

Column Names as Contracts

Sep 6, 2020 ·  [data](#), [workflow](#), [rstats](#)

Software products use a range of strategies to make promises or contracts with their users. Mature code packages and APIs document expected inputs and outputs, check adherence with unit tests, and transparently report code coverage. Programs with graphical user interfaces form such contracts by labeling and illustrating interactive components to explain their intent (for example, the “Save” button typically does not bulk-delete files).

Published data tables, however, exist in an ambiguous gray area; static enough not to be considered a “service” or “software”, yet too raw to earn the attention to user experience given to interfaces. This ambiguity can create a strange symbiosis between data producers and consumers. Producers may publish whatever data is accessible by the system or seems relevant, and consumers may be quick to assume tables or fields that “sound right” happen to be custom-fit for their top-of-mind question. Producers wonder why consumers aren’t satisfied, and consumers wonder why the data is never “right”.

Metadata management solutions aim to solve this problem, and there are many promising developments in this space including Lyft’s [Amundsen](#), LinkedIn’s [DataHub](#), and Netflix’s [Metacat](#). However, metadata solutions generally require a great degree of cooperation: producers must vigilantly maintain the documentation and consumers must studiously check it – despite such tools almost always living outside of either party’s core toolkit and workflow.

Using [controlled vocabularies](#) for column names is a low-tech, low-friction approach to building a shared understanding of how each field in a data set is intended to work. In this post, I’ll introduce the concept with an example and demonstrate how controlled vocabularies can offer lightweight solutions to rote data validation, discoverability, and wrangling. I’ll illustrate these usability benefits with R packages including `pointblank`, `collapsibleTree`, and `dplyr`, but we’ll conclude by demonstrating how the same principles apply to other packages and languages.

Controlled Vocabulary

The basic idea of controlled vocabularies is to define upfront a set of words, phrases, or stubs with well-defined meanings which can be used to index information. When these stubs are defined for different types of information and pieces together in a consistent order, the vocabulary becomes of descriptive grammar that we can use to describe more complex content and behavior.

In the context of a data set, this vocabulary can also serve as a latent contract between data producers and data consumers and carry promises regarding different aspects of the data lineage, valid values, and appropriate uses. When used consistently across all of an organization’s tables, it can significantly scale data management and increase usability as knowledge from working with one dataset easily transfers to another.

For example, imagine we work at a ride-share company and are building a data table with one record per trip. What might a controlled vocabulary look like?¹

Level 1: Measure Types

For reasons that will be evident in the examples, I like the first level of the hierarchy to generally capture a semi-generic “type” of the variable. This is not quite the same as data types in a programming language (e.g. `bool`, `double`, `float`) although everything with the same prefix should ultimately be cast in the same type. Instead, these data types imply both a type of information and appropriate usage patterns:

- ID: Unique identified for an entity.
 - Numeric for more efficient storage and joins unless system of record generates IDs with characters
 - Likely a primary key in some other table
- IND: Binary 0 or 1 indicator or an event occurrence
 - Because always 0 or 1, can be averaged to find proportion occurrence
 - Can consider calling IS instead of IND for even less ambiguity which case is labeled 1
- N: Count of quantity or event occurrence
 - Always a non-negative integer
- AMT: Sum-able real number amount. That is, any non-count amount that is “denominator-free”
- VAL: Numeric variables that are not inherently sum-able
 - For example, rates and ratios that cannot be combined or numeric values like latitude and longitude for which typical arithmetic operations don’t make sense
- DT: Date of some event
 - Always cast as a YYYY-MM-DD date
- TM: Timestamp of some event
 - Always cast as a YYYY-MM-DD HH:MM:SS timestamp
 - Distinguishing dates from timestamps will avoid faulty joins of two date fields arranged differently
- CAT: Categorical variable as a character string (potentially encoded from an ID field)

While these are relatively generic, domain-specific categories can also be used. For example, since location is so important for ride-sharing, it might be worth having ADDR as a level 1 category.

