` shortcut:

```
?get_property_values
```

`get_property_values` accepts the following parameters.

- **`dcids`** - vector OR single-column tibble/data frame of string(s) of `dcid` (s) to get property values for.
- **`prop`** - string identifying the property to get the property values for.
- **`out`** - optional boolean flag indicating whether the property is directed away from the given nodes. TRUE by default
- **`value_type`** - optional string identifying the node type to filter the returned property values by. NULL by default.
- **`limit`** - optional integer (up to 500) indicating the maximum number of property values returned aggregated over all given nodes. 100 by default.

When the `dcid` s are given as a column, the returned list of property values is a corresponding column where the i-th entry corresponds to property values associated with the i-th given `dcid` . Some properties, like containedInPlace (https://browser.datacommons.org/kg?dcid=containedInPlace), may have many property values. Consequently, the cells of the returned series will always contain a list of property values. Let's take a look:

```
# Call get_property_values. Because the return value will be a column, we ca
n
# assign it directly to a new column in our frame.
data$state_name = get_property_values(select(data, state), 'name')

# Display the frame
data
```

```
## # A tibble: 1 x 2
##   state    state_name
##   <chr>    <named list>
## 1 geoId/06 <chr [1]>
```

For each list in the returned column, we may need to expand each element of the list into its own row. This can easily be achieved by calling `unnest`.

```
data = unnest(data)
data
```

```
## # A tibble: 1 x 2
##   state    state_name
##   <chr>    <chr>
## 1 geoId/06 California
```

Working with Places in Data Commons

We would now like to get all Counties contained in California. This can be achieved by calling `get_property_values`, but because a large fraction of use cases will involve accessing geographical locations, the API also implements a function, **`get_places_in`**, to get places contained within a list of given places. `get_places_in` accepts the following parameters.

- **`dcids`** - vector or a single-column tibble/data frame of `dcid`s to get contained in places.
- **`placeType`** - string identifying type of places contained in the given `dcid`s to filter by.

When `dcid`s is specified as a column, the return value is a column with the same format as that returned by `get_property_values`. Let's call `get_places_in` to get all counties within California.

```
# Call get_places_in to get all counties in California. Here the type we use
is
# "County". See https://browser.datacommons.org/kg?dcid=County for examples
 of
# other nodes in the graph with type "County".
data$county = get_places_in(select(data, state), 'County')

# Display the frame
data
```

```
## # A tibble: 1 x 3
##   state    state_name county
##   <chr>    <chr>      <named list>
## 1 geoId/06 California <chr [58]>
```

Notice that both `state_name` and `county` are columns with lists. `unnest` will automatically flatten all columns that have lists in their cells.

```
# Call flatten frame
data = unnest(data)

# Display the frame
data
```

```
## # A tibble: 58 x 3
##    state    state_name county
##    <chr>    <chr>      <chr>
##  1 geoId/06 California geoId/06001
##  2 geoId/06 California geoId/06003
##  3 geoId/06 California geoId/06005
##  4 geoId/06 California geoId/06007
##  5 geoId/06 California geoId/06009
##  6 geoId/06 California geoId/06011
##  7 geoId/06 California geoId/06013
##  8 geoId/06 California geoId/06015
##  9 geoId/06 California geoId/06017
## 10 geoId/06 California geoId/06019
## # … with 48 more rows
```

Let’s now get the names for the county column, as we did for the state column.

```
# Get the names of all counties in California.
data$county_name = get_property_values(select(data, county), 'name')
data = unnest(data)

# Display the frame
data
```

```
## # A tibble: 58 x 4
##    state    state_name county      county_name
##    <chr>    <chr>      <chr>       <chr>
##  1 geoId/06 California geoId/06001 Alameda County
##  2 geoId/06 California geoId/06003 Alpine County
##  3 geoId/06 California geoId/06005 Amador County
##  4 geoId/06 California geoId/06007 Butte County
##  5 geoId/06 California geoId/06009 Calaveras County
##  6 geoId/06 California geoId/06011 Colusa County
##  7 geoId/06 California geoId/06013 Contra Costa County
##  8 geoId/06 California geoId/06015 Del Norte County
##  9 geoId/06 California geoId/06017 El Dorado County
## 10 geoId/06 California geoId/06019 Fresno County
## # … with 48 more rows
```

Working with Statistical Observations

Finally, we are ready to query for Employment statistics in Data Commons. Before proceeding, we discuss briefly about how Data Commons models statistics.

