sheets-db: Database powered by Google Spreadsheets

Aditya Kumar Ravikanti

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas School of Engineering in partial fulfillment of the requirements for the degree of Master of Science.

Project Committee:

Dr. Andy Gill: Chairperson

Dr. Perry Alexander

Dr. Prasad Kulkarni

Date Defended

The Project Committee for Aditya Kumar Ravikanti certifies
That this is the approved version of the following project:

sheets-db: Database powered by Google Spreadsheets

Committee:

_____

Chairperson

_____

_____

_____

Date Approved

Acknowledgements

I would like to take this opportunity to express sincere gratitude towards my advisor Dr. Andy Gill who has been a pillar of support throughout my masters project. Especially, I appreciate his patience, motivation, enthusiasm and resourcefulness which has helped me see through difficult times as I started my journey towards learning functional programming concepts. I can't emphasize more on how it helped in shaping my algorithmic and software design thinking and it will be truly beneficial for me in academics and professional career. I would also like to extend my special thanks to my graduate committee Dr. Perry Alexander and Dr. Prasad Kulkarni for being a part of my graduate school journey by helping me explore new areas of Computer Science through their courses and for timely help and suggestions.

It would be incomplete without mentioning the role of my family, especially my loving parents who have been a constant source of inspiration and without their support, I would not gone too far in life.

Abstract

Google spreadsheets or Google sheets is a web-based application program used for organization, analysis and storage of data in tabular form.

In this project, we develop sheets-db library which is a Haskell binding to Google Sheets API. sheets-db allows Haskell users to utilize Google spreadsheets as a light weight database. It provides various functions to create, read, update and delete rows in spreadsheets along with a way to construct simple structured queries.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Traditionally, applications use different databases such as Oracle, Postgres, MongoDB, MySQL, etc. to store data. Setting up this type of traditional database is a painful process which involves hosting database server, database management and requires people with special technical capabilities to retrieve and use information. The powerful features offered by traditional databases might not be required to certain class of applications. In this case using these traditional databases is an overkill which requires lots of effort in hosting, setting up, administration and learning curve to use. Also, non-technical users cannot read or manipulate data easily without knowledge of languages like SQL.

1.2 Google Spreadsheets

Google spreadsheets is a browser based application part of office suite offered by Google. It is used for organization, analysis and storage of data in tabular

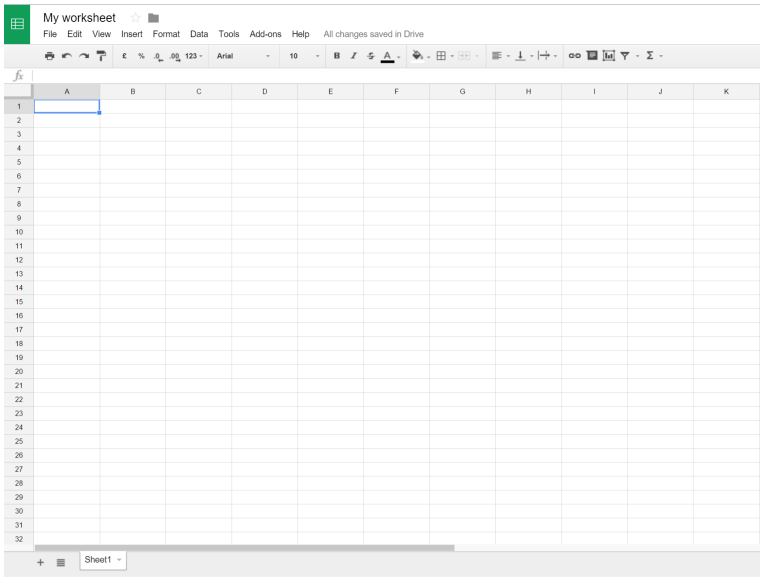

Figure 1.1. An example of Google spreadsheet

form. It is web-based solution similar to desktop-based Microsoft Excel. Google spreadsheets allows users to access their data through any browser and any desktop platform.

1.3 Google Sheets API

The Google Sheets API is a REST interface to Google spreadsheets which helps in developing client applications that can read and modify data in spreadsheets.

Capabilities of this API:

1. Managing the worksheets in Google Spreadsheets
2. Consuming the rows of a worksheet
3. Managing cells in a worksheet by position

Common use cases for this API:

1. Access spreadsheets data through any application for computation.

2. Generate statistics and showing them through different User Interfaces.

3. Automating storage of data through any application.

1.4 Motivation for sheets-db

To solve the problem of overhead involved in using traditional databases, a library can be developed to utilize spreadsheets as light weight database to work with data through Google Sheets API. Google's default spreadsheets interface can be utilized by non-technical users to read, modify and analyze data easily. Although, Sheets API does not offer powerful features as traditional databases, it is reliable and can completely achieve purposes of small applications.

This acted as a motivation to develop sheets-db, a Haskell library that can be added as a dependency to any Haskell application to automate storage and retrieval of data in spreadsheet.

Chapter 2

Terminology used

1. **spreadsheet:** "Google Spreadsheets document, created with the Google Spreadsheets user interface or with the Google Drive API" [1]. This is similar to traditional database, which is a collection of tables.

2. **worksheet:** "Named collection of cells within a spreadsheet" [1]. All spreadsheets must have one worksheet by default. This corresponds to a table in traditional database.

3. **list row:** "Row of cells in a worksheet, represented as a key-value pair, where each key is a column name, and each value is the cell value. The first row of a worksheet is always considered the header row when using the Sheets API, and therefore is the row that defines the keys represented in each row" [1]. This corresponds to a single row in a table.

Chapter 3

Using sheets-db

This chapter discusses the functions offered by sheets-db and examples to use sheets-db in any other application.

3.1 Obtaining Key of a spreadsheet and id of a worksheet

Create a new spreadsheet at https://docs.google.com/spreadsheets. Every spreadsheet has a worksheet by default and new worksheets can be created by clicking "+" button at the left hand corner. A spreadsheet is uniquely identified by Google using a key which is assigned when it is first created. When a spreadsheet is opened in the browser the URL looks like:

```
https://docs.google.com/spreadsheets/d/
1hIEq4AAauzI8INelQRIvgxBhmzX44qAB_1QQpFJQ2Xo/edit
```

The key of the spreadsheet created is colored and is required for sheets-db to identify your spreadsheet. A worksheet in a spreadsheet contains the data and the id of the worksheet is the serial number, starting from 1, obtained by counting

the worksheets at the bottom from left to right till the current worksheet. The key of the spreadsheet and the id of worksheet uniquely identify a worksheet.