Level 2: Measure Subjects

The best hierarchy varies widely by industry and the *overall contents of a database* – not just one table. Here, we expect to be interested in trip-attributed about many different subjects: the rider, driver, trip, etc. so the measure subject might be the logical next tier. We can define:

- DRIVER: Information about the driver
- RIDER: Information about the rider, the passenger who called the ride-share

- TRIP: Information about the trip itself
- ORIG: Information about the trip start (time and geography)
- DEST: Information about the trip destination (time and geography)
- COST: Information about components of the total cost (could be a subset of TRIP, but pertains to all parties and has high cardinality at the next tier, so we'll break it out)

Of course, in a highly normalized database, measures of these different entities would exist in separate tables. However, this discipline in naming them would still be beneficial so quantities are unambiguous when an analyst combines them.

Levels 3-n: Details

The first few tiers of the hierarchy are critical to standardize to make our “performance promises” and to aid in data searchability. Later levels will be measure specific and may not be worth defining upfront. However, for concepts that are going to exist across many tables, it is worthwhile to pre-specify their names and precise formats. For example:

- CITY: Should this be in all upper case? How should spaces in city name be treated?
- ZIP: Should 6 digit or 10 digit zip codes be used?
- LAT/LON: To how many decimals should latitude and longitude be geocoded? If the company only operate in certain geographic areas (e.g. the continental US), coarse cut-offs for these can be determined
- DIST: Is distance measured in miles? Kilometers?
- TIME: Are durations measured in seconds? Minutes?
- RATING: What are valid ranges for other known quantities like star ratings?

Terminal “adjectives” could also be considered. For example, if the data-generating systems spit out analytically unideal quantities that should be preserved for data lineage purposes, suffixes such as _RAW and _CLEAN might denote version of the same variable in its original and manicured states, respectively.

Putting it all together

This structure now gives us a grammar to compactly name 35 variables in table:

- ID_{DRIVER | RIDER | TRIP}: Unique identifier for party of the ride
- DT_{ORIG | DEST}: Date at the trip’s start and end, respectively
- TM_{ORIG | DEST}: Timestamp at the trip’s start and end, respectively
- N_TRIP_{PASSENGERS | ORIG | DEST} Count of unique passengers, pick-up points, and drop-off points for the trip
- N_DRIVER_SEATS: Count of passenger seats available in the driver’s car
- AMT_TRIP_{DIST | TIME}: Total trip distance in miles traveled and time taken
- AMT_COST_{TIME | DIST | BASE | FEES | SURGE | TIPS}: Amount of each cost component
- IND_SURGE: Indicator variable if ride caled during surge pricing
- CAT_TRIP_TYPE: Trip type, such as ‘Pool’, ‘Standard’, ‘Elite’
- CAT_RIDER_TYPE: Rider status, such as ‘Basic’, ‘Frequent’, ‘Subscription’
- VAL_{DRIVER | RIDER}_RATING: Average star rating of rider and driver
- ADDR_{ORIG | DEST}_{STREET | CITY | STATE | ZIP}: Address components of trip’s start and end
- VAL_{ORIG | DEST}_{LAT | LON}: Latitude and longitude of trip’s start and end

The fact that we can describe 35 variables in roughly 1/3 the number of rows already speaks to the value of this structure in helping data consumers build a strong mental model to quickly manipulate the data. But now we can demonstrate far greater value.

