BlueprintDB - AI Powered Dataset Generation for Testing

Ege Kabasakaloglu
MIT
egekabas@mit.edu

Tamar Korkotashvili
MIT
tkorkot@mit.edu

Nikoloz Birkadze
MIT
birka@mit.edu

ABSTRACT

Testing relational database systems demands high-quality datasets that balance realism, scalability, and customization—needs often unmet by existing solutions. In this paper, we introduce BlueprintDB, a system that addresses this gap with an AI-powered tool that generates datasets tailored to user-defined schemas. By integrating public data and generating realistic synthetic data, BlueprintDB ensures edge-case coverage, scalability, and relevance, streamlining testing and optimization for relational databases.

1 INTRODUCTION

As the reliance on big data continues to grow, the complexity of queries and database management is becoming increasingly challenging. During the development of relational database systems, high-quality data for testing is extremely important, both when checking for the correctness of the system and for the overall performance optimization process. Finding suitable data sets for testing is a cumbersome process, and it is extremely rare to find an available dataset that satisfies all the needs of the robust application development process.

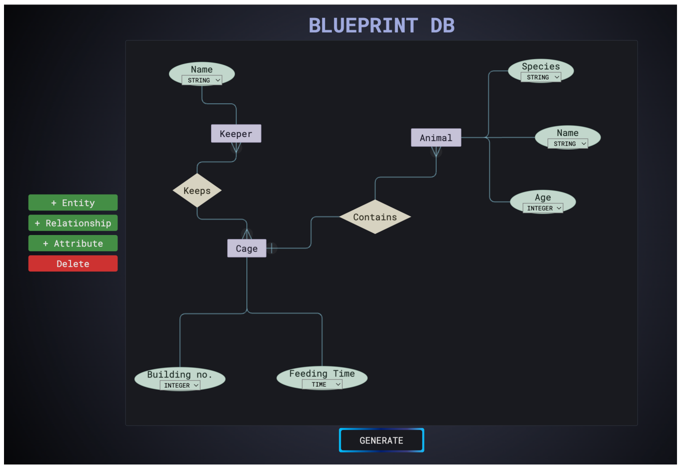
For instance, the size of the dataset is important for exhaustively searching for bugs, and also for testing the scalability of the performance of the developed systems. The realistic nature of the data set is important to predict how the implementation will perform when deployed for real-life usage. How well the tests cover edge cases is also crucial for finding problems with the implementation that might be hard to notice with naive brute-force testing. Moreover, on top of all this, users might want to customize the schema of the data to focus on which features are the most valuable to their implementations currently.

Often these desired properties conflict with each other, and there does not seem to be a simple way of achieving all or most of them at once. There are multiple fake-data generating tools on the internet, and multiple platforms for downloading already available datasets. There are even open-source data integration systems that try to connect datasets. However, none of these systems that we have encountered try to bridge the gap between relevance and utility, meaning that the systems either have fully realistic data, but are inflexible or inconvenient for testing, or they achieve robust testing suites at the expense of creating all of the data artificially.

To address all the problems mentioned above, we developed BlueprintDB, an automatic AI-powered dataset generation tool for testing. Using our system users are able to create their custom schemas using the relational ER Diagram, which we use for searching and downloading relevant public datasets and later populate with appropriate real-life and generated data. BlueprintDB aims to fit the available data to the structure of the provided schema, while also providing a large dataset with all edge cases of entity relationships represented.

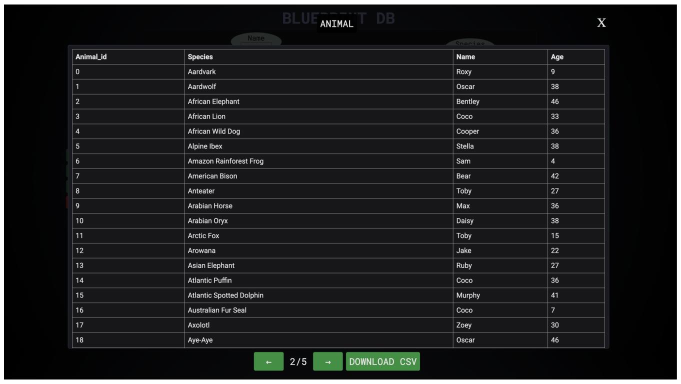
2 USER INTERFACE

To simplify the process of creating the test suite and provide a smooth experience for users whose needs might change over time, we created a graphical interface for users to visualize their desired schemas and add or remove entities and attributes from them as necessary. The UI uses the common ER diagram symbols and notations and can be simply navigated with a mouse, allowing users to create diagrams of arbitrary complexity and most importantly represent different types of relationships easily (one-to-one, one-to-many, many-to-many).