3.2 Registering your application on Google Developers Console for OAuth

The spreadsheet created belongs to a particular user and is not public. To access non-public user spreadsheets programatically through HTTP requests, the web requests need to pass an authentication token that validates the usage of data and also the application accessing it.

The protocol **OAuth 2.0** must be followed to accomplish the above process. A developer who wishes to use sheets-db in their application must follow the below steps to generate **client id** and **client secret** required for OAuth authentication process. Generating the client id and secret gives an application unique identity.

1. We must obtain OAuth 2.0 credentials (client id and secret) from the Google Developers Console.

2. After clicking the link above you will be redirected to login using your Google account.

3. After login click on the left side top corner navigation button to draw out the side panel.

4. Click the API Manager Menu item.

5. In the sub menu of API manager click on the Credentials item. Here you can create a new application which gets unique client id and secret for that application.

6. Create a new application by clicking on *"New Credentials"* drop down to choose *"OAuth Client ID"*.

7. Then choose the application type as *"other"* and give your application a name you wish and click on create.

8. A new application gets created. Client ID and Client Secret are displayed against the application's row. Copy these credentials for use in your application.

3.3 Initializing a worksheet in spreadsheet

The first row of the worksheet is the header row and must contain all the column names. A column without header name is not considered part of data. When a spreadsheet is first created appropriate header names must be manually created without spaces. Also the first row after the header row must contain dummy data. This is required to initialize the table as Sheets API doesn't return any data about column names if there are no rows by default.

3.4 Essential datatypes in sheets-db

Each worksheet is represented as a Sheet.

```
data Sheet = Sheet
{ key         :: String -- key of the spreadsheet
, worksheetId :: String -- id of current worksheet
, columns     :: [ColName] -- column names
, cid         :: String -- client id created in dev console
, csecret     :: String -- client secret created
} deriving (Show,Eq)


type ColName = T.Text
```

A spreadsheet data table is made from list of `Row`s. A `Row` is made of list of `Cell`s where each cell data corresponds to respective header column name.

```
data Cell = (:=)
{ colname :: !ColName
, value   :: Value
}
deriving (Typeable, Eq)


(=:) :: (Val v) => ColName -> v -> Cell
-- cell with given column name and typed value


(=?) :: (Val a) => ColName -> Maybe a -> Row
-- If Just value then return one cell row, otherwise return empty row
```

Each `Cell` is a key value pair of column name and the corresponding data from the worksheet cell. Special symbol function `(=:)` can be used to construct key value pairs of `Cell`. The valid values in the cell are instances of `class Val`.

```
class (Typeable a, Show a, Eq a) => Val a where
val :: a -> Value
cast :: Value -> API a


instance Val T.Text
instance Val String
instance Val Scientific
instance Val Bool
instance Val (Maybe Value)
instance Val UTCTime
instance Val POSIXTime
instance Val Float
instance Val Double
```

The 2 methods of the `class Val` are `val` and `cast`. `val` specifies how a particular data is converted to `Value` type and `cast` shows how a `Value` can be converted back to its original datatype.

Finally, the `Value` datatype:

```
data Value =
Number !Scientific |
String !T.Text |
Bool Bool |
UTC UTCTime |
Null
deriving (Typeable, Eq)


instance Show Value
```

sheets-db allows a user to perform simple structured queries based on the abilities given by google sheets API. A structured query can be constructed using the `Selector` datatype.

```
data Selector
= Gtr   ColName ST.Value
| Gteqr ColName ST.Value
| Ltr   ColName ST.Value
| Lteqr ColName ST.Value
| Eqr   ColName ST.Value
| And   Selector Selector
| Or    Selector Selector
| Everyting
deriving (Show, Eq)


(~>), (~>=), (~<), (~<=), (~=) :: (Val v) => ColName -> v -> Selector
k ~>  v = Gtr   k (val v)
k ~>= v = Gteqr k (val v)
k ~<  v = Ltr   k (val v)
k ~<= v = Lteqr k (val v)
k ~=  v = Eqr   k (val v)


(~&&~), (~||~) :: Selector -> Selector -> Selector
(~&&~) = And
(~||~) = Or


everyting :: Selector
everyting = Everyting
```

sheets-db also allows a user to sort based on a specific column either in ascending or descending order using the `Order` datatype.

```
data Order = Order
{ colName :: ColName
, reverse :: Bool
} | NoOrder -- Indicates no specific order is required
deriving (Show, Eq)
```

The `Select` constructor is used to construct `Selection` data on a `Sheet`.

Finally, `Query` constructor is used to construct a query.

```
data Selection = Select
{ selector :: Selector
, sheet    :: Sheet
}
deriving (Show, Eq)
-- Selects rows in spreadsheet that match selector


data Query = Query
{ selection :: Selection
, skip      :: Int  -- Number of initial matching documents to skip. Default = 0
, limit     :: Int -- Maximum number of documents to return, 0 = no limit. Default = 0
, sort      :: Order  -- Sort results by this order, NoOrder = no sort. Default = NoOrder
} deriving (Show, Eq)
```

An example query will look like this:

```
let q = Query (Select ((p "type" ~< (2 :: Float)) ~||~
                       (p "category" ~= "meat" ) sheet) 0 10
                       (Order (p "category") False)
```

3.5 API Monad Transformer

Every library requires some kind of error handling, logging and passing common configuration around. For achieving these functions `API` monad transformer is created using `ExceptT`, `WriterT` and `ReaderT`.

```
type API = ExceptT  String ( WriterT [String] ( ReaderT  Manager IO ))
```

Every operation in the library returns `API a`. As every monad transformer has a run method, `API` has one too.

```
runAPI :: API a -> IO (Either String a)
```

Use `runAPI` method to run `API` computations.

```
io :: IO a -> API a
```

Use `io` method to lift `IO` computations to `API` computations.

3.6 Constructing a Sheet

Various operations can be performed on a worksheet but before that `Sheet` data mentioned above must be constructed. To do that `access` method must be used.

```
access ::   String -- key of the spreadsheet
         -> String -- worksheet id of current spreadsheet
         -> String -- client Id of the application created in Dev Console
         -> String -- client Secret created in Dev Console
         -> API Sheet
```

The above method constructs a `Sheet` if all the credentials are correct.

