BFD: The Big Friendly Database
6.830 Final Project Report

Jonathan Kelly
Department of Computer Science
Massachusetts Institute of Technology
jgkelly@mit.edu

Michelle Lauer
Department of Computer Science
Massachusetts Institute of Technology
mflauer@mit.edu

Abstract

Currently, data is everywhere, and with it comes the potential to gain many previously unknown insights. However, this data remains inaccessible to many people who do not have the analytic expertise to learn from this data. It is our goal to change this, and allow users from any level of technical experience to understand their data. With this in mind, we present the Big Friendly Database (BFD), an end to end system designed to allow any user to transform an uninterpretable file into high quality results. From an online interface, users input csv files with the data they are interested in. They are then able to prune the data to the specific components that are relevant to them, and a custom database is created. Finally, BFD enables users to ask specific questions about the data in their custom database. Previous research endeavours use complex natural language processing (NLP) to allow users to ask unstructured questions over databases. However, user tests have found that these questions are sometimes misinterpreted, or require users to rephrase their questions into a form that the system can convert to SQL. Additionally, querying a database is different from asking a question in a search engine, because database queries typically require perfect results. Rather than attempting to process freeform user input, BFD provides users with a set of structured natural language options, which can be unambiguously translated to corresponding SQL queries. In doing this, we ensure that BFD always provides accurate results.

1 Introduction

In the last decade, the ubiquity and availability of data has increased dramatically. As hardware and software advance, and organizations increasingly recognize the value of storing the data they acquire, the volume of information being aggregated and recorded is constantly growing. With a simple search on Kaggle (a public website that hosts over 13,000 free datasets), anyone can acquire information ranging from California wine reviews to Black Friday consumer trends. The reason that companies are so focused on collecting large quantities of data is simple; they hope to discover actionable and, more importantly, marketable insights. However, conducting this type of analysis in practice often takes a team of specialized data scientists to tease these valuable insights out of the data. Unfortunately, for all the people who do not have a background in data analysis, this growing mass of data remains largely useless.

One important observation is that many people who are generally technically competent may still lack the technical expertise to work with large amounts of raw data. Being able to effectively use technology and quickly learn new pieces of software is a skill that has been subconsciously learned by many through the quick evolution of technology from desktops to laptops to smartphones. Practically anyone in the modern work force—from computer scientists, to financial analysts, to marketing professionals—would have little trouble searching for and downloading a dataset from a website like Kaggle. It is only the final step, the actual analysis of the acquired data, that prevents this group from converting data files into usable information.

Another important observation is that meaningful insights oftentimes do not require incredibly complex questions. For example, say that a doctor has seen an influx in flu patients recently. Some interesting things that she might want to know are the location of all the patients who have gotten sick in the last month, the number of patients who had received a flu shot before getting sick, or the efficacy of a certain drug when given to patients in a specific age range on this new strain of the flu. None of these questions are very complicated, and would be trivial for a trained data scientist to answer, but would likely be very difficult for an average doctor, who might instead manually go through and look at each file from her patients. Expand this to a larger scale for someone like a hospital administrator or a business analyst who is looking at data from thousands rather than tens or hundreds: manually going through the data goes from being inconvenient to infeasible.

With all of this in mind, we present BFD, a database system that is designed to empower people without any data analysis experience to get a range of insights from any dataset through the use of structured natural language queries. BFD is designed to not only be simple to use, but also simple to learn, making the barrier to go from dataset to insight extremely low. To use BFD, the user first uploads the dataset to the system, and then selects which information in the dataset is important to them. BFD then creates a new custom DB based on the information provided by the user. The user is then able to make a range of natural language structured queries on the dataset, with available options constantly being displayed and updated. Furthermore, BFD allows users to upload multiple files and combine them based on similar fields, essentially replicating SQL inner joins between the datasets. The user is then able to make structured queries over the new joined dataset.