To start, we create a small fake data set using our schema. For simplicity, I simulate 18 of the 35 variables listed above:

```
head(data_trips)
#>   ID_DRIVER ID_RIDER ID_TRIP   DT_ORIG   DT_DEST N_DRIVER_PASSENGERS
#> 1      8709     4484   3902 2019-04-06 2019-04-06                1
#> 2      9208     7204   3966 2019-01-20 2019-01-20                1
#> 3      2268     6449   8167 2019-05-05 2019-05-05                1
#> 4      7267     4612   5728 2019-10-24 2019-10-24                1
#> 5      4230     9646   7703 2019-01-25 2019-01-25                2
#> 6      6588     9897   3851 2019-04-12 2019-04-12                2
#>   N_TRIP_ORIG N_TRIP_DEST AMT_TRIP_DIST IND_SURGE VAL_DRIVER_RATING
#> 1           1           1    17.50953         0      2.362601
#> 2           1           1    34.83353         0      2.974465
#> 3           1           1    20.74241         0      2.000912
#> 4           1           1    23.94259         1      1.481275
#> 5           1           1    48.73168         0      3.622251
#> 6           1           1    42.26211         1      4.789015
#>   VAL_RIDER_RATING VAL_ORIG_LAT VAL_DEST_LAT VAL_ORIG_LON VAL_DEST_LON
#> 1      2.593877      41.05840      40.07335      78.48391      72.32824
#> 2      1.178165      40.37412      41.96093      77.01554     115.29323
#> 3      2.594083      40.90726      41.45727     117.02609      88.65927
#> 4      3.707060      41.18791      41.07091      92.59027      89.07035
#> 5      3.542982      41.94472      40.71440     115.76469     110.53817
#> 6      4.478668      41.09312      41.51065      82.63779      97.24406
#>   CAT_TRIP_TYPE CAT_RIDER_TYPE
#> 1      Elite      Frequent
#> 2      Pool       Frequent
#> 3      Elite      Subscription
#> 4      Pool       Frequent
#> 5      Pool       Frequent
#> 6      Pool       Frequent
```

Data Validation

The “promises” in variable names aren’t just there for decoration. They can actually help producers publish higher quality data my helping to automate data validation checks. Data quality is context-specific and requires effort on the consumer side, but setting up safeguards in any data pipeline can help detect and eliminate commonplace errors like duplicated or corrupt data.

Of course, setting up data validation pipelines isn't the most exciting part of any data engineer's job. But that's where R's pointblank package comes to the rescue with an excellent domain-specific language for common assertive data checks. Combining this syntax with dplyr's "select helpers" (such as [starts_with\(\)](#)), the same validation pipeline could ensure many of the data's promises are kept with no additional overhead. For example, N_ columns should be strictly non-negative and IND_ columns should always be either 0 or 1.

The following example demonstrate the R syntax to write such a pipeline in pointblank, but the package also allows rules to be specified in a standalone YAML file which could further increase portability between projects.

```
agent <-
  data_trips %>%
    create_agent(actions = action_levels(stop_at = 0.001)) %>%
    col_vals_gte(starts_with("N"), 0) %>%
    col_vals_gte(starts_with("N"), 0) %>%
    col_vals_not_null(starts_with("IND")) %>%
    col_vals_in_set(starts_with("IND"), c(0,1)) %>%
    col_is_date(starts_with("DT")) %>%
    col_vals_between(matches("_LAT(_|$)"), 19, 65) %>%
    col_vals_between(matches("_LON(_|$)"), -162, -68) %>%
    interrogate()
```

Pointblank Validation													
agent_2020-09-07_06:57:26 (2020-09-07 06:57:27)													
STEP		COLUMNS	VALUES	TBL	Eval	UNITS	PASS	FAIL	W	S	N	EXT	
1		col_vals_gte()	N_DRIVER_PASS... 0			100	100 1.00	0 0.00	—		—	—	
2		col_vals_gte()	N_TRIP_ORIG 0			100	100 1.00	0 0.00	—		—	—	
3		col_vals_gte()	N_TRIP_ORIG.1 0			100	100 1.00	0 0.00	—		—	—	
4		col_vals_gte()	N_DRIVER_PASS... 0			100	100 1.00	0 0.00	—		—	—	
5		col_vals_gte()	N_TRIP_ORIG 0			100	100 1.00	0 0.00	—		—	—	
6		col_vals_gte()	N_TRIP_ORIG.1 0			100	100 1.00	0 0.00	—		—	—	
7		col_vals_not_null()	IND_SURGE —			100	100 1.00	0 0.00	—		—	—	
8		col_vals_in_set()	IND_SURGE 0, 1			100	100 1.00	0 0.00	—		—	—	
9		col_is_date()	DT_ORIG —			1	1 1.00	0 0.00	—		—	—	
10		col_is_date()	DT_DEST —			1	1 1.00	0 0.00	—		—	—	
11		col_vals_between()	VAL_ORIG_LAT [19, 65]			100	100 1.00	0 0.00	—		—	—	
12		col_vals_between()	VAL_DEST_LAT [19, 65]			100	100 1.00	0 0.00	—		—	—	
13		col_vals_between()	VAL_ORIG_LON [-162, -68]			100	0 0.00	100 1.00	—		—	CSV	
14		col_vals_between()	VAL_DEST_LON [-162, -68]			100	0 0.00	100 1.00	—		—	CSV	

