Ver: View Discovery in the Wild

Yue Gong*, Zhiru Zhu*, Sainyam Galhotra, Raul Castro Fernandez
Department of Computer Science, The University of Chicago
Email: {yuegong, zhiru, sainyam, raulcf}@uchicago.edu
*Yue Gong and Zhiru Zhu contributed equally

***Abstract*—We present `Ver`[1], a data discovery system that identifies project-join views over large repositories of tables that do not contain join path information, and even when input queries are inaccurate. `Ver` implements a reference architecture to solve both the technical (scale and search) and human (semantic ambiguity, navigating a large number of results) problems of view discovery. We demonstrate users find the view they want when using `Ver` with a user study and we demonstrate its performance with large-scale end-to-end experiments on real-world datasets containing tens of millions of join paths.**

I. Introduction

The existence of large repositories of data such as those that originate from the combination of disparate databases [1], data lakes [2], open data portals [3], and cloud repositories [4] has the upside of offering opportunities to find valuable data for tasks such as machine learning, reporting, and data analytics. The downside is the resulting data discovery problem: identifying a combination of datasets useful for the downstream task even when these reside in different databases. For example, a machine learning engineer may need a training dataset that requires combining a table in a database with the one sitting on the enterprise data lake. Large volumes of often incomplete and noisy data without any join information, which we call *pathless table collections*, make solving data discovery difficult and thus hampers productivity.

There are several approaches to identify project-join views (PJ-views) over pathless table collections. Discovery systems such as Aurum [5], Goods [6], Auctus [7], Juneau [8], Josie [9], Table-Union [10], D3L [11] and libraries such as LSHEnsemble [12], and Lazo [13] help with identifying datasets that satisfy some relevance criteria whether via keywords [6] or programs [5]. Analysts can then combine the datasets to verify that it satisfies the view they need. Another approach is query-by-example (QBE) interfaces [14], [15] that lets users provide examples of the view they need. These can be adapted, with effort, to run on top of data discovery systems. Whether via keywords, programs, or QBE interfaces, the result of discovery queries over pathless table collections leads to ambiguous results that include semantically distinct results, duplicates or near-duplicates, and different versions of the data. This ambiguity further complicates identifying the right view. More generally, solving view discovery in the wild requires addressing the following challenges:

- **Challenge 1. Noisy Queries.** Users provide queries that represent their best knowledge of the data in the form of keywords, programs, and examples. User-provided input may or may not appear in the table collection and they may be noisy and incorrect.
- **Challenge 2. Noisy Join Paths.** Pathless table collections do not include join paths. Identifying true join paths automatically is impossible. We resort to identifying inclusion dependencies, a proxy to join paths, i.e., a join path implies an inclusion dependency but not the other way around.
- **Challenge 3. Large Number of Join Paths.** Large volumes of data result in a large number of join paths that require efficient data structures to represent and navigate them.
- **Challenge 4. Noisy Result Views.** A large number of join paths implies many views may satisfy a user query. Such result views will be noisy due to ambiguity in the user query and noisy join paths. Concretely, there will be duplicate result views, views that are contained within each other, others that are complementary, and others that will show contradictory values for the same key.
- **Challenge 5. Result View Navigation.** Result views will contain semantically ambiguous results, e.g., views with "work address" and "home address". Only users know the right context so the challenge is to elicit that context and use it to choose the view they need among all result views.

In this paper, we introduce a *reference architecture to identify PJ-views in the wild*. Discovering a PJ-view over pathless table collections requires understanding human preferences and requirements (which we refer to as a *human* problem) and solving a *technical* problem. The architecture divides the larger problem into smaller ones, each of which we tackle with a different component. Reference architectures help conceptualize problems and have been influential in advancing the field. For example, [16], [17] for relational databases and [18], [19], [20], [21] for data integration. The reference architecture we propose tackles the five challenges above. While challenges 1-3 are addressed by existing work, we introduce new techniques to address Challenge 4 and 5 in this paper, and demonstrate them as part of an implementation of the reference architecture. We make the following contributions:

- An end-to-end system, **`Ver`**, that identifies PJ-views among tens of millions of join paths. **`Ver`** implements the reference architecture for QBE-based interfaces. We choose QBE because it permits users to declare the table they would like to find, even when they do not know examples and can only provide attribute names (Section III). **`Ver`** relies on existing work [5] to address Challenges 1-3.

[1]code is available at https://github.com/TheDataStation/ver

- A **view presentation** component that helps humans to identify a good view among many. The approach uses a novel bandit-based approach to *learn* user-specific discovery preferences, and it uses abduction-based reasoning to quickly narrow down the search space, reducing the number of human interactions (Section IV).
- A **view distillation** component that automatically filters out views by classifying them into **4C categories**: *compatible*, *contained*, *complementary*, and *contradictory*. Besides helping with filtering, these categories produce information used by the view presentation component (Section V).

We conduct an IRB-approved user study (Institutional Review Board) [22] to validate **`Ver`**'s approach to the human problem. We conduct thorough experiments on queries from open data repositories that lack join paths.

II. Definitions and Problem Statement

$\mathcal{R}(A_1, \ldots, A_m)$ is a relation schema over m attributes, where A_i denotes the i^{th} attribute. A table D comprises a schema $\mathcal{R}(A_1, \ldots, A_m)$ and a set of tuples T where each tuple $t \in T$ is a specific instance of the schema.

In practice, tables do not look like the ideal defined above because they may *lack header information*, have *ambiguous names* and contain *dirty and noisy data*. More formally:

Definition 1 (Noisy structured data): A noisy data D is characterized by an incomplete schema information $\mathcal{R}(A_1, \ldots, A_m)$ where $A_i = \phi$ for missing header values and tuples T such that each tuple $t \in T$ contains at most m values.

In addition, a pathless collection may contain tables with contradictory values, e.g., two census tables with different population counts for the same states of the country. Formally, two tables D_i and D_j contradict if the tables contain different values for the same key. We discuss the detection of contradictions in Section V.

Definition 2 (Pathless table collection): A pathless table collection contains a set of noisy tables $\mathcal{D} = \{D_1, \ldots, D_n\}$ where each D_i is a noisy table and tables D_i, D_j may contain contradictory values.

PJ-example-query. A PJ-query (hereafter called query) contains (possibly noisy) example tuples of the desired output. The examples are a proxy to user's discovery requirements. The quality of examples depends on user's knowledge. Given a query q, there may be many tables that contain the input examples, and many combinations of these may satisfy q, resulting in a large number of candidate PJ-views.

Definition 3 (Noisy query): An input query q is a noisy table consisting of l example tuples, $\chi = \{\chi_1, \ldots, \chi_l\}$ where each χ_i is a noisy tuple denoting example values that are expected to be present in the desired output. The different columns in the examples χ are denoted by $\chi.A_i, \forall i \in \{1, \ldots, \tau\}$, where τ is the number of attributes in the input query.

PJ-views are constructed by joining datasets through keys. We first define a join path and then use it to discuss the effects of noise, followed by a formal problem statement.

Definition 4 (Join path): A join path P is defined as an ordered set of noisy tables $P \equiv \{D_1, \ldots, D_t\}$ such that tables D_j and D_{j+1} join for all $j < t$ via a key column $k \in D_j, D_{j+1}$ forming a chain of join operations.

Joinable datasets can be identified in the presence of key information, which is generally absent in pathless scenarios. A PJ-view V is the dataset formed after materializing the path P followed by projection, i.e., choosing the relevant columns.

Problem 1 (Project-Join view discovery over pathless table collections): Consider a pathless table collection $\mathcal{D}$ and a query q with examples χ. The goal is to construct a minimal candidate set of PJ-views $\mathcal{D}'$ that satisfy the user requirements.

Users may require the PJ-view to contain all examples χ, or any of them, depending on user and application. Formalizing these requirements is outside the scope of this work.

III. A Reference Architecture

Why a Reference Architecture? From a systems engineering perspective, a reference architecture is the materialization of a "divide and conquer" strategy that splits complex engineering problems into smaller parts. Thus, a reference architecture describes a collection of components and their interactions. In making a reference architecture concrete, we state our understanding of the problem and represent it as a concrete artifact that the community can scrutinize and improve.

Design Overview

Finding a PJ-view in the wild requires solving a *human* and a *technical* problem. We present a reference architecture for view discovery that contains components targeting both *human* and *technical* problems. Along with each component, we discuss implementation options, including those made by **`Ver`**. We also briefly discuss two novel components, View Distillation and View Presentation.

Overview. Algorithm 1 shows pseudocode (human components are highlighted in blue) and Fig. 1 shows **`Ver`**'s architecture inside a funnel, denoting the gradual reduction of views as data flows downstream. **`Ver`** builds a discovery index offline.

Discovery Engine and Index Creation (technical). *This component builds indices over pathless table collections: i) a join path index, which can be approximate; ii) retrieval indices over table names, values, attribute names and column similarity. The indices are available online, via the Engine's API to other components. (**Challenge 2**).*

The indices can be built using state-of-the-art methods such as Aurum [5] (that **`Ver`**'s implementation uses), Auctus [7], JOSIE [9], LSHEnsemble [12]. After the indices are built, users design and submit queries via a view-specification component (line 1).

View Specification (human). Discovery interfaces include spreadsheet-style, keyword search, APIs, natural language, and combinations of these. The reference architecture supports these interfaces via the View-Specification component. For QBE-based interfaces, as implemented by **`Ver`**, the input is a set of examples, χ, and the output of this stage is a set of example attributes and values. View-Specification

Algorithm 1: Ver Design Overview

```
Input  : Pathless table collection $\mathcal{D}$, Discovery Index $I$
Output: PJ-views $\mathcal{V}$
1  $\chi \leftarrow$ VIEW-SPECIFICATION($\mathcal{D}$)
2  CAND $\leftarrow \phi$
3  for $\chi.A_i \in$ COLUMNS($\chi$) do
4      CAND($A_i$) $\leftarrow$ COLUMN-SELECTION($\chi.A_i, \mathcal{D}, I$)
5      if MODE==Interactive then
6          CAND($A_i$) $\leftarrow$ Query CAND($A_i$)
7      CAND $\leftarrow$ CAND $\cup$ CAND($A_i$)
8  $\mathcal{V}_{PJ} \leftarrow$ JOIN-GRAPH-SEARCH(CAND, $\chi$)
9  $\mathcal{S} \leftarrow$ VIEW-DISTILLATION($\mathcal{V}_{PJ}$)
10 if MODE==Interactive then
11     $\mathcal{V} \leftarrow$ VIEW-PRESENTATION($\mathcal{V}_{PJ}, \mathcal{S}$)
12 else
13     $\mathcal{V} \leftarrow$ Rank $\mathcal{V}_{PJ}$ based on overlap score
14 return $\mathcal{V}$
```

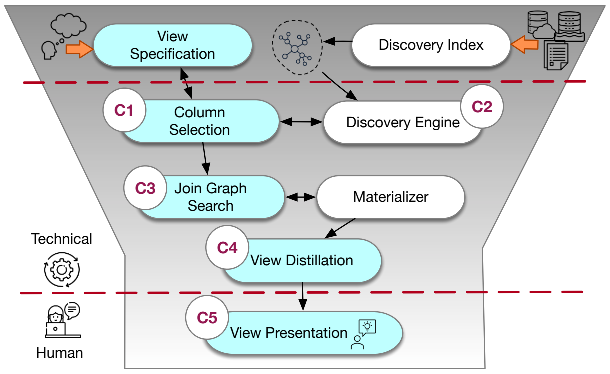


Fig. 1: **Ver** Reference Architecture. The funnel illustrates the progressive reduction of data as it flows downstream.

may interact with users when it detects the provided examples are ambiguous, or to offer examples to choose from.