3.7 Inserting a Row into worksheet

After initializing the `Sheet` data, various operations can be performed. To insert a new row into the worksheet a `Row` must be constructed using the `[ColName]` in the `Sheet` data and `(=:)` operator function.

For a spreadsheet with column names: name, category, healthiness, type. An example `Row` can be constructed by

```
toInsert :: Row
toInsert= [p "name"        =:  "papaya",
           p "category"    =:  "fruit",
           p "healthiness" =:  "adequate",
           p "type"        =:  (3 :: Float)]
```

Here p is alias for Text.pack function which is used very frequently. And insert function is

```
-- Adds a new row to google sheet after validation of data to be inserted
insert ::    Row -- The row to be inserted containing all column values
          -> Sheet -- The spreadsheet in which the data must be inserted
          -> API ()
```

Complete example illustrating insertion of a `Row`

```
clientid = "1086783968140-knfg08qu9onnn5b485veaskt2flr0loa.apps.googleusercontent.com"
clientsecret ="wRktcD6xif1_z6Xv-RhCatBB"


toInsert :: Row
toInsert = [p "name"        =:  "papaya",
            p "category"    =:  "fruit",
            p "healthiness" =:  "adequate",
            p "type"        =:  (3 :: Float)]


main = do
  let key = "1hIEq4AAauzI8INelQRIvgxBhmzX44qAB_1QQpFJQ2Xo"
  let doc = "1"


  let firstDocument = access key doc clientid clientsecret


  runAPI $ do
    sheet1 <- firstDocument
    insert toInsert sheet1
```

3.8 Removing a Row from worksheet

A `Row` can be removed by using the `remove` method. The `Row` to be removed must contain the id of the `Row` otherwise the operation will fail.

```
-- Used to delete a Row from the Google Sheet.
remove ::   Sheet -- The spreadsheet in which a row needs to be removed
         -> Row -- The row which is to be deleted with "id"
         -> API ()
```

3.9 Updating a Row in worksheet

After a `Row` is fetched from worksheet it can be changed and inserted back into the worksheet using the update function. The "id" field must exist else the operation fails. The operation also fails if all the column names are not present including the original cells which do not change.

```
update :: Sheet -- The spreadsheet in which a row needs to be updated
       -> Row -- Row to be updated along with "id" of row to be updated
       -> API ()
```

3.10 Find Rows in Spreadsheet

Structured queries can be constructed and rows satisfying these criteria are returned using the `find` method.

```
find :: Query -- Query to be run includes the sheet
     -> API [Row]
```

Example of find and query construction:

```
clientid = "1086783968140-knfg08qu9onnn5b485veaskt2flr0loa.apps.googleusercontent.com"
clientsecret ="wRktcD6xif1_z6Xv-RhCatBB"


main = do
  let key = "1hIEq4AAauzI8INelQRIvgxBhmzX44qAB_1QQpFJQ2Xo"
  let doc = "1" -- worksheet id


  let firstDocument = access key doc clientid clientsecret


  runAPI $ do
    sheet1 <- firstDocument
    rows <- find (Query (Select
                         ((p "type"     ~<  (2 :: Float)) ~||~
                          (p "category" ~=  "meat" )) sheet1) 0 0
                         (Order (p "category") False))
    io (mapM_ print rows)
```

Chapter 4

Approach

This chapter discusses implementation of sheets-db in detail. This mainly focuses on how main methods mentioned in the previous chapter are implemented and also describes about the various Haskell libraries used in the library.

4.1 Google Spreadsheet URL construction

sheets-db communicates with Google sheets REST API to achieve various functions described in previous chapter. Different URLs are constructed based on the requirement.

The basic URL used throughout the code is:

```
https://spreadsheets.google.com/feeds/list/<key>/<worksheetId>/private/full
```

This URL is known as list feed URL. A spreadsheet is uniquely identified by `<key>` and each spreadsheet can contain multiple worksheets which are uniquely identified by `<worksheetId>`.

4.1.1 URL to fetch rows

A GET request is sent to the list feed URL to retrieve the rows in the spreadsheet. This GET request returns content in XML but sheets-db retrieves rows in JSON format using the URL:

```
GET https://spreadsheets.google.com/feeds/list/<key>/<worksheetId>/private/full?alt=json
```

To get only the rows that meet a specified criteria structured query parameter "sq" can be used to construct query criteria.

```
GET https://spreadsheets.google.com/feeds/list/<key>/<worksheetId>/
private/full?alt=json&sq=<encoded-query>
```

A query example from Sheets API itself

```
age > 25 and height < 175
```

Every query should be constructed in this way and always be encoded to use as a parameter value with "sq" in the above URL.

4.1.2 URL to add new rows

To add a new row to the spreadsheet a post request should be sent to list feed URL along with the new row XML constructed as in the example shown below.

```
POST https://spreadsheets.google.com/feeds/list/<key>/<worksheetId>/private/full


<entry xmlns="http://www.w3.org/2005/Atom"
xmlns:gsx="http://schemas.google.com/spreadsheets/2006/extended">
<gsx:hours>1</gsx:hours>
<gsx:ipm>1</gsx:ipm>
<gsx:items>60</gsx:items>
<gsx:name>Elizabeth Bennet</gsx:name>
</entry>
```

The XML example is taken from Google sheets API documentation. In this XML hours, ipm, items, name correspond to the column names and the values of each XML node correspond to the values of the columns.

4.1.3 URL to update a row

After a GET request, an example row in the JSON format looks like:


```
{
        "id": {
                "$t": "https://spreadsheets.google.com/feeds/list/
                1hIEq4AAauzI8INelQRIvgxBhmzX44qAB_1QQpFJQ2Xo/od6/private/full/chk2m"
        },
        "updated": {
                "$t": "2015-11-25T19:08:07.834Z"
        },
        "category": [
        {
                "scheme": "http://schemas.google.com/spreadsheets/2006",
                "term": "http://schemas.google.com/spreadsheets/2006#list"
        }
        ],
        "title": {
                "type": "text",
                "$t": "Carrot"
        },
        "content": {
                "type": "text",
                "$t": "category: Vegetable, healthiness: Adequate, type: 1"
        },
        "link": [
        {
                "rel": "self",
                "type": "application/atom+xml",
                "href": "https://spreadsheets.google.com/feeds/list/
                1hIEq4AAauzI8INelQRIvgxBhmzX44qAB_1QQpFJQ2Xo/od6/private/full/chk2m"
```

```
        },
        {
                "rel": "edit",
                "type": "application/atom+xml",
                "href": "https://spreadsheets.google.com/feeds/list/
                1hIEq4AAauzI8INelQRIvgxBhmzX44qAB_1QQpFJQ2Xo/od6/private/full/chk2m/9a586cd3cmc72"
        }
        ],
        "gsx$name": {
                "$t": "Carrot"
        },
        "gsx$category": {
                "$t": "Vegetable"
        },
        "gsx$healthiness": {
                "$t": "Adequate"
        },
        "gsx$type": {
                "$t": "1"
        }
}
```


To update this row, we must send a PUT request to the edit URL along with the XML of the updated row. Edit URL can be parsed by finding value of "href" in the object with, ("rel" : "edit") in "link" value array.

