Demonstration of Ver: View Discovery in the Wild

Kevin Dharmawan[*]
kevin.dharmawan@ui.ac.id
University of Indonesia
Depok, Indonesia

Chirag A. Kawediya[*]
ckawediya@uchicago.edu
The University of Chicago
Chicago, USA

Yue Gong
yuegong@uchicago.edu
The University of Chicago
Chicago, USA

Zaki Indra Yudhistira
zaki.indra@ui.ac.id
University of Indonesia
Depok, Indonesia

Zhiru Zhu
zhiru@uchicago.edu
The University of Chicago
Chicago, USA

Sainyam Galhotra
sg@cs.cornell.edu
Cornell University
Ithaca, USA

Adila Alfa Krisnadhi
adila@cs.ui.ac.id
University of Indonesia
Depok, Indonesia

Raul Castro Fernandez
raulcf@uchicago.edu
The University of Chicago
Chicago, USA

ABSTRACT

We demonstrate **Ver**[1] [10], a data discovery system that identifies project-join views over large repositories of tables that do not contain join path information. Ver solves both the technical (scale and search) and human (semantic ambiguity, navigating a large number of results) problems of view discovery.

CCS CONCEPTS

• **Information systems → Information integration**.

KEYWORDS

Data Discovery, Data Integration, Query-by-Example

1 INTRODUCTION

Large data repositories that originate from different data sources, such as data lakes [12], open data portals [11], and cloud repositories [1], offer a wealth of data for machine learning, reporting, and analytics. However, they also bring a data discovery problem: finding the right combination of datasets for a specific task, especially when they are spread across different databases. For example, A data analyst at an e-commerce company may require product data from an enterprise data lake along with web traffic data from another database. This vast amount of often incomplete and noisy data, referred to as *pathless table collections* in [10], makes data discovery challenging and hampers the productivity of data analysts.

Many approaches have been developed to identify project-join views (PJ-views) over pathless table collections. Many data discovery systems [4, 5, 8, 9, 16] find datasets that satisfy certain relevance

criteria. Analysts can first locate relevant datasets using these systems and then merge those to obtain the final view. There are also discovery systems [7, 14] utilizing Query-by-Example (QBE) interfaces [18], which enables users to specify examples of the data they need and identify PJ-views directly. However, pathless table collections pose unique challenges to existing discovery systems. When conducting discovery queries on such collections, the results are frequently ambiguous, including semantically different results, and varying data versions. This ambiguity complicates the process of identifying the desired view. To tackle the problem of view discovery over pathless table collections, we have recently introduced an end-to-end system, **Ver** that identifies PJ-views among tens of millions of join paths and effectively navigates users to the desired result. The full research paper introducing the system is available [10]. **Ver** addresses the following challenges:

- **Challenge 1. Noisy Queries.** Users input queries based on their current understanding of the data. These inputs may not always align with the actual data in the table collection and could be imprecise or include mistakes.

- **Challenge 2. Noisy Join Paths.** In pathless table collections, join paths are not specified, making it unfeasible to automatically determine true join paths. Instead, we focus on detecting inclusion dependencies, which approximate true join paths. Therefore, join paths can be noisy and erroneous in our scenario.

- **Challenge 3. Large Number of Join Paths.** Large volumes of data lead to a large number of join paths. For example, there are 28.6M join paths in an open data repository with ~69K tables [10].

- **Challenge 4. Noisy Result Views.** When dealing with numerous noisy join paths and ambiguity in a query, there might be many result views satisfying a user's query. The results may contain duplicates, views contained in each other, complementary views, and even views with contradictory values.

- **Challenge 5. Result View Navigation.** Noisy result views bring a navigation problem. It is necessary to learn the preferences of users and guide users to the desired view.

In this paper, we demonstrate how **Ver** aids users in finding the desired data in a real-world scenario. We deploy **Ver** on Chicago Open Data [13] and showcase its capacity to assist a school counselor in identifying a view related to school information. **Ver** enables the counselor to specify the data needs via an example query, searches for relevant views, distills the results, and finally guides them to the right view by asking a series of data-related questions.

---

[*]Both authors contributed equally to this work.
[1]code is available at https://github.com/TheDataStation/ver

Related Work. Data discovery systems [2–5, 8] support dataset search and join discovery to enrich a given dataset. Specifically, MACO [2] enables users to augment an input dataset with additional correlated attributes. Auctus [4] is a dataset search engine, supporting spatio-temporal filtering and join discovery. However, these systems do not address the ambiguity and noise in both the query and the data, as well as the subsequent result navigation problem.

