Neo4J data model

Note: This is a design document for the refactored version of `twittercache` that will leverage a Neo4J database backend.

Summary of Neo4J graph database model

Recall that Neo4J uses the labelled property graph model. The elements of the labeled property graph model are nodes, relationships, properties, and labels.

- Nodes contain properties. Properties are key-value pairs where the key is a string and the value is a semi-arbitrary value
- Nodes can have one or more labels. Labels indicate the role of nodes within a dataset
- Relationships connect nodes. Relationships are directed, and have a start node and an end node.
- Relationships can have properties, just like nodes.

Schema

In our database, we will represent each Twitter user and each tweet as a node. We will indicate that user A follows user B by adding an edge from user A to user B.

Note to Nathan: For the time being, focus on the `User` node and the `follows` relationship, the rest of this scheme describes lower priority future work.

Nodes and node labels

- `User`: a user in the Twitter graph. Nodes with `User` label should have the following properties:
  - `sampled_at <dttm>`: Takes on a sentinel NULL value to indicate user has not yet been fully sampled, but partial user information was added to the graph at some point. Otherwise a datetime indicate when full information from `lookup_users()` was added to the graph.
  - `sampled_friends_at`: `NULL` unless we know that *all* of a users friends are in the graph becaues we explicitly added all information from `get_friends()`, in which case this should be the datetime when we added this information to the graph.
  - `sampled_followers_at`: Same as `sampled_friends_at` but for followers instead of friends.
  - `user_id <chr>`
  - `screen_name <chr>`
  - `protected <lgl>`
  - `followers_count <int>`
  - `friends_count <int>`
  - `listed_count <int>`
  - `statuses_count <int>`
  - `favourites_count <int>`
  - `account_created_at <dttm>`
  - `verified <lgl>`
  - `profile_url <chr>`
  - `profile_expanded_url <chr>`
  - `account_lang <lgl>`
  - `profile_banner_url <chr>`

    - profile_background_url <lgl>
    - profile_image_url <chr>
    - name <chr>
    - location <chr>,
    - description <chr>
    - url <chr>
- Tweet (ignore for now)
    - sampled_at <dttm>
    - status_id <chr>
    - created_at <dttm>
    - text <chr>
    - source <chr>
    - display_text_width <int>
    - is_quote <lgl>
    - is_retweet <lgl>
    - favorite_count <int>
    - rtweet_count <int>
    - quote_count <int>
    - reply_count <int>
    - hashtags <list>
    - symbols <list>
    - urls_url <list>
    - urls_t.co <list>
    - urls_expanded_url <list>
    - media_url <list>
    - media_t.co <list>
    - media_expanded_url <list>
    - media_type <list>
    - ext_media_url <list>
    - ext_media_t.co <list>
    - ext_media_expanded_url <list>
    - ext_media_type <chr>
    - lang <chr>
    - status_url <chr>

Relationship and relationship:

- follows: A relationship strictly between two Users. For example, both @alexpghayes and @karlrohe follow each other.
- tweeted: A relationship between a User and a Tweet
- reply_to: A relationship between a User and a Tweet
- reply_to: A relationship between two Tweets
- mentions: A relationship between a User and a Tweet
- quotes: A relationship between two Tweets

API brainstorm and some pseudocode

In the following our goal will always be to make a maximum of a single database query and a single API query for any given request from the user.

```
#' @param users A character vector of user ids (never screen names)
#'
```

```r
#' @return A tibble where each row corresponds to a User and each column
#'   to one of the User properties. If a user cannot be sampled, should
#'   return nothing for that user. If no users can be sampled, should
#'   return an empty tibble with appropriate columns.
#'
lookup_users <- function(users) {

  user_data <- db_lookup_users(users)

  not_in_graph <- setdiff(users, user_data$user_id)

  new_user_data <- add_new_users(not_in_graph)

  not_sampled <- user_data %>%
    filter(is.null(sampled_at)) %>%
    pull(user_id)

  upgraded_user_data <- sample_existing_users(not_sampled)

  bind_rows(user_data, new_user_data, upgraded_user_data)
}

db_lookup_users <- function(users) {
  # return a tibble where each row corresponds to a User and each column
  # to one of the User properties. when a user in not present in the
  # database, should not return a row in the output tibble for that
  # user. if no users are in the db should return an empty tibble with one
  # column for each User property
}

#' @return The tibble of user data, with one row for each (accessible)
#'   user in `users` and one column for each property of `User` nodes
#'   in the graph database.
add_new_users <- function(users) {
  # make sure to set sampled_at to Sys.time() and
  # sampled_friends_at and sampled_followers_at to NULL

  # return data on users
}

sample_existing_users <- function(users) {
  # pretty much the same as add_new_users(), except overwrite
  # existing data on each user rather than creating new user nodes
  # from scratch
}
```

```r
#' @param users A character vector of user ids (never screen names)
#'
#' @return A tibble where each row corresponds to a User and each column
#'   to one of the User properties. If a user cannot be sampled, should
#'   return nothing for that user. If no users can be sampled, should
#'   return an empty tibble with appropriate columns.
get_friends <- function(users) {
```

```
  # here we will need to query twice: once to ask who we actually
  # have *complete* friendship edges for, and then a second time to get
  # those friendship edges
  status <- friend_sampling_status(users)

  # if you can add new edges to the database without
  # creating duplicate edges (using MERGE?) you may be able
  # to combine the logic in the following steps
  new_edges <- add_new_friends(status$not_in_graph)
  upgraded_edges <- sample_existing_friends(status$sampled_friends_at_is_null)
  existing_edges <- db_get_friends(status$sampled_friends_at_not_null)

  # need to be careful about duplicate edges here. ideally
  # we guarantee that edges are unique somehow before this, but if not
  # we can use dplyr::distinct(), although this is an expensive operation
  bind_rows(new_edges, upgraded_edges, existing_edges)
}


add_new_friends <- function(users) {
  # set sampled_friends_at to Sys.time() and
  # sampled_at and sampled_followers_at to NULL

  # return friends of each user
}

sample_existing_friends <- function(users) {
  # set sampled_friends_at to Sys.time() and
  # *leave* sampled_at and sampled_followers_at at their
  # existing values

  # first remove outgoing edges from users to other nodes
  # in the database, as this information may be outdated
  # and we're about to update it anyway

  # return friends of each user
}

friend_sampling_status <- function(users) {

  # generate based on queries of user.sampled_friends_at node property

  list(
    not_in_graph = ...,
    sampled_friends_at_is_null = ...,
    sampled_friends_at_not_null = ...
  )
}
```

`get_followers()` logic should be nearly identical to `get_friends()` logic.

For Fan: `get_timeline()`

I haven't thought through how to support `get_timeline()` in detail, mostly because I'm unclear of the various relationships between tweets, and between tweets and users. In particular, what kind of relationships between tweets and users should we add and when?

Misc thoughts

- A fair amount of information may be missing for any given `User` node – we need to carefully think about the various states a `User` node might be in and how this will impact various graph queries
- Any user generated text is liable to contain Unicode abominations

Note that `rtweet` returns several fields containing geographic information that I think we should discard because that shit is creepy (I'm not clear which of the following are `User` versus 'Tweet properties either): - `place_url <chr>`, - `place_name <chr>` - `place_full_name <chr>`, - `place_type <chr>` - `country <chr>`, - `country_code <chr>` - `geo_coords <list>`, - `coords_coords <list>` - `bbox_coords <list>`

- To think about: user states: healthy, banned, suspected, removed, etc, etc