

Deeper: A Data Enrichment System Powered by Deep Web

Pei Wang[◇] Yongjun He^{◇†} Ryan Shea[◇] Jiannan Wang[◇] Eugene Wu[‡]
Simon Fraser University[◇] Nanjing University[†] Columbia University[‡]
{peiw, rws1, jnwang}@sfu.ca 141250047@smail.nju.edu.cn ew2493@columbia.edu

ABSTRACT

Data scientists often spend more than 80% of their time on data preparation. Data enrichment, the act of extending a local database with new attributes from external data sources, is among the most time-consuming tasks. Existing data enrichment works are resource intensive: data-intensive by relying on web tables or knowledge bases, monetarily-intensive by purchasing entire datasets, or time-intensive by fully crawling a web-based data source. In this work, we explore a more targeted alternative that uses resources (in terms of web API calls) proportional to the size of the local database of interest. We build Deeper, a data enrichment system powered by deep web. The goal of Deeper is to help data scientists to link a local database to a hidden database so that they can easily enrich the local database with the attributes from the hidden database. We find that a challenging problem is how to crawl a hidden database. This is different from a typical deep web crawling problem, whose goal is to crawl the entire hidden database rather than only the content relating to the data enrichment task. We demonstrate the limitations of straightforward solutions, and propose an effective new crawling strategy. We also present the Deeper system architecture and discuss how to implement each component. During the demo, we will use Deeper to enrich a publication database, and aim to show that (1) Deeper is an end-to-end data enrichment solution, and (2) the proposed crawling strategy is superior to the straightforward ones.

1 INTRODUCTION

Data enrichment (a.k.a. entity augmentation) is defined as enriching an entity table with new attributes extracted from external data resources. It is a crucial step in the data-science life-cycle because adding new attributes to the data can uncover new insights, and provide rich and highly predictive features, which enable data scientists to answer more interesting questions. As a simple example in entity resolution, the ability to enrich restaurant names with address, latitude, longitude, owner, and other attributes greatly improves the quality of the resolution process. Similarly, companies that develop prediction models routinely crawl web APIs and websites to generate additional features. For instance, a lead scoring company (anonymized) helps sales people rank potential clients to call based on their likelihood to generate a sale—this company crawls web APIs to augment the list of clients with information about the client, the client’s company, their industry, and other predictive metadata. This is a process that nearly every data journalist, data science student, and practitioner has experience in the process of analysis and model development [1].

The need for Deeper. Despite the importance of this problem, the tools to help an analyst make enrichment decisions are still evolving. One approach is to simply purchase entire datasets or data streams from a company or data market [4]. However, the

From Local DB			From Hidden DB
Restaurant	Address	City	Category
Boiling Point	4148 Main St	Vancouver	Hot Pot
Flamingo Chinese Restaurant	1652 SE Marine Drive	Vancouver	Dim Sum
Sun Sui Wah Restaurant	3888 Main Street	Vancouver	Seafood, Dim Sum

Keyword Search Interface




Figure 1: Enrich a local restaurant database with the category attribute extracted from Yelp.

analyst must be confident that the data will improve the model or application enough to justify the purchase cost. Although some data markets propose to price on a per-query [7, 9] or per-API call basis, how can an analyst quickly and cheaply decide *which* API calls to make in order make a purchasing decision?

Another approach is to use an existing data enrichment system [6, 11–13]. However, they focus on leveraging large existing corpuses, such as Web Tables and Knowledge Bases, that the analyst may not have access to, or they may not have the data that the analyst needs (e.g., Yelp ratings, Google Scholar citations).

A third approach, and the one we take, is to leverage the *Deep Web*. The Deep Web (or Hidden Database) is a database that can only be accessed through a restrictive query interface (e.g., keyword search, or form-like interface). There is a great opportunity to leverage *deep web* for data enrichment. First, many deep websites (e.g., Yelp, IMDB) contain rich and high-quality information about a large collection of entities. Second, they often contain some unique information about entities (e.g., Yelp ratings, Google Scholar citations) which cannot be found from anywhere else. Third, many of them provide Web Service APIs to facilitate data access. Although leveraging the deep web is promising, current work is primarily focused on techniques to fully crawl a given hidden database, by comprehensively filling in input forms or API elements.

