Package 'sparklyr.nested'

July 23, 2025

Title A 'sparklyr' Extension for Nested Data

Version 0.0.4

Maintainer Matt Pollock <mpollock@mitre.org>

Description A 'sparklyr' extension adding the capability to work easily with nested data.

Depends R (>= 3.3)

Imports sparklyr, jsonlite, listviewer, dplyr, rlang, purrr, tidyselect

Suggests testthat, reactR

License Apache License 2.0 | file LICENSE

SystemRequirements Spark: 1.6.x or 2.x

Encoding UTF-8

RoxygenNote 7.2.3

BugReports https://github.com/mitre/sparklyr.nested/issues

NeedsCompilation no

Author Matt Pollock [aut, cre], The MITRE Corporation [cph]

Repository CRAN

Date/Publication 2023-02-20 22:00:03 UTC

Contents

sdf_explode *Explode data along a column*

Description

Exploding an array column of length N will replicate the top level record N times. The i^th replicated record will contain a struct (not an array) corresponding to the i^th element of the exploded array. Exploding will not promote any fields or otherwise change the schema of the data.

Usage

```
sdf_explode(x, column, is_map = FALSE, keep_all = FALSE)
```

Arguments

x	An object (usually a spark_tbl) coercible to a Spark DataFrame.
column	The field to explode
is_map	Logical. The (scala) explode method works for both array and map column types. If the column to explode in an array, then is_map=FALSE will ensure that the exploded output retains the name of the array column. If however the column to explode is a map, then the map will have key/value names that will be used if is_map=TRUE.
keep_all	Logical. If FALSE then records where the exploded value is empty/null will be dropped.

Details

Two types of exploding are possible. The default method calls the scala explode method. This operation is supported in both Spark version > 1.6. It will however drop records where the exploding field is empty/null. Alternatively keep_all=TRUE will use the explode_outer scala method introduced in spark 2 to not drop any records.

Examples

```
## Not run:
# first get some nested data
iris_tbl <- copy_to(sc, iris, name="iris")
iris_nst <- iris_tbl %>%
  sdf_nest(Sepal_Length, Sepal_Width, Petal_Length, Petal_Width, .key="data") %>%
  group_by(Species) %>%
  summarize(data=collect_list(data))

# then explode it
iris_nst %>% sdf_explode(data)

## End(Not run)
```

sdf_nest *Nest data in a Spark Dataframe*

Description

This function is like `tidyr::nest`. Calling this function will not aggregate over other columns. Rather the output has the same number of rows/records as the input. See examples of how to achieve row reduction by aggregating elements using `collect_list`, which is a Spark SQL function

Usage

```
sdf_nest(x, ..., .key = "data")
```

Arguments

`x`	A Spark dataframe.
`...`	Columns to nest.
`.key`	Character. A name for the new column containing nested fields

Examples

```
## Not run:
# produces a dataframe with an array of characteristics nested under
# each unique species identifier
iris_tbl <- copy_to(sc, iris, name="iris")
iris_tbl %>%
  sdf_nest(Sepal_Length, Sepal_Width, Petal_Length, Petal_Width, .key="data") %>%
  group_by(Species) %>%
  summarize(data=collect_list(data))

## End(Not run)
```

sdf_schema_json *Work with the schema*

Description

These functions support flexible schema inspection both algorithmically and in human-friendly ways.

Usage

```
sdf_schema_json(
  x,
  parse_json = TRUE,
  simplify = FALSE,
  append_complex_type = TRUE
)

sdf_schema_viewer(
  x,
  simplify = TRUE,
  append_complex_type = TRUE,
  use_react = FALSE
)
```

Arguments

x	An R object wrapping, or containing, a Spark DataFrame.
parse_json	Logical. If TRUE then the JSON return value will be parsed into an R list.
simplify	Logical. If TRUE then the schema will be folded into itself such that {"name" : "field1", "type" : {"type" : "array", "elementType" : "string", "containsNull" : true}, "nullable" : true, "metadata" : { } } will be rendered simply {"field1 (array)" : "[string]"}
append_complex_type	Logical. This only matters if parse_json=TRUE and simplify=TRUE. In that case indicators will be included in the return value for array and struct types.
use_react	Logical. If TRUE schemas will be rendered using reactjson. Otherwise they will be rendered using jsonedit (the default). Using react works better in some contexts (e.g. bookdown-rendered HTML) and has a different look & feel. It does however carry an extra dependency on the reactR package suggested by listviewer.

See Also

sdf_schema

Examples

```
## Not run:
library(testthat)
library(jsonlite)
library(sparklyr)
library(sparklyr.nested)
sample_json <- paste0(
  '{"aircraft_id":["string"],"phase_sequence":["string"],"phases (array)":{"start_point (struct)":',
  '{"segment_phase":["string"],"agl":["double"],"elevation":["double"],"time":["long"],',
  '"latitude":["double"],"longitude":["double"],"altitude":["double"],"course":["double"],',
  '"speed":["double"],"source_point_keys (array)":["[string]"],"primary_key":["string"]},',
```

```
  '"end_point (struct)":{"segment_phase":["string"],"agl":["double"],"elevation":["double"],',
  '"time":["long"],"latitude":["double"],"longitude":["double"],"altitude":["double"],',
  '"course":["double"],"speed":["double"],"source_point_keys (array)":["[string]"],',
  '"primary_key":["string"]},"phase":["string"],"primary_key":["string"]},"primary_key":["string"]}'
)

with_mock(
  # I am mocking functions so that the example works without a real spark connection
  spark_read_parquet = function(x, ...){return("this is a spark dataframe")},
  sdf_schema_json = function(x, ...){return(fromJSON(sample_json))},
  spark_connect = function(...){return("this is a spark connection")},

  # the meat of the example is here
  sc <- spark_connect(),
  spark_data <- spark_read_parquet(sc, path="path/to/data/*.parquet", name="some_name"),
  sdf_schema_viewer(spark_data)
)

## End(Not run)
```

`sdf_select` *Select nested items*

Description

The `select` function works well for keeping/dropping top level fields. It does not however support access to nested data. This function will accept complex field names such as `x.y.z` where `z` is a field nested within `y` which is in turn nested within `x`. Since R uses "$" to access nested elements and java/scala use ".", `sdf_select(data, x.y.z)` and `sdf_select(data, x$y$z)` are equivalent.