2 Related Work

Giving non-technical users a way to interact with data is a recognized problem, and previous work has attempted to solve this with a variety of strategies. Many recent systems attempt to make searching easy for users by incorporating NLP, so that users are able to ask questions about data without needing to learn SQL or perform extensive data processing [2, 9, 8]. A major obstacle in these systems is that turning unconstrained user input into a SQL query is an extremely challenging NLP problem. If the system misinterprets the query, it might return results that are not only incorrect, but also difficult to distinguish from correct results. This means that unless the NLP to SQL process is perfect, it could misinform its users, which is unacceptable in many situations. Conversely, a problem for structured NLP systems is applying too many restrictions on the user. If trying to figure out how to express a question is as challenging as learning to write a SQL query, then the system loses its advantages.

The Natural Language Interface to Relational databases (NaLIR) is an example of an NLP-based model that claims to be usable in practice even for non-technical users to run complex queries [7]. Prior work mostly consisted of keyword-based search mechanisms with rigid structure and without the ability to scale to complex queries. NaLIR uses a parse tree to interpret the queries input by the user. For ambiguous queries, the system gives users a list of potential queries to select from in order to clarify their intended question. A majority of the novel system design in NaLIR surrounds the logic of constructing an appropriate tree structure for the user input. This tree representation can then be directly used for SQL query generation by converting the parse tree into an equivalent query tree. However NaLIR suffers from the problem aforementioned: misinterpreting the natural language queries. For some types of queries, NaLIR might only return a correct response around two thirds of the time.

Of the related work that we are aware of, the system most similar to BFD is DBPal [3]. In DBPal, users are not required to have prior knowledge about the database schema, but can learn about the contents of the database through DBPal's learned autocompletion model. A primary contribution of DBPal is their aim to make the schema more transparent and understandable to their users. To interpret the unconstrained user inputs, DBPal uses an underlying recurrent neural network, aiming to build a translation system robust enough to handle the various ways that humans naturally ask questions that would all translate to the same SQL query. For example, the same query should run whether a user says "show all patients with diagnosis of flu" or simply "get flu patients". The primary evaluation metric for the effectiveness of DBPal for users was to compare it to the previously discussed NaLIR system. Their grading scheme for evaluating the usability was to measure the amount of tries it took for a user to ask a question in order for the system to understand and execute the correct query, and they demonstrated successful improvements over the baseline NaLIR system. However, despite providing a learned auto completion feature for users, by allowing users to ask a question in many ways, they also introduced potential inaccuracies. This is demonstrated in their results: on their first test dataset, they were only able to achieve 76% accuracy, and on their second test dataset, they achieved less than 50% accuracy. Thus, while the DBPal method of NLP to SQL processing is significantly improved compared to previous work, its use case is still limited, as it provides no guarantees on the accuracy of the returned data.