4.1.4 URL to remove a row

In the above JSON, when we send HTTP DELETE request to the id URL the row will be removed.

```
DELETE https://spreadsheets.google.com/feeds/list/<key>/<worksheetId>/private/full/<Row ID>
```

4.2 Module GoogleRequest

To achieve various functions, GET, POST, PUT, DELETE HTTP requests need to be constructed and sent to Google sheet API server. To make these requests along with OAuth authentication, methods `get`, `post`, `put`, `delete` respectively are created in GoogleRequest.

These methods are utilized by `SheetDB` module.

4.2.1 OAuth Authentication and google-oauth2 library

Before an application can access private data using any Google API, it must obtain an access token that grants access to that API. A variable parameter called scope controls the set of resources and operations that an access token permits. During the access token request, our application sends one or more values in the scope parameter. For Google sheets API the scope we must send is

```
https://spreadsheets.google.com/feeds
```

When our application needs access to sheets data, it asks Google for above scope of access. Google displays an OAuth dialog to the user, asking them to authorize your application to request their private data. If the user approves, then Google gives our application a short-term access token. Our application can then perform different http requests (GET, POST, PUT, and DELETE) by attaching the access token to the request. If Google determines that our request and the token are valid, it performs the HTTP request. The google-oauth2 [3] library on hackage interacts with the Google OAuth2 authorization API to perform these functions for us. The example taken from the above link illustrates the process involved in using the library. The steps involved are:

1. Prompt the user for a verification code
2. POST that code to the Google API for a set of tokens (access and refresh)
3. Use the access token until it expires
4. Use the refresh token to get a new access token
5. Repeat from 3

```
import Data.Monoid
import Network.Google.OAuth2
import Network.HTTP.Conduit
import Network.HTTP.Types (hAuthorization)

import qualified Data.ByteString.Char8 as B8
import qualified Data.ByteString.Lazy.Char8 as L8

main :: IO ()
main = do
  let client = OAuth2Client clientId clientSecret
  scopes = ["https://www.googleapis.com/auth/drive"]
  token <- getAccessToken client scopes Nothing
  request <- parseUrl "https://www.googleapis.com/drive/v2/files"
  response <- withManager $ httpLbs $ authorize token request
  L8.putStrLn $ responseBody response
where
  authorize token request = request{ requestHeaders =
  [(hAuthorization, B8.pack $ "Bearer " <> token)] }
  -- Setup in Google Developers Console
  clientId = "..."
  clientSecret = "..."
```

The above example demonstrates the HTTP request process with the Google Drive API and it is similar for Google Sheets API. The exact steps are followed in GoogleRequest module for completing the authentication.

The library provides function called:

```
getAccessToken :: OAuth2Client
               -> [OAuth2Scope]
               -> Maybe FilePath -- File in which to cache the token
               -> IO OAuth2Token -- Refreshed token
```

Which takes `OAuth2Client` (Client ID and Client Secret), scope [https://spreadsheets.google.com and a file for caching the credentials once obtained from the server.

When we supply these parameters to the function and in case it executes for the first time it prompts the user for a verification code. It displays a URL which we need to use to generate verification code from the browser by authenticating with the google account that owns the sheet. After obtaining the verification code, supply to the prompt. Then the library uses the verification code to obtain a set of tokens (access and refresh) from Google OAuth2 authorization API. The credentials which are obtained are stored in the file we supplied as parameter, for reuse. We then use the access token for subsequent requests until it expires. We use the refresh token to get a new access token. These above steps are all done by `getAccessToken` function.

In `GoogleRequest` there are various functions that perform the above tasks by supplying correct parameters.

```
tokenOf :: Sheet -> IO String
tokenOf sheet = createToken $ OAuth2Client (cid sheet) (csecret sheet)


scopes = ["https://spreadsheets.google.com/feeds"]


createToken client = getAccessToken client scopes (Just "./key.txt")


authorize token request = request
{ requestHeaders =
(hAuthorization, B8.pack $ "Bearer " ++ token) : requestHeaders request }
```

The above `authorize` function is used to add the token header to the request to be sent to Google Sheets API for authentication.

4.2.2 HTTP requests with http-conduit

GoogleRequest uses http-conduit for making web requests.

```
httpLbs :: MonadIO m => Request -- Request to make
                     -> Manager -- Connection manager
                     -> m (Response ByteString)
```

`httpLbs` method is used to make HTTP requests. The Request data is constructed based on the function being performed.

The below table gives an overview of what are the contents of the Request in each `get`, `post`, `put`, `delete` methods of GoogleRequest and their return types.

GoogleRequest methods	Request method	Request Headers	Request Body	Return type	URL
get	GET	Authorization	None	responseBody	List feed URL
post	POST	Authorization, content-type, Gdata-version	XML of the row data	None	List feed URL
put	PUT	Authorization, content-type, Gdata-version	XML of row data to be modified	None	Edit URL of the row
delete	DELETE	Authorization	None	None	Id URL of the row

4.3 Module SheetDB

This module exposes core functionality of the library. The main methods are `find`, `insert`, `update`, and `remove`. Figure 4.1 shows an overview of the algorithm for `find`, `insert`, `update`, and `remove` methods.