Usage

```
sdf_select(x, ..., .aliases, .drop_parents = TRUE, .full_name = FALSE)
```

Arguments

`x`	An object (usually a `spark_tbl`) coercible to a Spark DataFrame.
`...`	Fields to select
`.aliases`	Character. Optional. If provided these names will be matched positionally with selected fields provided in `...`. This is more useful when calling from a function and less natural to use when calling the function directly. It is likely to get you into trouble if you are using `dplyr` select helpers. The alternative with direct calls is to put the alias on the left side of the expression (e.g. `sdf_select(df, fld_alias=parent.child.fld)`)
`.drop_parents`	Logical. If `TRUE` then any field from which nested elements are extracted will be dropped, even if they were included in the selected `...`. This better supports using `dplyr` field matching helpers like `everything()` and `starts_with`.

`.full_name` Logical. If `TRUE` then nested field names that are not named (either using a LHS `name=field_name` construct or the `.aliases` argument) will be disambiguated using the parent field name. For example `sdf_select(df, x.y)` will return a field named `x_y`. If `FALSE` then the parent field name is dropped unless it is needed to avoid duplicate names.

Selection Helpers

`dplyr` allows the use of selection helpers (e.g., see `everything`). These helpers only work for top level fields however. For now all nested fields that should be promoted need to be explicitly identified.

Examples

```
## Not run:
# produces a dataframe with an array of characteristics nested under
# each unique species identifier
iris_tbl <- copy_to(sc, iris, name="iris")
iris_nst <- iris_tbl %>%
  sdf_nest(Sepal_Length, Sepal_Width, .key="Sepal")

# using java-like dot-notation
iris_nst %>%
  sdf_select(Species, Petal_Width, Sepal.Sepal_Width)

# using R-like dollar-sign-notation
iris_nst %>%
  sdf_select(Species, Petal_Width, Sepal$Sepal_Width)

# using dplyr selection helpers
iris_nst %>%
  sdf_select(Species, matches("Petal"), Sepal$Sepal_Width)

## End(Not run)
```

`sdf_unnest` *Unnest data along a column*

Description

Unnesting is an (optional) explode operation coupled with a nested select to promote the sub-fields of the exploded top level array/map/struct to the top level. Hence, given `a`, an array with fields `a1`, `a2`, `a3`, then codesdf_explode(df, a) will produce output with each record replicated for every element in the `a` array and with the fields `a1`, `a2`, `a3` (but not `a`) at the top level. Similar to `tidyr::unnest`.

Usage

```
sdf_unnest(x, column, keep_all = FALSE)
```

Arguments

x	An object (usually a spark_tbl) coercible to a Spark DataFrame.
column	The field to explode
keep_all	Logical. If FALSE then records where the exploded value is empty/null will be dropped.

Details

Note that this is a less precise tool than using sdf_explode and sdf_select directly because all fields of the exploded array will be kept and promoted. Direct calls to these methods allows for more targeted use of sdf_select to promote only those fields that are wanted to the top level of the data frame.

Additionally, though sdf_select allows users to reach arbitrarily far into a nested structure, this function will only reach one layer deep. It may well be that the unnested fields are themselves nested structures that need to be dealt with accordingly.

Note that map types are supported, but there is no is_map argument. This is because the function is doing schema interrogation of the input data anyway to determine whether an explode operation is required (it is of maps and arrays, but not for bare structs). Given this the result of the schema interrogation drives the value o is_map provided to sdf_explode.

Examples

```
## Not run:
# first get some nested data
iris_tbl <- copy_to(sc, iris, name="iris")
iris_nst <- iris_tbl %>%
  sdf_nest(Sepal_Length, Sepal_Width, Petal_Length, Petal_Width, .key="data") %>%
  group_by(Species) %>%
  summarize(data=collect_list(data))

# then explode it
iris_nst %>% sdf_unnest(data)

## End(Not run)
```

---

struct_type *Spark Data Types*

Description

These function support supplying a spark read schema. This is particularly useful when reading data with nested arrays when you are not interested in several of the nested fields.

Usage

```
struct_type(sc, struct_fields)

struct_field(sc, name, data_type, nullable = FALSE)

array_type(sc, data_type, nullable = FALSE)

binary_type(sc)

boolean_type(sc)

byte_type(sc)

date_type(sc)

double_type(sc)

float_type(sc)

integer_type(sc)

numeric_type(sc)

long_type(sc)

map_type(sc, key_type, value_type, nullable = FALSE)

string_type(sc)

character_type(sc)

timestamp_type(sc)
```

Arguments

sc	A spark_connection
struct_fields	A vector or fields obtained from struct_field()
name	A field name to use in the output struct type
data_type	A (java) data type (e.g., string_type() or double_type())
nullable	Logical. Describes whether field can be missing for some rows.
key_type	A (java) data type describing the map keys (usually string_type())
value_type	A (java) data type describing the map values

Index