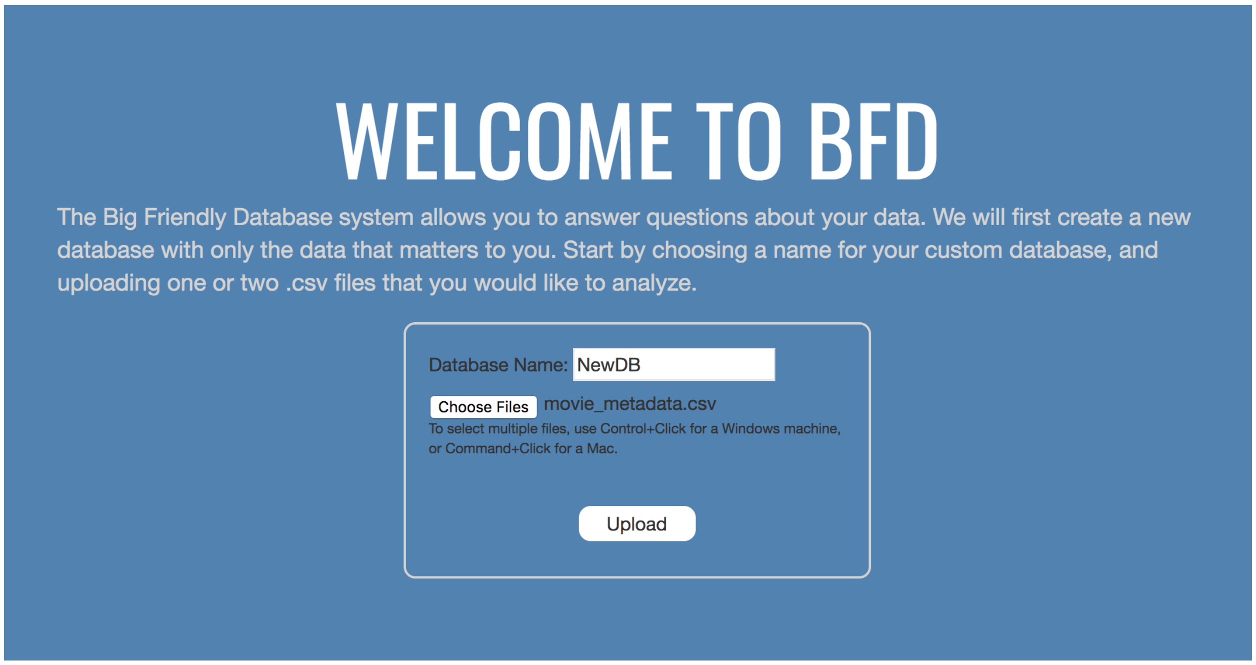
Some other work has been done aiming to accomplish the same high level goal of helping less technical users learn insights about their data. NaLIR and DBPal are the systems we thought were most similar to the one that we developed with regards to how users interact with the system, but multiple other systems have been developed that assume a range of technical experience and take different approaches to working with the user. Some earlier work includes keyword-search methods to extract information from database schemas [11, 4]. Other research into NLP-based approaches include [10] and [9]. Ava is one recent example of a highly interactive interface built to be used by data scientists, that asks users questions about their data and displays code and data analysis alongside the chatbot-like dialogue [6].

However, due to the challenges of correctly interpreting these users results, there is still ongoing research into more structured approaches [12, 5].

3 Introducing BFD

When designing BFD, we prioritized the usability and accuracy of our system. Previous systems rarely focused on how the actual database was built and often assumed the user already had a RDBMS set up. Since BFD is for non technical users, we wanted to build a complete end to end system. Thus BFD is designed for a user who starts out with just a csv file of data, as we have found that csv files are one of the most common ways that smaller datasets are stored. Additionally, to help the user understand what is going into their custom database, we allow the user to select exactly what parts of the csv they want. Finally, to address the accuracy issues and limited scope that previous NLP to SQL systems have had, we use structured natural language. While this takes away a degree of freedom from the user, it allows us to guarantee 100% accuracy and therefore be usable by a much broader audience than previous systems.

To use our BFD system, users start by uploading one or two csv files from the landing page shown in figure 1. We will be creating an entirely new database for the user, so we also give them the option to name this new database (which will be displayed to them in various locations throughout the interface later on).


WELCOME TO BFD

The Big Friendly Database system allows you to answer questions about your data. We will first create a new database with only the data that matters to you. Start by choosing a name for your custom database, and uploading one or two .csv files that you would like to analyze.

