Designing a robust system for Data Representation and Discovery in DataHub

Jesika Haria, MIT EECS 2014

I) ABSTRACT

DataHub is a hosted platform for organizing, managing, sharing, collaborating and making sense of data. This report focuses on the design and evaluation of data representation and data discovery for the DataHub application. This includes implementation of the Python ORM for the user's high-level control of the app, as well as defining ranked search over datasets, both by keyword and a given dataset.

II) INTRODUCTION

DataHub's intention is to become a Github-like repository for storing relational databases on the cloud. Our vision is for the users' data to not only be viewed and interacted with in a web browser or DataHub console, but also to be used to power applications, possibly built by other users, via SQL queries.

This necessitates the development of a robust data representation system that is both reliable and extensible, leading us to implement an Object Relational Mapping (ORM) for Python, which is a popular language for many modern-day web frameworks. The ORM simplifies publishing for the data creator as well as sharing and collaboration for the data consumer, thus making DataHub easily accessible to people with little coding experience.

Under this system, a user could define an Object Model that serves as a template for their data, analogously viewed as a table schema from the perspective of the database. They can now perform the standard functions of persistent storage – Create, Read, Update and Delete (CRUD) over their data. Through this process, the ORM layer ensures that the native Python class objects created or returned, while the parser converts the intended actions of the user into SQL commands that are executed over the real database. Since the ultimate goal is a combination of functionality and customizability, the user should retain the option of writing the raw SQL or even overwriting the CRUD methods themselves for added control.

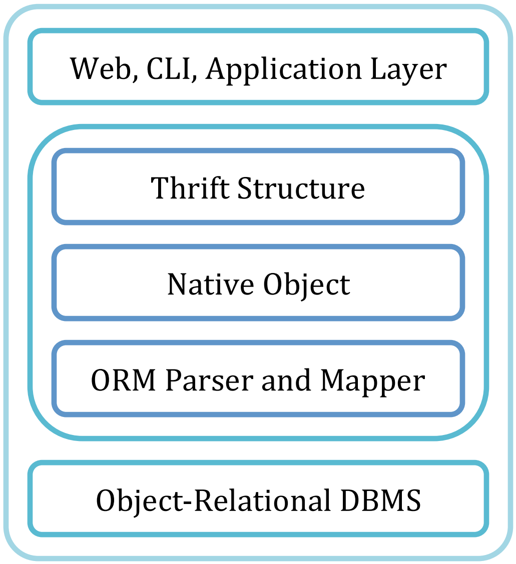

Figure 1: DataHub data representation

The native class objects are then manipulated by the user, eventually communicating to the Web Command Line Interface (CLI), website or application via a Thirft interface.

Considerable value is derived from users being able to interact with not only their own data, but also others' data. We can facilitate the data discovery process via search and recommendations. To be able to effectively capture the range of discovery cases, we must first understand both the interaction mode and the intent of the user.

The interaction mode of the user is whether the user is actively searching for tables using a particular keyword, or is just looking for recommendations of tables that might be useful to the table they are currently viewing. In the first case, searching over keywords requires matching canonicalized representations of the search term over a full-text index of the corpus. In the case of finding related datasets, the task is slightly more complicated in both understanding the meaning of the table as well as relevant rows.

The intent of the user defines the reason they are interested in other datasets. The data search being searched for is then a candidate for either a union or a join. A candidate dataset for unions represents can be identified by complementary rows using the same approximate schema. Making the process of understanding complementary rows without actually performing an expensive union over the datasets is a challenge, and there are numerous signals we will use to guide the characteristics of datasets of interest. The other type of combination of different data sources is a join, wherein the dataset being added provides a different direction over the approximate same set of rows. In this case, one must understand what columns are often seen together, and where possible, a semantic understanding of the column name and contents.

