Qualitative Join Discovery in Data Lakes using Examples

Mir Mahathir Mohammad
University of Utah
mahathir.mohammad@utah.edu

El Kindi Rezig
University of Utah
elkindi.rezig@utah.edu

ABSTRACT

Finding relevant datasets is critical in any data pipeline but becomes challenging when data lacks schemas or metadata, as in data lakes. This makes it hard to identify the joins needed to produce the desired dataset. In query-by-example (QbE) join discovery, users provide a query table with a few example values, aiming to find joins from data lake tables that produce datasets containing those examples. Current QbE methods rely only on syntactic similarity, while semantic join discovery methods do not support QbE interfaces that work with limited example values. Moreover, existing QbE join path discovery methods (1) assume that the matching tables are directly joinable with each other, whereas in practice, a join path might contain intermediate tables that don't match the query table; and (2) do not ensure that the example tuples are contained in the returned joined table. We propose SemDisc, an end-to-end join discovery system that provides (1) discovery of hybrid join paths using both equi-join and semantic joins across data lake tables, (2) produces join paths that may include intermediate tables that do not overlap with the query tables but are needed to build high-quality joins, and (3) ensures the returned tuples are semantically similar to the ones in the provided examples. SemDisc supports efficient querying of joinable tables using an index that keeps track of high-quality join paths. Our evaluation across diverse workloads and datasets shows that SemDisc yields an average precision of over 0.86 in finding the correct join paths across various benchmarks, which is more than a 3× improvement over state-of-the-art join discovery methods.

1 INTRODUCTION

Data lakes are now widespread, spanning public repositories (e.g., US Government's Open Data [1]) and organization-wide data lakes (e.g., MIT Data Warehouse [2]). Constructing datasets from data lake tables—known as data discovery—is challenging, as users must understand the content and how tables can join with each other. Unlike traditional databases, data lakes typically lack schemas or explicit relationships (e.g., primary/foreign keys), making this process manual and time-consuming.

Given a large data lake, if a user wishes to assemble a dataset that spans multiple tables, the absence of join information severely impedes progress. Without knowing which tables can be joined—and on which attributes—users struggle to efficiently and accurately construct datasets of interest. To address this, several join discovery systems [30, 37, 60] have emerged to automatically identify joinable tables by detecting columns with overlapping values. Yet, despite these advances, existing systems exhibit fundamental limitations, as we illustrate in the following example.

Example 1. Consider the MIT Data Warehouse [2], a centralized repository aggregating data from various administrative systems. Since individual entities (e.g., institutes) manage their own tables, schemas are inconsistent and value representations may vary. Lou, a data scientist, wants to build a dataset by providing a *query table*—a small set of example values with example column headers (Figure 1①)—to construct a dataset from the data lake that includes those examples. Sample tables from the data lake are shown in Figure 1②. Because no single table fully contains the query examples, a join discovery system must identify join paths across multiple tables. In this case, the `Fclt_organization` and `Fac_building` tables each partially match the query. To be effective, the data discovery system should satisfy the following requirements:

(R1) Hybrid Join Paths: Since data lakes contain heterogeneous tables, the data discovery system must identify join paths that align with Lou's intent expressed through the semantic types and example values, even when columns represent the same concept differently (e.g., `MATS SCI & ENG` vs. `Materials Science and Engineering`). As shown in Figure 1③, the resulting join path should include both semantic (referred to as *sem_join* in Figure 1③) and equi-joins to answer the query—for example, a semantic join between `Fclt_organization` and `Fclt_rooms` via `Name` and `Org Code`, along with two equi-joins. The final joined table is shown in Figure 1④.

(R2) Best Join Path with Hidden Tables: In Figure 1③, note that the `Fclt_rooms` and `Fclt_floor` tables do not contain values that overlap with the values or semantic types in the query table; however, it is necessary to link the `Fclt_organization` and `Fac_building` tables in one join path to answer the query. Case in point, the data discovery system must identify join paths that include "hidden" intermediate tables needed to reach a table relevant to the query. And those hidden tables can contain semantic or equi-join between them. We refer to such join paths as join paths with *hidden tables*. Existing QbE methods (e.g., [37]) enumerate all paths between column pairs but do not identify the *best* join path that is aligned with the query table.

(R3) Semantic Tuple Matching: Given a query table, Lou ideally wants the resulting joined tables to contain semantically matching tuples. For instance, in Figure 1①, the query tuple t_1 = (`East Building, Materials Engineering`) appears in the final joined table (Figure 1④) with a different syntactic form: (`Building East 62, Materials Science and Engineering`) (tuple attributes ordered according to query columns). The join discovery system must ensure that all query tuples are semantically preserved in the output. However, existing QbE systems [37, 60] do not guarantee or verify that the joined results include semantically equivalent representations of the input tuples.

To address the above requirements, we introduce SemDisc, a QbE data discovery system that features the following:

(1) Hybrid Join Path Discovery at Scale: To address **R1**, SemDisc precomputes and indexes high-quality hybrid join paths offline, enabling fast and efficient retrieval at query time for QbE queries.

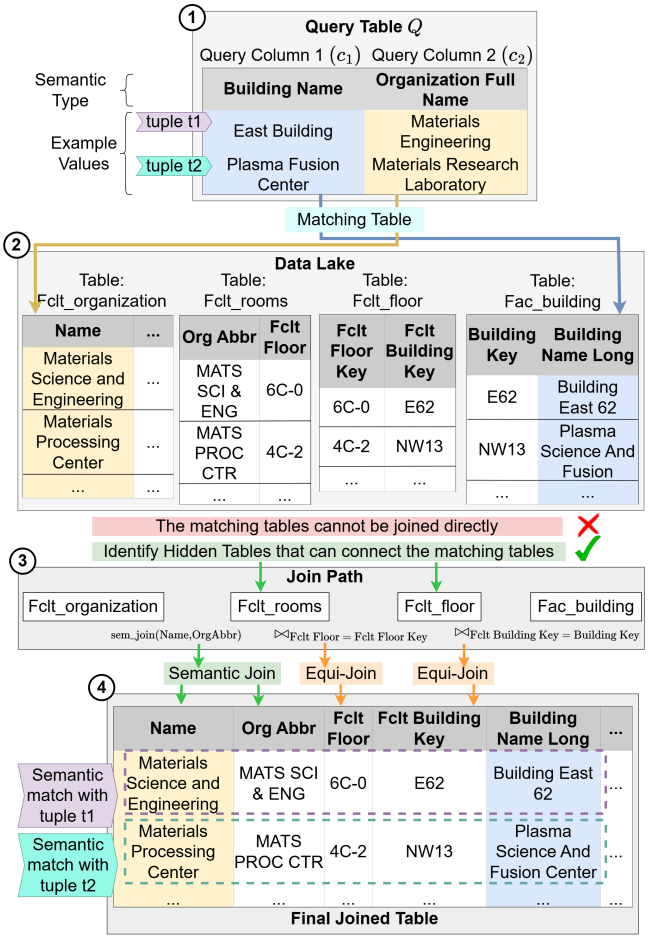

Figure 1: Overview of SemDisc. ① User provides a query table. ② SemDisc searches data lake tables; ③ finds hybrid join paths; and ④ returns the joined table.

(2) Join Discovery with Hidden Tables: To address requirement **R2**, SemDisc builds an Inverted Join Path Index for fast retrieval of join paths that include query table columns—and, when necessary, additional *hidden tables* that do not overlap with the query but are essential to reaching relevant tables. To our knowledge, SemDisc is the first data discovery system to support this capability.