Database Name: NewDB

movie_metadata.csv

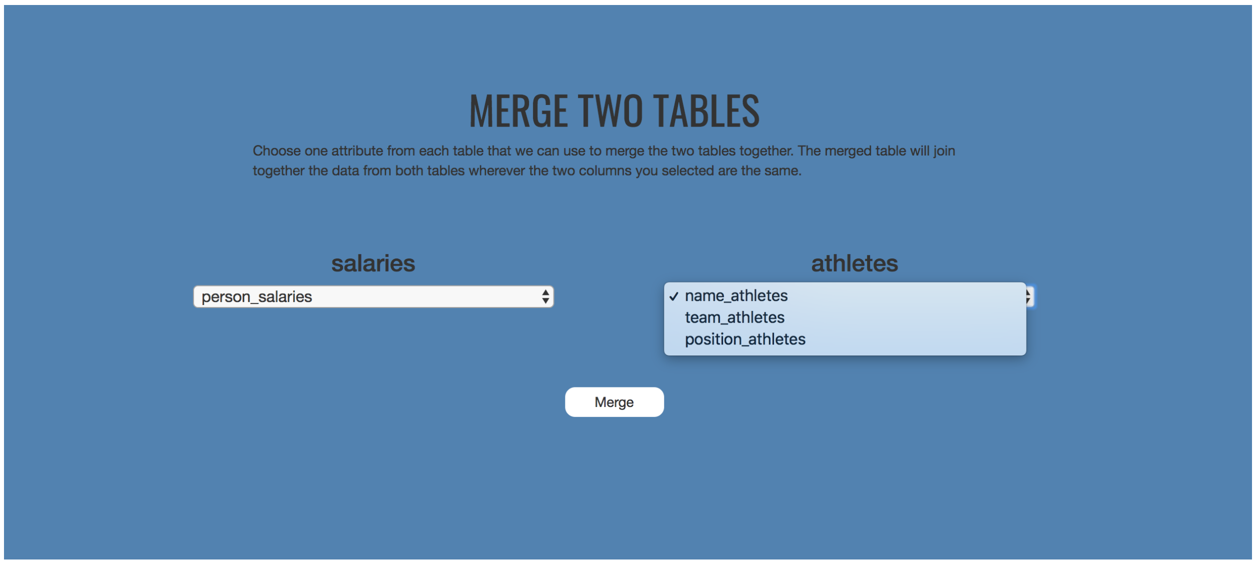
To select multiple files, use Control+Click for a Windows machine, or Command+Click for a Mac.

Figure 1: Landing page for the BFD interface

If they have uploaded two csv files, they are then taken to the merging page shown in figure 2, where they will select one column from each table in order to merge the two tables together.

Next, they are taken to the page shown in figure 3, where they select the columns from each table that they want in their custom database. They can additionally specify particular entries or ranges for each column if they will only ever be searching over a certain subset of the data. For example, if we have a user who is researching only action movies, but they have a csv file containing information about movies of all genres, they can use this page to choose only the action movies. In the next part of their workflow, when they are actually performing queries over their new database, only the action movies will appear as results.

The last phase of the user workflow is the search page, which will perform queries over their custom database. Figure 4 shows how users interact with a search bar that populates as they type with the possible questions that they can ask. These options will build incrementally depending on the contents of the database. Results will then be displayed below their search, show in figure 5.