Next, the COLUMN-SELECTION component selects the subset of tables containing user-provided examples, χ (lines 3-7).

COLUMN SELECTION (TECHNICAL). *This component must be designed to identify relevant data even when the input query is noisy, addressing **Challenge 1** above. The output of this component is a collection of candidate tables and columns.*

The candidate columns are processed by the JOIN-GRAPH-SEARCH component (line 8) to enumerate and materialize candidate PJ-views (addresses Challenge 3). The candidate view search space reduces, as shown in Fig. 1.

JOIN GRAPH SEARCH (TECHNICAL). *Given a set of candidate tables, an input query, and the discovery index providing join paths, this component identifies all* join graphs *that, when materialized, produce candidate PJ-views. The main goal of this component is to address the large join path space (**Challenge 3**). To materialize candidate PJ-views, the* JOIN-GRAPH-SEARCH *component uses a* MATERIALIZER, *a data processing component with the capacity to execute PJ queries.*

JOIN-GRAPH-SEARCH returns many *candidate views*. Ranking the views is hard because of users' differing search criteria. The VIEW-DISTILLATION component (line 9) summarizes the candidate views, further reducing the view search space as shown in Fig. 1.

VIEW DISTILLATION (TECHNICAL). *This component computes* categories *from the candidate PJ-views that include redundancy and containment in the views, as well as opportunities for unioning views and more. Some categories can be used to distill/summarize the views (**Challenge 4**). Others are shared with the downstream component.*

VIEW-PRESENTATION receives the distilled views. It can *rank* the views and return top-1 (for a full automated mode) or it can leverage the categories computed by VIEW-DISTILLATION to help users find the right data (lines 10-13).

VIEW PRESENTATION (HUMAN). VIEW-PRESENTATION *uses different question interfaces to elicit information from users via data questions. The questions are designed to narrow down the space until users find the desired view (**Challenge 5**). The component chooses what questions to ask, sequentially, using a bandit-based approach.*

Next, we present VIEW-PRESENTATION (Section IV), VIEW-DISTILLATION (Section V). We also introduce the implementation details of DISCOVERY-INDEX, COLUMN-SELECTION and JOIN-GRAPH-SEARCH in Appendix.

IV. VIEW PRESENTATION

Ideally, a query results in one PJ-view. In practice, ambiguity, redundancy, erroneous join paths, and large table repositories mean there may be hundreds of result views. **Ver** uses novel techniques to reduce the number of views automatically (Section V), but there is a limit to automation. Semantic ambiguity requires involving users to obtain the final view. The VIEW-PRESENTATION component analyzes the views and generates questions that, when answered, help rank and select views. For example, this component will ask a user if they want "home address" or "work address" in their output when it detects both in the views. By asking questions, users learn more about the schemas and datasets available, and thus refine their preferences and discovery needs. A key challenge is that different users may be able to respond to different questions and that their preferences evolve as they interact with the system. **Ver**'s VIEW-PRESENTATION component is based on two design principles that cater to varied user needs.

• **Mixed-Initiative Interface Design:** Lack of knowledge about available datasets inhibits users from effectively querying the identified set of views. However, users can answer questions about their desired view. Each user has a different understanding of the requirements and finds different interfaces to be more appropriate depending on that. Therefore, users may need different interface designs to answer questions. For example, some users could recognize phone number from the area code, while others might look for a pattern across values. Because different users will be able to answer different questions, **Ver** supports different question interfaces, each asking a different question type. This is motivated by previous mixed-initiative designs [23], [24]. Besides, **Ver** learns which interface to offer to a user according to their previous actions.

• **Adapt to evolving users' knowledge:** Users' understanding of their discovery need evolves as they interact with questions and learn about the schemas and data contained in the repository. **Ver** is designed so users can change their mind about previously answered questions, thus using newly acquired knowledge, without forcing users to start afresh.

Question Interface. To cater to the diverse preferences of users, **Ver** considers the following collection of different interface designs.

- **Dataset interface**: This interface shows users a candidate view to check if it satisfies user's requirements.
- **Attribute interface**: This interface shows users an attribute and asks if it should be present in the desired output.
- **Dataset Pair**: This interface shows users a pair of views and asks them to pick one. This interface is specifically designed to leverage 4C categorization of views (Section V).
- **Summary interface**: This interface summarizes a collection of views and checks if it is relevant for the desired output. We use a wordcloud to visualize the summary.

At every iteration, there are two choices to make: i) what interface design to choose (i.e., how to show the user the question); ii) what prioritization strategy to use to choose the question to show on the chosen interface. For example, if the algorithm chooses the attribute interface, then it could show an attribute similar to the input query or one that is different from others previously shown. We implement two prioritization strategies to order questions: i) distance of the question from the input query; ii) distance of dataset schema to input query. **Ver** supports other interface designs and prioritization strategies. Finally, users can always skip any question and **Ver** adapts to their responses. We discuss the relative usefulness of these interfaces in Section VI-A.

Ver's view presentation addresses the following problem

Problem 2: Given a collection of views $\mathbb{V}$ and access to a user who answers questions through interfaces $\mathbb{I}$, prioritize questions to identify the desired view while minimizing the number of queries to the user.

The VIEW-PRESENTATION component is designed to help users interact with the collection of candidate views and effectively navigate the result views. We now present the key insights we use to solve the problem and **Ver**'s view presentation algorithm.

A. Bandit-Based View Presentation Algorithm

A key design principle of the VIEW-PRESENTATION component is to not prune any views unless specifically discarded by the user. Instead, the VIEW-PRESENTATION component ranks the views, giving users the ability to revisit their choices as their knowledge evolves. It must balance the need for asking informative questions that help narrow down views with questions that the specific user may be able to answer. Merely learning first what question interface the user prefers (exploration stage) and then asking questions based on that interface alone (exploitation stage) would be brittle to users' changing knowledge and preferences. The algorithm is based on two insights: i) learning the best interface for a given user can be modeled as a bandit problem and; ii) the reward in the bandit problem should be based on a question's potential to reduce the number of candidate views.

Bandit-Based approach. A multi-arm bandit algorithm naturally models probabilistic user preferences. Each question interface is an arm, a question-answer pair is pulling an arm, and the reduction in candidate views after the answer is the reward. We design the algorithm off Exp3 [25] because i) it does not make assumptions about the reward distribution; ii) the expected reward is represented as the arm's weights and; iii) has provable guarantees. Exp3 uses an exponential growth function to adapt weights of arms that obtain a positive reward, and all arms start with the same weight and are considered independent from each other. We improve on this behavior by leveraging knowledge of the user's expected reward.

Question's reward. The reward of a question q is its expected information gain, defined as the maximum number of irrelevant views that are pruned if the user answers q. Information gain becomes the reward of the bandit formulation and thus it guides the questions that VIEW-PRESENTATION asks users.

Algorithm 2: VIEW-PRESENTATION

```
Input  : Candidate set of views $\mathbb{X}$, Set of question interfaces $\mathbb{I}$,
         Exploration factor $\gamma$
Output: Set of required views $\mathbb{S}$.
1  $\mathbb{S} \leftarrow \mathbb{X}$
2  while $j \in \{1, 2, \ldots, T\}$ do
3      for $I \in \mathbb{I}$ do
           /* Iterate over interfaces to estimate
              probability of selection                    */
4          $r(I) \leftarrow$ Estimate likelihood to answer question with interface $I$
5          $\chi(I) \leftarrow$ Calculate info gain if the question using $I$ is answered
6          $w(I) \leftarrow r(I) \times \chi(I)$
       /* Normalize to calculate probability          */
7      $p(I) \leftarrow (1-\gamma)\frac{w(I)}{\sum_{I \in \mathbb{I}} w(I)} + \gamma/|\mathbb{I}|, \forall I \in \mathbb{I}$
8      $I_c \leftarrow$ Draw randomly according to distribution $p$
9      response $\leftarrow$ Query the user using $I_c$
10     Update $r(I_c)$
11     if response $\neq$ Skip then
12         reward, $\mathbb{S} \leftarrow$ Use user response to remove irrelevant views
             from $\mathbb{S}$ and update ranking
13 return $\mathbb{S}$
```

Concretely, the probability of choosing an arm is as follows:

$$p(I) = (1-\gamma)\frac{w(I)}{\sum_{I \in \mathbb{I}} w(I)} + \frac{\gamma}{|\mathbb{I}|}$$

where $\mathbb{I}$ denotes the set of different question interfaces. γ determines the probability of exploring a random arm, while ignoring the expected reward, and $w(I)$ denotes the estimated value of the expected information gain on question interface I. Choosing $\gamma = 1$ is equivalent to an exploration strategy that chooses a random arm for every question while $\gamma = 0$ chooses an arm that relies on expected reward estimation for each question. The expected information gain $w(I)$ of an interface I is $r(I) \times \chi(I)$, where $r(I)$ is the probability that a user would answer the question with interface I, and $\chi(I)$ is the maximum reduction in candidate set size if the question using I is answered. Initially, users have not answered any questions and the estimates of expected gain are not accurate. Therefore, the approach is bootstrapped with the exploration strategy until $O(\log |\mathbb{I}|)$ questions have been asked for each interface. The estimated user behavior is then used to transform to a bandit-based approach.

Performance Guarantees. Theoretically, using Chernoff bound [26] we can show that $O(\log|\mathbb{I}|)$ questions per interface I yield an accurate estimate of the $r(I)$ with a probability of $1-\frac{1}{|\mathbb{I}|^2}$. Using prior results on the maximum coverage problem [27], greedily choosing the question that maximizes information gain is the best approximation of the optimal strategy. Therefore, the accurate reward estimation ensures effective interaction using a multi-arm bandit approach for VIEW-PRESENTATION.

B. The View Presentation Algorithm

Algorithm 2 first initializes the candidate set of views $\mathbb{S}$ (line 1) and iteratively queries the user until T iterations (lines 2–12). T is used to denote any stopping criterion, which could be when the user ends the session or a pre-defined parameter. In each iteration, the multi-arm bandit based approach estimates the expected reward of each arm to calculate the probability distribution of choosing each arm (lines 3–7). The question interface is chosen according to this distribution. After choosing the interface I_c, a question that has the maximum information gain $\chi(I)$ is asked to the user. User's response is subsequently used to either update $r(I)$ or the candidate set of views.