(3) Semantic Tuple Validation during Query: SemDisc finds join paths that are likely to result in tables containing tuples that are semantically similar to the ones provided by the user in the example query table (requirement **R3**).

Contributions.

(1) We formalize the problem of discovering optimal join trees (of which join paths are a special case) with hidden tables for a given query table and prove that it is NP-hard (Section 3).
(2) We propose approximation techniques to efficiently compute the join graph, which encodes both semantic and equi-joins between data lake columns, along with effective pruning techniques to retain only high-quality join edges (Section 4).
(3) We develop an Inverted Join Path Index to support fast retrieval of the top-k join paths for a given query table (Section 5).
(4) We introduce an efficient algorithm to select join paths that are likely to contain tuples that semantically match the query table tuples (Section 6).
(5) We conduct an extensive experimental evaluation demonstrating the effectiveness and efficiency of SemDisc across multiple real-world datasets and against state-of-the-art data discovery baselines (Section 7).

2 RELATED WORK

SemDisc bridges three key problem spaces: (1) join path discovery [65, 69, 75], (2) Query-by-Example (QbE) data discovery [29, 49, 58, 61, 77], and (3) data lake search [8, 9, 12, 23, 24, 28, 30, 32, 39, 57]. As summarized in Table 1, SemDisc is the first end-to-end data discovery system to support the following capabilities for answering Query-by-Example queries: (1) hybrid join paths combining semantic and equi-joins; (2) join paths with *hidden tables*; (3) retrieval of joined tables that semantically match all query tuples; and (4) semantic column-level matching between query and data lake tables (data lake search).

Equi-join Path Discovery. Numerous systems focus on discovering equi-joinable tables in data lakes [65, 69, 75]. Aurum [30] constructs a join graph (an enterprise knowledge graph) that users can query through composable primitives. JOSIE [75] efficiently estimates set similarity using prefix filters [14], enabling large-scale equi-join discovery. An extension of Aurum [31] augments this graph with new links derived from semantic matchers and ontologies connecting tables by name or attribute. LSH Ensemble [76] computes Jaccard set containment between a query domain and candidate columns, while D^3L [10] employs multiple LSH indexes to capture schema- and instance-level features for joinable-column search across datasets.

DBXplorer [3] was among the first systems to support keyword-based data discovery, identifying join paths across tables that collectively contain the user's terms. However, it assumes a fixed relational schema with explicit primary–foreign key relationships. In contrast, SemDisc targets schema-less data lakes lacking predefined relationships, where discovering join paths requires reasoning over both equi- and semantic joins.

Semantic Join Path Discovery. Recent work has explored semantic join discovery in data lakes. DeepJoin [22] fine-tunes a pre-trained language model to measure semantic joinability between columns. It embeds all columns, indexes them using a Hierarchical Navigable Small World (HNSW) [52] graph, and retrieves the top semantically joinable columns for a given query via Approximate Nearest Neighbor search. Its precursor, PEXESO [21], embeds column values into high-dimensional vectors and applies pivot-based filtering [15] to prune candidate comparisons. WarpGate [17] similarly embeds column values and leverages Locality-Sensitive Hashing [13] to efficiently locate joinable columns.

Unlike these column-level systems, SemDisc discovers hybrid join paths that combine equi- and semantic joins across multiple tables and can include hidden tables not referenced in the query.

Snoopy [38] learns embeddings based on semantic joinability rather than syntax, retrieving top-k semantically joinable columns. SemDisc extends this by supporting tuple-level semantic matching and hidden-table discovery. Models such as TaBERT [72], TURL [18],

Ditto [50], and Starmie [27] produce structure-aware embeddings that encode relationships among table columns and cells. While SemDisc can plug in such embeddings, it functions as an end-to-end join discovery system—integrating query-by-example, hidden table support, and semantic tuple validation beyond embedding-based similarity.

Query-by-Example Data Discovery. The Query-by-Example (QbE) paradigm has long been studied in relational databases with fixed schemas [29, 49, 58, 61, 77]. Our work adopts a similar interface for schema-less data lakes. Ver [37] is the state-of-the-art QbE data discovery system in this setting, returning top join paths for a user-provided query table. DICE [60] also discovers join paths from example tuples, using user feedback to refine results. Both Ver and DICE rely on MinHash [11] to approximate Jaccard similarity between columns when constructing the join graph. However, they operate purely on equi-joins, relying on syntactic overlap and ignoring semantic similarity and data heterogeneity. THETIS [16] extends QbE-style search to semantic table retrieval using knowledge graphs to rank tables by relevance, but it does not perform join-path discovery.

DataXFormer [5] addresses QbE transformation discovery: given example input–output column pairs, it identifies joins that reproduce the transformations. Yet it assumes all source attributes lie in one table and all targets in another, reducing the problem to a two-table search with functional dependencies (1–1 joins). In contrast, SemDisc handles attributes spread across multiple tables without assuming such dependencies, resulting in a substantially larger and more realistic search space. Discovery-by-navigation approaches [55] build keyword-centric graphs that guide users from general to specific attributes. These are designed for semantic exploration, not join-path construction. SemDisc instead builds a query-agnostic semantic join graph, enabling efficient join-path retrieval for any query table at runtime.

Method Name	Query Type	Equi-Joins	Semantic Joins	Semantic Search	Hybrid Join Paths	Joins w/Hidden Tables	Semantic Tuple Validation
Starmie[27]	Target Table or Column	✗	✗	✓	✗	✗	✗
SemProp[31]		✗	✗	✓	✗	✗	✗
WarpGate[17]		✗	✓	✓	✗	✗	✗
PEXESO[21]		✗	✓	✓	✗	✗	✗
Gen-T[26]		✓	✗	✗	✗	✗	✓
Nexus[36]		✓	✗	✗	✗	✗	✓
SANTOS[43]		✗	✗	✓	✗	✗	✗
DeepJoin[22]		✓	✓	✓	✗	✗	✗
ALITE[44]		✓	✗	✗	✗	✗	✗
JOSIE[75]		✓	✗	✗	✗	✗	✗
S3D[35]		✓	✗	✓	✗	✗	✗
MF-Join[65]		✓	✗	✗	✗	✗	✗
SMS-Join[69]		✓	✓	✗	✗	✗	✗
BLEND[25]		✓	✗	✗	✗	✗	✗
Metam[34]		✓	✗	✗	✗	✗	✗
Snoopy[38]		✓	✓	✓	✗	✗	✗
Ver[37]	Examples	✓	✗	✗	✗	✓	✗
THETIS[16]	Examples	✗	✗	✓	✗	✗	✗
STR[74]	Keyword	✗	✗	✓	✗	✗	✗
Solo[66]	NL	✗	✗	✓	✗	✗	✗
Aurum[30]	SRQL	✓	✗	✗	✗	✗	✗
SemDisc	Example values and types	✓	✓	✓	✓	✓	✓

Table 1: Feature comparison of data discovery methods. (NL = Natural Language, SRQL = Source Retrieval Query Language)

3 PROBLEM DEFINITION AND SYSTEM OVERVIEW

Modern data lakes consist of a large number of heterogeneous tables collected from diverse sources, often without consistent schemas or explicit relationships [36, 56]. As a result, discovering how tables in a data lake relate to one another is a non-trivial challenge. This section formalizes the problem setting of SemDisc, our system for semantic join discovery across large-scale data lakes. We begin by establishing the fundamental concepts and notation used throughout the paper (Section 3.1, and Section 3.2). We then formally define our problem and establish its hardness (Section 3.3).

3.1 Preliminaries

Semantic Types. Following prior work [33, 63, 68], the semantic type of a column c, denoted $type(c)$, represents the real-world concept shared by its values (e.g., `Person`, `BirthPlace`).