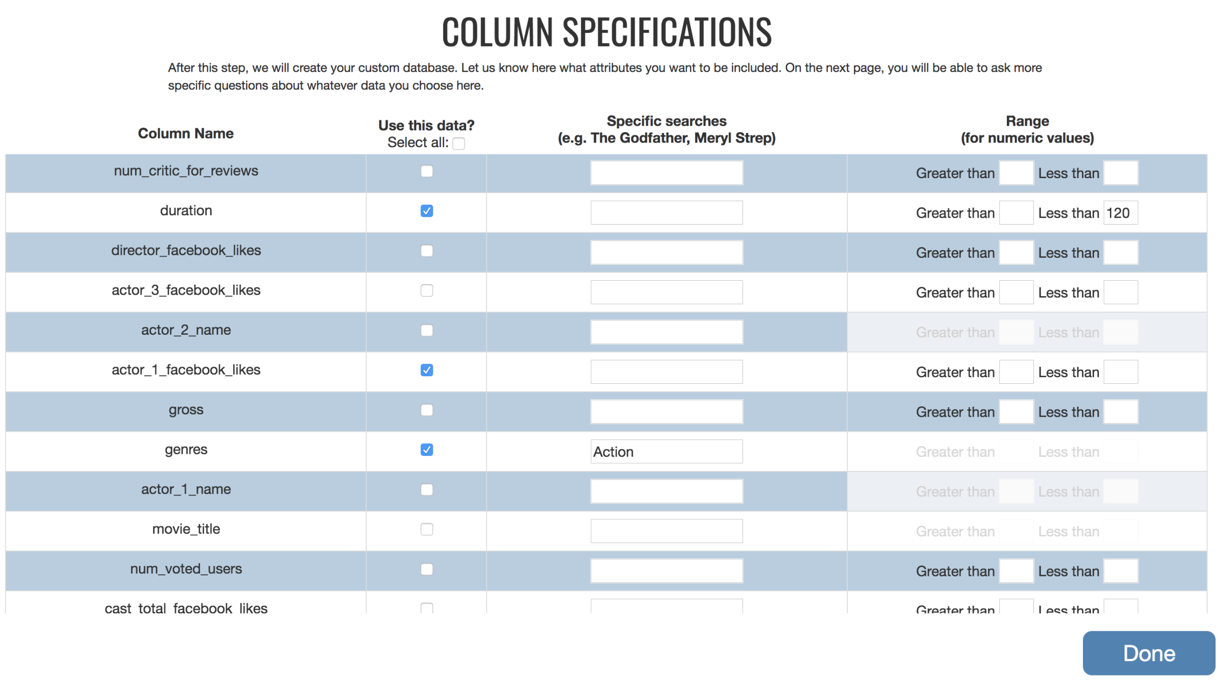


MERGE TWO TABLES

Choose one attribute from each table that we can use to merge the two tables together. The merged table will join together the data from both tables wherever the two columns you selected are the same.

salaries	athletes
person_salaries	✓ name_athletes team_athletes position_athletes

Figure 2: If two tables were uploaded, users are taken to this merging page. The dropdown titles match the names of the csv files they uploaded, and the dropdown choices are populated using the column labels from those files.

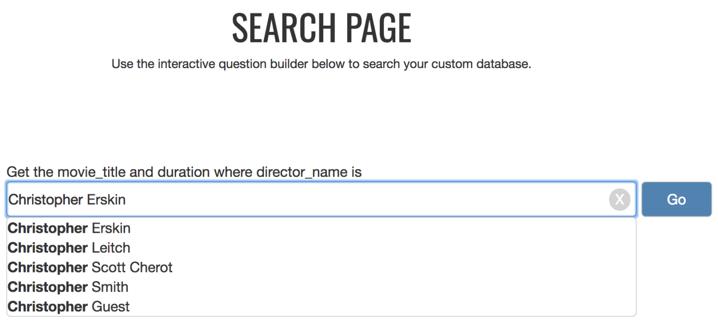


COLUMN SPECIFICATIONS

After this step, we will create your custom database. Let us know here what attributes you want to be included. On the next page, you will be able to ask more specific questions about whatever data you choose here.

Column Name	Use this data? Select all:	Specific searches (e.g. The Godfather, Meryl Strep)	Range (for numeric values)
num_critic_for_reviews			Greater than Less than
duration	✓		Greater than Less than 120
director_facebook_likes			Greater than Less than
actor_3_facebook_likes			Greater than Less than
actor_2_name			Greater than Less than
actor_1_facebook_likes	✓		Greater than Less than
gross			Greater than Less than
genres	✓	Action	Greater than Less than
actor_1_name			Greater than Less than
movie_title			Greater than Less than
num_voted_users			Greater than Less than
cast_total_facebook_likes			Greater than Less than

Figure 3: Users can specify which columns (or subsets of those columns) they want to be included in their custom database.