Example of an ER diagram created with BlueprintDB

Afterward, simply clicking the generate button automatically sends a request to the server, which generates the datasets using the methods that we will discuss in the later sections of this paper. On the front end, we also display the outcome of scraping and generation to the user and additionally provide the option to download the whole dataset.



Animal_id	Species	Name	Age
0	Aardvark	Roxy	9
1	Aardwolf	Oscar	38
2	African Elephant	Bentley	46
3	African Lion	Coco	33
4	African Wild Dog	Cooper	36
5	Alpine Ibex	Stella	38
6	Amazon Rainforest Frog	Sam	4
7	American Bison	Bear	42
8	Anteater	Toby	27
9	Arabian Horse	Max	36
10	Arabian Oryx	Daisy	38
11	Arctic Fox	Toby	15
12	Arowana	Jake	22
13	Asian Elephant	Ruby	27
14	Atlantic Puffin	Coco	36
15	Atlantic Spotted Dolphin	Murphy	41
16	Australian Fur Seal	Coco	7
17	Axolotl	Zoey	30
18	Aye-Aye	Oscar	46

The created data being displayed to the user

3 DATASET LOOKUP

To fill the table in our ER diagram with real data, we scrape the web for databases that might be appropriate sources for the current diagram. To do this, we first prompt ChatGPT to generate a list of appropriate search strings, then we use the kaggle API to get the list of available datasets corresponding to each search strings. Among the results, we do a search and choose only the ones that are in a .csv format (there are also datasets with images and similar content that we do not want) and download the most suited one, determined by the title/description and the searched columns.

Here, an important design choice was whether to try search for one dataset that could be used for the entire er diagram, or to search individually for each table in the dataset. The first option would have the ability of providing us with data that was more coherent among the different tables in the diagram. However, it was extremely limiting to what types of ER diagrams could effectively have data generated for them, and our online sources did not have diverse enough datasets to make it feasible. Therefore, we opted to download different datasets for each table in the ER diagram, except for in specific situations that will be discussed in section 4.3.

4 DATA GENERATION

4.1 Column Matching

After fetching proper datasets for each table in the ER diagram, we must figure out effectively which columns in the dataset correspond best with each column of the table in the ER diagram. Finding a one-on-one match might be problematic because the downloaded dataset might contain different naming conventions or might be created in a different context than the ER diagram. To solve this issue, we tried multiple different approaches after ultimately deciding on one.

4.1.1 Word2Vec. Our initial approach was to use Word2Vec to generate vectors for each column in the downloaded dataset and the ER diagram, then use cosine similarity to find how closely related each pair of columns is. This approach has many advantages:

(1) Word2Vec fits into memory and is very fast
(2) We get a numerical score on how good of a match two columns are, which we can use to decide whether to use that match or generate fake data instead
(3) We can get access to how good of a match all columns in the downloaded dataset are to each column in the ER diagram table, allowing us to make a decision with a grand view, and not in a naive greedy way

Despite these benefits, Word2Vec has a huge disadvantage that makes it unusable: It cannot generate vectors for words that are not in its vocabulary, which could be a common occurence given that we randomly download datasets that can name their columns using arbitrary conventions. This is where our second attempt comes into play.

4.1.2 Fasttext. Fasttext is very similar to Word2Vec in that it is a lightweight model that provides vector embeddings for words. However, it has one feature that Word2Vec does not have, it can create vector embeddings for words that are not in its dictionary, by creating subwords from a given word (for example, 'running' could be split into 'run', 'unn', 'nni', 'nin', 'ing'), then compute a vector embedding for the word by combining the information provided by subwords.

Using Fasttext, we developed an algorithm that would match columns one by one, starting with the highest cosine-similarity match, removing those two values, and continuing, until the cosine similarity was below a predetermined threshold.

Although this approach seems good on paper, our testing during the prototyping phase showed that the subword embeddings seemed to fail potential matches between words that should have very similar meanings (like "employee id" and "EMP_ID' or "popularity" and "popular"), which was why we did not end up using it.

4.1.3 ChatGPT. The final idea we tried, and then included in our system, was to use ChatGPT to find appropriate column matchings.

The main disadvantage of this method compared to prior ones is that it might be slower (it requires multiple API calls for good results). However, we already use ChatGPT calls in other parts of the project, and the column matching has a very small effect on the overall runtime (as data scraping and fake data generation are much bigger bottlenecks). Moreover, it is an integral part of creating a dataset that is as realistic as possible, so we decided to neglect the runtime drawbacks.