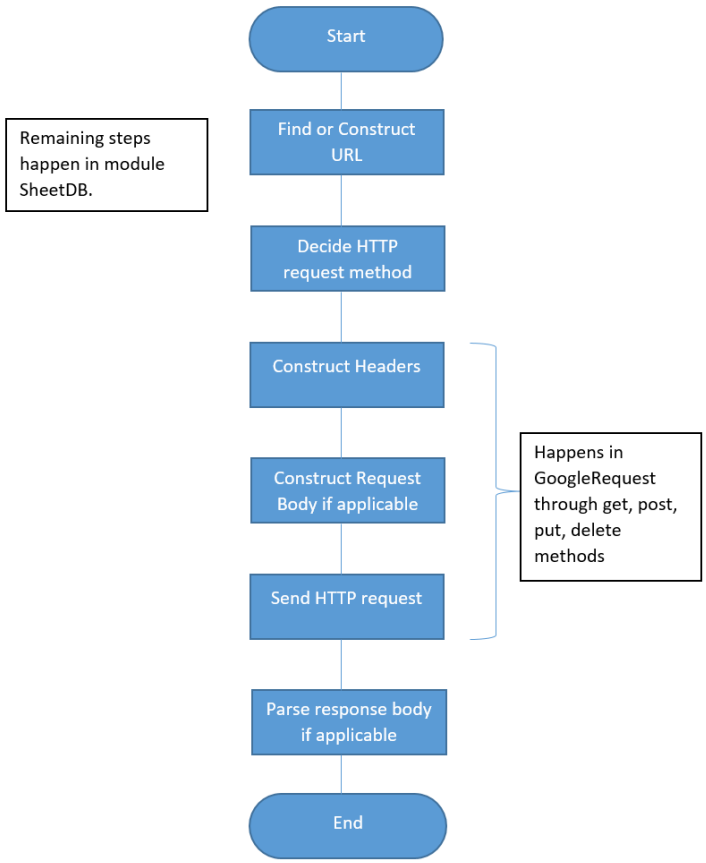

Figure 4.1. General process for find, insert, update, remove

4.3.1 Find method

`find` method in `SheetDB` is used to get the rows from a spreadsheet and perform basic structured queries. Figure 4.2 illustrates the steps in `find` operation.

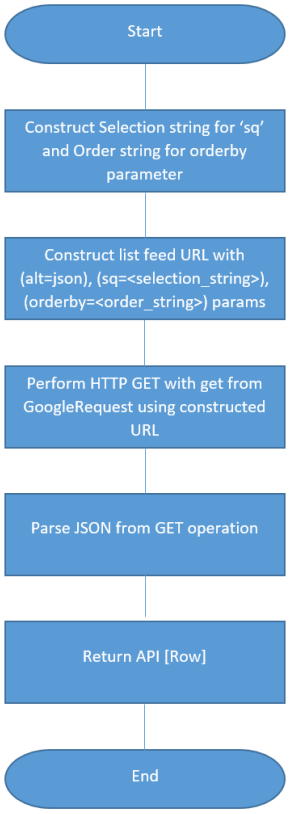

Figure 4.2. find algorithm

4.3.1.1 Construction of find URL

`find` method takes `Query` type as input and `Query` constructor takes `Selection` and `Order` type data. The `Selection` data is converted to string, encoded and passed as 'sq' parameter value in list feed URL.

```
data Selector
= Gtr   ColName ST.Value
| Gteqr ColName ST.Value
| Ltr   ColName ST.Value
| Lteqr ColName ST.Value
| Eqr   ColName ST.Value
| And   Selector Selector
| Or    Selector Selector
| Everyting
deriving (Show, Eq)

-- Converts Selector to String. example: Gtr "age" 25 -> age > 25
selectorQueryUrl :: Selector -> String
selectorQueryUrl (Gtr   colName value) = T.unpack colName ++ " > "  ++ show value
selectorQueryUrl (Gteqr colName value) = T.unpack colName ++ " >= " ++ show value
selectorQueryUrl (Ltr   colName value) = T.unpack colName ++ " < "  ++ show value
selectorQueryUrl (Lteqr colName value) = T.unpack colName ++ " <= " ++ show value
selectorQueryUrl (Eqr   colName value) = T.unpack colName ++ " = "  ++ show value
selectorQueryUrl (And   sel1    sel2)  = selectorQueryUrl sel1 ++ " and " ++ selectorQueryUrl sel2
selectorQueryUrl (Or    sel1    sel2)  = selectorQueryUrl sel1 ++ " or "  ++ selectorQueryUrl sel2
selectorQueryUrl Everyting = ""

-- Encodes Query parameter
encodeSelectorQueryUrl :: String -> String
encodeSelectorQueryUrl q = B8.unpack $ urlEncode True $ B8.pack q
```

The `Order` data in `Query` specifies the sort order based on a particular column. Google Sheets API currently supports sorting only on single column. The `Order` data also has "reverse" constructor which specifies ascending or descending order of rows. The `Order` data is converted to string and encoded as other parameters.

```
data Order = Order
{ colName :: ColName
, reverse :: Bool
} | NoOrder -- Indicates no specific order is required
deriving (Show, Eq)
```

All the parameters "sq", "orderby", "reverse", "alt" are all created in `makeQueryUrl` method

```
makeQueryUrl :: Query -> API String
makeQueryUrl q = do
        let
                spreadsheet  = sheet $ selection q
                order        = sort q
                param1       = [("alt","json")] :: [(String,String)]
                selectString = selectorQueryUrl $ selector $ selection q
                encWithSp    = encString False ok_param
                param2       = [("sq", selectString) | not (null selectString)] ++ param1
        let
                params = case order of
                                NoOrder       -> param2
                                Order col rev -> param2 ++
                                [ ("orderby", "column:" ++ T.unpack col)
                                , ("reverse", map toLower (show rev))
                                ]
        url <- formURL (key spreadsheet) (worksheetId spreadsheet) params
        return (exportURL url)
```

List feed URL template used is:

```
urlTemplate = "https://spreadsheets.google.com/feeds/list/${key}/${worksheetid}/private/full"
```

The above template is converted to find URL with relevant parameters using `formURL` method.

```
-- Takes spreadsheet key and worksheet id along with parameters to construct URL
formURL :: String -> String -> [(String,String)] -> API URL
formURL key worksheetid params = do
        let template           = T.pack urlTemplate
        let keyPattern         = T.pack "${key}"
        let worksheetidPattern = T.pack "${worksheetid}"
        let keyUrl             = T.replace keyPattern (T.pack key) template
        let keyWorksheetUrl    = T.replace worksheetidPattern (T.pack worksheetid) keyUrl
        case importURL $ T.unpack keyWorksheetUrl of
                Just url ->
                        return $ foldl add_param url params
                Nothing ->
                        die ["Not an URL:", T.unpack keyWorksheetUrl]
```

4.3.1.2 Parsing find response JSON

After an HTTP GET request is performed on above `Query` URL, a JSON is returned, which is parsed and rows are extracted.