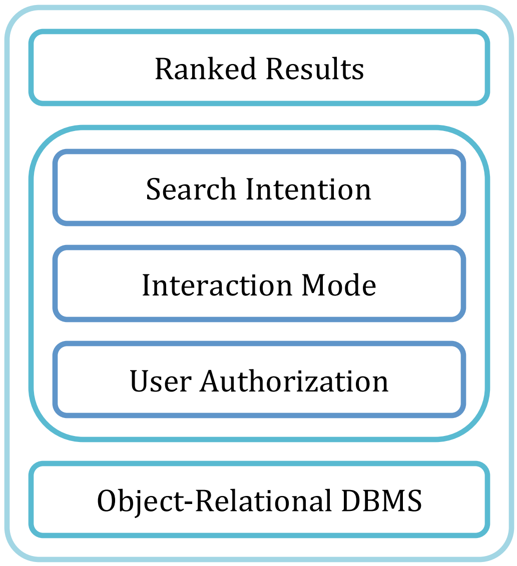

Figure 2: DataHub search

The search then feeds into the notion of ranking, wherein the retrieved search results have to be displayed to the user in the order of relevance. To this effect, we might be able to use data quality metrics to determine which of the matching queries are most likely to be relevant. Of course, the order of ranking for each user is likely to be different (as opposed to just the search results, which over public datasets, will be the same for all users). Rankings can be customized on a per-user basis by implementing user-based collaborative filtering. This is achieved by determining similarity of users on a predefined scale and then giving similar users similar results.

III) PREVIOUS WORK

In this section, we detail the past work done on hosted data publishing and collaboration efforts, as well as the two technical areas of ORMs and search/ ranking.

Google Fusion Tables (Fusion) is an experimental tool for publishing and viewing data on the web, collaborating on public datasets and visualizing this information [1]. In theory, the motivation behind DataHub and Fusion is extremely similar. As it currently stands, Fusion allows for rudimentary data ingestion and storage, primarily in the form of online tables and spreadsheets. However, since it is not backed by the support and robustness of a relational database, there is no schema enforcement. The publishing option that is currently offered is in the form of an iframe on a webpage, so one can display the data but there is no way to programmatically access and build on top of it. Additionally, it lacks the fine-grained querying and filtering support that SQL provides. Hence, User Defined Functions (UDFs) also cannot be implemented and executed, leading to poor flexibility for the data publisher and consumer. By firmly basing our data in a relational database system, DataHub attempts to bridge these very gaps.

For data representation, DataHub chooses to implement a simplified ORM. ORMs have been used in popular web development frameworks such as Django or Flask for Python [2] and ActiveRecord for Ruby on Rails [3]. However, because of their niche syntax, one cannot easily port an application from one framework to another, even within a single language. Additionally, since these are tied to their own backend systems, there is not a lot of flexibility regarding where the data comes from, or defining specific actions to be taken by overriding the in-built CRUD methods. DataHub's ORM is simply language specific in order to support methods (so that any given web application framework using that language can be supported), but the data can and is passed around via a Thrift binary object, that is easily converted to the native class object.

Most major DBMS have full text support via Apache Lucene [4], based in Java. Fusion allows for search based on specific columns of the dataset, for similar columns in other datasets [5]. These are primarily join candidates. It seems that their approach is mainly keyword and entity matching based for a given (user-selected) column. There remains a lot more work to be done in automatic inference of related columns from other papers, especially in terms of semantic understanding of the column names and contents. Furthermore, Fusion does not support unions, while DataHub will endeavor to do so, as an integral part of data collaboration.

IV) SYSTEM ARCHITECTURE AND DESIGN DECISIONS

In this section, we describe the System Architecture for DataHub's ORM and data discovery module.