In the example above, just 7 lines of portable table-agnostic code end up creating 14 data validation checks. The results catch two errors. Upon investigation², we find that our geocoder is incorrectly flipping the sign on longitude!

One could also imagine writing a linter or validator of the variables names themselves to check for typos, outliers that don't follow common stubs, etc.

Data Discoverability

On the user side, a controlled vocabulary makes new data easier to explore. Although is is not and should not be a replacement for a true data dictionary, imagine how relatively easy it is to understand the following variables' intent and navigate either a searchable tab of visualization of the output.

To make some accessible outputs, we can first wrangle the column names into a table of their own.

```
cols_trips <- names(data_trips)
cols_trips_split <- strsplit(cols_trips, split = "_")
cols_components <- data.frame(
  variable = cols_trips,
  level1 = vapply(cols_trips_split, FUN = function(x) x[1], FUN.VALUE = character(1)),
  level2 = vapply(cols_trips_split, FUN = function(x) x[2], FUN.VALUE = character(1)),
  level3 = vapply(cols_trips_split, FUN = function(x) x[3], FUN.VALUE = character(1))
)
head(cols_components)
#>      variable level1 level2 level3
#> 1 ID_DRIVER   ID DRIVER   <NA>
#> 2 ID_RIDER   ID  RIDER   <NA>
#> 3 ID_TRIP    ID   TRIP   <NA>
#> 4 DT_ORIG    DT   ORIG   <NA>
#> 5 DT_DEST    DT   DEST   <NA>
#> 6 N_DRIVER_PASSENGERS N DRIVER PASSENGERS
```

Part of the metadata, then, could make it particularly easy to search by various stubs – whether that be the measure type (e.g. N or AMT) or the measure subject (e.g. RIDER or DRIVER).³[DT output with searchable columns³

```
library(DT)
datatable(cols_components, filter = list(position = 'top', clear = FALSE))
```

Show 10 entries

Search:

	variable	level1	level2	level3
	<input type="text" value="All"/>	<input type="text" value="All"/>	<input type="text" value="All"/>	<input type="text" value="All"/>
1	ID_DRIVER	ID	DRIVER	
2	ID_RIDER	ID	RIDER	
3	ID_TRIP	ID	TRIP	
4	DT_ORIG	DT	ORIG	
5	DT_DEST	DT	DEST	
6	N_DRIVER_PASSENGERS	N	DRIVER	PASSENGERS
7	N_TRIP_ORIG	N	TRIP	ORIG
8	N_TRIP_DEST	N	TRIP	DEST
9	AMT_TRIP_DIST	AMT	TRIP	DIST
10	IND_SURGE	IND	SURGE	