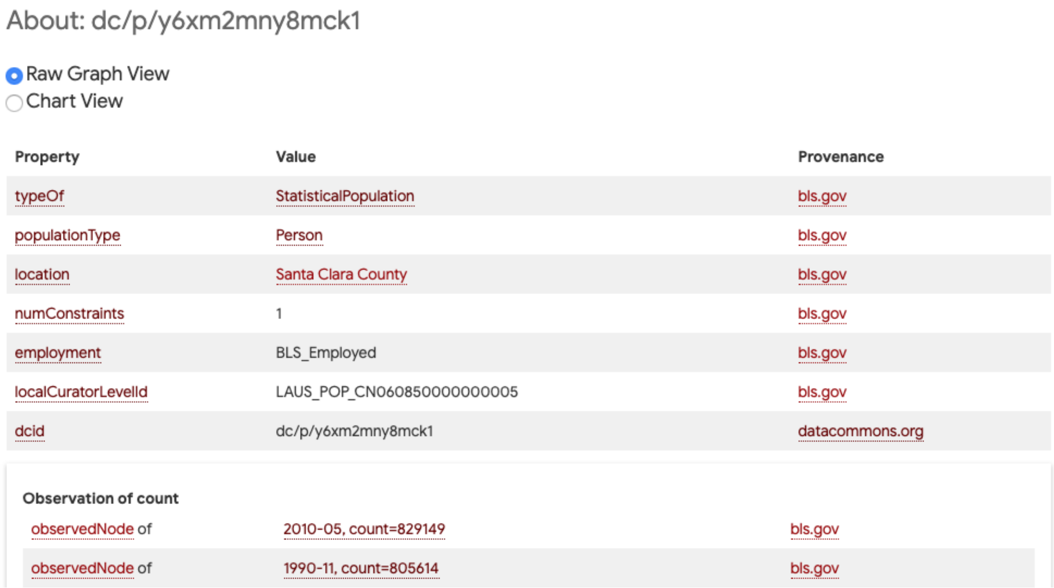
Statistical observations can be separated into two types of entities: the *statistical population* that the observation is describing, and the *observation* itself. A statistic such as

> The number of employed individuals living in Santa Clara in January 2018 was 1,015,129

can thus be represented by two entities: one capturing the population of *all persons who are unemployed* and another capturing the observation 1,015,129 made in January 2018. Data Commons represents these two entity types as `StatisticalPopulation` and `Observation` . Let's now focus on each separate one.

StatisticalPopulations

Consider the node's browser page (https://browser.datacommons.org/kg?dcid=dc/p/y6xm2mny8mck1&db=) representing the `StatisticalPopulation` of all employed persons living in Santa Clara County.



About: dc/p/y6xm2mny8mck1

● Raw Graph View
○ Chart View

Property	Value	Provenance
typeOf	StatisticalPopulation	bls.gov
populationType	Person	bls.gov
location	Santa Clara County	bls.gov
numConstraints	1	bls.gov
employment	BLS_Employed	bls.gov
localCuratorLevelId	LAUS_POP_CN060850000000005	bls.gov
dcid	dc/p/y6xm2mny8mck1	datacommons.org

Observation of count

observedNode of	2010-05, count=829149	bls.gov
observedNode of	1990-11, count=805614	bls.gov

The population of employed individuals in Santa Clara County

At the top of the browser page, we are presented a few properties of this node to consider:

- The **`typeOf`** this node is `StatisticalPopulation` as expected.
- The **`populationType`** is `Person` telling us that this is a statistical population describing persons.
- The **`location`** of this node is Santa Clara County (https://browser.datacommons.org/kg?dcid=geoId/06085) telling us that the persons in this statistical population live in Santa Clara County.
- The **`dcid`** property tells us the `dcid` of this node
- The **`localCuratorLevelId`** tells us information about how this node was uploaded to the graph. For the purposes of this tutorial, we can ignore this field.