Data Lake Tables. We denote data lake $\mathcal{D}$ as a collection of tables where $\mathcal{D} = \{T_1, T_2, \ldots, T_n\}$. Each table $T \in \mathcal{D}$ consists of a set of columns $attr(T) = \{c_1, c_2, \ldots, c_z\}$ and a set of *data lake tuples* $tup(T) = \{t_1, t_2, \ldots, t_r\}$. We refer to a specific column c_i in table T as $T.c_i$, with its cardinality denoted by $|c_i|$. For simplicity, we omit the table name when the referenced column is clear from context. For a table T containing a tuple t_i, we use $t_i[c_j]$ to denote the value of tuple t_i in column c_j, respectively. Each column c_j is associated with a semantic type, denoted $type(T.c_j)$, and a set of values denoted $vals(T.c_j)$.

Query Table. A query table, denoted Q, follows the same structural form as any $T \in \mathcal{D}$, but is provided by the user to express their query. Its tuples, called *query tuples* ($t_i \in tup(Q)$), contain example values illustrating the user's intent. Each query column $c_j \in attr(Q)$ likewise has an associated semantic type $type(Q.c_j)$ specified by the user.

An example query table Q is illustrated in Figure 1① where $attr(Q) = \{c_1, c_2\}$. For instance $type(Q.c_1)$ ="Building Name", and $vals(Q.c_1)$ =["East Building", "Plasma Fusion Center"].

To support joins between heterogeneous tables with varying value representations, we use a pre-trained language model [59] (PLM) to embed attribute values. This enables the system to join semantically similar values despite syntactic differences. For both semantic and equi-joins, value matches are defined via vector similarity, as detailed below.

Definition 1 (Semantic Match). *Let $u, v \in \mathcal{U}$ be two strings ($\mathcal{U}$ denotes the universe of string values from data lake columns). Let $emb : \mathcal{U} \to \mathbb{R}^d$ be an embedding function that maps each string to a d-dimensional ℓ_2–normalized vector ($\|emb(x)\|_2 = 1$), where $d \in \mathbb{N}$. Let $s : \mathbb{R}^d \times \mathbb{R}^d \to [0, 1]$ be a similarity function, and $\theta \in [0, 1]$ be a similarity threshold. We define the Semantic Match between u and v as follows:*

$$M^s_\theta(u, v) = \mathbb{1}\Big(s(emb(u), emb(v)) \geq \theta\Big) \tag{1}$$

Equation 1 evaluates to 1 if the $emb(u)$ and $emb(v)$ have a cosine similarity of at least θ, and 0 otherwise. We use cosine similarity as

the similarity function s for its proven usage in finding semantically similar values [17, 22, 53], and we use SBERT [59] as the embedding-generating model *emb*. Because our goal is to compare the semantic similarity between individual string values—independent of their surrounding table context—we selected SBERT as our embedding model. As shown in our experiments (Section 7), SBERT consistently outperformed several table-aware embedding models, where contextual table structure provides limited additional signal.

We use Equation 1 to capture both equi-joins and semantic joins. Setting the threshold $\theta = 1$ restricts matches to be made only between values with exactly the same embeddings, which is the case of equi-joins.

3.2 Column Joinability

Before defining joinability between columns, we define $c_{i,j}$ as the subset of values in column c_i that semantically match at least one value in column c_j ($i \neq j$), according to Equation 1. Formally:

$$c_{i,j} = \{u \in c_i \mid \exists v \in c_j \text{ such that } M^s_\theta(u, v) = 1\}.$$

In previous work [21, 22], the semantic joinability from a source column c_1 to a target column c_2 is defined as follows:

$$jn^s_\theta(c_1, c_2) = \frac{|c_{1,2}|}{|c_1|} \tag{2}$$

Note that Equation 2 is asymmetric, meaning that the value of joinability changes if the order of columns is reversed- e.g., $jn^s_\theta(c_1, c_2) \neq jn^s_\theta(c_2, c_1)$. This is because previous work only focused on finding a join between a given source column and a target column. In contrast, our focus is on discovering joins across *any* pair of columns and tables, i.e., we want the joinability score to be the same between a pair of columns regardless of their order. To align with this setting, we propose a symmetric joinability definition that is invariant to column order and accounts for semantic type similarity.

Definition 2 (Semantic Joinability). *Given two columns c_1 and c_2, the semantic joinability between c_1 and c_2 is defined as follows:*

$$jn(c_1, c_2) = jn(c_2, c_1) = M^s_\theta(type(c_1), type(c_2)) \cdot \frac{1}{2}\left(\frac{|c_{1,2}|}{|c_1|} + \frac{|c_{2,1}|}{|c_2|}\right) \tag{3}$$

Equation 3 defines joinability between columns c_1 and c_2 as the product of their semantic type similarity and the average proportion of semantically matching values across both directions, ensuring a balanced and symmetric measure of alignment. Setting $\theta = 1$ reduces the measure to an equi-join. Unlike prior asymmetric formulations [21, 22], our definition is order-independent—$jn(A, B) = jn(B, A)$—capturing joinability as a commutative relation consistent with the symmetry of inner joins.

Join Graph. After introducing semantic joinability, a natural question arises: how do we keep track of all columns that are semantically joinable? To address this, we introduce a join graph—a data structure that encodes join relationships across tables in the data lake. This graph serves as the central data structure of SemDisc, capturing all semantically joinable columns and enabling the extraction of join paths that satisfy a user's query (Section 6). The join graph encodes joinability between tables in $\mathcal{D}$. Formally, it is defined as an undirected graph $G = (V, E)$ where:

- $V = \{T_i \mid T_i \in \mathcal{D}\}$, the set of nodes, corresponds to the tables in the data lake.
- $E \subseteq V \times V$, the set of undirected edges, represents join relationships between pairs of tables.

Each $edge = (T_i, T_j) \in E$ such that $i \neq j$ is characterized by:

(1) A label $edge.label = (c_i, c_j)$, where $c_i \subseteq \text{attr}(T_i)$ and $c_j \subseteq \text{attr}(T_j)$ are the sets of join columns from tables T_i and T_j, respectively.
(2) A weight $edge.weight \in [0, 1]$, where $edge.weight$ is the joinability (Equation 3) of T_i and T_j using the columns c_i, c_j, i.e., $jn(c_i, c_j)$
(3) A type $edge.type \in \{\text{semantic, equi-join}\}$, which specifies whether the edge represents a semantic join or an equi-join.

Definition 3 (Hybrid Join Path). *In a join graph $G = (V, E)$, a hybrid join path P is a simple path connecting a sequence of tables $tables(P) = \{T_1, T_2, \ldots\}$. Consecutive tables T_i and T_{i+1} are joined via columns $A \in attr(T_i)$ and $B \in attr(T_{i+1})$ using either a semantic join (cosine similarity threshold $\theta \in [0, 1)$) or an equi-join ($\theta = 1$).*

Candidate Tables and Columns. For a given query table Q with l columns $\{Q.c_1, Q.c_2, ..., Q.c_l\}$, we define the set of their respective best-matching candidate columns from the data lake as $C_{set}(Q) = \{c_1, c_2, ..., c_l\}$ that are respectively contained in candidate tables $T_{set}(Q) = \{T_1, T_2, ..., T_l\}$. Let $T(P)$ be a table generated from a join path P containing all tables $T_{set}(Q)$ projected on the candidate columns $C_{set}(Q)$, i.e., $attr(T(P)) = \{c_1, c_2, ..., c_l\}$.