Showing 1 to 10 of 18 entries

Previous

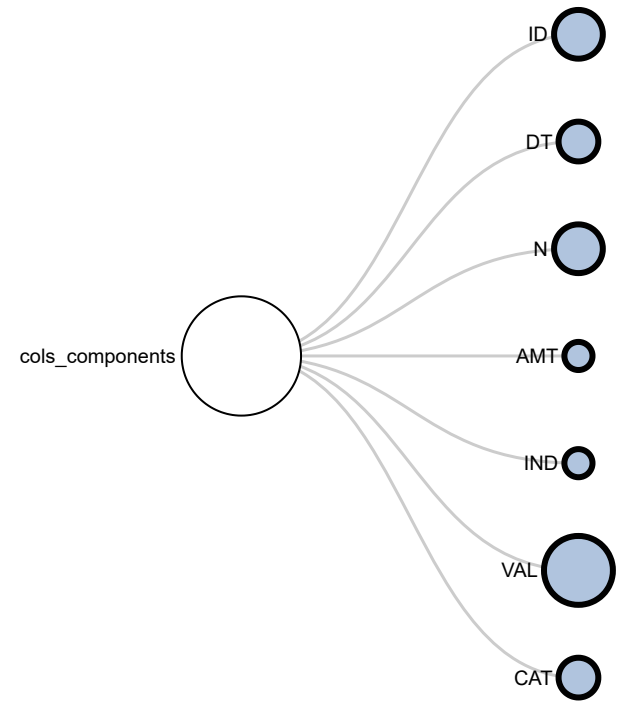
1

2

Next

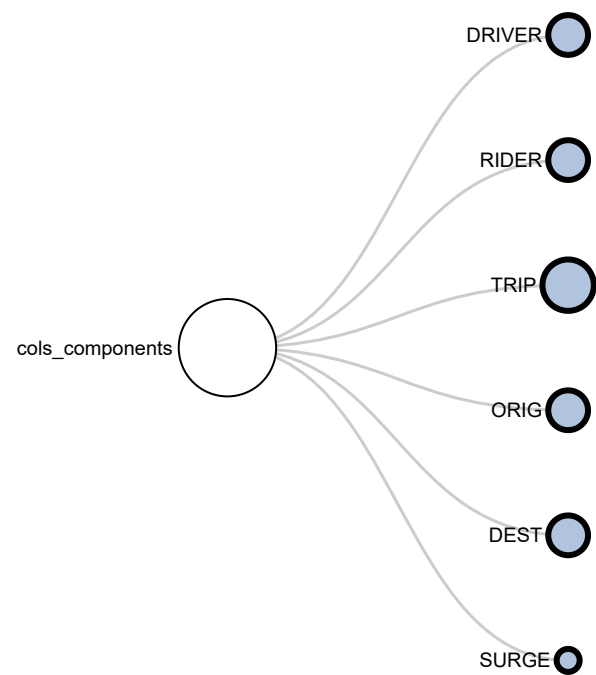
Similarly, we can use visualization to both validate and explore the available fields. Below, data fields are illustrated in a tree.

```
library(collapsibleTree)
collapsibleTree(cols_components,
  hierarchy = paste0("level", 1:3),
  nodeSize = "leafCount"
)
```



Depending on the type of exploraiion being done, it might be more convenient to drilldown first by measure subject. collapsibleTree flexibly lets us control this by specifying the hierarchy.

```
collapsibleTree(cols_components,
  hierarchy = paste0("level", c(2,1,3)),
  nodeSize = "leafCount"
)
```



These naming conventions are particularly friendly to a “passive search” via an IDE with autocomplete functionality. Simply typing “N_” and pausing or hitting tab might elicit a list of potential options of count variables in the data set.

More broadly, driving this standardization opens up interesting possibility for variable-first documentation. As our grammar for describing fields becomes richer and less ambiguous, it’s increasingly possible for users to explore a variable-first web of quantities and work their way back to the appropriate tables which contain them.

Data Wrangling

Controlled, hierarchical vocabularies also make basic data wrangling pipelines a breeze. By programming on the column names, we can appropriately summarize multiple pieces of data in the most relevant way.

For example, the following code uses `dplyr`’s “select helpers” to sum up count variables where we might reasonably be interested in the total and find the arithmetic average of indicator variables to help us calculate the proportion of occurrences of an event (here, the incidence of surge pricing).