There are two other properties defined: `numConstraints` and `employment` . These two properties help us describe entities contained in this statistical population. Properties used to describe the entities captured by a StatisticalPopulation are called *constraining properties*. In the example above, `employment=BLS_Employed` is a constraining property that tells us the Statistical Population captures employed persons. `numConstraints` denotes how many constraining properties there are, and in the example above, `numConstraints=1` tells us that `employment` is the only constraining property.

To query for `StatisticalPopulation` s using the Data Commons R API Client, we call **`get_populations`** . The function accepts the following parameters.

- **`dcids`** - vector OR single-column tibble/dataframe of `dcid` (s) identifying `Place` s of `StatisticalPopulation` s to query for.
- **`population_type`** - string identifying the population type of the `StatisticalPopulation` .
- **`constraining_properties`** - optional named list mapping constraining property to the value that the `StatisticalPopulation` should be constrained by. Empty by default.

When a column is provided to `dcids` , the return value is a column with each cell populated by a single `dcid` and not a list. This is because the combination of `dcids` , `population_type` , and optionally `constraining_properties` always map to a unique `StatisticalPopulation` `dcid` if it exists.

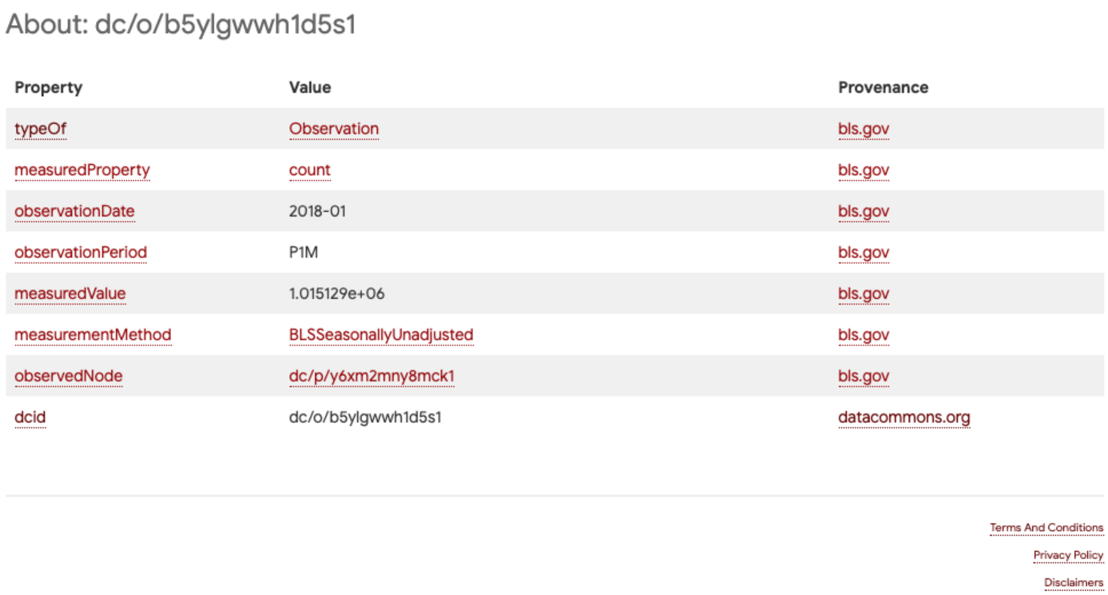
Let’s call `get_populations` to get the populations of all employed individuals living in counties specified by the `county` column of our tibble.

```
# First we create the constraining_properties
props <- list(employment = 'BLS_Employed')

# We now call get_populations.
data$employed_pop <- get_populations(select(data, county), 'Person', constra
ining_properties=props)

# Display the DataFrame. Notice that we don't need to flatten the frame.
print(head(data))
```

```
## # A tibble: 6 x 5
##   state    state_name county      county_name      employed_pop
##   <chr>    <chr>      <chr>       <chr>            <chr>
## 1 geoId/06 California geoId/06001 Alameda County   dc/p/69mjbvx3c6m38
## 2 geoId/06 California geoId/06003 Alpine County    dc/p/38cz2egkw07rb
## 3 geoId/06 California geoId/06005 Amador County    dc/p/d0tgg3hcqwevh
## 4 geoId/06 California geoId/06007 Butte County     dc/p/fn9p8qqppptdf
## 5 geoId/06 California geoId/06009 Calaveras County dc/p/cgf7kpwczz56b
## 6 geoId/06 California geoId/06011 Colusa County    dc/p/dll18skxd9gl2
```