2 SYSTEM OVERVIEW

Ver introduces a reference architecture for view discovery, incorporating components designed to address both human and technical aspects of the problem.
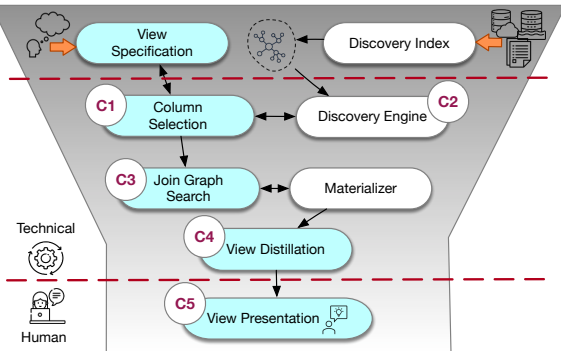

Figure 1: Ver Reference Architecture. The funnel shows the reduction of data as it flows downstream. (Figure from [10])

Discovery Engine and Index Creation (technical). *This component is responsible for building indices for pathless table collections, which includes: i) an index for approximate join paths, and ii) various retrieval indices that span table names, values, attribute names, and similarities between columns. All these indices are made accessible to other modules through the Engine's API (**Challenge 2**).*

Ver employs Aurum [5] to create the discovery indices. Alternative approaches [8, 17] can be integrated. Once these indices are established, users can then proceed to design and submit their queries using the VIEW-SPECIFICATION component.

View Specification (human). Various discovery interfaces, such as keywords, APIs, natural language, and combinations of those can be integrated into the VIEW SPECIFICATION component. **Ver** implements Query-By-Example (QBE) interfaces, in which the user can specify a set of examples to indicate the data they want.

Subsequently, COLUMN-SELECTION identifies and selects the subset of tables containing the examples provided by the user.

Column Selection (technical). *This component identifies candidate tables and columns even in the presence of noisy input queries, addressing **Challenge 1***.

Join Graph Search (technical). *This component identifies all join graphs that can merge the candidate tables utilizing the discovery index that provides the join paths. A join graph yields a PJ-view when materialized using MATERIALIZER component.*

JOIN-GRAPH-SEARCH generates a large number of *candidate views*. Ranking these views is challenging due to variations in users' search criteria. To address this, VIEW-DISTILLATION reduces the view search space by summarizing the candidate views.

View Distillation (technical). *This component categorizes candidate PJ-views, identifying view pairs like redundancy, containment, and potential opportunities for unioning views. Certain categories can be used to summarize the views (**Challenge 4**).*

VIEW-PRESENTATION receives the distilled views and can rank them. It returns the first-ranked view in a fully automated mode, or, alternatively, it can utilize the categories computed by VIEW-DISTILLATION to assist users in pinpointing the most relevant data.

View Presentation (human). *VIEW-PRESENTATION uses a variety of question interfaces to ask questions about the resulting views, thereby learning user preferences and knowledge. Through answering questions, users gain further insights about the resulting views, being guided to the desired view. (Challenge 5).*

After a user submits an example query, **Ver** utilizes the offline discovery indices to locate relevant datasets that match user-specified examples, identify join graphs to combine candidate datasets, and materialize the join graphs to generate candidate views.

Ideally, a query returns one PJ-view. In practice, ambiguity, redundancy, erroneous join paths, and large table repositories can lead to hundreds of result views. **Ver** uses two novel components VIEW-DISTILLATION and VIEW-PRESENTATION to reduce the view search space and interactively guide users to the right view. Next, We give an overview of VIEW-PRESENTATION and VIEW-DISTILLATION. Please refer to our full paper [10] for more details.

2.1 View Distillation

VIEW-DISTILLATION reduces the view search space by first classifying candidate views into four categories and then applying a distillation strategy.

4C Categories. VIEW-DISTILLATION classifies candidate views into the following 4 categories.

- **Compatible** view pair: Two candidate views, V_1 and V_2, are compatible if their sets of rows are identical, $(V_1 \cap V_2) = V_1 = V_2$.
- **Contained** view pair: A view, V_1, is contained by another view, V_2, if and only if $V_1 \subset V_2$.

The categorization of views as either *Complementary* or *Contradictory* depends on their candidate keys. In any given view V, the candidate key, $K(V)$, is a set of attributes that uniquely identify each tuple within V.