IV.I) ORM

The ORM is represented in the form of a BaseModel class that serves as the super class for the user-defined models, Field classes that define the mapping to a SQL field and a parser module that handles and transforms the appropriate requests. To understand the architecture better, let's take an example of MIT CSAIL wanting to make a list of all Faculty Awards public, while the NYT wants to aggregate all relevant sources of Faculty Awards from MIT and write an article.

i) BaseModel

The BaseModel class will be extended by any user-defined model, and contains the shell methods for CRUD. In this context, the model methods for a sample Faculty instance, along with the SQL translation are as follows:

```
f = Facul-
ty.create(faculty_params) →
INSERT INTO faculty
({param_names}) VALUES
({param_values})

f = Faculty.find(params_id) →
SELECT * FROM faculty WHERE
id={params_id}

f = Facul-
ty.find_by(faculty_params) →
SELECT * FROM faculty WHERE
{param_name=param_value for
param in faculty_params}

f.update(faculty_params) →
UPDATE faculty SET
({param_names})=({param_values}
) WHERE id={f.id}

f.delete() →
DELETE FROM faculty where
id={f.id}
```

Here, `params_id` refers to the unique id of that Faculty object, or its primary key in the database table. `faculty_params` are the list of the parameters specified according to the instance attributes of the Faculty object, and are themselves instance of Field sub-classes.

Design Decisions: CRUD was chosen as the basic set of operations that could be combined in different ways to generate more complex queries. The syntax for these CRUD operations was chosen to be as close as possible to the very intuitive ActiveRecord syntax for Rails. Since the methods originate from the superclass BaseModel, they can be conveniently overwritten as per the user's choice.

ii) Field classes

The BaseField class currently has a size, `primary_key`, not_null and unique attribute that represents default presets for the different Fields. These are further subdivided into:

```
CharField (default size 30)
IntegerField
BooleanField
DateTimeField
TimestampField
```

These are the standard fields that Postgres supports as column types. To make the process more intuitive, we have also added the notion of `ForeignKey`, which essentially provides a reference by string match to another Python class defined in the same models file.

For example, this is one example of how the Faculty and Awards objects from the previous example could be represented:

```
class Faculty(models.BaseModel):
  id = mod-
els.IntegerField(primary_key=True)
  name = models.CharField(size=30)
  email = mod-
els.CharField(size=30,
unique=True)
  age = models.IntegerField()
  dob = mod-
els.DatetimeField(not_null=True)
  isMale = models.BooleanField()
  award_id = mod-
els.ForeignKey('Award')

class Award(models.BaseModel):
  id = mod-
els.IntegerField(primary_key=True)
  name = models.CharField(size=35)
```

Design Decisions: In constructing the entire class structure for the fields, it was important to support at least all the basic fields that were valid for a Postgres database. A limited number of higher order functions such as `primary_key` (essential for indexing), `not_null` and `unique` (basic validation) were also implemented to support the more powerful aspects of relational databases. Some presets regarding the size of the fields were set in case the user forgets to add them manually.

iii) Parser module

The parser module is used for translating the Python code into SQL statements as described in i), and executing them via a DHConnection object over the database. When objects are returned from the database in the form of a DHObject (which is a Thrift object), this module also converts the returned object into an instance of the returned Model class, while preserving the attributes as a direct mapping. Since Python objects can have attributes dynamically added to them, and the mapping to be supported is initially restricted to exact strings only, it is possible to achieve the translated object by setting the user's object attribute as the same-name attribute from the incoming Thrift object.

Design Decisions: For the parser, a key decision that was made initially was to retain the same names for columns as well as instance attributes. Eventually, we could separate the two and construct a mapping, but its maintenance requires too much effort for the relative benefit at this stage.

IV.II) SEARCH

This section details the architecture of the Search system, based on a full-text index such as Apache Lucene. Search could be given as

```
Keyword | Dataset → Ranked Dataset
```

i) Keyword search

Keyword search can be used to search over either row values or column names. The former applies in the event that one is looking for something in the current dataset. In this case, full-text support from Lucene should be able to handle the queries. This can be extended to search for column names for other datasets. In this case, a simple string search will not yield a lot of matches. Instead we will need to first rewrite the term being searched for, match it against our index, pass it to a scorer in order to rank the matched results and finally perhaps into a blender to normalize scores and generate results over the possible outcomes.

a) Rewriter

The rewriter takes the string from the search box and first strips the punctuations, converts to lower case, tokenizes the search term if needed. At this point, the search term is still almost in its original form, with some minor cleaning. Now, the rewriter can expand the query into its synonyms, aliases and stems: other canonical representations of the search term, while attaching a probability score with each interpretation of the keyword. The differnet expressions can be combined with AND, OR, XOR and NOT operations to generate arbitrarily complex representations.