The main idea is very simple, provide ChatGPT with the list of the columns from the dataset and the list of the columns from the ER diagram table, and ask it to find a matching. However, it turns out it was extremely important to finetune the exact query text so that ChatGPT behaves well. Our main observations was that:

(1) Always use column names, not indices. While indices are more convenient to work with afterwards, they lower ChatGPT's ability to reason greatly.
(2) If you want ChatGPT to leave some columns unmatched, explicitly ask it to name the ones that were left unmatched. Otherwise, no matter what you ask, it will try to make matches that are not good fits.
(3) Finally, provide an example of the expected output.

With this in mind, an example query would look like

```
Match the names in the columns to the keywords
Give the answer in the following
format 'column_name, keyword_name',
where each tuple is on a new line
If you don't think a good match exists,
you can leave the column unmatched by
providing the tuple 'column_name, UNMATCHED'
Please dont add any extra text,
as I will be parsing your output
Moreover, each keyword should be
matched to at most one column
An example input would be:
Columns: name, profession, location, hobby
Keywords: name, city, job, parents, siblings
An example output to this input would be:
name, name
profession, job
location, city
hobby, UNMATCHED
Here, hobby was left unmatched as there was
no good match It is EXTREMELY important that
you leave columns unmatched if you think
no good match exists
Now I will provide you with the input
you should respond to
Columns: payment, age, gender
Keywords: RECORD_NBR, PAY_YEAR,
DEPARTMENT_NO, DEPARTMENT_TITLE,
JOB_CLASS_PGRADE, JOB_TITLE,
EMPLOYMENT_TYPE, JOB_STATUS, MOU, MOU_TITLE,
REGULAR_PAY, OVERTIME_PAY, ALL_OTHER_PAY, TOTAL_PAY,
CITY_RETIREMENT_CONTRIBUTIONS,
BENEFIT_PAY, GENDER, ETHNICITY
```

And we get a well-behaved answer

```
payment, TOTAL_PAY
age, UNMATCHED
gender, GENDER
```

After we know which columns in the ER diagram's tables match which columns in the downloaded dataset the best, we can simply populate the matched columns with values from the dataset and record the appropriate relationships between them.

4.2 Data Generation

It is possible that we cannot find any appropriate data to populate one of the columns, it might be too unusual or specific for us to find any dataset with a matching column during web scraping. In this case, we populate the column with data generated by ChatGPT. We devised two possible ways to do this.

(1) Provide ChatGPT with column names and rows partially populated with the methods described in the previous section, and prompt ChatGPT to fill in the missing values. This method has the advantage of preserving relations between different columns, as it is often not accurate to assume the columns of a table are uncorrelated. However, this approach has the disadvantage of being extremely slow and unreliable. Based on the small unit tests that we ran, ChatGPT only rarely managed to utilize the extra information to create more coherent data and the runtime cost was large enough that it was almost unusable for our aim of generating large exhaustive datasets for testing.
(2) Provide ChatGPT with the names of the unpopulated columns one by one, and ask it to create as many possible values for each column as possible (independent of all other columns). Then, we can populate columns by randomly choosing from the list of possible values ChatGPT provided. This approach assumes that columns are uncorrelated, and although often not a correct assumption, is a large benefit for our runtime, as we only have to query once for column instead of every time for a new row.
As a potential drawback, we discovered that ChatGPT is not good at giving you the requested number of options. To fix that, we tried to add support for integer and float types of columns and instead ask ChatGPT to provide us with potential minimum and maximum values and a distribution functions. As it turns out ChatGPT is not the best at providing the distribution function, but it can provide minimum and maximum values. Even though we favored the first approach, we

plan to represent the realistic distribution better in the future.

4.3 Generating Relations

After generating the data for each of the tables in the diagram, what remains to be done is to fill in the relations between tables. Let us assume there exists a relationship R between two tables T1 and T2. In our algorithms, we consider two potential scenarios.