An example of JSON returned:

```
{
        "version": "1.0",
        "encoding": "UTF-8",
        "feed": {
                "xmlns": "http://www.w3.org/2005/Atom",
                "xmlns$openSearch": "http://a9.com/-/spec/opensearchrss/1.0/",
                "xmlns$gsx": "http://schemas.google.com/spreadsheets/2006/extended",
                "id": {
                        "$t": "https://spreadsheets.google.com/feeds/list/
                        1hIEq4AAauzI8INelQRIvgxBhmzX44qAB_1QQpFJQ2Xo/od6/private/full"
                },
                "updated": {
                        "$t": "2015-11-25T19:08:07.834Z"
                },
                "category": [
                {
                        "scheme": "http://schemas.google.com/spreadsheets/2006",
                        "term": "http://schemas.google.com/spreadsheets/2006#list"
```

```
}
],
"title": {
        "type": "text",
        "$t": "Sheet1"
},
"link": [
{
        "rel": "alternate",
        "type": "application/atom+xml",
        "href": "https://docs.google.com/spreadsheets/d/
        1hIEq4AAauzI8INelQRIvgxBhmzX44qAB_1QQpFJQ2Xo/edit"
},
{
        "rel": "http://schemas.google.com/g/2005#feed",
        "type": "application/atom+xml",
        "href": "https://spreadsheets.google.com/feeds/list/
        1hIEq4AAauzI8INelQRIvgxBhmzX44qAB_1QQpFJQ2Xo/od6/private/full"
},
{
        "rel": "http://schemas.google.com/g/2005#post",
        "type": "application/atom+xml",
        "href": "https://spreadsheets.google.com/feeds/list/
        1hIEq4AAauzI8INelQRIvgxBhmzX44qAB_1QQpFJQ2Xo/od6/private/full"
},
{
        "rel": "self",
        "type": "application/atom+xml",
        "href": "https://spreadsheets.google.com/feeds/list/
        1hIEq4AAauzI8INelQRIvgxBhmzX44qAB_1QQpFJQ2Xo/od6/private/full?alt=json"
}
],
"author": [
{
        "name": {
                "$t": "adityaravikanti.vnit"
        },
        "email": {
                "$t": "adityaravikanti.vnit@gmail.com"
        }
}
],
"openSearch$totalResults": {
        "$t": "2"
},
"openSearch$startIndex": {
```

```
        "$t": "1"
},
"entry": [
{
        "id": {
                "$t": "https://spreadsheets.google.com/feeds/list/
                1hIEq4AAauzI8INelQRIvgxBhmzX44qAB_1QQpFJQ2Xo/od6/private/full/cokwr"
        },
        "updated": {
                "$t": "2015-11-25T19:08:07.834Z"
        },
        "category": [
        {
                "scheme": "http://schemas.google.com/spreadsheets/2006",
                "term": "http://schemas.google.com/spreadsheets/2006#list"
        }
        ],
        "title": {
                "type": "text",
                "$t": "Pork Shoulder"
        },
        "content": {
                "type": "text",
                "$t": "category: meat, healthiness: questionable, type: 1"
        },
        "link": [
        {
                "rel": "self",
                "type": "application/atom+xml",
                "href": "https://spreadsheets.google.com/feeds/list/
                1hIEq4AAauzI8INelQRIvgxBhmzX44qAB_1QQpFJQ2Xo/od6/private/full/cokwr"
        },
        {
                "rel": "edit",
                "type": "application/atom+xml",
                "href": "https://spreadsheets.google.com/feeds/list/
                1hIEq4AAauzI8INelQRIvgxBhmzX44qAB_1QQpFJQ2Xo/od6/private/full/cokwr/14nmk169a0mpo7"
        }
        ],
        "gsx$name": {
                "$t": "Pork"
        },
        "gsx$category": {
                "$t": "meat"
        },
        "gsx$healthiness": {
```

```
                "$t": "questionable"
            },
            "gsx$type": {
                "$t": "1"
            }
        },
        {
            "id": {
                "$t": "https://spreadsheets.google.com/feeds/list/
                1hIEq4AAauzI8INelQRIvgxBhmzX44qAB_1QQpFJQ2Xo/od6/private/full/cpzh4"
            },
            "updated": {
                "$t": "2015-11-25T19:08:07.834Z"
            },
            "category": [
            {
                "scheme": "http://schemas.google.com/spreadsheets/2006",
                "term": "http://schemas.google.com/spreadsheets/2006#list"
            }
            ],
            "title": {
                "type": "text",
                "$t": "Bubblegum"
            },
            "content": {
                "type": "text",
                "$t": "category: candy, healthiness: Super High, type: 1"
            },
            "link": [
            {
                "rel": "self",
                "type": "application/atom+xml",
                "href": "https://spreadsheets.google.com/feeds/list/
                1hIEq4AAauzI8INelQRIvgxBhmzX44qAB_1QQpFJQ2Xo/od6/private/full/cpzh4"
            },
            {
                "rel": "edit",
                "type": "application/atom+xml",
                "href": "https://spreadsheets.google.com/feeds/list/
                1hIEq4AAauzI8INelQRIvgxBhmzX44qAB_1QQpFJQ2Xo/od6/private/full/cpzh4/1cm513nm0e853k"
            }
            ],
            "gsx$name": {
                "$t": "Bubblegum"
            },
            "gsx$category": {
```

```
                        "$t": "candy"
                    },
                    "gsx$healthiness": {
                        "$t": "Super High"
                    },
                    "gsx$type": {
                        "$t": "1"
                    }
                }
                ]
        }
}
```


In the above JSON the value of "entry" key is an array of objects which correspond to rows in the spreadsheet. In each such object the column names are the keys and the cells are values. Each column key starts with "gsx$".