Observations

At the bottom of the page describing the `StatisticalPopulation` of all employed persons living in Santa Clara County is a list of observations made of that population. This is the browser page (https://browser.datacommons.org/kg?dcid=dc/o/b5ylgwwh1d5s1) for the node representing the observed count of this population for January 2018.



About: dc/o/b5ylgwwh1d5s1

Property	Value	Provenance
typeOf	Observation	bls.gov
measuredProperty	count	bls.gov
observationDate	2018-01	bls.gov
observationPeriod	P1M	bls.gov
measuredValue	1.015129e+06	bls.gov
measurementMethod	BLSSeasonallyUnadjusted	bls.gov
observedNode	dc/p/y6xm2mny8mck1	bls.gov
dcid	dc/o/b5ylgwwh1d5s1	datacommons.org

Terms And Conditions
Privacy Policy
Disclaimers

The number of employed individuals in Santa Clara County

In this page, there are a few properties to consider:

- The **`typeOf`** this node is `Observation` as expected.
- The **`measuredProperty`** is `count` telling us that this observation measures the number of persons in this statistical population.
- The **`measurementMethod`** is `BLSSeasonallyUnadjusted` to indicate how the observation was adjusted. We can click on that link to see what `BLSSeasonallyUnadjusted` means.
- The **`observationDate`** is `2018-01` . This date is formatted by ISO8601 (https://en.wikipedia.org/wiki/ISO_8601) standards.
- The **`observedPeriod`** is `P1M` to denote that this observation was carried over a period of 1 month.
- The **`observedNode`** which tells us what the node is being observed by the observation. Here its value is `dc/p/y6xm2mny8mck1` which is the `dcid` of the population of employed persons living in Santa Clara.

The final property of interest is **`measuredValue`** . This property tells us that the raw value observed by the observation (whose value is 1,015,129) in this case. The `measuredValue` is also a *statistic type* associated with the observation. For a single observation, there could be many statistics that describe it. One would be the raw value represented by `measuredValue` , while others include `meanValue` , `medianValue` , `marginOfError` , and more.

These parameters are useful for deciding what values to provide to the API. To query for `Observation` s using the R API Client, we call **`get_observations`** which accepts the following parameters.

- **`dcids`** - vector OR single column tibble/data frame of `dcid` s to get observations for.
- **`measured_property`** - string identifying the measured property.
- **`stats_type`** - string identifying the statistical type for the observation.
- **`observation_date`** - string of the associated observation date in ISO8601 format.
- **`observation_period`** - string specifying the observation period. Required if the Observations you are querying for specify this field. NULL by default.
- **`measurement_method`** - string specifying the measurement method. Required if the Observations you are querying for specify this field. NULL by default.