Definition 4 (Semantic Tuple Match). *We consider a given query table Q and a joined table $T(P)$ materialized from join path P. We say that the materialized table $T(P)$ satisfies the Semantic Tuple Match ($STM(Q, T(P))$) condition for query table Q if the following expression is satisfied:*

$$\forall\, t \in tup(Q); \exists\, t' \in tup(T(P)) : \sum_{i=1}^{l} M^s_\theta(t[c_i],\ t'[c_i]) = l \tag{4}$$

Intuitively, Equation 4 ensures that every tuple in the query table has at least one semantically similar match in $tup(T(P))$. Here, $tup(Q)$ and $tup(T(P))$ denote the sets of tuples in the query table Q and the materialized table $T(P)$, respectively. The Semantic Tuple Match condition holds if and only if every tuple $t \in tup(Q)$ has a corresponding tuple $t' \in tup(T(P))$ such that each column value $t[c_i]$ semantically matches $t'[c_i]$ for $i \leq l$.

For example, in Figure 1①, the query tuple t_1 = (`East Building`, `Materials Engineering`) is semantically matched with the joined table tuple (`Building East 62`, `Materials Science and Engineering`) (Figure 1④). The joined table's attributes are projected and ordered to align with the query columns. As a result, t_1 has a corresponding tuple in the joined table that semantically matches on all attributes. In fact, both query tuples, t_1 and t_2, have at least one such matching tuple. Thus, the joined table in Figure 1④ satisfies the Semantic Tuple Match condition for query Q.

3.3 Problem Definition

Our search space consists of *join trees* – acyclic connected subgraphs of the join graph, where each edge represents either a semantic or an equi-join. A join path is a special case of a join tree restricted to a single path. Thus, our task can be formulated as

finding a join tree that: (1) maximizes joinability and includes a given set of candidate tables, and (2) satisfies the Semantic Tuple Match condition while maximizing tuple cardinality (because we want to favor join paths that have a higher number of rows). We formalize these as Problem 1 and Problem 2 below and establish their hardness.

Problem 1. ***(Weight-Optimal Join Tree for a Query Table Problem)**: Given a query table Q, we want to compute the join tree $P^*(Q)$ (P^*, for short) such that: (1) P^* maximizes the sum of edge weights from the join graph (because we want to include the most joinable tables); (2) P^* contains all the tables in $T_{set}(Q)$;(3) P^* may contain other tables that are not in $T_{set}(Q)$ (hidden tables), but are necessary to join all the tables in $T_{set}(Q)$; and (4) The total count of tables in P^* does not exceed a budget table count $\mathcal{L}$.*

Here, $T(P^*)$ is the materialized table of P^* projected on the candidate columns $C_{set}(Q)$, and the projected columns follow the same order as the columns of query Q.

Proposition 1. *Problem 1 is NP-Hard by reduction from the Minimum-Weight Steiner Tree problem (proof in Appendix A.1).*

Problem 2. ***(Optimal Join Tree for a Query Table Problem)**: We extend Problem 1 by requiring the Semantic Tuple Match condition to hold and, among all weight-optimal trees, maximize the tuple cardinality of the returned join tree. Formally, find a join tree P^* such that (i) it satisfies the weight-optimality conditions (Problem 1), (ii) $STM(Q, T(P^*))$ holds ($T(P^*)$ is the join tree materialization), and (iii) $|T(P^*)|$ (tuple cardinality) is maximized.*

Proposition 2. *Problem 2 is NP-Hard because Problem 1 is a special case of Problem 2, making Problem 2 NP-Hard. Proof by restriction (full proof in Appendix A.2).* □

Because enumerating all join trees is computationally expensive, SemDisc focuses on join *paths*– a constrained subset of join trees. We introduce heuristics to efficiently identify high-quality join paths—those that yield non-empty results and high column-level joinability (Section 6).

3.4 System Overview

Figure 2 illustrates the workflow of SemDisc. We divide the architecture into two phases: **(1) Offline Phase:** This is where SemDisc efficiently computes the join graph (Figure 2①) where SemDisc ingests a data lake of tables, leverages an LLM to annotate data lake columns with descriptive semantic types, and approximates semantic joinability between columns. To enable efficient join path search, SemDisc constructs an index (Figure 2②) by first building a join graph with semantic and equi-join edges, applying qualitative pruning to retain high-quality edges, and then creating an Inverted Join Path Index for fast retrieval of join paths. **(2) Online Phase:** In this phase (Figure 2③), the user submits a query table, and SemDisc matches the query table's example values and semantic types with data lake tables. SemDisc then applies a novel heuristic to semantically match query tuples with tuples from the joined paths without materializing them. SemDisc then queries the Inverted Join Path Index to get the top-k join paths that contain the candidate tables, along with hidden tables if needed. Next, we describe the building blocks of SemDisc in detail.

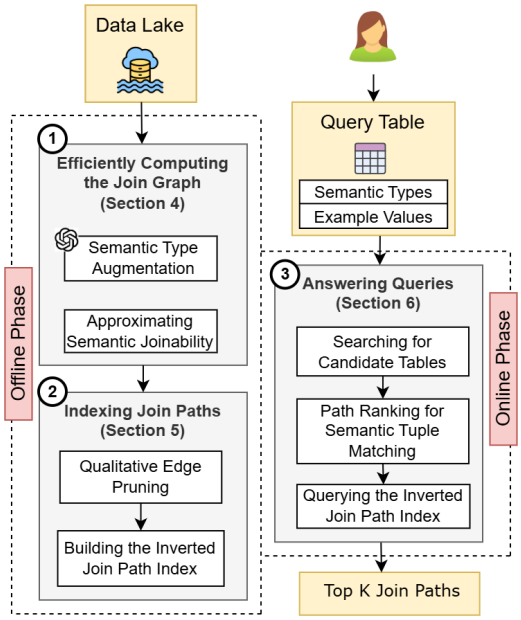

Figure 2: Architecture of SemDisc: ① Build join graph → ② Index join paths → ③ Retrieve top-k join paths.

4 EFFICIENTLY COMPUTING THE JOIN GRAPH

Constructing a join graph over large data lakes is costly, requiring joinability checks across many column pairs and managing numerous edges. We propose efficient, sketch-based methods to estimate semantic column joinability and reduce graph complexity. Figure 2① illustrates this offline step. Section 4.1 describes semantic type extraction, and Section 4.2 details our joinability approximation techniques.

4.1 Semantic Type Augmentation

What if two columns have a high overlap of values but refer to different real-world entities? For instance, columns `student_id` and `part_id` might share many values but do not refer to the same real-world entity, and thus cannot be joined. To make matters worse, we cannot rely on the column headers to be descriptive. To solve this problem, we resort to semantic type annotation of all columns in the data lake. Column type annotation is an active research area [33, 63, 68]. SemDisc augments all the data lake columns with LLM-generated semantic types. We prompt GPT-4o [7] with sample values from the data lake columns and other co-occurring columns in the same table (to give it context). The LLM then provides suggested semantic types for each column.

The prompt for the LLM to extract semantic types of the columns in a table has the following structure:

- **Instruction:** An instruction elaborating that the LLM needs to find the semantic types of the columns in a table.

```
Below is a list of column ids and a sample of their
values separated by commas from a table. Give detailed
elaborate semantic type for each column in the
following format:
```

- **Output Format:** The structure of the output list containing the column header and semantic type in each list item.

```
[{"id": columnid1, "semantictype":typename1},
{"id": columnid2, "semantictype":typename2}, ...]
```

• **Input Columns:** The columns are listed one by one with their header value and a sample of the 20 most occurring values.

```
id: <COLUMN_HEADER>
sample values: <SAMPLE_VALUES>,
```

For an example prompt, refer to Appendix B.