For example, the rewritten expression for the keyword 'mit' could be expanded to:

```
((1.0: (mit)) OR (0.9: (massachu-
setts institute of technology)) OR
(0.4: (michigan institute of tech-
nology))
```

Design Decisions: Rewriting the query into a probabilistic logical expression seems to be a sophisticated approach to keyword matching, rather than the raw string itself. This is extremely essential because it is unlikely that people will refer to the same object in the same manner. For example, states names of colleges could be either written as CA or California, and neither the searcher nor the table provider should be penalized for that. Of course, as we deviate from the original term, we are less and less sure of our predictions and inferences, which prompts the inclusion of a weighted match.

b) Indexer

The indexer now accepts this rewritten expression and matches all expressions against the full-text index provided by Lucene in order of weights. However, it still needs to decode that AND means that the results should only be returned if both expressions match, while OR can be treated as multiple simultaneous requests, and hence are independent. Since NOT is also supported in Lucene, XOR can be modeled as the union minus the intersection of the results. The indexer passes to the scorer the meta-data regarding the match. This includes a Thrift structure with the following information:

```
struct DHSearchResult{
  1: i32 table_id,
  2: string table_name,
  3: i32 date_created,
  4: i32 creator_id,
  5: i32 table_views_downloads,
  6: i32 table_rating,
  7: string matched_col_name,
  8: string matched_col_type,
  9: i32 DHQuality
}
```

Design Decisions: Running multiple queries in response to a single request is not odd, and is even preferred for search result variety and completeness. Some meta-data about the match is gathered in order to help the scorer rank the output of the indexer.

c) Scorer

This module is responsible for the ranking of the datasets being returned. Although it is expected that the ranking algorithm is likely to change and be adaptive over time, some of the features considered for the first iteration are:

`date_created` – Newer tables tend to be more relevant in a whole where data is always changing, so one can add a number inversely proportional to how long ago the table was created.

`creator_id` – With the concept of user rankings, tables provided by strong and reliable data providers should be preferred. They would be ranked similarly to the table rating score.

`table_views_downloads` – Since one cannot trust just an average score (a new table might have just one 5-star rating whereas an old rating might have an average of 4.6 from 100 users), the number of table views and downloads are important metrics, measured as a percentile of the most viewed/ downloaded table.

`table_rating` – the higher the rating, the higher the score to assign, on a scale of 0 to 1 based on the average of the user ratings, where 1 represents perfect rating over all users.

`matched_col_name` – this score, ranging from 0 to 1, represents how much the names themselves match in a conceptual context. A score of 1 indicates the same string, whereas a score of 0 indicates completely orthogonal concepts, like ‘cat’ and ‘victor’. The intention is to use the ConceptNet 5 API in order to return these scores.

`matched_col_type` – given a common category and enough training data, one can infer what a column’s type ought to be. For example, ‘number of awards’ is likely to be an IntegerField, while ‘state’ is likely to be CharacterField. Based on the sureity of the prediction, this score can take values from -1 to 1. Here, 1 is where the algorithm is relatively certain that the column name is the right type, 0 represents lack of information, while -1 represents that the algorithm is fairly certain the returned column name is of the wrong type.

`DHQuality` – DataHub will be providing a DHQuality metric that would take homogenity and sparsity of the table into account and generate a DHQuality score to be added to the scoring decision.

Design Decisions: Scoring is essential to good search quality, and these metrics will change as the system gets more used over time. Eventually, it would make sense to customize the results by including some user-based collaborative filtering.

d) Blender

The blender does the function of aggregating the results and scores, normalizing where necessary before returning the top-few results.