The Aeson library is used to convert raw JSON bytestring to Aeson Value data, which makes it convenient to traverse and fetch specific values we need. The `parseSheetJson` method in `SheetDB` is used to parse and make list of `Row`'s.

```
-- Converts the Aeson Value of the sheet to list of rows.
-- Parses the json tree and iterates through it to produce the list.
parseSheetJson :: A.Value -> API [Row]
parseSheetJson json =
        case parseMaybe parseSheet json of
                Just rows -> return rows
                Nothing   -> die ["Parse error", take 100 $ show json]
        where
                parseSheet = withObject "value" $ \obj -> do
                        feed    <- obj  .: "feed"
                        entry   <- feed .: "entry"
                        columns <- getColumnHelper entry
                        let parseRows = withObject "object" $ \obj2 -> do
                                let
                                        makePairs :: T.Text -> Parser Cell
                                        makePairs key = do
                                                valObject <- obj2 .: T.append (T.pack "gsx$") key
                                                val <- (valObject .: "$t") :: Parser A.Value
                                                let
                                                cell = case val of
                                                        AT.String x -> key =: x
                                                        AT.Bool   b -> key =: b
                                                        AT.Number n -> key =: n
                                                        _  -> key =: (Nothing :: Maybe ST.Value)
                                                return cell
                                currRow <- mapM makePairs columns
                                linkId  <-  obj2   .: "id"
                                link    <- (linkId .: "$t" ) :: Parser A.Value
                                let
                                        rowid = case link of
                                                AT.String u -> idKey =:  u
                                                _   -> idKey =: (Nothing :: Maybe ST.Value)

                                return $ rowid : currRow
                        let parseEntry = withArray "array" $
                                         \arr -> return $ map parseRows (V.toList arr)
                        rows <-parseEntry entry
                        sequence rows
```

All the above process is done part of `find` method.

```
find :: Query  -> API [Row]
find q = do
        let sel = selection q
        when (not $ validate q) $ do
                die ["Query skip or Query limit may be invalid."]
        let spreadsheet = sheet sel
        url           <- makeQueryUrl q
        token         <- io $ tokenOf (sheet sel)
        jsonValueForm <- getJson token url
        rs            <- parseSheetJson jsonValueForm
        let
        rows
                | skip q == 0 && limit q == 0 = rs
                | skip q >  0 && limit q == 0 = drop (skip  q) rs
                | skip q == 0 && limit q >  0 = take (limit q) rs
                | otherwise = take (limit q) $ drop (skip q) rs
        return rows
```

4.3.2 Insert Method

`insert` method in SheetDB is used to insert a new row into the spreadsheet. Figure 4.3 illustrates the steps in `insert` operation.

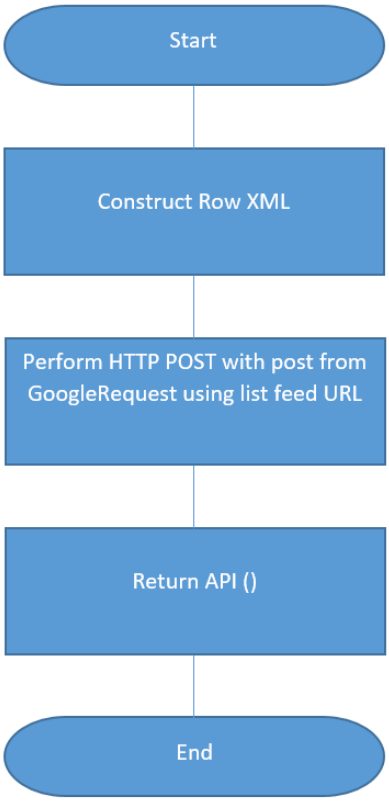

Figure 4.3. insert algorithm

4.3.2.1 Constructing XML from Row

Google sheets API server specifies the XML format of the row to be inserted and was mentioned earlier about the format.

`insert` method takes `Row` data to be inserted and constructs XML in the format accepted by Google Sheets API. Along with construction `insert` method validates data that is being inserted by validating if the column names are valid

and also checks if all the columns are present.

xml [8] library is used to construct the XML.

```
-- Checks if Row has all the Columns of the corresponding spreadsheet
isvalid :: Row -> [ColName] -> Bool
isvalid row cols = flip all row $ \(key := _) -> key `elem` cols


isEqualLength :: [a] -> [b] -> Bool
isEqualLength x y = length x == length y


xmlns = Attr
{ attrKey = unqual "xmlns"
, attrVal = "http://www.w3.org/2005/Atom"
}


xmlnsgsx = Attr
{ attrKey = unqual "xmlns:gsx"
, attrVal = "http://schemas.google.com/spreadsheets/2006/extended"
}


cellToXml :: Cell -> Element
cellToXml (key := v) = do
        let
                valstring = case v of
                        ST.String x -> T.unpack x
                        _           -> show v
        unode ("gsx:" ++ T.unpack key) valstring


-- Create xml string from list of key value tuples used for insert method
rowToXml :: Row -> String
rowToXml row
        = showElement
        $ add_attrs [xmlns, xmlnsgsx]
        $ unode "entry"
        $ map cellToXml row
```

`rowToXml` method converts a `Row` to XML String which looks like:

```
<entry xmlns="http://www.w3.org/2005/Atom"
xmlns:gsx="http://schemas.google.com/spreadsheets/2006/extended">
        <gsx:hours>1</gsx:hours>
        <gsx:ipm>1</gsx:ipm>
        <gsx:items>60</gsx:items>
        <gsx:name>Elizabeth Bennet</gsx:name>
</entry>
```

After constructing the XML, a HTTP POST request is performed using the `post` method in GoogleRequest.

All the above process is performed in `insert` method.

```
-- Adds a new row to google sheet after validation of data to be inserted
insert :: Row -> Sheet -> API ()
insert rawRow sheet = do
        let cols = columns sheet

        when (not $ isEqualLength rawRow cols) $ do
                die ["Length is different"]

        when (not $ isvalid rawRow cols) $ do
                die ["Columns mismatch, required", show cols, "got", show rawRow]

        let xml = rowToXml rawRow

        url    <- formURL (key sheet) (worksheetId sheet) []
        token  <- io $ tokenOf sheet
        status <- post token (exportURL url) xml

        when (status /= status201) $ do
                die ["Status returned: ", show status]
```

4.3.3 Update Method

`update` method in `SheetDB` is used to update an existing `Row` with new data. Figure 4.4 illustrates the steps in `update` operation.