The above three approaches have a key aspect in common: they require large amount of resources, either existing resources in the form of web tables, monetary resources to purchase full datasets, or time resources to fully crawl the deep web. The goal of Deeper is to develop a targeted enrichment system to quickly enrich the analyst’s existing dataset sufficiently to build their application or decide where to commit the above resources. In the long term, Deeper should *enable data scientists to enrich local data using any given hidden database in the Web and complete the data-enrichment task within minutes*.

In this paper, we present our solution to fulfill part of the vision, by focusing on hidden databases behind keyword search APIs. That is, a hidden database takes as input a keyword-search query (e.g., “Thai Restaurant”) and returns the top-*k* results that match the keywords based on an unknown ranking function. Even with the assumption, the system is already challenging to build.

Deep Web Crawling (Section 2). Deep web crawling is the main technical challenge. Since a hidden database can be accessed through a keyword-search API, we need to determine which queries should be issued to the hidden database in order to crawl the related data. Note that this is different from existing works, whose objective is to crawl the entire hidden database. For example, suppose we want to enrich a restaurant table of 10 records. There is no need to crawl the entire Yelp but only the records that can match the 10 records. We formalize the problem and discuss the limitations of straightforward solutions. We propose SMARTCRAWL, a novel deep web crawling approach to overcome the limitations. The experimental results [3] show that SMARTCRAWL can save the number of queries by up to 10× compared to the straightforward solutions.

End-to-End System (Section 3). Deeper is an end-to-end data enrichment system. On one end, a user uploads an entity table and selects a keyword-search API; on the other end, the user gets the enriched entity table. To enable the end-to-end experience, we need to develop some other system components, such as schema matching and entity resolution. Schema matching matches the (inconsistent) schemas between a local database and a hidden database; entity resolution finds matching record pairs between a local database and the crawled part of a hidden database. We discuss how each component is implemented as well as how to put them together.

We have built the Deeper system¹ and made a demo video². The system was used in a data science class at Simon Fraser University and helped the students to reduce the time spent on data enrichment from hours to minutes. In our demonstration, we will let the participants to test all the functionalities of Deeper within a web application. The users can upload a dataset or use our example data, and then perform data enrichment using one or multiple deep web sources. We will show that the underlying algorithm is not only cost-efficient but also robust to data errors. Our system now supports integration from three APIs, namely DBLP, Yelp, and AMiner, and it can be easily extended to other hidden databases with keyword search supported.

2 LOCAL-TABLE-AWARE DEEP WEB CRAWLING

In this section, we present our solution to the *local-table-aware deep web crawling* problem.

Problem Definition. Consider a local database \mathcal{D} with $|\mathcal{D}|$ records and a hidden database \mathcal{H} with $|\mathcal{H}|$ (unknown) records. Each record describes a real-world entity. We call each $d \in \mathcal{D}$ a *local record* and each $h \in \mathcal{H}$ a *hidden record*. Without loss of generality, we model a local database and a hidden database as two relational tables.

Local records can be accessed freely; hidden records can be accessed by issuing queries through a *keyword-search interface*. Let q denote a keyword-search query, and $q(\mathcal{H})_k$ denote the top- k hidden records that match the query. We say a local record d is covered by a query q if and only if there exists $h \in q(\mathcal{H})_k$ such that d and h refer to the same real-world entity.

Definition 2.1 (Local-table-aware deep web crawling). Given a budget b , a local database \mathcal{D} , and a hidden database \mathcal{H} , the goal

of local-table-aware deep web crawling is to find a set of b queries such that they can cover as many local records in \mathcal{D} as possible.