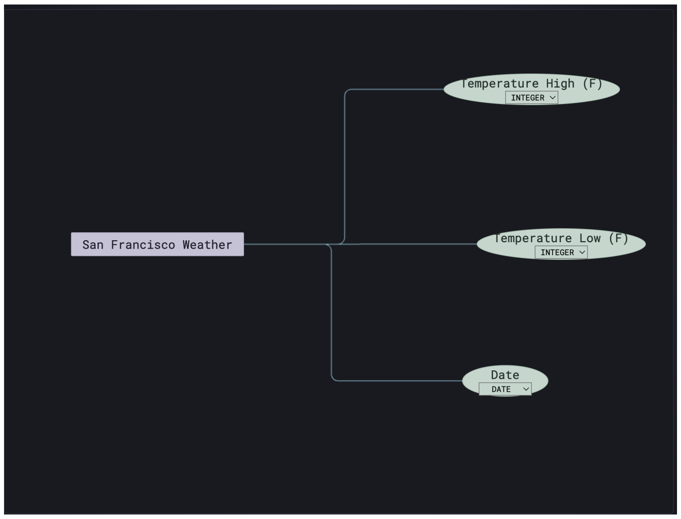
(1) **T1 and T2 are populated using the same dataset:** If the tables T1 and T2 were populated from the same downloaded dataset, then filling in the relations become trivial. If two rows r1 in T1 and r2 in T2 get their data from the same row in the downloaded dataset, then we can infer that there must exist the relation R between them. However, our data scraping algorithm randomly choosing the same dataset for two tables T1 and T2 is extremely unlikely. Thus, we modify our algorithm in the following way: Whenever we download a dataset for any table T1, we loop over the tables T2 that have a relation to T1, and if a sufficient number of columns of T2 can be populated using the same dataset, instead for scraping the web for another dataset for T2, we use the same one, and populate the relation between T1 and T2 accordingly, with real relations. We could also try populating all the tables T3 that have a relationship to T2 as well, and so on, almost doing a graph traversel. However, we saw very sharply diminishing returns with this graph traversel addition (as the downloaded datasets have around 10 columns maximum), so didn't include it in our final system.

(2) **T1 and T2 are populated using different datasets:** In this case, it is extremely hard to infer if a relation exists between two rows r1 belonging to T1 and r2 belonging to T2. There is no information that can be deduced as the original datasets are completely unrelated. Moreover, using ChatGPT to make inferences would require ChatGPT to consider $|T1| \cdot |T2|$ possibilities, making it unfeasible due to runtime concerns. The current solution we implement is to generate these relations randomly, making sure to cover all edge cases in the meantime (for example in a many-to-many relationship, a tuple can have 0, 1, or n other tuples assigned to it.) If there are no 0 or 1 relationships, we randomly delete 10% of n relationships present for representation, and in the case when there are no n relationships, we generate random connections between 2 tables for 10% of the entries adding a random selection of 1 to 5 new entries for each id. Furthermore, if there are any attributes associated with the relationship we can generate them as described in the section 4.2.

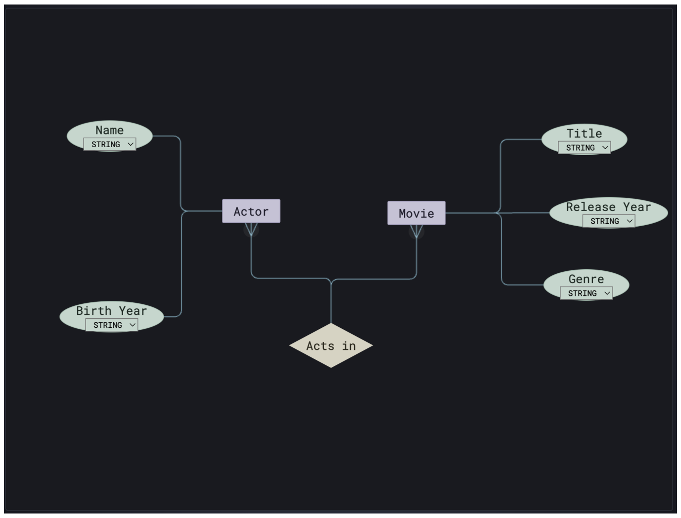
Overall, this inability to infer relations and fall back in random generation in case (2) is one of the biggest obstacles for our system, and potential ways to remedy this are discussed in the Future Work section of the paper.

5 RESULTS

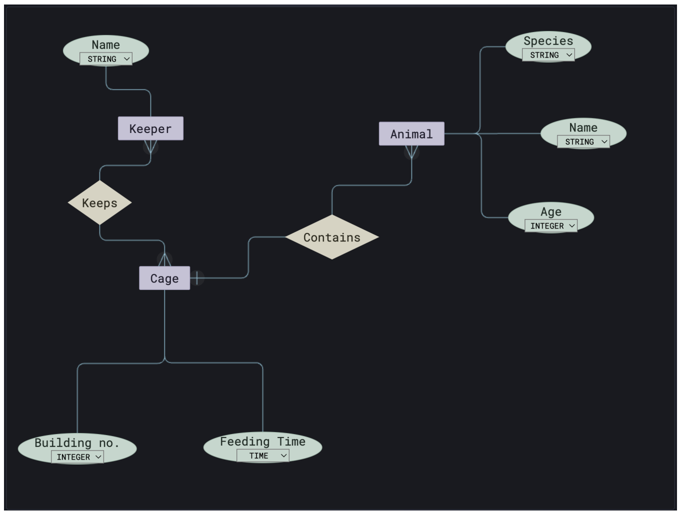
Besides testing for correctness in all the different parts of our system, we also tested the effectiveness of our system on 15 curated examples of ER diagrams. We split the ER diagrams into 3 groups (easy, medium, and hard) based on how complicated they were: decided by the number of entities, attributes, and relations and how likely we would be able to find real-life datasets that are good matches.