One thing to note is that not all `Observation` s will have a property value for `observationPeriod` and `measurementMethod` . For example, the number of housing units with 3 bedrooms in Michigan (https://browser.datacommons.org/kg?dcid=dc/o/6x1emqkzvvjq2&db=) does not have `observationPeriod` while number of Other Textile Mills (https://browser.datacommons.org/kg?dcid=dc/o/mc6yc1v1004y5&db=) does not have `measurementMethod` . These parameters are thus optional arguments to the `get_observations` API function.

When a column is provided to `dcids` , the return value is again a column with each cell populated by the statistic and not a list of statistics. The combination of the above parameters always map to a unique observation if it exists. If the statistic does not exist for the given parameters, then the cell will contain `NaN` .

Let's get the `measuredValue` of observations of the column of populations we just queried for.

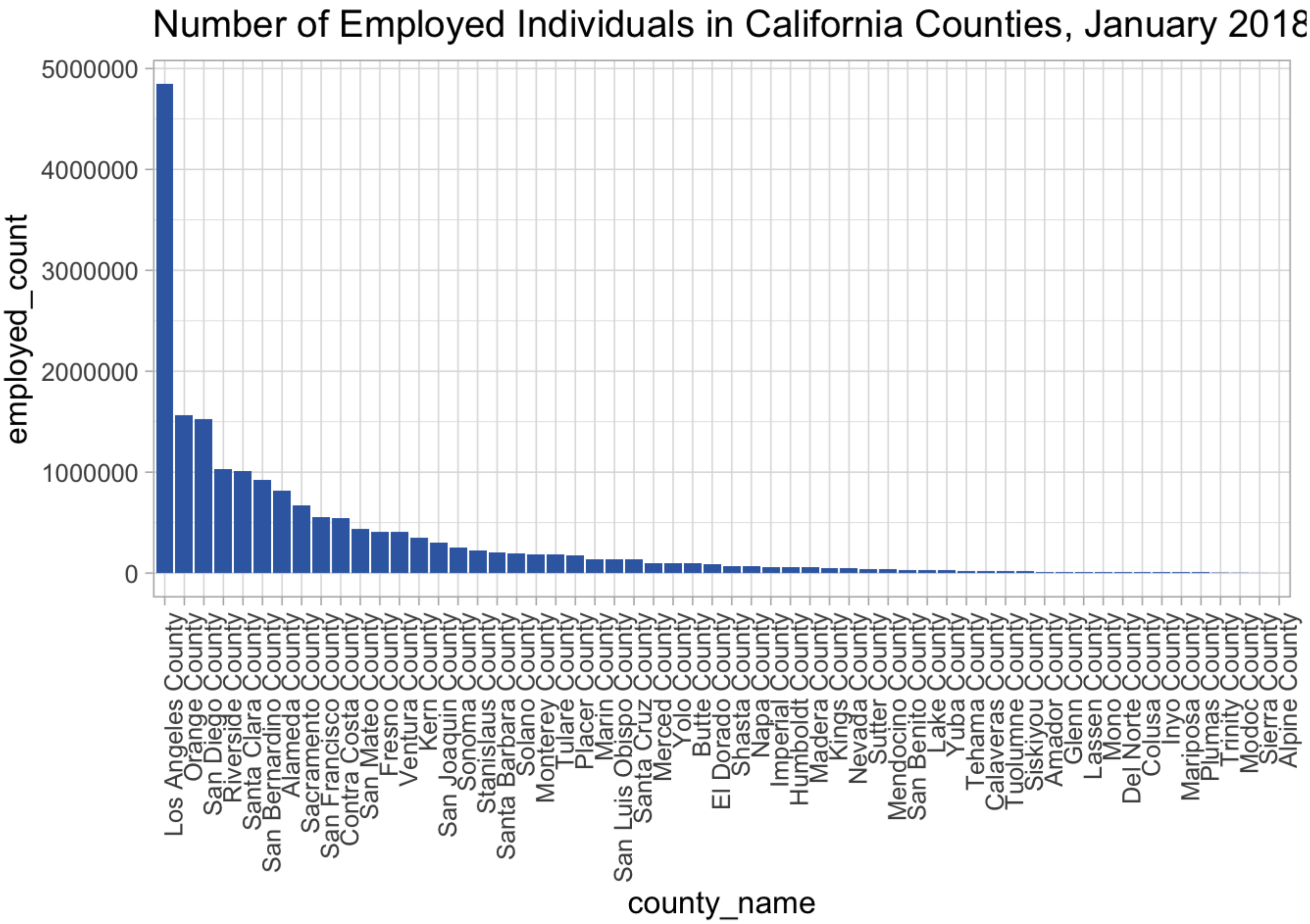
```
# Call get_observations We are passing into the parameters the values that w
e
# saw in the link above.
data$employed_count <- get_observations(
    select(data, employed_pop),
    'count',
    'measuredValue',
    '2018-01',
    observation_period='P1M',
    measurement_method='BLSSeasonallyUnadjusted')

# Display the DataFrame
print(head(data))
```

```
## # A tibble: 6 x 6
##   state   state_name county    county_name   employed_pop    employed_cou
nt
##   <chr>   <chr>      <chr>     <chr>         <chr>                    <db
l>
## 1 geoId/… California geoId/06… Alameda Coun… dc/p/69mjbvx3c…         8197
43
## 2 geoId/… California geoId/06… Alpine County dc/p/38cz2egkw…            5
28
## 3 geoId/… California geoId/06… Amador County dc/p/d0tgg3hcq…          138
83
## 4 geoId/… California geoId/06… Butte County  dc/p/fn9p8qqpp…          967
62
## 5 geoId/… California geoId/06… Calaveras Co… dc/p/cgf7kpwcz…          202
53
## 6 geoId/… California geoId/06… Colusa County dc/p/dll18skxd…           85
00
```