Note what our controlled vocabulary and the implied “contracts” have given us. We aren’t summing up fields like latitude and longitude which would have no inherent meaning. Conversely, we can confidently calculate proportions which we couldn’t do if there was a chance our indicator variable contained nulls or occasionally used other numbers (e.g. 2) to denote something like the surge severity instead of pure incidence.

```
library(dplyr)
```

```
data_trips %>%
  group_by(CAT_RIDER_TYPE) %>%
  summarize(
    across(starts_with("N_"), sum),
    across(starts_with("IND_"), mean)
  )
#> # A tibble: 3 x 5
#>   CAT_RIDER_TYPE N_DRIVER_PASSENGERS N_TRIP_ORIG N_TRIP_DEST IND_SURGE
#>   <chr>          <int>          <dbl>          <dbl>          <dbl>
#> 1 Basic          51             30             30             0.533
#> 2 Frequent       48             32             32             0.469
#> 3 Subscription  55             38             38             0.421
```

Addendum on Other Languages

The above examples use a few specific R packages with helpers that specifically operate on column names. However, the value of this approach is language agnostic since most popular languages for data manipulation support character pattern matching and wrangling operations specified by lists of variable names. We will conclude with a few examples.

Generating SQL

Although SQL is a hard language to “program on”, many programming-friendly tools offer SQL generators. For example, using `dbplyr`, we can use R to generate SQL code that sums up all of our count variables by rider type without having to type them out manually.

```
library(dbplyr)

df_mem <- memdb_frame(data_trips, .name = "example_table")

df_mem %>%
  group_by(CAT_RIDER_TYPE) %>%
  summarize_at(vars(starts_with("N_")), sum, na.rm = TRUE) %>%
  show_query()

#> <SQL>
#> SELECT `CAT_RIDER_TYPE`, SUM(`N_DRIVER_PASSENGERS`) AS `N_DRIVER_PASSENGERS`, SUM(`N_TRIP_ORIG`) AS `N_TRIP_ORIG`, SUM
#> FROM `example_table`
#> GROUP BY `CAT_RIDER_TYPE`
```

R - base & data.table

However, we aren't of course limited just to tidyverse style coding. Similarly concise workflows exists in both base and data.table syntaxes. Suppose we wanted to summarize all numeric variables. First, we can use `base::grep` to find all column names that begin with N_.

```
cols_n <- grep("^N_", names(data_trips), value = TRUE)
print(cols_n)
#> [1] "N_DRIVER_PASSENGERS" "N_TRIP_ORIG"          "N_TRIP_DEST"
```

We can define the variables we want to group by in another vector.

```
cols_grp <- c("CAT_RIDER_TYPE")
```

These vectors can be used in aggregation operations such as `stats::aggregate`:

```
aggregate(data_trips[cols_n], by = data_trips[cols_grp], FUN = sum)
#>   CAT_RIDER_TYPE N_DRIVER_PASSENGERS N_TRIP_ORIG N_TRIP_DEST
#> 1      Basic          51             30          30
#> 2    Frequent          48             32          32
#> 3 Subscription          55             38          38
```

Or with data.table syntax:

```
library(data.table)
dt <- as.data.table(data_trips)
dt[, lapply(.SD, sum), by = cols_grp, .SDcols = cols_n]
#>   CAT_RIDER_TYPE N_DRIVER_PASSENGERS N_TRIP_ORIG N_TRIP_DEST
#> 1:    Frequent          48             32          32
#> 2: Subscription          55             38          38
#> 3:      Basic          51             30          30
```

python pandas


Similarly, we can use list comprehensions in python to create a list of columns names matching a specific pattern (cols_n). This list and a list to define grouping variables can be passed to pandas's data manipulation methods.

```
import pandas as pd
cols_n = [vbl for vbl in data_trips.columns if vbl[0:2] == 'N_']
cols_grp = ["CAT_RIDER_TYPE"]
data_trips.groupby(cols_grp)[cols_n].sum()

#>              N_DRIVER_PASSENGERS  N_TRIP_ORIG  N_TRIP_DEST
#> CAT_RIDER_TYPE
#> Basic                      51           30.0           30.0
#> Frequent                    48           32.0           32.0
#> Subscription                 55           38.0           38.0
```





- 1. Again, vocabularies should span a database – not just an individual dataset, but for simplicity we just talk through a smaller example. [↪](#)
- 2. the output of pointblank is actually an interactive table. For blog-specific reasons, I show only the png here. [↪](#)
- 3. As with pointblank, DT output is interactive, but my blog unfortunately reacts poorly to the extra JavaScript so for now I'm only showing an image [↪](#)





Emily Riederer

Senior Analytics Manager



One of the challenges with all these syntaxes is portability. For example, if you have a system that understands geospatial data and you export data to a system that does not, do you change the name of the column? I would say yes, but that might be unexpected behavior downstream.