4.2 Approximating Semantic Joinability

To build hybrid join paths, SemDisc calculates the joinability between columns for both semantic and equi-joins using the same framework, which allows SemDisc to leverage a single unit of measurement to evaluate the quality of edges in the join graph for both semantic and equi-joins. SemDisc computes semantic edges in the join graph G for all pairs of tables with semantically joinable columns. These semantic edges capture relationships between columns based on their content similarity. To efficiently estimate the semantic joinability of two columns c_1 and c_2 from tables T_1 and T_2 respectively, SemDisc follows a three-step process, allowing both semantic and equi-joinable edges in the same graph:

(1) **SimHash Generation:** For each value in columns c_1 and c_2, SemDisc generates a SimHash [13] value of its vector embedding. SimHash is a *locality-sensitive hashing (LSH)* technique that maps similar high-dimensional embeddings into the same subspace. Using SimHash, SemDisc ensures that similar embeddings (i.e., column values) are assigned the same hash value.

To generate the SimHashes of values for a given data lake, SemDisc generates a fixed set of b random hyperplanes, where the length of each hyperplane is equal to the length d of each embedding extracted from data lake values by the embedding model. Formally, the hyperplanes are $h_1, h_2, ..., h_b \in \mathbb{R}^d$. Each data lake embedding is projected on these hyperplanes, assigning 0 or 1 for each of the b hyperplanes. Each embedding of the values in the data lake is encoded into a SimHash of bit count b, e.g. `Materials Science and Engineering` is encoded into `01110111` where the value of b is 8. Semantically joinable values are likely to be close to each other in the vector space, and hence likely to be assigned the same hash value, reducing the problem of finding semantically joinable value pairs to finding equi-joinable value pairs between two columns. For example, `MATS SCI & ENG` is encoded into the same SimHash value `01110111` as `Materials Science and Engineering`.

Now, the SimHash values can be joined using equality between them, allowing us to build hybrid paths where we can evaluate the join paths for both semantic and equi-joins. The procedure of selecting the SimHash bit count b for a given cosine similarity is described in the experiments section (Section 7.3).

(2) **MinHash Signature Computation:** To summarize the set of SimHash values generated for c_1 and c_2, SemDisc computes a fixed-size *MinHash signature* [11] for each column. MinHash is a popular LSH technique used to estimate the Jaccard similarity between sets efficiently. It works by creating compact, fixed-size signatures that approximate the extent of overlap between two sets.

(3) **Approximate Semantic Joinability:** Finally, SemDisc calculates the Jaccard similarity between the MinHash signatures of c_1 and c_2 to approximate the joinability of Equation 3. The Jaccard similarity estimates the overlap of SimHashes between two columns. Formally, this process is summarized as follows:

$$w(c_1, c_2) = J(minhash(simhash(c_1)), minhash(simhash(c_2))) \quad (5)$$

J denotes the Jaccard similarity, $simhash(c)$ denotes the list of SimHash values of column c. For non-string columns, we ignore the SimHash encoding layer and directly use the column values to create the MinHash signatures per column. The weight $w(c_1, c_2)$ corresponds to the undirected join edge e in the join graph connecting tables T_1 and T_2, with label (c_1, c_2) denoting the pair of columns involved in the join. The higher the value of $w(c_1, c_2)$, the more joinable c_1 and c_2 are in the case of both equi-join and semantic join edges. Now, we have a join graph containing both semantic join and equi-join edges, where weights capture the degree of joinability between two columns, allowing us to build hybrid join paths that contain both types of edges.

Time Complexity. Using SimHash reduces the time complexity of finding semantically similar value pairs between two columns. Calculating joinability between a column pair c_1, c_2 directly using embeddings has a time complexity of $O(|c_1|\cdot|c_2|\cdot d)$. Calculating joinability using the MinHash signature pair that was generated from the SimHashes has a total time complexity of $O((|c_1|+|c_2|) \cdot m + m)$, where $O((|c_1|+|c_2|) \cdot m)$ is required for hashing the values of 2 columns using m hash functions. $O(m)$ is the time required to evaluate the Jaccard similarity between two MinHash signatures of size m. The final time complexity is $O((|c_1|+|c_2|) \cdot m)$ which is significantly lower than $O(|c_1|\cdot|c_2|\cdot d)$ since $m << d$.

5 INDEXING JOIN PATHS

The join graph G may contain $O(\binom{|\mathcal{D}|}{2} \cdot {C_{max}}^2)$ edges, where $|\mathcal{D}|$ is the number of tables in the data lake and C_{max} is the maximum count of columns in a table. For example, in one of the data lakes we evaluated (OpenData dataset of LakeBench [19]), the join graph contained over 1.7 million edges (Table 4ⓕ). This makes finding join paths (simple paths in the graph) computationally expensive. We outline strategies to significantly reduce the edge count in the join graph by removing low-quality edges (Figure 2②).

5.1 Qualitative Edge Pruning

SemDisc applies Algorithm 1 to prune join edges based on multiple signals. For each table pair T_i, T_j (vertices in G), and their corresponding set of edges $edges(T_i, T_j)$ (line 4), SemDisc retains only the top-ranked edge based on edge weight (lines 10–11). The pruning process consists of two main stages:

(1) **Semantic Type Similarity (line 5):** SemDisc evaluates whether the semantic types of column c_i and c_j are similar by computing the cosine similarity between their LLM-generated type embeddings, using the similarity threshold θ (Equation 1).

(2) **Value Joinability (line 7):** SemDisc checks if the approximate semantic joinability score between columns c_i and c_j (as defined in Equation 5) meets or exceeds a threshold θ_j. Only the top-ranked edge is retained, and all others are discarded via the *prune_edges* function (line 11).

Algorithm 1: Join Graph Pruning

Input: G (the join graph)
Output: Pruned join graph

for *table pair* T_i, T_j *in* G **do**
  Initialize $best\_edges \leftarrow []$
  $edge\_ranks \leftarrow []$
  for u(*column* c_i, *column* c_j) *in* $edges(T_i, T_j)$ **do**
    if $M^s_\theta(type(c_i), type(c_j)) = 0$ **then**
      continue
    if $w(c_i, c_j) \geq \theta_j$ **then**
      $best\_edges \leftarrow best\_edges \cup u$
      $edge\_ranks[u] \leftarrow w(c_i, c_j)$
  $top\_edge \leftarrow u^*$ where $edge\_ranks[u^*]$ has the maximum value among edges in $edge\_ranks$
  $G \leftarrow prune\_edges(best\_edges, u^*)$
return G

Building the Join Paths. After pruning the join graph, SemDisc generates all possible join paths $\mathcal{P} = \{P_1, P_2, ...\}$, which correspond to simple paths in the join graph G, between all the pairs of nodes of G where the simple paths have at most $\mathcal{L}$ nodes. $|\mathcal{P}|$ is the count of join paths and each join path $P_i \in \mathcal{P}$ $(i \leq |\mathcal{P}|)$ is a simple path in G, i.e., the vertices traversed in a simple path in G. The join paths in $\mathcal{P}$ are not guaranteed to have a non-zero tuple count (cardinality). Therefore, SemDisc performs cardinality estimation to filter out join paths that produce zero tuples from a large collection of join paths. As cardinality estimation is a well-studied problem [6, 41, 45, 45–47, 62, 67, 70, 71] and not the focus of our contribution, we adopt an existing sampling-based cardinality estimation technique [48]. Since semantic joins are reduced to equi-joins using SimHash, estimating cardinality for hybrid paths becomes possible using existing cardinality estimation techniques for equi-joins. The estimated cardinality is used as a quality signal for a join path, i.e., the more tuples a join path yields, the better.