Ranking Views. Given a collection of questions Q to the user, an expected utility score is calculated for each view to rank them. Mathematically, the utility score of a dataset D is a weighted sum of the view's utility according to each question:

$$\sum_{Q_i \in Q} s_{Q_i} \times (P(D \text{ satisfies user needs} | Q_i = \checkmark) \cdot P(Q_i = \checkmark))$$

where $Q_i = \checkmark$ denotes that the question Q_i is answered correctly and s_{Q_i} is 1 if D is considered to satisfy user requirements by Q_i's response, -1 if D is considered irrelevant by Q_i's response and 0 otherwise. The probability P(D satisfies user needs $| Q_i = \checkmark$) is inversely proportional to the number of views that Q_i captures and the probability that a user would answer a question is used as a proxy estimate for P($Q_i = \checkmark$). Note that this score is calculated only for candidate views that are not pruned by user's responses.

V. VIEW DISTILLATION

VIEW-DISTILLATION consists of two parts: i) categorizing pairs of result views (i.e., candidate views); ii) applying a distillation strategy to reduce the number of result views.

4C Categories. Result views with the same schema are classified into the following categories:

Definition 5 (Compatible view pair): Two candidate views, V_1 and V_2, are compatible (denoted by $V_1 \equiv V_2$) if they have the same set of rows, $(V_1 \cap V_2) = V_1 = V_2$.

Definition 6 (Contained view pair): A view, V_1, contains another view, V_2, when $V_2 \subset V_1$, that is, when all rows of V_2 are contained in V_1.

A pair of views may be *Complementary* or *Contradictory*, based on their candidate keys:

Definition 7 (Candidate key): A candidate key, $\mathcal{K}(V)$, is a set of attributes in an output view, V, that uniquely identify each row in $R \in V$.

Now we define complementary and contradictory pairs.

Definition 8 (Complementary view pair): Two views, V_1 and V_2 are complementary if the two views have the same candidate key, $K(V_1) = K(V_2)$, and the views have overlapping rows $V_1 \cap V_2 > 0$ but are neither contained nor compatible.

Definition 9 (Contradictory view pair): Two views, V_1 and V_2 are contradictory if the two views have the same candidate key, $K(V_1) = K(V_2)$, and a key value yields different rows in V_1 and V_2.

Note. We categorize a pair of views as contradictory or complementary with respect to a candidate key. Therefore, views V_1 and V_2 may be contradictory with candidate key k_1 and complementary with key k_2. Our VIEW-DISTILLATION component exposes all candidate relationships for further downstream processing.

The first step in VIEW-DISTILLATION is to identify and label pairs of candidate views with one of the 4C categories. Then, a distillation strategy automatically prunes views based on their 4C category. Views are nodes in a graph. An edge is labeled with the category of the nodes it links. More formally:

Problem 3: Given a collection of views $\mathbb{V}$, identify a labelled graph G where the irrelevant views are pruned and edges that can be categorized as 4C are labelled accordingly.

Algorithm 3: View Distillation

```
Input  : $\mathbb{V}$, collection of views
Output: $G$, graph with edges categorized as 4C
1  $G \leftarrow$ ADD-NODES($\mathbb{V}$), $\mathbb{V} \leftarrow$ IDENTIFY-KEYS($\mathbb{V}$)
2  $\mathbb{V} \leftarrow$ SCHEMA-BASED-BLOCKS($\mathbb{V}$)
3  for $\mathcal{V} \in \mathbb{V}$ do
4      for $1 \leq i, j \leq |\mathcal{V}|, i > j$ do
5          $V_i \leftarrow \mathcal{V}[i], V_j \leftarrow \mathcal{V}[j]$
           /* Iterate over the views, compare rowwise hashes $H$ */
6          if $H[V_i] = H[V_j]$ then
7              $G[(V_i, V_j)] =$ Compatible
8              $\mathcal{V} \leftarrow \mathcal{V} \setminus \{V_i\}$
9          else if $H[V_i] \subset H[V_j]$ or $H[V_j] \subset H[V_i]$ then
10             $G[(V_i, V_j)] =$ Contained
11             $\mathcal{V} \leftarrow \mathcal{V} \setminus \{V_i, V_j\} \cup \{V_i \cup V_j\}$
12         else if $H[V_i] \cap H[V_j] \neq \phi$ then
               /* Initialize any overlapping view pair as complementary */
13             $G[(V_i, V_j)] =$ Complementary
       /* Second phase: Identify contradictions */
14     $I \leftarrow$ INDEX($\mathcal{V}$)
15     for $k \in I.keys()$ do
16         $\mathcal{C} \leftarrow$ GROUP($I[k]$)
           /* Identify pairs in different groups */
17         for $(V_i, V_j) : V_i, V_j \in I[k],\ \mathcal{C}[V_i] \neq \mathcal{C}[V_j]$ do
18             $G[(V_i, V_j)] =$ Contradictory
```

`Ver`'s VIEW-DISTILLATION implementation uses the following insights to classify views into 4C categories and construct the graph G. First, it uses the transitivity property to not compare any pair of views whose categorization can be inferred from prior comparisons. For compatibility, if $V_1 \equiv V_2$

and $V_2 \equiv V_3$, then $V_1 \equiv V_3$. And for containment, **`Ver`** maintains the largest view for categorization, i.e. if $V_1 \subseteq V_2$, then **`Ver`** distills out V_1 and keeps V_2 as V_1's representative. Second, it partitions candidate views into SCHEMA-BASED-BLOCKS. This ensures that pairs of views are compared only if they share the same schema. Third, it hashes each view using a row-wise hash function (say H), i.e. $H(V)$ maps V to a set of values where each value corresponds to a different row. This hash map helps to efficiently find compatible and contained pairs of views. Fourth, it identifies approximate key columns [28], [29] and constructs an inverted index that maps each value in a key column to the corresponding rows and views that contain that value. This index helps to identify rows that have contradictory values and hence contradictory views.

Algorithm 3 presents the pseudocode of **`Ver`**'s VIEW-DISTILLATION. First, it initializes a graph G where each view is added as a node, identifies keys in each view (line 1) and partitions the collection of views $\mathbb{V}$ into different blocks based on their schema (line 2). These schema-based blocks are processed sequentially to populate G with 4C categories (line 3). The categorization process operates in two phases. The first phase (line 4-13) hashes all views (hash value of a view V is denoted by $H(V)$) and compares hashed values to check containment and compatibility (line 6-11). A pair of views V_i and V_j that overlap and have the same key but are not contained or compatible are marked as complementary (line 12-13). These pairs are later updated to be contradictory if the second phase identifies any contradictions. All previously described comparisons are performed on the hash of each view. The hash function maps each view to a set of values, where each value in the set corresponds to a row in the view and we employ a cache to not hash any view multiple times. The second phase constructs an inverted index over the values in the key column(s) of each view in $\mathcal{V}$. This index maps each key value (say k) to a list of rows that contain the value k (denoted by $I[k]$). **`Ver`** iterates over all values of the key column and identifies contradictions among the rows that contain the value (line 16-18). Specifically, it groups all duplicate rows that contain a key value k together (line 16) and pairs of views not in the same group are labeled contradictory (line 17-18).

Distillation Strategy. Algorithm 3 merges distillation with graph construction. It applies a distillation strategy that deduplicates compatible views and keeps the largest contained view. Alternative strategies can be implemented based on the target use. This strategy helps reduce the search space of views that VIEW-PRESENTATION component needs to consider.

Complexity Analysis. A crucial step of Algorithm 3 is hashing, which requires $O(n)$ time, where n is the total number of candidate views. Other than that, **`Ver`** partitions the set of views into different schema blocks and compares views within a block. In the worst case, it may compare hashes of all pairs of non-compatible views within a block to check containment, i.e. requiring total complexity of $O(n+\alpha\Gamma^2)$, where α denotes the number of distinct schemas and Γ denotes the maximum number of distinct views sharing the same schema. However, the distillation property of keeping the largest contained view helps reduce complexity in practical scenarios (median reduction ratio of more than 18%). For contradiction and complementary categorization, calculation of key is the most time-consuming step, which requires processing all views. The subsequent steps of constructing the inverted index and processing each key value in the index are relatively efficient. Consider a key value k which is present in t different rows, out of which γ are distinct values that contradict each other. The complexity of **`Ver`**'s grouping approach to process the key k in the inverted index takes $O(t)$ running time to identify all contradictions involving k. Therefore, this step has complexity linear in the number of contradictions.

VI. EVALUATION

In this section, we answer these research questions:

- **RQ1**: Is **`Ver`** effective in navigating users to the view that satisfies their requirements? (human problem)
- **RQ2**: Is VIEW-DISTILLATION useful for reducing the view choice space and is VIEW-DISTILLATION scalable? (technical problem)
- **RQ3**: End-to-end evaluation of **`Ver`** (technical problem)
- **Qualitative Study (QS).** We discuss qualitative differences with QBE systems.

In Appendix. C, we include several **Microbenchmarks** where we explore the effect of various query and data parameters on **`Ver`**'s performance.

Datasets and Workload. We use three real-world large-scale datasets in the evaluation. Detailed statistics of these three datasets are shown in Table I.

Dataset	#Tables	#Columns	∼# Joinable Columns	∼ Total #Rows	Size
ChEMBL	70	446	435	140M	6.5GB
WDC	10000	39939	11.6M	140K	45MB
Open Data	69407	2955305	28.6M	900M	119GB

TABLE I: Characteristics of Datasets

- **ChEMBL**: ChEMBL [30] is a database of bioactive molecules with drug-like properties. ChEMBL is large in terms of total data size. However, it has a relatively small number of tables and joinable columns.
- **WDC**: WDC is a subset of the web tables corpus [31] containing 10K tables crawled from the web. It has more than 10 million pairs of joinable columns.
- **Open Data** [32] This dataset consists of 69K open datasets collected from the Open Data Portal Watch [33], [34] which catalogs and monitors 262 open data portals such as NYC Open data, finances.worldbank.org, etc.

System Setup. We ran all experiments on a Ubuntu server with 500GB memory and an Intel(R) Xeon(R) CPU with 48 cores and 2.3GHz speed each. We built **`Ver`** using python3.6. **`Ver`** uses Aurum to find join paths without using schema information. In **ChemBL**, we ignore the schema information to simulate pathless scenario and instead use schema to evaluate ground truth. When searching for join graphs, **Ver** uses by default a maximum of two hops, $\rho = 2$. We set the clustering threshold θ to 1, and the expected number of output views k to be the total number of join graphs so we materialize all join graphs generated from JOIN-GRAPH-SEARCH.

*A. RQ1: Is **Ver** effective in navigating users to the view that satisfies their requirements?*

We conducted a within-subjects user study to answer this question. We give participants a task and expose them to two systems: VIEW-PRESENTATION as explained in this paper, and a ranking of views as produced by overlap-based ranking mechanism of FASTTOPK [35]. Their goal is to identify a view that satisfies the task.

Participants. We recruited 18 students with diverse backgrounds (CS, Economics, Math) from the University. We did not record any personally identifiable information.