- **Complementary** view pair: Two views, V_1 and V_2, are complementary if they have the same candidate key $K(V_1) = K(V_2)$ and overlapping rows $|V_1 \cap V_2| > 0$, without being contained or compatible with each other.
- **Contradictory** view pair: Two views V_1 and V_2 are contradictory when they share a candidate key $K(V_1) = K(V_2)$, but there exists a same key value that leads to different rows in each view.

VIEW-DISTILLATION efficiently classifies views into 4C categories by i) using the transitivity property to not compare any pair of views whose categorization can be inferred from prior comparisons; ii) creating row-level hashes to find compatible and contained views efficiently; iii) utilizing an inverted index that maps each key value to their rows to accelerate the detection of complementary and contradictory views.

Distillation Strategy. VIEW-DISTILLATION implements the following distillation strategy: i) deduplicate compatible views, ii)

retain the largest contained view, iii) union complementary views. Alternative strategies can be applied for specific use cases. **Ver** implements this strategy to help reduce view search space that View-Presentation needs to consider.

Complementary and contradictory view pairs are useful signals for users to prune noisy views and steer towards the correct view. These view pairs are subsequently passed to View-Presentation to generate data questions.

2.2 View Presentation

view-presentation analyzes the views and generates questions that aid in ranking and selecting the most relevant views. For instance, it might ask a user about their preference between "home address" and "work address" if it identifies both in the views. By answering these questions, users gain a better understanding of the available schemas and datasets, enabling them to fine-tune their preferences and exploration requirements.

Question Interface. To capture users' diverse preferences effectively, **Ver** designs the following question interfaces:
• **Dataset interface**: This interface shows a view to a user and asks them if it is their desired view.
• **Attribute interface**: This interface shows a user an attribute and asks whether it should be included in the intended output.
• **Dataset Pair**: This interface displays a pair of views to users, prompting them to choose one. It is specially designed to utilize the 4C categorization of views (Section 2.1).
• **Summary interface**: This interface displays a summary for a set of views (e.g., a word cloud), and asks a user if it is relevant for the desired output.

View-Presentation has two choices to make at each iteration: i) what question interface to choose; ii) how to prioritize questions to show on the chosen interface. For example, if the dataset interface is chosen, then it could show a candidate view that is most relevant to the input example query, or a view with the highest data quality. **Ver** implements two strategies to prioritize questions: i) semantic distance of the question from the input example query; ii) semantic distance of dataset schema to input query. **Ver** supports other interface designs and strategies for prioritizing questions. Additionally, users can always skip questions and **Ver** adapts to their responses.

Bandit-Based View Presentation Algorithm. To effectively navigate users to the desired view, View-Presentation needs to ask questions that i) narrow down view choice space and ii) users are capable of answering. Naive approaches that rely solely on a user's initial preference for a question interface and then continue to ask questions based on this chosen interface may not adapt well to changes in the user's knowledge and preferences.

To address this challenge, View-Presentation employs a multi-arm bandit algorithm to model users' evolving preferences. In the algorithm, each question interface represents an arm, and answering a question is pulling an arm. The reward of a question is its information gain, defined as the maximum number of views that can be pruned after answering it. It also uses a parameter to balance the choice of interface with the most information gain (exploitation) and explore a random question interface (exploration) at each iteration. In summary, View-Presentation achieves a balance between choosing questions that users are capable of answering and

questions that prune many irrelevant views, by adaptively learning and accommodating users' preferences.

Ranking Views. View-Presentation keeps all views unless explicitly discarded by a user, allowing users the flexibility to revisit their selections as their knowledge evolves. It ranks the views according to their capacity to fulfill users' answered questions. More concretely, The score of a view is calculated as a weighted sum of the view's utility for each question.

3 DEMONSTRATION

To demonstrate how **Ver** aids users in finding the desired view, we design a real-world scenario: Anna is a school counselor in Chicago and she wants to help parents and students choose which schools to attend. The data she needs is a table that contains information about every public school in Chicago, including the school name, its curriculum/degree type (ex: IB, General Education, etc), and its latest rating based on Chicago's School Quality Rating Policy (SQRP) [6] as shown in Table 1.

School Name	Type	Level
Ogden Intl. High School	IB	Level 1
Hyde Park High School	General Education	Level 2
...	...	...