5.2 Building the Inverted Join Path Index

Going from a set of candidate tables (data lake tables with values or types that match those in the query table) to a join path requires searching for all join paths in $\mathcal{P}$ that contain the candidate tables (and possibly intermediate tables needed to join the candidate tables with each other). We formally define the *Join Path Query* problem as follows:

Definition 5. *(Join Path Query): Given a set of candidate tables for a query* Q, $T_{set}(Q) = \{T_1, T_2, ..., T_l\}$, *we want to find all the join paths* $P_{match} \subset \mathcal{P}$, *such that for every path* $P \in P_{match}$: $T_{set}(Q) \subset tables(P)$.

Naively, we could solve the Join Path Query problem using a linear search over $\mathcal{P}$, where, for each path of at most size $\mathcal{L}$, we check whether l candidate tables exist in that path. This approach has a time complexity of $O(l \cdot \mathcal{L} \cdot |\mathcal{P}|)$. Instead, to quickly find all the join paths containing a set of tables, SemDisc builds the Inverted Join Path Index offline to enable answering join path queries efficiently.

The Inverted Join Path index is a binary matrix with a row for each table in $\mathcal{D}$ and a column count equal to the number of simple paths in $\mathcal{P}$. Each column represents a join path from $\mathcal{P}$, and each row represents a table from $\mathcal{D}$. SemDisc builds the Inverted Join Path Index as follows:

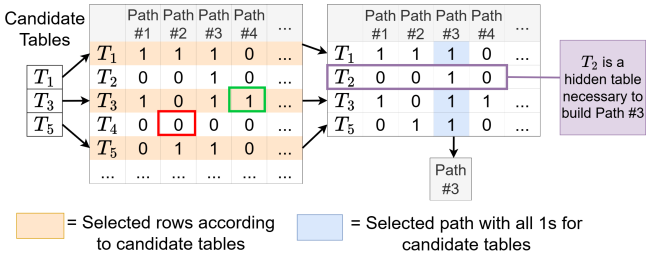

Figure 3: Illustration of querying the Inverted Join Path Index to get join paths having all the candidate tables.

(1) SemDisc initializes a matrix with dimension $|\mathcal{D}| \times |\mathcal{P}|$ with all zero entries.
(2) For each table T_i $(i \leq |\mathcal{D}|)$ in a given join path P_j in $\mathcal{P}$, SemDisc locates the row in the inverted index (i^{th} row among $|\mathcal{D}|$ rows), and sets the column j of that row to 1.

To find all join paths consisting of a set of candidate tables, SemDisc first retrieves the rows corresponding to the candidate tables from the Inverted Join Path Index. From the selected rows, SemDisc searches for the columns that have all 1s in them. From those columns, SemDisc returns the corresponding join paths containing the candidate tables (details in Section 6.3).

Example 2. We show an example of an Inverted Join Path Index in Figure 3. A cell value is 1 if the corresponding table is in the respective join path. For example, the cell value 1 marked in green in Figure 3 indicates that table T_3 is found in Path 4. The cell value 0 marked in red indicates that T_4 does not exist in Path 2.

To answer a join path query from the Inverted Join Path Index of Figure 3, say we want to find the join paths that consist of candidate tables T_1, T_3, and T_5. SemDisc first selects the rows corresponding to these tables (highlighted in orange), and among those rows, finds the columns that have all 1s in them (highlighted in blue). Notice that Path 3 consists of candidate tables T_1, T_3, T_5 with an additional table T_2, which is not a candidate table, but it is a *hidden table* needed to build Path 3. We conclude that Path 3 consists of all the tables T_1, T_3, and T_5 with an additional hidden table T_2.

Time Complexity. The time complexity for building the Inverted Join Path Index is $O(|\mathcal{P}| \cdot \mathcal{L})$ because SemDisc has to iterate over all $|\mathcal{P}|$ paths and each path has at most $\mathcal{L}$ tables, where $\mathcal{L}$ is the maximum allowed table count to build the join path. Time complexity of querying the Inverted Join Path Index is given in Section 6.3.

6 ANSWERING QUERIES

When a query table Q is submitted in the *online phase*, SemDisc must efficiently identify the most relevant join paths. As shown in the system architecture (Figure 2③), this process begins by retrieving a set of candidate tables for each query column, as described in Section 6.1. Next, SemDisc constructs candidate paths by combining these tables and ranks them to prioritize those most likely to satisfy the semantic tuple match condition (Section 6.2). Finally, in Section 6.3, we present how SemDisc queries the Inverted Join Path Index to retrieve join paths that include the tables in each candidate path.

6.1 Searching for Candidate Tables

For each column $c_i \in attr(Q)$, the objective of SemDisc is to find the top tables consisting of a column described by c_i. For each column $Q.c_i$, SemDisc fetches the top λ data lake columns that best match query column $Q.c_i$ using Algorithm 2.

Algorithm 2: Get Candidate Columns for Query Columns

```
   Input  : Query table Q : attr(Q) = {c_1, ..., c_l}, number of candidates λ
   Output : List of candidate column sets CsetList(Q)
1  C_all ← [ ]
2  for i = 1 to l do
3      C_i ← HNSW_ANN(emb(type(Q.c_i)), λ)
4      C'_i ← []
5      foreach c ∈ C_i do
6          /* Checking if the values in the query table are
               semantically contained in a candidate column c   */
7          if set(simhash(Q.c_i)) ⊆ set(simhash(c)) then
8              Append c to C'_i
9      Append C'_i to C_all
10 CsetList(Q) ← Cartesian product of column lists in C_all
11 return CsetList(Q)
```

Filtering Candidate Columns. SemDisc identifies candidate columns by retrieving the top-k approximate nearest neighbors (ANN) for each query column's semantic type embedding using HNSW [52] for efficient similarity search [51, 52, 54] (line 3, Algorithm 2). It then filters out columns whose precomputed SimHashes (Section 4.2) do not overlap with those of the query's example values (line 7), ensuring that all retained columns are semantically similar to the query values. The resulting column lists are combined via a Cartesian product to form CsetList(Q), capturing all candidate column combinations to be evaluated through the Inverted Join Path Index. In the next step, SemDisc ranks these combinations to prioritize those most likely to yield joined tuples matching the query table tuples. Refer to Appendix D for the time complexity analysis of Algorithm 2.

6.2 Path Ranking for Semantic Tuple Matching

Each $C_{\text{set}}(Q) \in$ CsetList(Q) contains one candidate column per query column, drawn from different data lake tables. To avoid materializing all join paths, which is computationally expensive, SemDisc uses a fast scoring heuristic (Algorithm 3) that assigns a *reward* to each C_{set}, estimating the likelihood that it will satisfy the semantic tuple match condition (Definition 4). The heuristic operates as follows: (1) Columns in C_{set} are grouped by their source tables (line 2). (2) Groups with only one column are ignored, as they do not contribute to tuple-level matches (line 4). (3) For each group and each query tuple t, the algorithm computes the intersection of SimHash index sets across all group columns to find if query table tuples have a partial match with data lake tuples (lines 7-8). (4) If any tuple yields an empty intersection, the candidate set is marked as invalid (line 9). (5) Otherwise, the group contributes one point to the overall reward (line 13-14).

Only valid candidate sets are retained, and each is assigned its final reward score. SemDisc then ranks all valid $C_{\text{set}}(Q)$ entries by their reward. Each $C_{\text{set}}(Q) = \{c_1, \ldots, c_l\}$ corresponds to a candidate path $T_{\text{set}}(Q) = \{T_1, \ldots, T_l\}$, and the collection of all T_{set} sets defines the join path search space considered in the next phase.