Study Procedure. We design the study to ensure internal validity. Each participant attends a 30-min training session to learn the interface design. During the session, we describe the study, give a tutorial on each interface, and ask participants to solve two randomly-chosen *trial* queries using **Ver** and FASTTOPK. The goal of the trial task is to familiarize participants with the interfaces. After finishing the trial tasks, each participant solves two different queries with **Ver** and FASTTOPK, respectively. The order in which the participant uses **Ver** and FASTTOPK is randomized to avoid ordering and learning effects. Participants work in isolation to avoid biasing each other. After finishing the tasks, participants answer a short survey about their experience with both systems.

Task Setup. Participants are exposed to 4 (2 trial, 2 study tasks) of the 5 queries shown in Table II from WDC [31] dataset. We chose a diverse set of queries involving numerical and textual attributes that generated semantically ambiguous results.

Query	Example	**Ver** #Views	FASTTOPK #Views
Find views containing IATA code of airports in any of these states in the US.	Indiana, Georgia, Virginia, Illinois, Connecticut	397	2255
Find views containing churches in any of these states in the US	Indiana, Georgia, Virginia, Illinois, Connecticut	397	2255
Find views containing newspaper companies in any of these cities.	San Diego, Boston, Philadelphia	394	838
Find views containing population of any of these countries.	China, Japan, United States	566	2235
Find views containing the number of births per 1000 population in any of these countries.	China, Japan, United States	566	2235

TABLE II: Tasks used in the user study and # Views generated by **Ver** / FASTTOPK.

We manually verified each participant's output and record whether they found a relevant view that answers the query.

Interface Setup. We setup **Ver**'s VIEW-PRESENTATION with different types of question interfaces. The interface asks users if they want to include a specific attribute, collection of attributes or an individual dataset (as discussed in Section IV). We use two different prioritization strategies for each interface: one based on the distance of the question from the input query and other based on the distance of the datasets corresponding to the questions from the input query. We use pre-trained word2vec embeddings to calculate distance. In each interaction, the user can either skip or answer the question or explore the ranking of views to select one view. The scoring model we adopted in FASTTOPK presents a ranking of views allowing the user to manually explore the options and pick the one that satisfies the input query.

Data Collection and Results. We log the interactions of each participant with the system for subsequent analysis. We measured interactions and outcomes to answer the following questions (Table III presents the study results):
Q1. Does the user find the relevant view? 16/18 participants identify the correct view with **Ver** versus only 6 when using the FASTTOPK ranking. 12 participants finished the task without finding any dataset using FASTTOPK versus only 1 with **Ver**. The results are statistically significant: we run Fisher's exact test and obtain a p-value of 0.002. Due to this result, we confirm the sample size is adequate for this study.
Q2. Which system would you prefer to search datasets? 12 participants prefer searching for datasets with **Ver** and 5 prefer FASTTOPK (1 participant was not sure).
Q3. If you are to forward the query and the dataset you chose using **Ver** same question for FASTTOPK) to someone else. How confident are you to share the identified search result for the input query? 14 participants were confident with the result they found with **Ver**. We cannot measure confidence for FASTTOPK because 12 participants did not find any view.
Q4. How difficult is to use **Ver**? and Q5. How difficult is to answer multiple choice questions with **Ver**? 14 participants deemed using **Ver** easy and intuitive and 4 disagreed. Anecdotally, some participants mentioned that questions asked by **Ver** are easy to answer and do not require in-depth analysis. Our discussion with the participants revealed that different users preferred different interface designs. For example, some students verified the attribute names before choosing a view while others verified a sample of the records.
Time taken The median participant using **Ver** finds the view within 101 seconds (median) and with a median of 3 interactions. They take 93 seconds (median) when using FASTTOPK.

B. RQ2: Does VIEW DISTILLATION *reduce the number of views and is* VIEW-DISTILLATION *scalable?*

We evaluate the effectiveness in reducing result views and the scalability of VIEW-DISTILLATION.

Noisy Query Generation. Each query consists of a collection of 2-column, 3-row example values. To generate the query, we first find a PJ-query that produces a result we call the *ground truth PJ-view*. Columns in the *ground truth PJ-view*

Q1. Does the user find a relevant view?		
	Ver	FastTopK
Found	**16***	6
Not Found	2	12
*Result is statistically significant with p-value of 0.002		

Q2. Which system would you prefer to search datasets?		
Ver	FastTopK	Unsure
12	5	1

Q3. Confidence in the identified search result		
	Ver	FastTopK
Confident	**14**	6
Not Confident	4	8

Q4. How difficult is to use Ver?	
Intuitive	Not Intuitive
14	4

Q5. How difficult is to answer multiple choice questions with Ver?	
Easy	Difficult
16	2

TABLE III: Summary of survey results.

Query	Noise level	Original	C_1	C_2	C_3 worst case	C_3 best case
ChEMBL Q1	Zero	38	36	36	20	20
	Med	20	18	18	8	8
	High	33	31	31	21	21
ChEMBL Q2	Zero	59	58	54	51	47
	Med	32	32	30	30	29
	High	41	38	35	32	30
ChEMBL Q3	Zero	58	33	29	23	23
	Med	44	21	17	12	14
	High	44	21	17	14	14
ChEMBL Q4	Zero	23	17	14	14	14
	Med	83	74	68	62	59
	High	83	74	68	62	59
ChEMBL Q5	Zero	24	18	15	15	15
	Med	64	57	51	46	46
	High	33	23	20	20	20
WDC Q2	Zero	44	39	21	8	6
	Med	42	37	19	5	3
	High	39	34	15	6	5
WDC Q3	Zero	20	20	20	20	4
	Med	15	15	15	15	3

TABLE IV: Effect of view distillation based on 4C signals on number of view. We excluded queries that have less than 10 original number of views.

are called *ground truth columns*. Then, we generate the 2×3 input queries according to three strategies, *Zero Noise*, *Medium Noise*, and *High Noise*. *Zero noise* means we sample values from the *ground truth columns*. In *Medium noise* we sample $\frac{2}{3}$ values from the *ground truth columns* and $\frac{1}{3}$ from a *noise column*, which is a column with a Jaccard Containment of more than 0.8 with respect to the *ground truth column*. Finally, in *High noise* we sample $\frac{1}{3}$ values from the *ground truth column* and $\frac{2}{3}$ from the *noise columns*.

We generate 5 ground truth queries by sampling join graphs from the ground truth views of ChEMBL and WDC. For each ground truth query, we generate one noisy user query consisting of example values for each of the 3 noise levels. For each noisy user query, we obtain input PJ-views to View-Distillation by getting candidate columns via column-selection component and feeding them to Join-Graph-Search and Materializer.

*1) **Compatible and Contained (C1 and C2):*** Each group of compatible views is reduced to a single view. When views are contained, we keep the larger one. The C_1 and C_2 columns in Table IV show the number of views left after pruning compatible views and contained views, respectively.

Insights of C1. Around 50% of candidate PJ-views in Q3 of ChEMBL are compatible because many tables have more than one candidate key. For example, one pair of compatible views are being materialized using the same join tables: *assays* and *cell_dictionary*, but one view's join key is *cell_name* and the other's join key is *cell_description*; since there is a one-to-one mapping between *cell_name* and *cell_description*, the views they produced are identical.

Insights of C2. Q2 of WDC pruned 18 contained views for zero/medium and 19 for high noise level. The majority of the output views in the output share the same attributes: *State* and *Newspaper Title*. We found that each pair of contained views are joined using different tables but the same join key, *State*. The join key values of one join path are subsumed by the join key values of other join paths, thus the resulting view is contained in another view.

*2) **Complementary (C3):*** We union complementary views. The complementarity of views depends on their key. In this experiment, we consider the key that leads to the least reduction (worst case column) and the largest reduction (best key column). Table IV shows that unioning tables reduces the number of views in most queries. In the worst case, tables do not union with each other, as in the case of the WDC Q3.

Insights of C3. The reason WDC Q2 can union many complementary views even in the worst case is that all candidate PJ-views that share the same list of attributes, *State* and *Newspaper Title*, are joined by two tables using the join key *State*; one join table containing the attribute *Newspaper Title* is the same for all the views, while the other containing the attribute *State* is different for each view. The join tables that are different have different coverage of *State* values. Therefore, a lot of candidate PJ-views are complementary based on the candidate key *State* (the worst case).

For ChEMBL queries, typically one pair of views does not share their contradictory rows with any other pair of views, so no matter which candidate key we choose, it can always lead to unionable complementary views since the contradictory relationships are not transitive across views. For some queries such as Q5, many views do not have valid candidate keys, so there are no unionable views.

*3) **Contradictory (C4):*** We construct contradictions from contradictory view pairs by grouping all views that share the same contradiction together. Given a contradiction, we do not have an automated way of choosing a view. However, we calculate the value of pruning by measuring the worst case and best case reduction in the candidate set size. We sort contradictions in descending order according to their degree of discrimination–the number of views that agree with one side of the contradiction. Then, we select the contradictions sequentially and consider two cases: (a) where the selection

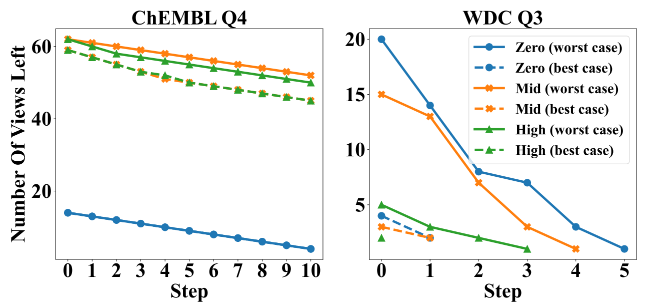


Fig. 2: Number of views left at each step after pruning views.

leads to the largest reduction of the views (best-case) and (b) where it leads to the least reduction of the views (worst-case).

Fig. 2 shows the number of remaining views after each step (for a maximum of 10 steps) for a selection of queries that present discriminative and non-discriminative contradictions. As expected, when contradictions are not discriminative, the reduction is limited, such as in mid/high noise queries of ChEMBL Q4 in the worst case. However, there are cases where contradictions are quite discriminative and the signal proves effective in reducing the set size, such as in the worst case queries of WDC Q3.

Insights of C4. In ChEMBL, the output of Q4 mid/high noise queries contains many candidate PJ-views that are joined by varied join tables and keys. The contradictions in candidate views are mainly due to wrong join paths. For example, one view that is partly joined by the two tables *component_sequences* and *component_class* on the shared attribute *component_id*, while the other view is joined by the two tables *component_sequences* on attribute *description* and *target_dictionary* on attribute *pref_name*, when projecting the final attributes, *organism* and *pref_name*, these two views have contradictions. *component_id* is a better join key than *description*. However, as we do not know the join information of the input data, we can only perform join operations using all the attributes that are being considered as valid join keys by the discovery engine.

Moreover, since the contradictions in ChEMBL mainly arise due to different join paths, they typically do not share the same contradictions across multiple views. Each contradictory signal only contains two views. Therefore, the maximum number of views we prune at each step is 1. For WDC queries, however, we are able to prune multiple views in Q3, since the contradictory views have many shared contradictory rows, thus each contradictory signal contains many views. And since we prioritize presenting the more discriminatory contradictions first, we prune multiple views even in the worst case.