NAIVECRAWL. A naive approach is to enumerate each record in \mathcal{D} and then generate a query to cover it. For example, a query can be a concatenation of restaurant name and address attributes. In fact, OpenRefine (a state-of-the-art data wrangling tool) has been using this approach to crawl data from web services [2]. However, this approach has two drawbacks. First, it suffers from high query cost because it needs to issue one query for every record. For example, suppose $|\mathcal{D}| = 1,000,000$. NAIVECRAWL has to issue one million queries to the hidden database. If one wants to enrich the data using the Google Map API (2500 free requests per day), she will spend up to $\frac{1,000,000}{2500} = 400$ days to complete the job. Second, NAIVECRAWL is not robust to data errors. Each query issued by NAIVECRAWL tends to be very long because it aims to retrieve a specific hidden record. A long query is more likely to be affected by data errors than a short query. If a query contains some errors, it may not be able to retrieve the data that we are looking for. For example, suppose “Sun Sui Wah Restaurant” is falsely written as “Sun Sui Wah Restaurant 12345”. If the issued query is “Sun Sui Wah Restaurant 12345”, a hidden database may not return the matching restaurant to us.

SMARTCRAWL. To overcome the drawbacks, we propose SMARTCRAWL, a new local-table-aware deep web crawling strategy. It has two stages: query pool generation and query selection.

- In the first stage, SMARTCRAWL initializes a *query pool* by extracting the queries from \mathcal{D} . The query pool does not only contain specific queries (like what NAIVECRAWL has) but also general queries (e.g., “Noodle”) that can cover multiple local records at a time.
- In the second stage, SMARTCRAWL first selects the query q^* with the largest *benefit* from the query pool, issues q^* to the hidden database, removes the covered records from \mathcal{D} , and updates the query pool. This iterative process will repeat until the budget is exhausted or the local database is fully covered.

Compared with NAIVECRAWL, SMARTCRAWL generates both specific and general queries. A hidden database typically sets the top- k restriction with k in the range between 10 to 1000 (e.g., $k = 100$ for Google Search API, $k = 1000$ for DBLP Publication API). Suppose $k = 100$. At best, SMARTCRAWL can use a single query to cover 100 records, which is 100 times better than NAIVECRAWL. Furthermore, since general queries tend to contain fewer keywords, it is less sensitive to data errors.

Benefit Estimation. The main challenge is how to estimate the benefit of issuing a query before issuing it. This is a “chicken-and-egg” problem. We solve the problem by estimating the query benefit based on a *hidden database sample*. There is a large body of work on deep web sampling [5, 10, 14], aiming to randomly choose a set of records from a hidden database. Recently, Zhang et al. [14] propose efficient techniques that can create an unbiased sample of a hidden database as well as an unbiased estimate of the sampling ratio by issuing keyword-search queries. SMARTCRAWL applies these techniques to create a hidden database sample *offline*. Note that the sample only needs to be created once and can be reused by any user who wants to match their local database with the hidden database. We propose a number of effective estimators, prove

¹<http://deeper.sfucloud.ca/>

²<http://tiny.cc/deeper-video>

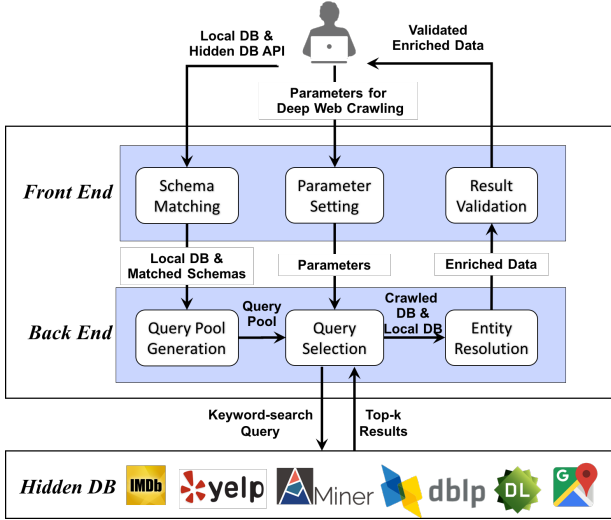


Figure 2: The Deeper System Architecture.

some good theoretical properties, and validate their effectiveness experimentally [3].