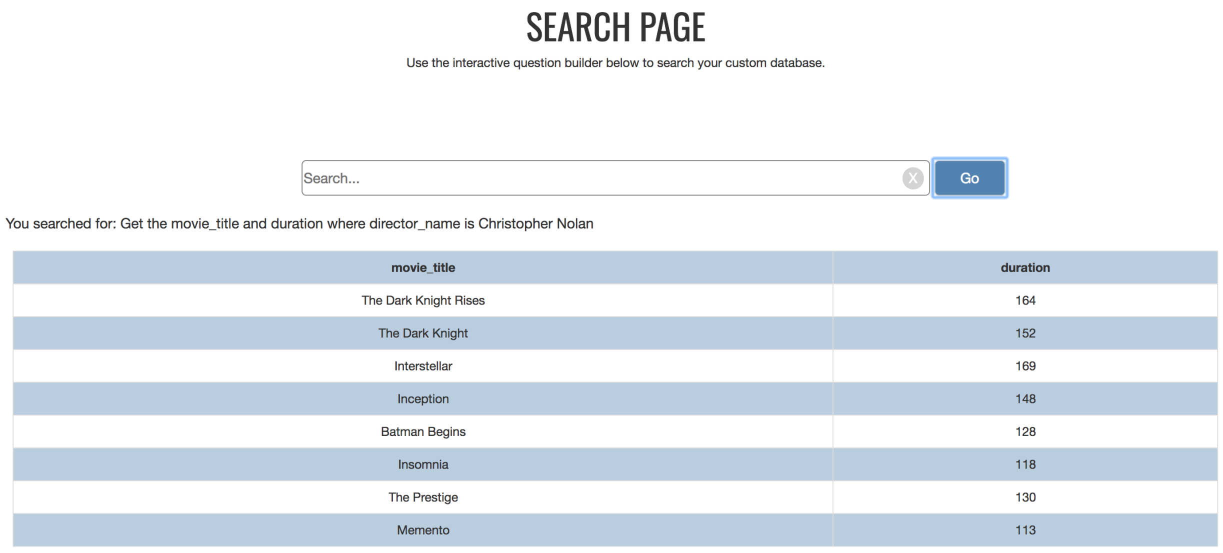
SEARCH PAGE

Use the interactive question builder below to search your custom database.

Get the movie_title and duration where director_name is

Christopher Erskin

Figure 4: Search page with a responsive dropdown menu of query options that adjusts as the user types.



SEARCH PAGE

Use the interactive question builder below to search your custom database.

You searched for: Get the movie_title and duration where director_name is Christopher Nolan

movie_title	duration
The Dark Knight Rises	164
The Dark Knight	152
Interstellar	169
Inception	148
Batman Begins	128
Insomnia	118
The Prestige	130
Memento	113

Figure 5: Results will be displayed after a user executes their query, along with their original request.

4 System Design

4.1 Technology Stack

The backend language we used to build our system is Python. The UI is designed using the standard stack of Javascript, HTML, and CSS, with jquery and ajax used to communicate with the backend. The backend server handles requests and updates from the frontend using flask, and the custom database is a SQLite database. Our implementation can be found here: `https://github.com/mflauer/bfd`.

4.2 Custom Database Creation

In the first step of the process, the user uploads either one or two csv files. For each file uploaded, we initially read the headers of the file, and then save a copy of the uploaded file. In the case that two files are uploaded, we assume the user wants to do a join, and so the headers from each file get sent back to the frontend. The user is then able to select which header from each file to perform the join on, and their selections get sent back to the server. That information is then stored, and during the final creation of the database, an inner join between the two uploaded files is performed on the specified columns.

After choosing the join columns, or if the user uploaded a single file, they are able to choose exactly which columns and values that they want in their custom database. They have the option of choosing which columns are included, choosing multiple specific values from each column included, and if the column consists of numeric values, choosing ranges of values to include. Once the user has input what aspects of the uploaded file they are interested in, that information is sent to the backend and their custom database is built. While the user is able to select and include all of the data from the csv file, we believe a majority of the time they will not need the entire dataset. The ability to select which information to include is thus beneficial in two main ways; firstly it ensures that the user has a sense of exactly what is in their database, and secondly it makes the database lightweight, allowing for faster processing.

Some prior work has relied entirely upon pandas dataframes for interacting with data, so we wanted to explore this technology. However, rather than keeping all of the data as a pandas dataframe, after a user has specified the data they want included in their final database, we convert this pandas dataframe into a .db file. After this file has been created, the rest of the users interactions with the database will be performed on the backend through SQL queries on this database. We chose to use SQLite for this lightweight implementation as it interacts well with Python, but this could easily be swapped out with a different relational database management system.