4) Scalability: In this experiment, we evaluate the scalability of VIEW-DISTILLATION using 50 randomly sampled queries from the OpenData dataset along with 3 datasets built using random samples of $25\%, 50\%$, and 75% tables from the original[2]. Figure 3 shows the distribution of the number

[2]The subsampling was performed to ensure that all datasets present in a smaller size version are also present in the larger sample.

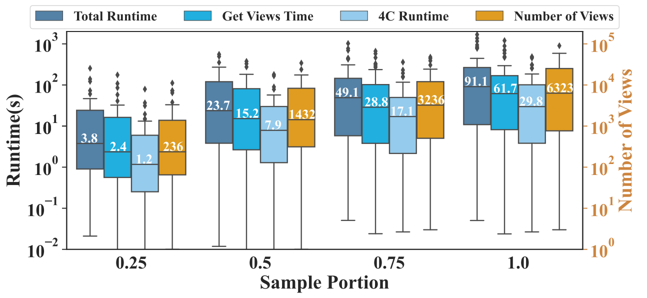
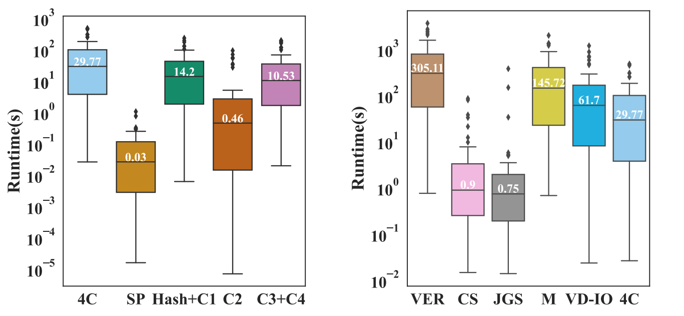


Fig. 3: Total Runtime, Get Views Time, and 4C Runtime of VIEW-DISTILLATION for each sample portion on left y-axis, and number of views on the right y-axis.



(a) 4C Runtime for different steps (b) Total Runtime of Ver over 50 queries

Fig. 4: (a) Time to execute different steps in 4C for sample portion=1 (SP=Schema Partition). (b) CS=COLUMN-SELECTION, JGS=JOIN-GRAPH-SEARCH, M=MATERIALIZER, VD-IO=Get Views Time in VIEW-DISTILLATION, 4C=4C Runtime in VIEW-DISTILLATION.

of views for each subsample (see 2nd y-axis). Note that we use boxplots to report the min, max, 25th, median, 75th, and max runtimes given the varied complexity of queries. The total runtime (y-axis) grows linearly with the number of views. Concretely, and as a proxy summary statistic, we observe that **`Ver`** takes 1.16 seconds to calculate 4C categories for around 236 views (median for 25% sample size) and around 30 seconds for 6500 views (median of 100% sample size); we observe similar growth when comparing other percentiles. The algorithm's scalability is limited by the time to read views from disk, which grows with the total amount of data. The total '4C Runtime' is small in comparison.

Figure 4(a) zooms into the time taken by 4C ('4C Runtime' in Figure 3) to understand the impact of different parts of Algorithm 3. The schema group classification and identification of contained views are efficient and require less than 0.5 seconds. Hashing dominates runtime as it needs to hash each row in the set of candidate views. The time to find contradictory or complementary views involves two main steps: i) find candidate keys for each view, which is linear in the number of total rows; ii) find the actual contradictions based on the inverted index, which is linear in the number of contradictions. When the number of contradictions is small the time to find candidate keys dominates.

Effectiveness of View Distillation. In this experiment, we evaluated the reduction ratio (fraction of view pruned) of merging compatible and contained views for the 100% sample size. We observed that 50% of the queries had a reduction ratio of more than 17.5%, while 25% had a reduction ratio of 63%. This demonstrates the effectiveness of VIEW-DISTILLATION to efficiently prune contained and compatible views.

C. *End-to-End Evaluation*

We first present an end-to-end experiment of Ver using different implementations of VIEW-SPECIFICATION. We then study the QBE implementation in more detail to understand the effect of different baselines for different components.

1) Alternative View Specification Implementations: In this experiment, we implement 3 view specification methods i) QBE (Ver's default); ii) Keyword search; and iii) Attribute search. We use 10 randomly chosen queries from OpenData. Every query runs within 11 minutes (for ≈ 27K views) with QBE interface, 13 minutes (≈ 500 views) for keyword interface and 30 minutes (≈ 1000 views) for attribute interface. The views generated by keyword and attribute interfaces contain a large number of columns as compared to QBE, contributing to higher running time for these implementations. We further run VIEW-DISTILLATION to merge compatible and contained views, followed by VIEW-PRESENTATION with a simulated user. We simulated the user to answer questions correctly. We observe that the user identified the ground truth view in as few as 20 queries for around 500 views and less than 100 queries for 3000 views. This evaluation demonstrates the effectiveness of our VIEW-DISTILLATION and PRESENTATION to effectively prune the search space and help the user identify relevant views. In terms of runtime, VIEW-PRESENTATION produces questions for users in less than 10^{-3} seconds.

We now dive deeper into the QBE implementation to understand the intricacies of our implementation.

2) Runtime Comparison: In this experiment, we report the distribution of runtimes for the sample of 50 queries used to evaluate 4C's scalability.

Figure 4 (b) shows that the total runtime is below 305.11 seconds for 50% of the queries. The bottleneck is the MATERIALIZER and the time taken to read views from the disk, which require 145 and 62 seconds for 50% of the queries, respectively. MATERIALIZER's runtime is linear with the number of join graphs generated for a query. We use pandas library in Python to materialize the join and read the view from disk, which could be optimized by using a database.

The median runtime of COLUMN-SELECTION and JOIN-GRAPH-SEARCH is less than 1 second. There are fewer than 3 outlier queries which take more than 100 seconds for COLUMN-SELECTION because these queries are too general (numerical values without semantic meaning) and the input query is present in more than $100K$ datasets. We do not report the time taken by VIEW-PRESENTATION as it requires human-in-the-loop. However, we observe that the median time taken

Ground Truth Hit Ratio								
Zero Noise			Mid Noise			High Noise		
SA	SB	CS	SA	SB	CS	SA	SB	CS
1.0	1.0	1.0	1.0	0.08	1.0	1.0	0.02	0.96

TABLE V: Ground truth hit ratio over 150 queries in input workload split by noise level in the input query (SA: Select-All, SB: Select-Best, CS: Column-Selection).

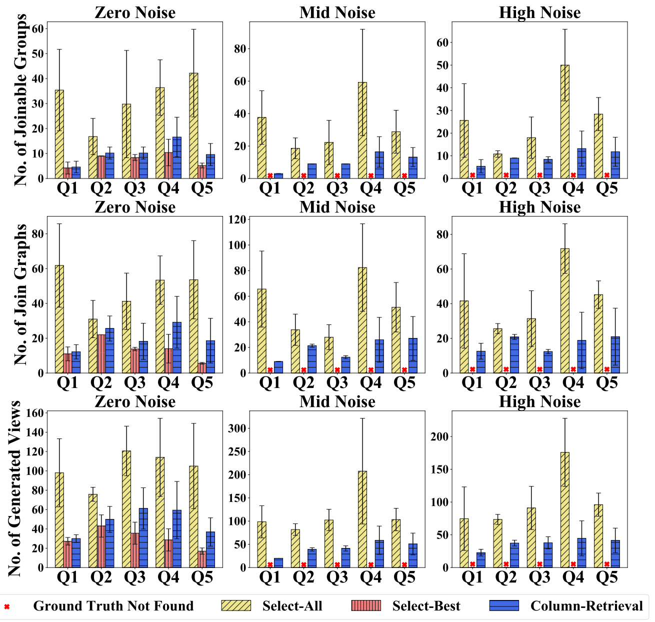


Fig. 5: #joinable groups, join graphs and views on ChEMBL

to initialize the component is 73 seconds but it takes less than 0.5 millisecond per question. Low latency of asking a question helps to ensure interactive performance of **`Ver`**.

3) RQ3: Column Selection and Join Graph Search: In this section we ask: do COLUMN-SELECTION and JOIN-GRAPH-SEARCH find relevant PJ-views given a noisy input query over pathless table collections?

Workload. We generate noisy user queries as described in Section VI-B. We generate 5 noisy user queries consisting of example values for each ground truth view and for each of the 3 noise levels. This results in a total of 150 noisy user queries across both datasets and noise levels.

Baselines. We compare the COLUMN-SELECTION component in **`Ver`** with two other baselines:

- SELECT-ALL. This baseline (implemented from FAST-TOPK [35]) selects any column that contains at least an example from the input query.
- SELECT-BEST. This baseline selects the column that contains the highest number of examples from the input query. This is the selection strategy implemented by *SQuID* [36].

After running the baselines, we feed the returned candidate columns to the JOIN-GRAPH-SEARCH component that finds joinable groups of tables identified in the **Join Graph Enumeration** stage, identifies join graphs, and materializes them to produce the set of candidate PJ-views.

Experiment and Metrics. We obtain the output set of can-

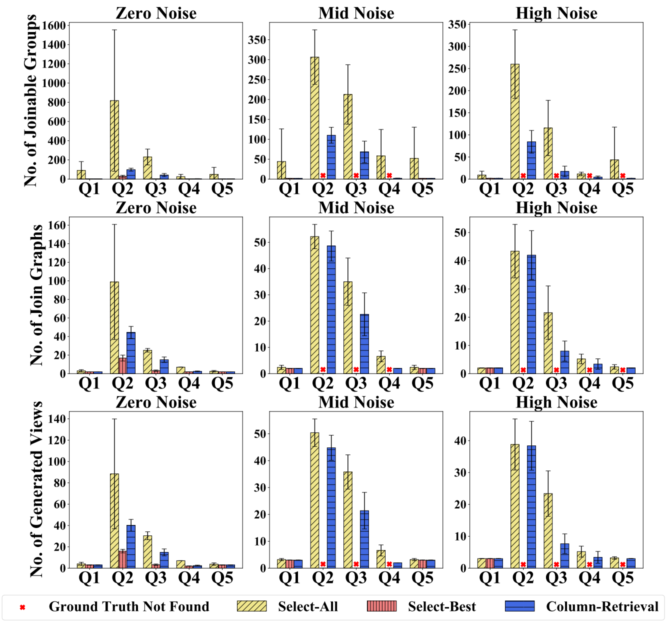


Fig. 6: #joinable groups, join graphs and views on WDC

didate PJ-views for each of the 150 input user queries in the workload as described in **Noisy Workload Generation**. We consider two metrics. First, whether the *ground truth view* is part of the candidate PJ-views, i.e., the system finds the required view. In particular, we measure the *Ground Truth Hit Ratio* that determines the ratio of input queries for which the system finds the *ground truth view*. Second, we measure the size of the set of candidate PJ-views. Given two systems that find the *ground truth view*, we prefer the one producing the smaller set of candidate PJ-views. Smaller candidate PJ-views indicate lower runtime (as we will demonstrate later) and more importantly, facilitates the job of the VIEW-PRESENTATION stage, e.g., consider a human who needs to look at each view in the candidate set to select the right one.