Algorithm 3: Semantic Tuple Match

```
   Input: List of candidate column sets CsetList(Q) for query table Q
   Output: Each candidate set assigned a reward or discarded
1  foreach C_set(Q) ∈ CsetList(Q) do
2      Group columns in C_set(Q) by table ID into groups; reward ← 0;
         valid ← True
3      /* Evaluate each group of tables                          */
4      foreach group G in groups with |G| > 1 do
5          foreach tuple t in Q do
6              idx ← all tuple indices of table(G)
7              foreach column c ∈ G do
8                  q ← query column for c
                     idx ← idx ∩ index_set(t[q], c)
9              if idx = ∅ then
10                 valid ← False
11         if valid then
12             // There is a partial tuple match
13             reward ← reward + 1
14     if valid then
15         Assign reward to C_set(Q)
16     else
17         Remove C_set(Q) from CsetList(Q)
18 Sort CsetList(Q) by reward (descending)
```

Example 3. Figure 4 illustrates reward-based path ranking. In path 1, all columns come from different tables, forming only single-column groups; no reward is assigned, so the total score is zero. Path 2 includes two columns from the same table (*Buildings*), forming a multi-column group. Intersecting their SimHash index sets across query tuples yields non-empty results, giving a reward of 1. Hence, SemDisc prioritizes path 2 over path 1 as it better preserves semantic matches with the query table.

Time Complexity. In Algorithm 3, for a single candidate column set $C_{\text{set}}(Q)$, grouping columns in line 2 takes $O(l)$ where l is the query column count. For a $C_{\text{set}}(Q)$, the worst case occurs when all query columns are matched from the same table, for which line 4 iterates only once. Line 5 iterates r times, r being the query table tuple count. If all candidate columns originate from the same table, line 7 iterates l times. Set intersection takes $O(|T|)$ where T is the table of the group and $|T|$ is the cardinality of T. Therefore, the total time complexity for one $C_{\text{set}}(Q)$ becomes $O(r \cdot l \cdot |T|)$.

6.3 Querying the Inverted Join Path Index

SemDisc uses the Inverted Join Path Index to answer join path queries for each candidate table set (tables that correspond to the candidate columns in CsetList(Q) in Algorithm 2). In addition to the Inverted Join Path Index, SemDisc maintains a hash map that maps each data lake table to the list of join paths that contain that table. Given a set of candidate tables $T_{\text{set}}(Q)$, SemDisc gets the join paths containing $T_{\text{set}}(Q)$ as follows: (1) Select the rows of the Inverted Join Path Index that correspond to each candidate table and additionally, find the candidate table $T' \in T_{\text{set}}(Q)$ for which the path count from the hash map is minimum. (2) Among the columns in the inverted index corresponding to the paths of T', select the columns containing all 1's in the selected rows. (3) Extract the

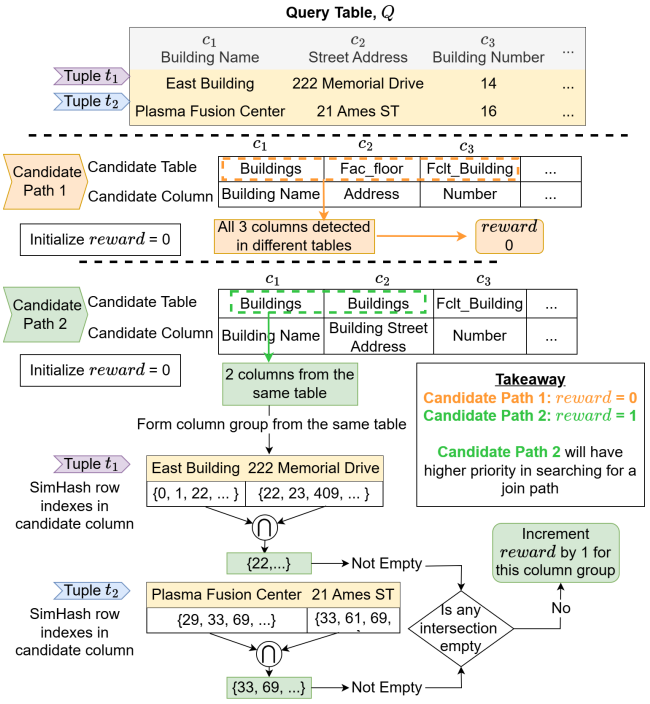

Figure 4: Demonstration of Algorithm 3 on two candidate paths.

corresponding join paths. (4) Materialize the top-K paths, projecting only the columns relevant to the query table Q.

Time Complexity. The time complexity of querying the Inverted Join Path Index is $O(l \cdot \min_{T \in T_{\text{set}}(Q)} |\{P | P \in \mathcal{P} \text{ and } T \in P\}|)$. l is the time required to check whether a column has all 1's in the selected rows. The last term of the time complexity is the number of paths for table T' in the hash map, where $T' \in T_{\text{set}}(Q)$ is the table containing the least number of paths in the hash map among the tables of $T_{\text{set}}(Q)$.

7 EXPERIMENTAL STUDY

In this evaluation study, we answer the following research questions:

- **RQ1:** How effective is SemDisc in finding the correct join paths for a given query table (Section 7.1)?
- **RQ2:** How do different embedding models perform at QbE join path discovery (Section 7.1)?
- **RQ3:** How efficient is SemDisc in returning join paths for a given query table (Section 7.2)?
- **RQ4:** What are the best hyperparameters for SemDisc (Section 7.3)?

Datasets. We evaluate SemDisc on real-world data lakes and three benchmarks: the Text-to-SQL workload Spider [73], the data discovery benchmark LakeBench [19], and the large relational collection SchemaPile [20]. Statistics are shown in Table 2. All real-world data lakes lack explicit schemas—only column names are available, and no relationships are known.

SchemaPile (SCP) contains 22,989 databases (34.9K tables with data, 109K PK–FK links) across multiple domains; we use the public SchemaPile-perm version. **Spider (SP)** includes 200 databases, of which 156 contain join queries (Table 3). For both, we remove PK–FK relationships to test systems' ability to recover joins without schema hints, using the original schemas as ground truth. **LakeBench (LB)** provides ground-truth joinable column pairs from its OpenData dataset.

We also include four real-world data lakes: **DrugCentral (DC)** [64], an online pharmacological database; **MIT Data Warehouse (MD)** [2], a centralized institutional data repository; and two from **data.gov** [1]: **U.S. Fish and Wildlife Service (FWS)** with ecological and geographic datasets, and **Centers for Disease Control and Prevention (CDC)** with public health records on disease, mortality, and drug usage.

Table 2: General statistics of the datasets.

Data Lake	Table Count	Tuple Count	Column Count	Mean Tuple Count	Mean Column Count
SchemaPile (SCP)	34.9K	1.06M	1.33M	28.66	7.00
Spider (SP)	802	73,164	4,097	91.23	5.11
LakeBench (LB)	500	491,352	10,229	982.70	20.46
DrugCentral (DC)	69	52,993	443	768.01	6.42
MITDWH (MD)	167	86,883	1,930	520.26	11.56
CDC	467	354,980	10,112	843.18	24.02
FWS	256	99,884	5,211	401.14	20.93

Table 3: Statistics of Spider (left) and distribution of join-path lengths (right).