3 THE DEEPER SYSTEM

Figure 2 shows the Deeper system architecture. The system is divided into two parts: a front-end user interface, which handles the interaction with end users, and a back-end infrastructure, which handles the interaction with hidden databases.

3.1 Front-End User Interactions

The front end of Deeper is a web-based user interface (see Figure 3). It allows users to upload their data, do schema matching, configure parameters for deep web crawling, refine the output of an entity resolution algorithm, and download the enriched table.

In the beginning, a user uploads a local CSV file and selects a hidden database API (e.g., DBLP Publication API). The system first parses the file into a table.

Schema Matching. The local table and the hidden database may have some attributes matched but with different representations. The user needs to perform schema matching to link the corresponding attributes between them. Figure 4 illustrates the user interface of our schema matching component. The top one shows the attributes in the local table and the bottom one gives the schema of the hidden database. By clicking the attributes in order, the user can build the correspondences between attributes, where the correspondences are indicated by the numbers behind the attribute names.

Parameter Setting. The user needs to specify a number of parameters for the deep web crawling component, including query budget, the number of threads, and the number of queries issued per second. After that, the user clicks the “Try it now” button, and a progress circle shows the running time and how much work are done currently. The back end generates queries from the table, issues queries, and parses the returned results. Detailed implementation will be presented in Section 3.2.

Result Validation. Once a set of records are crawled, we link them to the local table. However, entity resolution (ER) is a challenging problem. It is hard to develop a perfect machine-only ER approach.

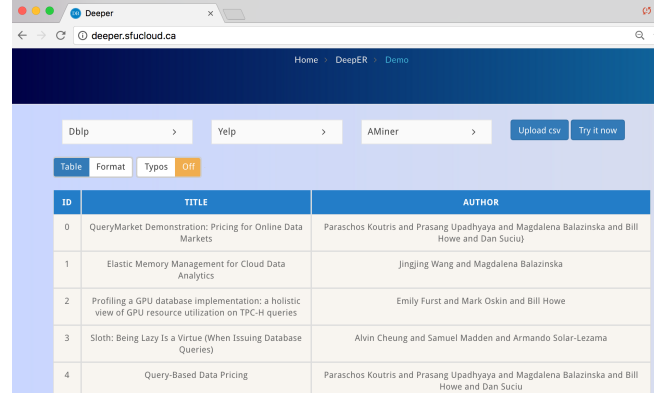


Figure 3: Deeper Web-based User Interface.

Therefore, we allow the user to validate the final data-enrichment result. Figure 5 shows an example. Each row represents an enriched record, where ID, TITLE, and AUTHOR come from the local table, and INFO.TITLE, INFO.AUTHORS.AUTHOR.*, INFO.VENUE, and INFO.YEAR are from DBLP. We highlight the enriched record (i.e., the 3rd row) that the system is not certain about. The user can decide whether to keep it or not.

Once the user is satisfied with the enriched table, she can download it as an CSV file.

3.2 Back-End Implementations

The back end is responsible for crawling data from a hidden database, and performing ER between local data and crawled data.

Query Pool Generation. Once a local table is uploaded to the server, we first generate a query pool for the table. The query pool consists of general queries and specific queries. For general queries, we use the same set of queries as NAIVECRAWL. For specific queries, we use the frequent pattern mining algorithm [8] to extract a collection of frequent keyword sets (e.g., “memory” and “data analytics”).

Query Selection. Query selection is an iterative process. We propose a number of optimization techniques to speed up this process [3]. For example, to compute query benefit, we need to know how many records in the local table contain a query. A naive way is to scan the entire local table and then check whether each record contains the query or not. Instead, we build an inverted index over the local table. The inverted index is a hash map which maps each keyword to an inverted list—the list of records containing the keyword. Given a query with a set of keywords, we retrieve the inverted list of each keyword, and then compute the intersection of the inverted lists, without the need to scan the whole table.