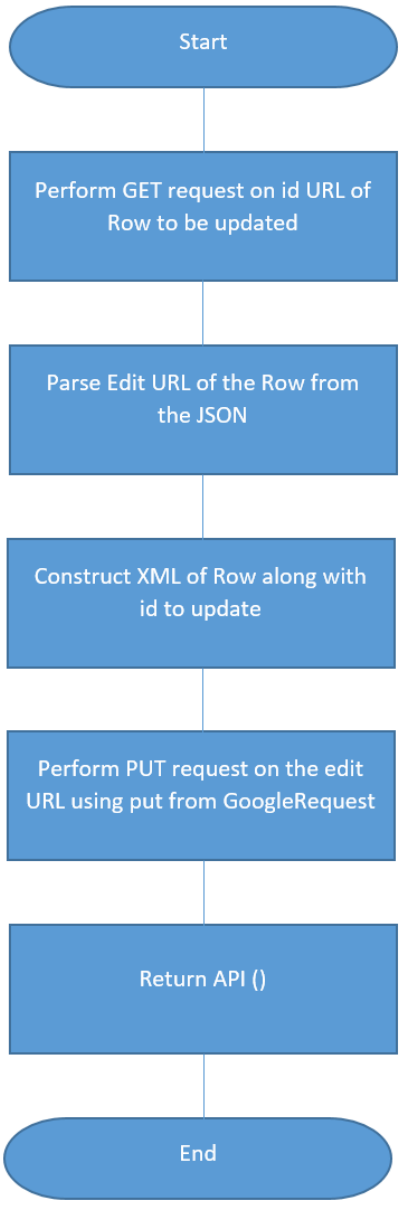

Figure 4.4. update algorithm

4.3.3.1 Parse Edit URL

Each `Row` has an edit URL which must be used to perform a PUT request with updated `Row`. The edit URL changes after each update so there is a need to fetch the row JSON and parse edit URL before an update. In the JSON example mentioned previously, the row object in the "entry" contains "id" field, which is a URL, identifies a row uniquely in spreadsheet. The row id URL can be used to perform a GET request to get the JSON corresponding to the row.

After JSON is fetched the edit URL is parsed from it using this method.

```
parseEditURL :: A.Value -> API String
parseEditURL json =
        case parseMaybe parseRow json of
                Just (AT.String url) -> return (T.unpack url)
                _                    -> die ["Cannot parse url:", take 100 $ show json]
        where
                parseRow = withObject "value" $ \obj -> do
                        entry <- obj   .: "entry"
                        link  <- entry .: "link" :: Parser A.Value
                        let
                                isEdit = withObject "object" $ \rowobj -> do
                                        rel <- rowobj .: "rel" :: Parser A.Value
                                        return $ rel == AT.String "edit"
                                parseURLSObjs = withArray "array" $ \arr ->
                                        filterM isEdit (V.toList arr)
                        editurlobjlist <- parseURLSObjs link
                        let
                                editurlobj = head editurlobjlist
                                extract    = withObject "object" $ \obj ->
                                        obj .: "href" :: Parser A.Value
                        extract editurlobj
```

`update` method takes `Row` as argument and this `Row` data must contain "id"

field that helps to know which corresponding row needs to be updated in spreadsheet. If the "id" field is not present the operation fails.

The `Row` data is converted to XML in the same way as insert method mentioned previously. After the XML is constructed a HTTP PUT request is performed using the `GoogleRequest`'s `put` method.

All the above process is done as part of `update` method.

```
update :: Sheet -> Row -> API ()
update sheet toRow = do
        val           <- look idKey toRow
        token         <- io $ tokenOf sheet
        jsonValueForm <- getJson token (T.unpack val ++ "?alt=json")
        url           <- parseEditURL jsonValueForm

        let cols = idKey : columns sheet

        when (not $ isEqualLength toRow cols) $ do
                die ["Length is different"]

        when (not $ isvalid toRow cols) $ do
                die ["Columns mismatch, required", show cols, "got", show toRow]

        let xml = rowToXml toRow

        token  <- io $ tokenOf sheet
        status <- put token url xml

        when (status /= status200) $ do
                die ["Status returned: ", show status]
```

4.3.4 Remove Method

`remove` method in `SheetDB` is used to remove an existing row from spreadsheet. Figure 4.5 illustrates the steps in `remove` operation.

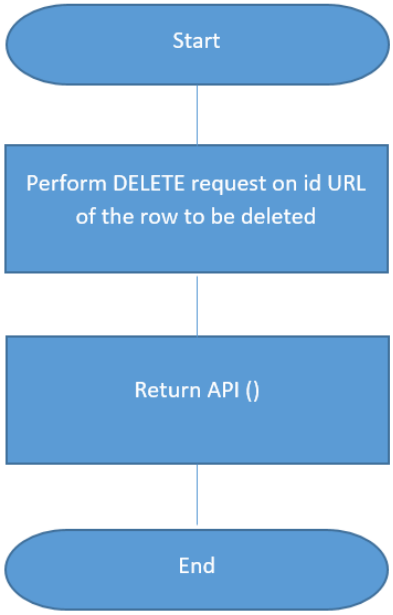

Figure 4.5. remove algorithm

`remove` method utilizes the `delete` method in GoogleRequest and returns `API ()`. And finally, the actual `remove` method.

```
-- Used to send a delete request to the Google Sheet.
remove :: Sheet -> Row -> API ()
remove sheet r = do
        url    <- look idKey r
        token  <- io $ tokenOf sheet
        status <- delete token (T.unpack url)

        when (status /= status200) $ do
                die ["Status returned:", show status]
```

Chapter 5

Conclusion and Future Work

Currently the spreadsheets should be manually created by the user as one creates tables in traditional database but this can be automated by using Google Drive API. The queries offered by Google Sheets API is very limited and has a lot of scope for improvement, as and when google adds new features to its API they can be added to the library.

Sheets-db has features that will be useful to applications that do not need full power of traditional database but needs a simple row based storage and retrieval mechanism from a Haskell application. For example, if an application's build generates benchmarks data of the application each time it is built, sheets-db can be used to push the benchmarks data into a spreadsheet and the application's author can check how the application is performing over time by plotting graphs using sheets-db or using the graph capabilities offered by spreadsheets UI. This library can be a great utility for such purposes and must be actively used instead of traditional database.

References

[1] Google sheets api. https://developers.google.com/google-apps/spreadsheets.

[2] Google spreadsheets. https://docs.google.com/spreadsheets.

[3] P. Brisbin. https://hackage.haskell.org/package/google-oauth2.

[4] I. S. Diatchki. https://hackage.haskell.org/package/url.

[5] A. Gill. https://hackage.haskell.org/package/mtl-2.2.1.

[6] T. Hannan. https://hackage.haskell.org/package/bson-0.3.2.1.

[7] T. Hannan. https://hackage.haskell.org/package/mongodb.

[8] G. Inc. https://hackage.haskell.org/package/xml.

[9] B. O'Sullivan. https://hackage.haskell.org/package/aeson.

[10] M. Snoyman. https://hackage.haskell.org/package/http-conduits.

[11] J. Soma. https://github.com/jsoma/tabletop.