Design Decisions: The blender module must be included to preserve flexibility, and in the future support searching over different verticals (column names, concepts, etc)

ii) Dataset recommendation

The second kind of search is over a dataset, and involves judging candidates for unions (adding more rows to the same columns) and joins (adding more columns to roughly the same rows).

a) Union

Given a dataset, we can rank other datasets on the basis of a few of these criteria:

Column name and type percentage match (fuzzy Jaccard distance) – This is represented as a score from 0 to 1, where 1 is a complete ideal match between a table and itself. Jaccard distance measures the similarity of two sets by diving the intersection by the union. Here, we will consider matches that aren't exact string matches, but also matches that represent the same general concept.

Row overlap – While some amount of overlap in rows is acceptable, for a successful join, the table being joined must offer sufficient new information in the form of added rows. This is also a (0,1) score, but 1 now represents no overlap in rows (desirable).

DHQuality – Data quality will be taken into account in the scoring system.

Design Decisions: The feature list is continuously growing, and these are but estimates for good matches. With a union, the default behavior upon deciding to match will be to reserve the current table's rows in the case of overlap (later on, this can be extended and customized).

b) Join

The best candidates for joins are those tables that complement the insight from one table. There is an inherent tradeoff here – one needs enough similarity in the concept in order to be relevant (no point comparing faculty award list to the number of Disney movies watched in that state), while at the same time the columns cannot be too similar for fear that no new information is really provided (year of obtaining professorship vs number of years as a professor, given that we know the current year). In order to achieve this, we use a few more metrics in addition to those already described for the union:

Co-occurrence – We can model the co-occurrence of certain concepts or column names to train a Markov Chain on frequently co-occuring columns. What this means is that if a users table is pretty popular, and hence the columns, name, age and gender appear together frequently. Hence, given name and age, one can predict with a high probability that the next column will be gender. This effect is amplified with an inverse document frequency. The more unusual the combination of columns, the stronger the prediction can be. Markov Chains are intuitive and generally generate good practical results. This is an example of an item-based collaborative filter.

Alternatively, one could also implement as user-based collaborative filter to recommend tables that a particular user must view based on the previous actions of similar users.

V) EVALUATION

V.I) ORM

There are not many quantitative metrics as such for the ORM, but it must be thoroughly tested to ensure that ACID guarantees hold, and that these transactions are fast enough. The benchmarks can be timed against popular ORMs for Django and Rails.

V.II) SEARCH

The data will be evaluation with respect to Fusion tables. We start randomly from a node and view related tables that Fusion suggests, importing them as links in the graph. We then add some unrelated links to add noise to the system, changing the ratio of noise for every trial. This will allow us to compare our ranking algorithm against that of Fusion's.

For cases where there can be no direct comparison made toe Fusion data (for example, unions), we can run user experiments to determine satisfaction.

With more users on the DataHub network, conventional search metrics such as CTR, Table activity (download), Precision (how many times the first result shown is picked) etc can be logged and monitored.

VI) CONCLUSION

Through this paper, we describe the framework that DataHub rests on, and two crucial aspects of data representation and data discovery. We discussed the use of ORMs for easily storing, managing, interacting with and editing data; and delved into the communication and translation mechanisms, along with a glimpse of proposed usage. We also saw the different modes of data discovery for users, and proposed mechanisms and ranking metrics for ordering retrieved data. We finally discussed strategies for the evaluation of our proposed approach, with the view that greater usage of DataHub would enhance our signaling mechanisms, in turn leading to higher quality of service.

VII) ACKNOWLEDGEMNETS

The author would like to thank Anant Bhardwaj and Professor Samuel Madden, CSAIL for their continued support and supervision of this project. Many thanks to Adam Marcus and Eugene Wu for their insightful comments and thought-provoking questions.

VIII) REFERENCES

[1] *About Fusion Tables*, https://support.google.com/fusiontables/answer/2571232?hl=en
[2] *Models and databases*, django, https://docs.djangoproject.com/en/dev/topics/db/
[3] *Active Record Query Interface,* http://guides.rubyonrails.org/active_record_querying.html
[4] *Apache Lucene Core,* http://lucene.apache.org/core/
[5] *Finding Related Tables,* A. Sarma et al, https://amplab.cs.berkeley.edu/wp-content/uploads/2012/06/finding-related-tables.pdf