Also, as your data architecture evolves, your syntax may need to evolve as well. For example, the world used to be all CSV/RDBMS related with columns and rows being the best way to represent things. Then, JSON and NoSQL came along and turned things on their ear. I would be interested to see how you might approach a more complex data model with the syntax.

Great article!

bvancil commented 3 months ago

Thank you so much for writing this! I especially appreciate the depth of thought you manifested in metadata discoverability and data testing.

I'm curious to know, if you were dealing with many probabilities (perhaps predicted in a modeling step), would you prefer extending the level 1 schema, e.g. `P_SUBJECT_DETAIL` or `PRB_SUBJECT_DETAIL` , or using the level 1 `VAL_` prefix, e.g. `VAL_SUBJECT_DETAIL_PROB` ?

jeffzi commented 3 months ago

Thanks for the nice (and very complete) write-up.

I follow similar naming conventions but I use *level 1 - measure types* as a suffix instead of a prefix. In my opinion it makes it easier to identify similar *level 2 - measure subjects*. It's subjective, but I also find it more natural to read.

For example: `N_TRIP_ORIG`, `N_TRIP_DEST`, `AMT_TRIP_DIST` becomes `TRIP_ORIG_N`, `TRIP_DEST_N`, `TRIP_DIST_AMT`

emilyriederer commented 3 months ago

Owner

[@morgango](#) - Thanks so much for those references! I've enjoyed reading up on both Hungarian notation and ECS. I love seeing the similarities / differences. Even more, thank you for flagging those challenges.

I didn't emphasize it in the article, but I also hope conventions like these could help preserve data lineage (since the same quantities should naturally end up with the same names), but your point on GIS and different system requirements is fascinating when you must either sacrifice the standard or break the contract. Like you, I'm inclined to sacrifice the standard to ensure the contract holds (e.g. change the name if the data can no longer perform as specified), but it is uncomfortable.

I'd have to think more about NoSQL, but for JSON, my first guess is that you might define appropriate names at different levels of nesting. The [GitHub API](#) I think does a decent job of this with entities as the top level and some level of standardization in the fields within each. Do you think that could work?

Thanks so much for writing back! I really love hearing these corner cases.

emilyriederer commented 3 months ago

Owner

[@bvancil](#) - That's an interesting question! I think it probably depends a bit how I would intend to use those probabilities, but my initial reaction would be to add to the level 1. My reasoning is that I probably have fundamentally different validation checks to perform on these (e.g. should be bounded between 0 and 1) and potentially different use cases. Of course, it's just as possible to use select helpers (`ends_with()`) or regex (`.*_PROB$`) to still handle variables the same if they are in the suffix, but to me this feel like "first-class" information. I tend to think of the suffix as more like adjectives, like `AMT_SUBJECT_3MO` or `AMT_SUBJECT_6MO` might reflect trailing 3 month and 6 month observations for the same measure of the same subject.

emilyriederer commented 3 months ago

Owner

[@jeffzi](#) - Thank you for sharing! You do make a good point that when one is at the beginning and the other is at the end, both can be highlighted. I do like that when I think about the autocomplete sense as well; perhaps users would rather "search" based on the entity of interest (for which a measure might or might not exist) rather than a measure 🤔 . I'll have to try out that way, too!

Related

- [A beginner's guide to Shiny modules](#)
- [projmgr: Managing the human dependencies of your projects](#)
- [RMarkdown Driven Development: the Technical Appendix](#)
- [RMarkdown Driven Development \(RmdDD\)](#)
- [Resource Round-Up: Latent and Lasting Documentation](#)

[LICENSE: CC BY-NC-SA 4.0](#)



© Emily Riederer 2020 · Made with the [R hugodown](#) package and the [Academic theme](#) for [Hugo](#). · Also shared on [R-bloggers](#).