Entity Resolution. Once data is crawled, for each record in the local table, we need to check whether there is a matching one in the crawled data. We implemented a similarity-based ER approach. The approach computes the similarity between two records based on a similarity function (e.g., Jaccard, Edit Distance). If the similarity value is larger than a threshold (e.g., 0.5), they are considered as matching; otherwise, non-matching. It is worth noting that ER is an independent component in our system. If one wants to use another ER approach (e.g., learning-based), she just needs to implement it and plug it into our system. In the future, we plan to add more built-in ER approaches to Deeper.

SCHEMA MATCHING

YOUR SCHEMA

ID: 0

title: 1

author: 2

API SCHEMA

info.key: 0

info.title: 1

info.authors.author.*: 2

@score

info.url

info.venue

info.volume

info.year

info.type

@id

url

SUBMIT

Figure 4: Schema Matching.

	ID	TITLE	AUTHOR	INFO.TITLE	INFO.AUTHORS.AUTHOR.*	INFO.VENUE	INFO.YEAR
	0	QueryMarket Demonstration: Pricing for Online Data Markets	Paraschos Koutris and Prasang Upadhyaya and Magdalena Balazinska and Bill Howe and Dan Suciu	QueryMarket Demonstration - Pricing for Online Data Markets	Paraschos Koutris Prasang Upadhyaya Magdalena Balazinska Bill Howe Dan Suciu	PVLDB	2012
▲	1	Elastic Memory Management for Cloud Data Analytics	Jingjing Wang and Magdalena Balazinska	Elastic Memory Management for Cloud Data Analytics	Jingjing Wang Magdalena Balazinska	USENIX Annual Technical Conference	2017
⚡	1	Elastic Memory Management for Cloud Data Analytics	Jingjing Wang and Magdalena Balazinska	Toward elastic memory management for cloud data analytics.	Jingjing Wang Magdalena Balazinska	BeyondMR@SIGMOD	2016

Figure 5: Result Validation.

Python Library. We have implemented the back end of Deeper, and open-sourced the Python library for developers³. The system is easy to extend. Developers can easily extend the library by adding new Web APIs or customize their own ER algorithms.

4 DEMONSTRATION PROPOSAL

We aim to demonstrate that (1) Deeper is an *end-to-end* data enrichment solution, and (2) SMARTCRAWL is *superior* to NAIVECRAWL. Imagine Bob is a PhD student. He collects a list of interesting DB papers, but each paper has only title and author attributes. He wants to add venue, year, and citation# attributes to the data.

End-to-End Experience. He first loads the data to the Deeper system and selects the DBLP publication API (see Figure 3). Then, he clicks the “Try it now” button. The system will pop up a window and ask him to do schema matching (see Figure 4). After that, the system will send the data along with the schema-matching result to the back end, and start to enrich the data. Once this process is finished, the system will show an enriched table (see Figure 5). Bob can validate the result and check whether there are some mistakes. Since the DBLP API does not have the citation# attribute, Bob has to get it from another API (e.g., the AMiner API). To do so, Bob just needs to select the AMiner API and repeats the above process.

Superiority of SMARTCRAWL. As discussed in Section 2, SMARTCRAWL is superior to NAIVECRAWL for two reasons: i) it can save a lot of queries, ii) it is more robust to data errors. To demonstrate these points, Deeper provides a functionality called *error ingestion*. Bob can use it to add errors to his data. When we say 5% errors are ingested, it means that we randomly select x% of the records, and for each record, we either add a random word or delete an existing word. Bob can run SMARTCRAWL and NAIVECRAWL at the same time, and compare their performance.