Wrapping Things Up

Now that we have our tibble populated, we can analyse the data. Let’s plot the counts in a bar chart to see what we find.

```
# Make a deep copy
final_data <- data
# Sort by employment count
final_data <- final_data[order(final_data$employed_count, decreasing = TRUE
),]
# make county_name an ordered factor so ggplot2 doesn't try to sort it...
final_data$county_name <- factor(final_data$county_name, levels = final_data
$county_name)

theme_set(theme_light(base_size = 12))
options(scipen=999)
g <- ggplot(final_data, aes(x=county_name, y=employed_count)) +
  geom_bar(stat = "identity", fill = "#396AB1") +
  theme(axis.text.x = element_text(angle = 90, hjust = 1)) +
  ggtitle("Number of Employed Individuals in California Counties, January 20
18")

g
```



This wraps up our tutorial on how to use the Data Commons R API Client to access statistics in the knowledge graph and view it in a tibble. From this tutorial we should now know

- How statistics are modeled in the graph

- What Node API methods are available for accessing these statistics
  - `get_property_values`
  - `get_places_in`
  - `get_populations`
  - `get_observations`
- How to use API methods to access these statistics

We did not cover 2 Node API methods:

- get_triples: get all triples (subject-predicate-object) where the specified node is either a subject or an object.

- get_property_labels: get property labels of each specified node.

Remember that you can learn more about these functions, or any function, using the R `help(function)` command or the `?function` shortcut. To see a list of all functions in a package, use the R `ls(pos = "package:datacommons")` command.

Lastly, one may wish to perform a more nuanced analysis of employment. For example, to really understand employment rates in a given county, we may wish to normalize the counts we queried for by the counts of various populations such as:

- All persons in a county.
- All persons of working age in a county.
- All persons with or without a disability in a county.

We hope that with the tools provided in this notebook, such analysis will be easier and quicker to perform. Happy hunting!