Results. Table V shows the *Ground Truth Hit Ratio* across the 150 queries, for each baseline, and grouped by different input query noise levels in the X-axis. When there is no noise in the input query, then all baselines perform well and find the *ground truth view*. As the noise in the input query increases, the SELECT-BEST strategy crumbles because of its over-reliance on columns that contain all values in the input query. This demonstrates that when input queries contain noise, the SELECT-BEST strategy is inadequate.

With both SELECT-ALL and COLUMN-SELECTION consistently finding the *ground truth view*, the next question is at what cost. Fig. 5 and Fig. 6 show results for ChEMBL and WDC respectively. Each 3×3 grid shows the size of the set of candidate PJ-views at the top for the three noise levels. The results clearly indicate that for all queries across both datasets and noise levels, the set of the candidate PJ-views is always significantly larger in the case of SELECT-ALL than in the case of COLUMN-SELECTION. Since both baselines find the *ground truth view*, the smaller sets are preferred.

The SELECT-ALL retrieval strategy selects many columns

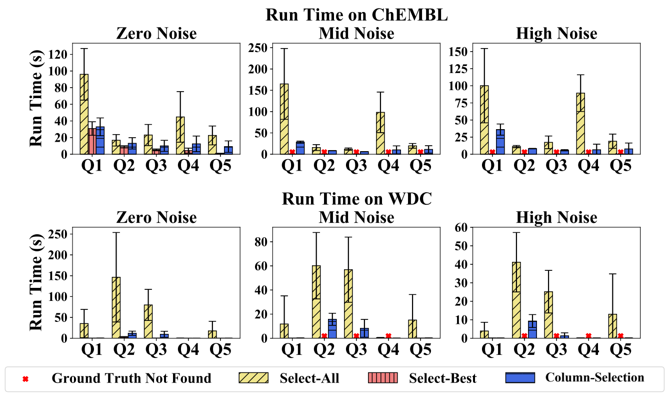


Fig. 7: Run time of COLUMN-SELECTION + JOIN-GRAPH-SEARCH + MATERIALIZER on ChEMBL and WDC

and produces much larger joinable groups than necessary (see *No. of Joinable Groups* in Fig. 5 and Fig. 6). Larger joinable groups, in turn, lead to a larger number of join graphs–sometimes up to $4\times$ more, see *No. of Join Graphs* in Fig. 5 and Fig. 6)–which results in more candidate PJ-views. Fig. 7 shows that larger sets of candidate PJ-views leads to higher runtime. The runtime of COLUMN-SELECTION + JOIN-GRAPH-SEARCH is an order of magnitude lower than the SELECT-ALL strategy.

D. QS: Analysis of existing QBE systems

SQuID [36] does not scale to pathless table scenarios. *SQuID* precomputes an abduction-ready database (αDB) that requires human input to select the pairs of table key and attributes of interest from other tables. Without human input the number of combinations grows large, especially given that in pathless table collections some join path information will be wrong. Given that the size of αDB can be as large as the original table—for example, *component_sequences* of ChEMBL is 5.9M and results in an αDB size of 8.1M—the storage footprint would multiply. Therefore, we were only able to test *SQuID* on a dataset containing a handful of tables. Without a deep understanding of the input dataset—which we lack in pathless table collections—we can only provide limited information to the system to compute the αDB, thus resulting in poor query performance. A more reasonable use of *SQuID* is to employ it as a downstream component to **`Ver`**, where the input is a narrowed-down version of candidate PJ-views.

Duoquest [38] lets users specify a natural language query (NLQ) along an input table containing example tuples and additional information about the attributes (Table Sketch Query). It outputs candidate SQL queries ranked from highest to lowest confidence based on the user's input queries. Navigating that ranking presents challenges to the user in pathless table scenarios because the candidate query may consist of incorrect join tables or keys due to noise in data. An additional presentation phase is necessary to navigate the user among the candidate queries. A more reasonable use of *Duoquest* is to employ it as an implementation of the VIEW-SPECIFICATION component, giving users alternative ways of describing their queries.

	Technique	View Specification		Column Selection	Discovery Engine	Join Graph Search	View Distillation	View Presentation
		Input Type	Handles Noise		Require PK/FK			
QBE	SQuID [36]	Relational	N	Automatic	Y	Online	N	N
	S4: FastTopK [35]	Relational	Y	Automatic	Y	Online	Individual signal	N
	MWeaver [37]	Relational	N	Automatic	Y	Online	Individual signal	N
	DuoQuest [38]	Natural language	Y	Automatic	Y	Online	Individual signal	N
QRE	TALOS [15]	N		Automatic	Y	Online	Individual signal	N
	PALEO-J [39]	Ranks tuples	N	Automatic	Y	Online	Individual signal	N
	SQLSynthesizer [40]	Relational	N	Automatic	Y	Online	Individual signal	N
	REGAL+ [41]	N		Automatic	Y	Online	N	N
Data Disc.	Aurum [5]	N		N	N, Offline index	Online	N	N
	Josie [9]	N		N	N, Offline index	Online	N	N
	TableUnion [10]	N		N	N, Offline index	Online	N	N
	Lazo [13]	N		N	N, Offline index	Online	N	N
	LSHEnsemble [12]	N		N	N, Offline index	Online	N	N
	PEXESO [42]	N		N	N, Offline index	Online	N	N
VP	Voyager [43]	N		N	N	N	N	Y
	SeeDB [23]	N		N	N	N	N	Y
	NorthStar [44]	N		N	N	N	N	Y
	RONIN [45]	N		N	N	N	N	Y
	Ver	Relational	Y	Automatic, Interactive	N, Offline index	Online	Individual, Dependent signal	Y

TABLE VI: Overview of SOTA. VP = View Presentation and Data Disc. = Data Discovery. N means the system does not implement the component. Individual signals means the approach only computes a statistic over candidate views. Dependent signals consider dependence between candidate views like 4C signals.

VII. Related Work

We use Table VI to overview the related work.

View Specification. Common specification interfaces are keywords [7], APIs, table search [8], and QBE (relational in the table). Each of these interfaces caters to different discovery needs and can be plugged into our reference architecture. Although we implemented QBE in **Ver**, we evaluated the performance of alternative view specification implementations such as keywords, and attribute-based search interface.

Data Discovery. Some discovery engines [5], [11], [8], [10], [46] identify join paths in repositories and their emphasis is often effectiveness and scalability. There are many other techniques that identify join paths using techniques such as summaries [47], etc. Most discovery engines leverage key detection techniques as a building block to identify join paths [28], [29]. Complementary work to identifying join paths is on automatically detecting transformations to expand the set of joinable columns [48], [49], [50]. Any improvement to the detection of join paths can be incorporated into the reference architecture presented here. The 'Discovery Engine' column shows what techniques depend on existing join paths and how the navigation is done (column 'Join Graph Search').

View Distillation. Ranking is a well-explored topic in data management and many approaches naturally leverage this technique to sort candidate results, see column 'View Distillation' in the table, where Individual Signal refers to the ranking score. Orthogonally, view summarization [51], and automated data exploration techniques concentrate in sorting through data and offer an alternative way of navigating the data, like view distillation. Unlike distillation, none of these techniques leverage 4C categories.

View Presentation. There is related work on designing effective visualizations to assist users in exploring pathless data collections [32], [52], [45] and view recommendations [53], [54]. These techniques can be useful to implement the human components of **Ver**. Table VI summarizes some techniques.

Applying QBE to Pathless Table Collections. We present the prior QBE systems in Table VI. Other QBE techniques [37], [55], [35], [56], [15], [57], [40], [58], [59], [39], [60], [61], [62] are not a system, are not designed to handle pathless scenarios and do not focus on **Challenges 4 and 5**.

Supports Pathless? Existing QBE [36], [38], [35], [55], [37] and Query Reverse Engineering (QRE) systems [59], [63], [15], [57] are designed for databases with well-defined path information, i.e., primary key/foreign key relationships, and they will generate spurious results when executed over noisy join paths. Bonifati et al. [56] learns join predicates without assuming the existence of join paths. But the approach requires performing a Cartesian product on relevant tables, which introduces a scalability challenge even in moderate size databases. Other techniques that use QBE interface, either assume a different data format, such as knowledge bases [64], or consider a different view specification interface, such as exemplar queries in [65], [66], [67], [68] that define a more general notion of queries than query-by-example.

VIII. Conclusions

We presented a reference architecture for the discovery of PJ-views over pathless table collections, and a system, **Ver**, that addresses both technical and human problems of view discovery. **Ver** efficiently addresses the challenges of large-scale pathless table collections by combining different components, including VIEW-DISTILLATION and VIEW-PRESENTATION.

IX. Acknowledgement

We thank the Chameleon cloud platform [69] for providing the resources necessary to conduct this research. We thank all participants of our user study and the anonymous reviewers for helping us improve the work. This work was partially supported by the NSF Convergence Accelerator (Award Number #2040718) and NSF grant #2030859.

APPENDIX

A. Discovery Index Construction

Ver leverages Aurum [5] to efficiently build a discovery index over large collections of data. **Ver** uses the following functions provided by Aurum:

SEARCH-KEYWORD(TARGET, FUZZY). Given an input string, it returns columns that contain the string in either the attribute name, or in the values, as specified by TARGET. The match can be exact or fuzzy, after specifying a maximum Levenshtein distance.

NEIGHBORS(THRESHOLD). Given an input column, it returns all neighbors with a Jaccard containment [13] above the input THRESHOLD.

GENERATE-JOIN-GRAPHS(TABLES, ρ). Given a set of tables, it returns all join graphs connecting input tables, via inclusion dependencies. Each edge in the join graph has a maximum number of hops, ρ.

B. Column Selection

COLUMN-SELECTION generates a set of columns $\text{CAND}(A_i)$ given user-provided input examples, $\chi.A_i$. To deal with noisy inputs, the component clusters candidate columns and returns clusters with top-θ scores. Setting $\theta = 1$ returns clusters with the highest overlap with $\chi.A_i$ (there could be ties). In contrast, $\theta = \infty$ returns any column with non-empty overlap. This relaxed design makes the component more robust to noisy inputs than methods based on *exact containment* [36], [38], and more efficient than methods based on *non-empty containment*.