4.3 Structured User Queries

As mentioned above, previous research has found that in more complex NLP-based systems where users can type unstructured questions into query databases, accuracy of the returned results is often a problem. NaLIR attempted to counteract this by displaying multiple rewordings of the users phrase, while DBPal took an alternate approach, displaying potential queries as the user typed. As discussed in section 2, the success of these approaches still had significant limitations. As we believe that accuracy is of critical importance for users, we prioritized it. In a similar way to DBPal, we display possible queries to the user, which are updated as the user types. However, unlike DBPal, when using BFD the user must select one of the provided options, thereby removing any ambiguity of the returned response. This structured NLP works nicely with the building of a custom database, since during the database creation process, the contents of their database becomes much more transparent, and thus the potential queries displayed will be easier to understand.

4.3.1 Supported Query Structure

The existing BFD system supports a subset of all possible SQL queries. While ideally the system would support any SQL query, given the scope of the project and the amount of time allotted, we chose to implement the queries we thought would be the most useful. We also tried to make each aspect of the system as modular as possible, so that additional features could be added without too much overhead.

With this in mind, we first implemented simple matching queries: return the tuples where a certain column has value equal to X. Since the users must choose options from a dropdown, we begin by sending a list of all columns to the frontend. Similarly, when selecting the value for the chosen column, we retrieve and display every possible value of that particular column. After matching with one column value, we expanded to matching multiple column values, and returning any subset of columns from the database. In SQL terms, this corresponds to simple projection and equality selection.

From there we expanded to handle more complex queries. We allow users to make aggregate queries such as sum, min, max, and average, as well as counts of column values. For the aggregates, we also allow further pruning based on matching specific column values. We also support getting the a range of the top or bottom values from any column, replicating SQL's `ORDER BY` and `LIMIT`.

4.3.2 SQL Query Generation

An important feature of BFD that is different from most other work in the space is that every input question maps to exactly one SQL query. The solution we implemented ended up being able to progressively write the SQL as the user typed, without needing to maintain much internal state. As discussed previously, we only implemented a subset of all SQL queries, so we have not proven that this strategy of incrementally writing the SQL query extends to all queries.

We were able to accomplish this by phrasing questions (as much as we could) such that they presented information to the user in the same order that a SQL command is built. In doing so, we assume the user with relative ease would be able to switch between different phrasings of the same question. For example, we directly convert "How many [tablename] ... " into `SELECT COUNT(*)` .... Then, there was typically some associated logic to put these pieces together in a way that made sense in English (i.e. adding words like "and" and "is" to connect the logical elements).

While ideally we would be able to phrase all questions in the same order that the SQL query is built, it is not always possible. SQL functions that are placed at the end of a query such as `ORDER BY` and `LIMIT` typically appear at the beginning of the question in English. To handle this, we have specific logic set up in our SQL builder to handle and store the state associated with these types of queries. Another alternative that we did not explore would have been to incorporate aggregates as filters over the full list of results, so users could sort or limit the results after the initial query had been run.

5 Preliminary Testing

Most of the testing done on our BFD system was with a dataset from Kaggle of IMDB movie data [1]. We initially thought of the types of basic questions that a non-technical user might be curious about or need to know about this kind of data. We built our first iteration to support these queries. We then did some preliminary user testing by asking test users what types of questions they might want to ask of a movie database, and found that our system was capable of answering a majority of these questions. We added support for the questions that could not be answered, like selecting the top values of a certain column.

One of our design goals was flexibility. Thus, we built our system in such a way that it is able to support csv inputs from any data file with headers and data, like many of the files that can be downloaded from sites like Kaggle that provide public datasets. To test this, we downloaded a range of datasets and tried out our system with each. While these trials illuminated a few bugs in our system (which we subsequently fixed) for the most part as long as there were column headers, our system worked as expected.