Table 1: The Desired View of Anna

Anna has explored several methods to collect data, but none met her needs: i) She checked the Chicago Public Schools (CPS) website for school-specific information, but manually reviewing each school was too time-consuming; ii) She searched the Chicago open data portal for "schools" datasets. Despite finding over a hundred datasets, none individually contained all the needed information; iii) Anna also consulted ChatGPT, which cannot provide a complete, real-time table, and suggested the same CPS website approach she had already tried. Next, we describe how **Ver** addresses Anna's problem, as illustrated in Fig 2.

Index Creation (Offline). Ver first builds a data discovery index over Chicago Open Data during the offline stage (before the demo starts). The retrieval index over table names, values, and attribute names is implemented in DuckDB [15]. The join path index is built using Aurum [5].

Step 1: View Specification. Anna knows the information about a few schools in Chicago. For example, *Ogden International High School* has the type *IB* and the rating *Level 1*; *Hyde Park High School* has the type *General Education* and the rating of *Level 2*. She would like the final view to include these examples. The QBE interface of View-Specification enables Anna to specify the example data to illustrate the view she desires. As shown in Fig. 2, Anna inputs descriptive column names and a few example rows to specify the view she wants. She can use the buttons above the table to adjust the shape of the example table and click "Find Views" to let the system search for views according to the specified examples.

Step 2: Generating Candidate Views. Ver then searches for candidate columns that match the specification using its discovery index, searches all join graphs to combine candidate columns, and finally materializes all candidate graphs to generate the views for
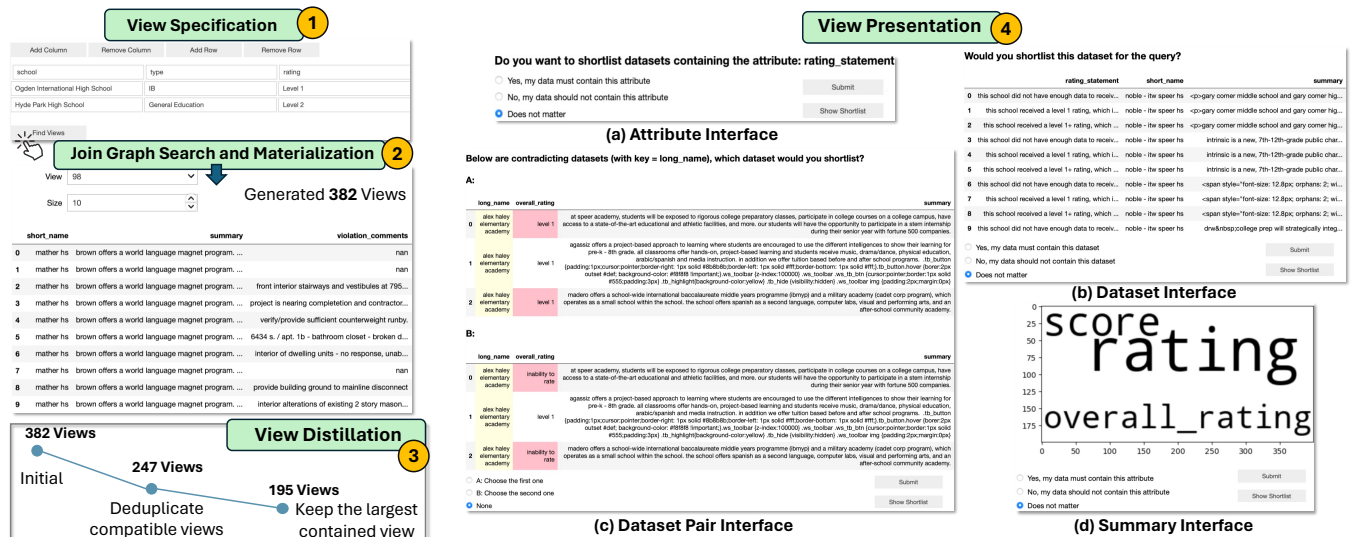
Figure 2: Ver Interface in Jupyter Notebook. Left (1-3): Specify example queries, search for join graphs, generate and distill views; Right (4): View Presentation Question Interfaces.

the user. **Ver** generates 382 views for Anna's example query. She can browse each generated view using **Ver**'s API.

Step 3: View Distillation. Manually examining 382 views is laborious for Anna. **Ver** uses View-Distillation to reduce the view search space. It deduplicates compatible views, keeps the largest view for contained views, and unions complementary views. After that, there are 195 views left.