Algorithm 4: COLUMN-SELECTION

```
Input  : Example value set $\chi.A_i$ for a column $A_i$, Discovery Index
         $I$, Clustering threshold $\theta$
Output: $\text{CAND}(A_i)$ Candidate columns and scores.
1  $\text{CAND}(A_i) \leftarrow \phi$
2  for $e \in \chi.A_i$ do
3      $\text{CAND}(e) \leftarrow \text{SEARCH-KEYWORD}(e)$
4      $\text{CAND}(A_i) \leftarrow \text{CAND}(A_i) \cup \text{CAND}(e)$
5  $\mathcal{C} \leftarrow \text{CONNECTED-COMPONENT}(\text{CAND}(A_i), I)$
6  for $Cluster \in \mathcal{C}$ do
7      $\text{SCORE}(Cluster) = \max_{col \in Cluster}(|col \cap \chi.A_i|)$
8  $\mathcal{C}' \leftarrow$ top-$\theta$ clusters in $\mathcal{C}$ based on SCORE
9  $\text{CAND}(A_i) \leftarrow \bigcup_{T \in \mathcal{C}'} T$
10 return $\text{CAND}(A_i)$
```

Algorithm 4 presents the COLUMN-SELECTION algorithm. First, it identifies all columns that have a non-empty overlap with the input examples (lines 2-4). These columns are then clustered by finding connected components over the hypergraph constructed by the DISCOVERY ENGINE (line 5). To identify the connected components, it uses the discovery NEIGHBORS function. Each cluster is assigned a score that corresponds to the maximum number of examples contained in any column in the cluster (line 7). This stage supports both *automatic mode* and *interactive mode*. In the *automatic mode*, top-θ clusters are selected based on their scores (line 8).

Rationale. Isolating this component in the architecture has two advantages: i) it detects ill-specified queries that lead to too many unnecessary retrieved columns; ii) it offers a point of interaction with VIEW-SPECIFICATION to help users select the right column clusters. Identifying interactive policies to present clusters is outside the scope of this work; we focus on providing the mechanism.

C. Join Graph Search and Materializer

The JOIN-GRAPH-SEARCH and MATERIALIZER components construct candidate PJ-views by joining the tables returned by COLUMN-SELECTION. The pseudocode in Algorithm 5 operates in two steps.

1) Join Graph Enumeration (lines 1-10) enumerates all possible combinations of the columns returned by COLUMN-SELECTION, identifies joinable groups of their corresponding tables and find all join graphs in a joinable group. A join graph consists of a group of columns connected via join paths. Given a pair, c_i, c_j, of non-joinable columns, no combination of columns involving c_i or c_j can be joined. Non-joinable pairs are cached to skip computation.

2) Ranking and Materialization (lines 11-12) materializes the top-k views as ranked according to the discovery engine score. The discovery engine ranks views according to how well join graphs approximate PK/FK, and according to the size of the join graph; smaller graphs rank higher. Views are materialized incrementally via a materializer built on top of Pandas [70]; it can be adapted to use an embedded database [71], [72] or a processing engine with the *external table* feature [73] to achieve better performance.

Algorithm 5: JOIN-GRAPH-SEARCH

```
Input  : QBE-style query $\chi$, candidate columns CAND, expected
         number of output views $k$
Output: $\mathcal{V}_{PJ}$: list of candidate PJ-views
/* Step 1: Join Graph Enumeration                    */
1  $\tau \leftarrow |\text{COLUMNS}(\chi)|$
2  $\Gamma \leftarrow \{(c_1, \ldots, c_\tau) : \forall c_i \in \text{CAND}(A_i)\}$
3  $\Gamma_{join} \leftarrow \phi$
4  for $r \equiv (c_1, \ldots, c_\tau) \in \Gamma$ do
       /* Get join graphs between source
          tables of $r$ with $\rho = 2$                   */
5      $P \leftarrow \text{GENERATE-JOIN-GRAPHS}(r.tables(), 2)$
6      if $\exists c_i, c_j \in r \mid P(c_i, c_j) = \phi$ then
7          Remove all candidates $r \in \Gamma$ containing columns $c_i$ and $c_j$
             such that $P(c_i, c_j) = \phi$
8          Continue
9      else
10         $\Gamma_{join} \leftarrow \Gamma_{join} \cup P$
/* Step 2: Ranking and Materialization              */
11 $\Gamma'_{join} \leftarrow$ Select top $k$ join candidates based on join score generated
     by the discovery engine.
12 $\mathcal{V}_{PJ} \leftarrow \text{MATERIALIZE-VIEWS}(\Gamma'_{join})$
13 return $\mathcal{V}_{PJ}$
```

D. Microbenchmarks

In this section, we explore the effect of various query and data parameters on **Ver**'s performance.

*1) **Varying Discovery Index Quality**:* The quality of the discovery indices built by the Discovery Engine has an effect on the set of candidate PJ-views generated by **Ver** (**Challenge 2**). We study its effect by modifying the default threshold Aurum uses to compute the hypergraph. In particular, we use thresholds 0.8 (the default), 0.7, 0.6, and 0.5. These thresholds

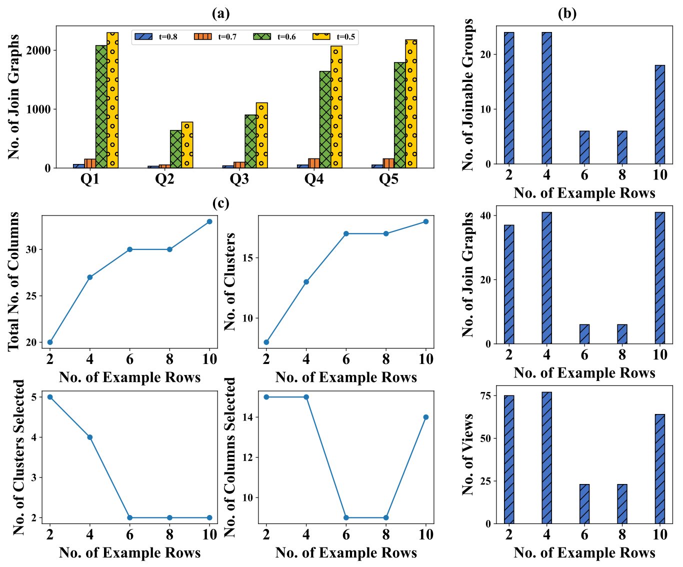


Fig. 8: **(a)** #join graphs under different t on ChEMBL. Aurum produces 435 (0.8), 582 (0.7), 2143 (0.6), and 2947 (0.5) joinable column pairs. **(b)** #joinable groups, join graphs, views of different sample size queries **(c)** #columns, clusters, selected clusters and selected columns of different sample size queries

lead to 435, 582, 2143, and 2947 joinable column pairs, respectively. As the thresholds lower, the schema quality worsens, leading to more spurious join paths. These configurations evaluate the effect of schema quality on the discovery of PJ-views over pathless table collections, i.e., **Challenge 2**. As expected, Fig. 8(a) shows that the number of join graphs increase as the schema quality worsens. As shown in the previous section, this leads to a higher number of join graphs, and consequently, higher runtimes.

Conclusion. When no join path information is available in the schema, inferring the join paths automatically leads to noisy and incorrect ones, which in turn, has an effect on the scalability of the problem (**Challenge 2**). In conclusion, i) discovering PJ-views on pathless table collections is strictly more difficult than on settings with perfect schemas and; ii) investment in better join path discovery algorithms would highly benefit this problem as well.

2) ***Vary the number of Rows in the Query:*** We increase the number of rows inside one query view and observe its effect on the number of joinable groups, join graphs and views (i.e. the search space). As shown in Fig. 8(b), the relationship between the number of rows and the search space is not monotonous. Increasing the number of rows in a query view can cause the search space to shrink or grow. This is because there are two factors affecting the search space conversely when the number of rows in a query increases.

Enlarge the search space. Fig. 8(c) shows that the total No. of columns before clustering increase as the No. of rows increase and the No. of column clusters will increase as well.

Shrink the search space. Fig. 8(c) indicates that the number of clusters that COLUMN-SELECTION selects will decrease as the No. of examples increases. This is because the score of the ground truth column and its corresponding cluster increases.

Conclusion. Intuitively, more rows should lead to fewer candidate PJ-views at the end, and previous work has demonstrated this when schemas are well-formed. But in pathless table collections this is not the case as demonstrated here.

3) ***Vary the number of Columns in the query:*** We study the effect of increasing the number of columns in the input query. We choose query with 2, 3 and 4 columns on ChemBL. Unlike varying the number of rows, the results of this experiment are intuitive: higher number of columns in the input lead to higher number of join graphs, candidate PJ-views, and runtime, we do not plot the data for space reasons.

References

[1] A. Doan, A. Halevy, and Z. Ives, *Principles of data integration*. Elsevier, 2012.

[2] F. Nargesian, E. Zhu, R. J. Miller, K. Q. Pu, and P. C. Arocena, "Data lake management: challenges and opportunities," *PVLDB*, vol. 12, no. 12, pp. 1986–1989, 2019.

[3] N. Huijboom and T. Van den Broek, "Open data: an international comparison of strategies," *European journal of ePractice*, vol. 12, no. 1, pp. 4–16, 2011.

[4] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia, "Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics." CIDR, 2021.

[5] R. Castro Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker, "Aurum: A data discovery system," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 1001–1012.

[6] A. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang, "Goods: Organizing google's datasets," in *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, 2016, pp. 795–806.

[7] F. Chirigati, R. Rampin, A. S. R. Santos, A. Bessa, and J. Freire, "Auctus: A dataset search engine for data augmentation," *CoRR*, vol. abs/2102.05716, 2021. [Online]. Available: https://arxiv.org/abs/2102.05716

[8] Y. Zhang and Z. G. Ives, "Juneau: data lake management for jupyter," *PVLDB*, vol. 12, no. 12, 2019.

[9] E. Zhu, D. Deng, F. Nargesian, and R. J. Miller, "Josie: Overlap set similarity search for finding joinable tables in data lakes," in *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*, 2019, pp. 847–864.

[10] F. Nargesian, E. Zhu, K. Pu, and R. Miller, "Table union search on open data," *PVLDB*, vol. 11, pp. 813–825, 2018.

[11] A. Bogatu, A. A. A. Fernandes, N. W. Paton, and N. Konstantinou, "Dataset discovery in data lakes," in *ICDE*, 2020, pp. 709–720.

[12] E. Zhu, F. Nargesian, K. Q. Pu, and R. J. Miller, "Lsh ensemble: Internet-scale domain search," *PVLDB*, vol. 9, no. 12, 2016.

[13] R. C. Fernandez, J. Min, D. Nava, and S. Madden, "Lazo: A cardinality-based method for coupled estimation of jaccard similarity and containment," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 1190–1201.

[14] M. M. Zloof, "Query by example," in *Proceedings of the May 19-22, 1975, national computer conference and exposition*, 1975, pp. 431–438.

[15] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy, "Query by output," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 535–548. [Online]. Available: https://doi.org/10.1145/1559845.1559902

[16] J. M. Hellerstein, M. Stonebraker, and J. Hamilton, *Architecture of a database system*. Now Publishers Inc, 2007.

[17] F. Faerber, A. Kemper, P.-Å. Larson, J. Levandoski, T. Neumann, A. Pavlo *et al.*, *Main memory database systems*. Now Publishers, 2017.

[18] L. Seligman, P. Mork, A. Halevy, K. Smith, M. J. Carey, K. Chen, C. Wolf, J. Madhavan, A. Kannan, and D. Burdick, "Openii: an open source information integration toolkit," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 1057–1060.

[19] Z. G. Ives, N. Khandelwal, A. Kapur, and M. Cakir, "Orchestra: Rapid, collaborative sharing of dynamic data." in *CIDR*, 2005, pp. 107–118.