6 Conclusion and Future Work

We believe that we successfully achieved our design goals by creating a proof of concept of an end to end system that is easy for users to understand, guarantees high accuracy results, and is significantly more intuitive than SQL for non-technical users. However, in order for this system to be production-ready, there are some aspects which have not been built out to completion. While the current system only supports the predicates, projections, and basic aggregates described in section 4.3.1, a complete system would have the depth necessary to handle any possible query. Additionally, there is very minimal error handling on the frontend or backend. If a user were confused about how to use the system, adding informative error messages would definitely be important to include.

Through the iterative process of building the parser, we started by only supporting a structure matching `SELECT [col] FROM table WHERE [col] = [colValue];`. We were able to relatively easily add support for additional predicates and aggregates by giving users options that were in understandable English, but that kept the backend logic from being overly complex. This system could be improved by performing user testing to optimize the wording and options presented to the users. Additionally, evaluating our own backend infrastructure, we think that another promising and potentially more scalable way to organize the backend would be to create a "Command" interface, and have all commands implement this interface. Each time a user made a selection on the frontend, a corresponding Command would be instantiated on the backend, keeping the backend state more organized. With this kind of structure, each Command could keep track of its own portion of SQL query, update relevant internal state (in some cases i.e. aggregates), and store its user-facing representation to be displayed in the search dropdown.

Another potential area for improvement is handling datasets of larger sizes. Our system handles this to some extent by having users select the subset of columns from their dataset(s) that they are interested in, thereby allowing us to build a smaller database. However, we did not consider optimizing how the new database was created or stored. Depending on the original dataset size, their custom database might still be very large. If this were the case, further optimizations on how we store, create, and query the custom database would be very useful.

References

[1] Imdb 5000 movie dataset. `https://www.kaggle.com/carolzhangdc/imdb-5000-movie-dataset`.

[2] Ion Androutsopoulos, Graeme D Ritchie, and Peter Thanisch. Natural language interfaces to databases–an introduction. *Natural language engineering*, 1(1):29–81, 1995.

[3] Fuat Basik, Benjamin Hättasch, Amir Ilkhechi, Arif Usta, Shekar Ramaswamy, Prasetya Utama, Nathaniel Weir, Carsten Binnig, and Ugur Cetintemel. Dbpal: A learned nl-interface for databases. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1765–1768. ACM, 2018.

[4] Sonia Bergamaschi, Francesco Guerra, Matteo Interlandi, Raquel Trillo-Lado, and Yannis Velegrakis. Quest: a keyword search system for relational data based on semantic and machine learning techniques. *Proceedings of the VLDB Endowment*, 6(12):1222–1225, 2013.

[5] Lukas Blunschi, Claudio Jossen, Donald Kossmann, Magdalini Mori, and Kurt Stockinger. Soda: Generating sql for business users. *Proceedings of the VLDB Endowment*, 5(10):932–943, 2012.

[6] Rogers Jeffrey Leo John, Navneet Potti, and Jignesh M Patel. Ava: From data to insights through conversations. In *CIDR*, 2017.

[7] Fei Li and HV Jagadish. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84, 2014.

[8] Yunyao Li, Huahai Yang, and HV Jagadish. Nalix: an interactive natural language interface for querying xml. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 900–902. ACM, 2005.

[9] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th international conference on Intelligent user interfaces*, pages 149–157. ACM, 2003.

[10] Marta Vila, M Antònia Martí, and Horacio Rodríguez. Paraphrase concept and typology. a linguistically based and computationally oriented approach. *Procesamiento del Lenguaje Natural*, 46:83–90, 2010.

[11] Jeffrey Xu Yu, Lu Qin, and Lijun Chang. Keyword search in relational databases: A survey. *IEEE Data Eng. Bull.*, 33(1):67–78, 2010.

[12] John M Zelle and Raymond J Mooney. Learning to parse database queries using inductive logic programming. In *Proceedings of the national conference on artificial intelligence*, pages 1050–1055, 1996.