We constructed a local database of 10,000 papers from major DB venues, and added 5% and 50% data errors to it, respectively. We varied the number of queries issued, and compared the number of enriched records using SMARTCRAWL and NAIVECRAWL. Figure 6(a)

³<https://pypi.python.org/pypi/deeperlib>

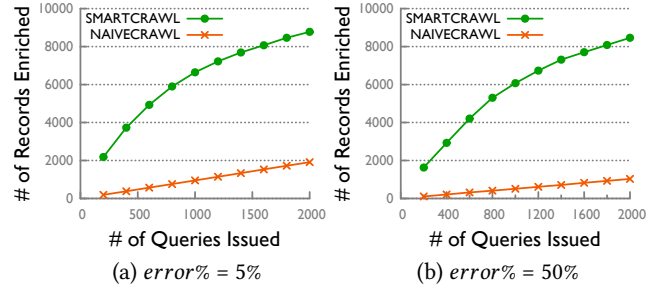


Figure 6: Comparisons of SMARTCRAWL and NAIVECRAWL.

and (b) show the results. First, we can see that SMARTCRAWL saved a lot of queries. For example, in Figure 6(a), SMARTCRAWL can enrich 2000 records by issuing only 250 queries but NAIVECRAWL needs to issue 2000 queries, which is about 10× more expensive. Second, we can see that SMARTCRAWL is more robust to data errors. For example, in the case of *error%* = 5%, SMARTCRAWL and NAIVECRAWL can use 2000 queries to enrich 8775 and 1914 local records, respectively. When *error%* was increased to 50%, SMARTCRAWL can still enrich 8463 local records (only missing 3.5% compared to the previous case) while NAIVECRAWL can only enrich 1031 local records (46% less than the previous case).

5 CONCLUDING REMARKS

This paper described a demonstration of Deeper, a system to incrementally and quickly enrich local databases by leveraging deep web APIs—specifically keyword-based APIs in the current system. Our intention is to use this both as a tool for data science practitioners to improve models and make decisions about where to commit data enrichment resources, and as part of data science curricula by making it easy to leverage deep web APIs as part of homework and projects. In this latter setting, we have tested it in an SFU data science class, and helped the students to save hours of time to enrich their local databases.

REFERENCES

- [1] 5 Ways to Use Enrichment. <https://blog.clearbit.com/5-ways-to-use-clearbits-enrichment-api/>. Accessed: 2017-07-12.
- [2] OpenRefine Reconciliation Service. <https://github.com/OpenRefine/OpenRefine/wiki/Reconciliation-Service-API>. Accessed: 2018-01-15.
- [3] P. Wang, R. Shea, J. Wang and E. Wu. SmartCrawl: Deep Web Crawling Driven by Data Enrichment. <http://deeper.sfucloud.ca/Deeper/>.
- [4] M. Balazinska, B. Howe, and D. Suciu. Data markets in the cloud: An opportunity for the database community. *PVLDB*, 4(12):1482–1485, 2011.
- [5] Z. Bar-Yossef and M. Gurevich. Random sampling from a search engine’s index. *J. ACM*, 55(5):24:1–24:74, 2008.
- [6] M. J. Cafarella, A. Y. Halevy, and N. Khoussainova. Data integration for the relational web. *PVLDB*, 2(1):1090–1101, 2009.
- [7] S. Deep and P. Koutris. The design of arbitrage-free data pricing schemes. *arXiv preprint arXiv:1606.09376*, 2016.
- [8] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, pages 1–12, 2000.
- [9] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu. Query-based data pricing. *Journal of the ACM (JACM)*, 62(5):43, 2015.
- [10] F. Wang and G. Agrawal. Effective and efficient sampling methods for deep web aggregation queries. In *EDBT*, pages 425–436, 2011.
- [11] M. Yakout, K. Ganjam, K. Chakrabarti, and S. Chaudhuri. Infogather: entity augmentation and attribute discovery by holistic matching with web tables. In *SIGMOD*, pages 97–108, 2012.
- [12] M. Yang, B. Ding, S. Chaudhuri, and K. Chakrabarti. Finding patterns in a knowledge base using keywords to compose table answers. *PVLDB*, 7(14):1809–1820, 2014.
- [13] M. Zhang and K. Chakrabarti. Infogather+: semantic matching and annotation of numeric and time-varying attributes in web tables. In *SIGMOD*, pages 145–156, 2013.
- [14] M. Zhang, N. Zhang, and G. Das. Mining a search engine’s corpus: efficient yet unbiased sampling and aggregate estimation. In *SIGMOD*, pages 793–804, 2011.