[20] D. Deng, R. C. Fernandez, Z. Abedjan, S. Wang, M. Stonebraker, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, and N. Tang, "The data civilizer system." in *Cidr*, 2017.

[21] C. Chen, B. Golshan, A. Y. Halevy, W.-C. Tan, and A. Doan, "Biggorilla: An open-source ecosystem for data preparation and integration." *IEEE Data Eng. Bull.*, vol. 41, no. 2, pp. 10–22, 2018.

[22] "Uchicago irb https://sbsirb.uchicago.edu/about/."

[23] M. Vartak, A. Parameswaran, N. Polyzotis, and S. R. Madden, "Seedb: automatically generating query visualizations," 2014.

[24] E. Horvitz, "Principles of mixed-initiative user interfaces," in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, 1999, pp. 159–166.

[25] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, "The nonstochastic multiarmed bandit problem," *SIAM journal on computing*, vol. 32, no. 1, pp. 48–77, 2002.

[26] H. Chernoff, "A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations," *The Annals of Mathematical Statistics*, pp. 493–507, 1952.

[27] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher, "An analysis of approximations for maximizing submodular set functions—i," *Mathematical programming*, vol. 14, no. 1, pp. 265–294, 1978.

[28] Z. Chen, V. Narasayya, and S. Chaudhuri, "Fast foreign-key detection in microsoft sql server powerpivot for excel," *PVLDB*, vol. 7, no. 13, pp. 1417–1428, 2014.

[29] L. Bornemann, T. Bleifuß, D. V. Kalashnikov, F. Naumann, and D. Srivastava, "Natural key discovery in wikipedia tables," in *Proceedings of The Web Conference 2020*, 2020, pp. 2789–2795.

[30] A. Gaulton, L. J. Bellis, A. P. Bento, J. Chambers, M. Davies, A. Hersey, Y. Light, S. McGlinchey, D. Michalovich, B. Al-Lazikani, and J. P. Overington, "ChEMBL: a large-scale bioactivity database for drug discovery," *Nucleic Acids Research*, vol. 40, no. D1, pp. D1100–D1107, 09 2011. [Online]. Available: https://doi.org/10.1093/nar/gkr777

[31] O. Lehmberg, D. Ritze, R. Meusel, and C. Bizer, "A large public corpus of web tables containing time and context metadata," in *Proceedings of the 25th International Conference Companion on World Wide Web*, 2016, pp. 75–76.

[32] K. Hu, S. Gaikwad, M. Hulsebos, M. A. Bakker, E. Zgraggen, C. Hidalgo, T. Kraska, G. Li, A. Satyanarayan, and Ç. Demiralp, "Viznet: Towards a large-scale visualization learning and benchmarking repository," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019, pp. 1–12.

[33] J. Mitlöhner, S. Neumaier, J. Umbrich, and A. Polleres, "Characteristics of open data csv files," in *2016 2nd International Conference on Open and Big Data (OBD)*. IEEE, 2016, pp. 72–79.

[34] S. Neumaier, J. Umbrich, and A. Polleres, "Automated quality assessment of metadata across open data portals," *Journal of Data and Information Quality (JDIQ)*, vol. 8, no. 1, pp. 1–29, 2016.

[35] F. Psallidas, B. Ding, K. Chakrabarti, and S. Chaudhuri, "S4: Top-k spreadsheet-style search for query discovery," ser. SIGMOD '15, 2015, p. 2001–2016.

[36] A. Fariha and A. Meliou, "Example-driven query intent discovery: Abductive reasoning using semantic similarity," *PVLDB*, vol. 12, no. 11.

[37] L. Qian, M. J. Cafarella, and H. V. Jagadish, "Sample-driven schema mapping," ser. SIGMOD '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 73–84. [Online]. Available: https://doi.org/10.1145/2213836.2213846

[38] C. Baik, Z. Jin, M. Cafarella, and H. V. Jagadish, "Duoquest: A dual-specification system for expressive sql queries," ser. SIGMOD '20, 2020, p. 2319–2329.

[39] K. Panev, N. Weisenauer, and S. Michel, "Reverse engineering top-k join queries," in *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*, B. Mitschang, D. Nicklas, F. Leymann, H. Schöning, M. Herschel, J. Teubner, T. Härder, O. Kopp, and M. Wieland, Eds. Gesellschaft für Informatik, Bonn, 2017, pp. 61–70.

[40] S. Zhang and Y. Sun, "Automatically synthesizing sql queries from input-output examples," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'13. IEEE Press, 2013, p. 224–234. [Online]. Available: https://doi.org/10.1109/ASE.2013.6693082

[41] W. C. Tan, M. Zhang, H. Elmeleegy, and d. Srivastava, "Regal$<$sup$>$ $+ <$ /sup$>$: Reverse engineering spja queries," *PVLDB*, vol. 11, no. 12, p. 1982–1985, Aug. 2018. [Online]. Available: https://doi.org/10.14778/3229863.3236240

[42] Y. Dong, K. Takeoka, C. Xiao, and M. Oyamada, "Efficient joinable table discovery in data lakes: A high-dimensional similarity-based approach," 2021.

[43] K. Wongsuphasawat, Z. Qu, D. Moritz, R. Chang, F. Ouk, A. Anand, J. Mackinlay, B. Howe, and J. Heer, "Voyager 2: Augmenting visual analysis with partial view specifications," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, 2017, pp. 2648–2659.

[44] T. Kraska, "Northstar: An interactive data science system," *PVLDB*, vol. 11, no. 12, pp. 2150–2164, 2018.

[45] P. Ouellette, A. Sciortino, F. Nargesian, B. G. Bashardoost, E. Zhu, K. Q. Pu, and R. J. Miller, "Ronin: Data lake exploration," *PVLDB*, vol. 14, no. 12, p. 2863–2866, jul 2021. [Online]. Available: https://doi.org/10.14778/3476311.3476364

[46] R. El Kindi, A. Bhandari, A. Fariha, B. Price, A. Vanterpool, A. Bowne, L. McEvoy, and V. Gadepally, "Examples are all you need: Iterative data discovery by example in data lakes," 2021.

[47] A. Santos, A. Bessa, F. Chirigati, C. Musco, and J. Freire, "Correlation sketches for approximate join-correlation queries," in *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, 2021, pp. 1531–1544.

[48] Z. Jin, Y. He, and S. Chauduri, "Auto-transform: learning-to-transform by patterns," *PVLDB*, vol. 13, no. 12, pp. 2368–2381, 2020.

[49] Y. He, X. Chu, K. Ganjam, Y. Zheng, V. Narasayya, and S. Chaudhuri, "Transform-data-by-example (tde) an extensible search engine for data transformations," *VLDB*, vol. 11, no. 10, pp. 1165–1177, 2018.

[50] S. Zhang and K. Balog, "Web table extraction, retrieval, and augmentation: A survey," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 11, no. 2, pp. 1–35, 2020.

[51] M. Joglekar, H. Garcia-Molina, and A. Parameswaran, "Interactive data exploration with smart drill-down," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 1, pp. 46–60, 2017.

[52] N. Bikakis and T. Sellis, "Exploration and visualization in the web of big linked data: A survey of the state of the art," *arXiv preprint arXiv:1601.08059*, 2016.

[53] X. Zhang, X. Ge, P. K. Chrysanthis, and M. A. Sharaf, "Viewseeker: An interactive view recommendation tool." in *EDBT/ICDT Workshops*, 2019.

[54] X. Zhang, "Interactive view recommendation," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2849–2851.

[55] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik, "Discovering queries based on example tuples," ser. SIGMOD '14. Association for Computing Machinery, 2014, p. 493–504.

[56] A. Bonifati, R. Ciucanu, and S. Staworko, "Learning join queries from user examples," *ACM Trans. Database Syst.*, vol. 40, no. 4, Jan. 2016. [Online]. Available: https://doi.org/10.1145/2818637

[57] A. Das Sarma, A. Parameswaran, H. Garcia-Molina, and J. Widom, "Synthesizing view definitions from data," in *Proceedings of the 13th International Conference on Database Theory*, ser. ICDT '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 89–103. [Online]. Available: https://doi.org/10.1145/1804669.1804683

[58] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava, "Reverse engineering complex join queries," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 809–820. [Online]. Available: https://doi.org/10.1145/2463676.2465320

[59] H. Li, C.-Y. Chan, and D. Maier, "Query from examples: An iterative, data-driven approach to query construction," vol. 8, no. 13, p. 2158–2169, Sep. 2015. [Online]. Available: https://doi.org/10.14778/2831360.2831369

[60] C. Wang, A. Cheung, and R. Bodik, "Synthesizing highly expressive sql queries from input-output examples," *SIGPLAN Not.*, vol. 52, no. 6, p. 452–466, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062365

[61] W. C. Tan, M. Zhang, H. Elmeleegy, and D. Srivastava, "Reverse engineering aggregation queries," *PVLDB*, vol. 10, no. 11, p. 1394–1405, Aug. 2017. [Online]. Available: https://doi.org/10.14778/3137628.3137648

[62] D. V. Kalashnikov, L. V. Lakshmanan, and D. Srivastava, "Fastqre: Fast query reverse engineering," ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 337–350. [Online]. Available: https://doi.org/10.1145/3183713.3183727

[63] K. Panev, S. Michel, E. Milchevski, and K. Pal, "Exploring databases via reverse engineering ranking queries with paleo," *PVLDB*, vol. 9, pp. 1525–1528, 09 2016.

[64] N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri, "Querying knowledge graphs by example entity tuples," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 10, pp. 2797–2811, 2015.

[65] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas, "Exemplar queries: Give me an example of what you need," *PVLDB*, vol. 7, no. 5, pp. 365–376, 2014.

[66] A. Bonifati, U. Comignani, E. Coquery, and R. Thion, "Interactive mapping specification with exemplar tuples," *ACM Transactions on Database Systems (TODS)*, vol. 44, no. 3, pp. 1–44, 2019.

[67] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas, "Exploring the data wilderness through examples," in *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*, 2019, pp. 2031–2035.

[68] R. Pimplikar and S. Sarawagi, "Answering table queries on the web using column keywords," *arXiv preprint arXiv:1207.0132*, 2012.

[69] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons learned from the chameleon testbed," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[70] W. McKinney *et al.*, "pandas: a foundational python library for data analysis and statistics," *Python for High Performance and Scientific Computing*, vol. 14, no. 9, pp. 1–9, 2011.

[71] R. D. Hipp, "SQLite," 2020. [Online]. Available: https://www.sqlite.org/index.html

[72] M. Raasveldt and H. Mühleisen, "Duckdb: An embeddable analytical database," in *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1981–1984. [Online]. Available: https://doi.org/10.1145/3299869.3320212

[73] M. Cai, M. Grund, A. Gupta, F. Nagel, I. Pandis, Y. Papakonstantinou, and M. Petropoulos, "Integrated querying of sql database data and s3 data in amazon redshift." *IEEE Data Eng. Bull.*, vol. 41, no. 2, pp. 82–90, 2018.