Step 4: View Presentation. As the number of remaining views is still too large for Anna to manually go through, View Presentation analyzes the views and generates questions that when answered, help rank and select views, effectively navigating Anna to the desired view. View Presentation provides Anna with 4 question interfaces: Fig 2a is an Attribute interface, in which Anna is asked whether a specific attribute (in this case, rating_statement) should be present in the view, Fig. 2b shows the Dataset interface, in which Anna is shown a candidate view and asked if it satisfies her requirements or whether it should be eliminated, Fig. 2c displays the Dataset Pair interface, in which Anna can choose between two contradictory views and choose which one aligns with her requirements, and Fig. 2d shows the Summary interface, in which Anna is shown a word cloud of attributes in the data to determine whether it contains the data relevant to her needs. Anna can always skip any question by choosing "Does not matter" and **Ver** adapts to the responses. The user can also stop at any time and check the current ranking of views by clicking "Show Shortlist" button.

Finally, the counselor, Anna, finds the desired view have a high rank in the view list, and obtain the school information she needs.

Interactivity. During the demonstration, we will walk participants through Anna's use case. In the view specification phase, they can formulate example queries using the QBE interface. Following this, they can explore the join graphs generated by **Ver** and browse the candidate views. This process will help them grasp the impact of noisy join paths. Lastly, they can engage with View Presentation to answer data questions and narrow the scope of views. This step will allow participants to understand how View Presentation mitigates noise in the data and disambiguates result views. We will prepare Chicago Open Data [13] and 5 example queries for users to explore.

REFERENCES

[1] M. Armbrust and et al. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *CIDR '21*.
[2] J. Becktepe, M. Esmailoghli, M. Koch, and Z. Abedjan. 2023. Demonstrating MATE and COCOA for Data Discovery. In *SIGMOD '23*. 119–122.
[3] A. Bogatu, A. A. A. Fernandes, N. W. Paton, and N. Konstantinou. 2020. Dataset Discovery in Data Lakes. In *ICDE '20*. IEEE, 709–720.
[4] S. Castelo and et al. 2021. Auctus: a dataset search engine for data discovery and augmentation. *Proc. VLDB Endow.* 14, 12 (jul 2021), 2791–2794.
[5] R. Castro Fernandez and et al. 2018. Aurum: A Data Discovery System. In *ICDE '18*. IEEE, 1001–1012.
[6] Chicago Public Schools. 2023. School Quality Rating Policy (SQRP). https://www.cps.edu/about/district-data/metrics/sqrp/. Accessed on April 12, 2024.
[7] A. Fariha and et al. 2018. SQuID: Semantic Similarity-Aware Query Intent Discovery. In *SIGMOD '18*. 1745–1748.
[8] R. C. Fernandez and et al. 2019. Lazo: A cardinality-based method for coupled estimation of jaccard similarity and containment. In *ICDE '19*. 1190–1201.
[9] S. Galhotra, Y. Gong, and R. Fernandez. 2023. Metam: Goal-Oriented Data Discovery. In *ICDE*. IEEE Computer Society, 2780–2793.
[10] Y. Gong, Z. Zhu, S. Galhotra, and R. C. Fernandez. 2023. Ver: View discovery in the wild. In *ICDE '23*. IEEE, IEEE, 503–516.
[11] N. Huijboom and T Van den Broek. 2011. Open data: an international comparison of strategies. *European journal of ePractice* 12, 1 (2011), 4–16.
[12] F. Nargesian and et al. 2019. Data lake management: challenges and opportunities. *Proc. VLDB Endow.* 12, 12 (aug 2019), 1986–1989.
[13] Chicago Data Portal. 2024. *Chicago Data Portal.* Retrieved January 12, 2024 from https://data.cityofchicago.org
[14] F. Psallidas and et al. 2015. S4: Top-k Spreadsheet-Style Search for Query Discovery. In *SIGMOD '15*. 2001–2016.
[15] M. Raasveldt and H. Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *SIGMOD '19*. ACM, 1981–1984.
[16] Y. Zhang and Z. G. Ives. 2019. Juneau: data lake management for Jupyter. *Proc. VLDB Endow.* 12, 12 (aug 2019), 1902–1905.
[17] D. Zhu and et al. 2019. Josie: Overlap set similarity search for finding joinable tables in data lakes. In *SIGMOD '19*. 847–864.
[18] M. M. Zloof. 1975. Query by example. In *AFIPS '75*. ACM, 431–438.