Example of an easy ER diagram



Example of an medium ER diagram

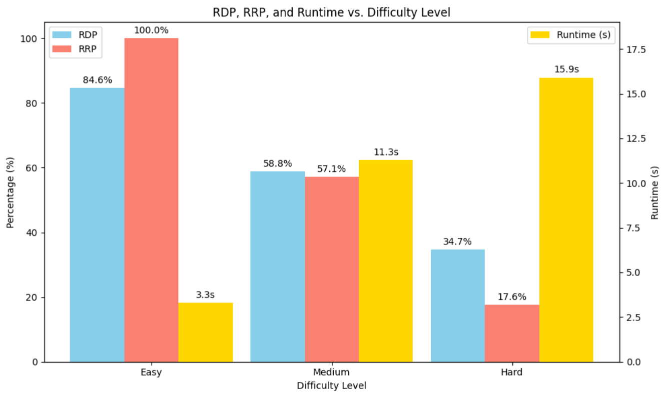


Example of a hard ER diagram

Moreover, for each generated result for each diagram, we measured three metrics:

(1) **RDP (Real Data Percentage):** What percent of the generated data (not including its relations) is real (from real-life datasets)
(2) **RRP (Real Relation Percentage):** What percent of the relations are real (inferred from the downloaded dataset as opposed to generated randomly, as described in section 4.3)
(3) **Runtime:** How long does it take for the system to generate the data.

The results are summarized in the graph below.



Because we avoided choices that lead to intensive computation requirements in the previous sections, the run-time was within reasonable bounds for all ER diagram types. Moreover, an interesting result is that RRP seems to drop more steeply than RDP, confirming our suspicions in 4.3 that accurately inferring relations is the main obstacle for our system.

The code for the entire developed system can be accessed at the following link: https://github.com/nik-ab/blueprint-db

6 FUTURE WORK

Although we were very satisfied with the results we obtained, there are still many areas to explore to improve the system.

6.1 Fine-Tuned Models

In the final version of the project, the AI model that we used for all the different parts of the project was ChatGPT turbo 3.5. However, this might not have been the best option available. For example, for the column matching part, a fine-tuned model that specializes in word or specifically column matching would be more performant than a general-purpose LLM, and possibly even more runtime efficient. A similar argument could be made for the data generation part.

6.2 Better Data Scraping

We used Kaggle to download data sets because of the ease of use of its API and the consistency of the datasets provided. During the development process, we considered using Google's Dataset Search platform, which interestingly enough has all the Kaggle datasets linked as well. However, the platform does not support API calls anymore, and after doing initial tryouts with the platform we discovered that many dataset download links are deeply nested. In our development of the system we preferred the simple and reliable process of getting data from kaggle. However, it is not impossible to implement a different data-scraping tool that can overcome this nesting issue even without an available API: this would enable us to get access to many different datasets that are usually posted on university or government websites which are open to public but unfortunately not available on platforms like Kaggle.

6.3 Additional Ways Generating Relations

As mentioned in Section 4.3, improvements could be made to the generation of relationships when the tables in a relationship are populated using different datasets.

(1) **Better Randomness:** When generating the relations, we used the simplifying assumption that the relations were uniformly distributed. However, this is not always the case in real life: For example, people who already have many friends are more likely to make more friends in the future. It might be a good idea to support multiple distributions and allow users the option to choose between them in the UI.

(2) **AI Inference:** Even though the computation requirement for AI inference to determine relations was too high for us for this project, if given better resources, there are some optimizations that could make it feasible. For example, randomly choosing a subset of the $|T1| \cdot |T2|$ relation possibilites and asking the AI to pick among those, or maybe using a lightweight specially trained model, that is not a bulky LLM like ChatGPT.

7 CONCLUSION

BlueprintDB demonstrates promising results in generating realistic and scalable datasets tailored to diverse testing needs. The system effectively balances customization, relevance, and edge-case coverage, addressing critical challenges in database testing. While these initial outcomes are encouraging, there is ample room for enhancements. Future work will focus on refining data integration techniques, expanding schema customization options, and conducting extensive testing to further validate and improve the system's capabilities.