Individual Database Count	156
Total Join Queries	6464
Total Unique Join Sequences	768
Max Table Count in a Join	5
Mean Join Path Per Database	4.076
Maximum Paths in a Database	22

Path Length	Join Path Count
2	452
3	164
4	21
5	3

Metrics. For a given query table, we measure the ability of various baselines to return the joined table that semantically matches the query table. Following previous work [40, 66], we use the precision-at-K metric (P@K) aggregated over all query tables. For a set of queries Q, we calculate P@K as follows:

$$P@K = \frac{|\{Q \in Q | \exists P \in \text{top K paths for } Q : STM(Q, T(P))\}|}{|Q|} \quad (6)$$

$P@K$ is the fraction of all queries for which at least one of the top K paths satisfies the Semantic Tuple Match condition.

We also conducted a series of experiments to evaluate the performance of SimHash approximation using the *F1 score*. The F1 score is defined as:

$$\text{F1} = \frac{2TP}{2TP + FN + FP} \quad (7)$$

where TP, FN, and FP denote the number of true positives, false negatives, and false positives, respectively. They are defined separately for each experiment in which the F1 score is reported.

Baselines. We compare SemDisc against the following data discovery systems: **Gen-T** [26] reconstructs a source table from a data lake by discovering related tables through SPJU (Select– Project– Join–Union) operators such as outer union, projection, and subsumption. It assumes, as part of the query table, a unique key column for joins, which we provide. **DataXFormer** [5] performs example-driven transformation discovery by finding a join from a set of source columns to a given target column. **Ver** [37] is the state-of-the-art QbE join discovery system supporting equi-joins only; we use its default hop count of two. **WarpGate** [17] discovers semantic

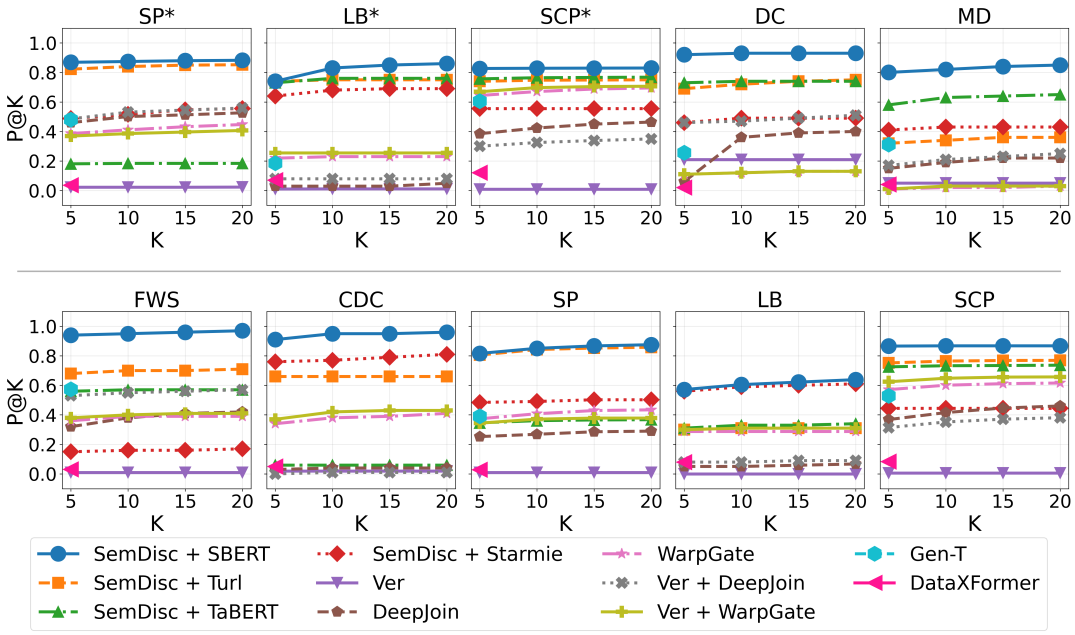

Figure 5: P@K plots for 11 baselines across 3 equi-join workloads and 7 semantic join workloads.

joins via SimHash-based similarity. For QbE, we encode each query column, retrieve the top-10 candidate tables, and test the first K combinations for joinability. **DeepJoin** [22] fine-tunes a language model to find semantically joinable columns; we follow the same top-10/K-combination procedure as for WarpGate. **TaBERT** [72], **TURL** [18], and **Starmie** [27] are Transformer-based table encoders capturing column, row, and text relations. We substitute SBERT in SemDisc with each model (SemDisc +TaBERT, SemDisc +TURL, SemDisc +Starmie) to assess effectiveness; SemDisc +SBERT is the default. **Ver+DeepJoin** and **Ver+WarpGate** are hybrid baselines combining Ver's join-graph traversal with DeepJoin's fine-tuned or WarpGate's SimHash-LSH joinability models.

Workload. We evaluate the baselines using query table set Q in an end-to-end manner for each data lake. We generate these query table sets for two types of joins: 1) Equi-join, and 2) Semantic join. **Equi-join Workload:** We use three benchmark workloads from Spider, LakeBench and SchemaPile since they all come with join paths: **(1) Spider:** We sample join queries spanning 156 databases, using 5 random rows from each result as query tables. These SQL queries serve as ground truth, providing the exact join paths. **(2) LakeBench:** Since LakeBench only includes ground truth for two-table joins, we construct longer join paths transitively (e.g., A joins with B, then B with C, etc.). These join paths serve as the ground truth path for query table generation. **(3) SchemaPile:** We build join paths spanning across 22,989 databases using 109,092 PK-FK relationships given in the benchmark. Query tables are built using 5 random rows and 2 to 5 random columns from each materialized join path. We refer to these equi-join workloads as SP*, LB*, and SCP* for Spider, LakeBench, and SchemaPile, respectively. **Semantic Join Workload.** We build the ground-truth join graph by computing cosine similarity between all value embeddings (Equation 3), yielding exact tuple-level semantic edges. From this graph, we sample 100 join paths (≤10M rows), materialize them, and select 2–5 columns per path, including at least one from the first and last tables to hide intermediates. Each query table contains five sampled rows with LLM-assigned semantic types. Similarity thresholds and parameters are listed in Table 4. We generate semantic join workloads for all datasets in Table 2 and refer to them by their acronyms (e.g., CDC for the CDC dataset, SP for Spider, LB for LakeBench, and SCP for SchemaPile). SemDisc 's join path index is built from all join paths using sketch-based edges (MinHash, SimHash; Section 5); random sampling occurs only when forming Q for evaluation. P@K experiments test whether, given a query table derived from a sampled ground-truth join path, SemDisc retrieves top-k paths satisfying the Semantic Tuple Match condition (Definition 4). Real-world examples with hidden tables are shown in Appendix C.

System setup. All experiments are conducted on a Linux-based high-performance machine with 400 GB RAM, 40 2.75 GHz AMD EPYC 9454 CPU cores, and an NVIDIA A800 GPU.

7.1 End-to-end Evaluation

This experiment evaluates the baselines in answering query tables using the top-K join paths. We compute $P@K$ (Equation 6) across datasets and report results in Figure 5.

Comparison of Equi-join Baselines. Across the three equi-join workloads (SP*, LB*, SCP*) in Figure 5, SemDisc achieves the highest P@K over equi-join baselines Ver, Gen-T, and DataXFormer, even though these benchmarks contain only PK–FK joins and no semantic edges. SBERT+SimHash maps syntactically identical value pairs to the same SimHash and MinHash sketches, which creates the equi-join edges. Join paths that match the query table are then retrieved from the inverted join-path index.

In contrast, after detecting candidate tables for each query column, Ver arbitrarily orders candidate tables before exploring join paths with a fixed hop count, missing the correct paths. Gen-T assumes an explicit ID column as the first query attribute; if absent, it performs a greedy search to locate one. In data lakes, this often fails because no feasible join path exists to a table containing that

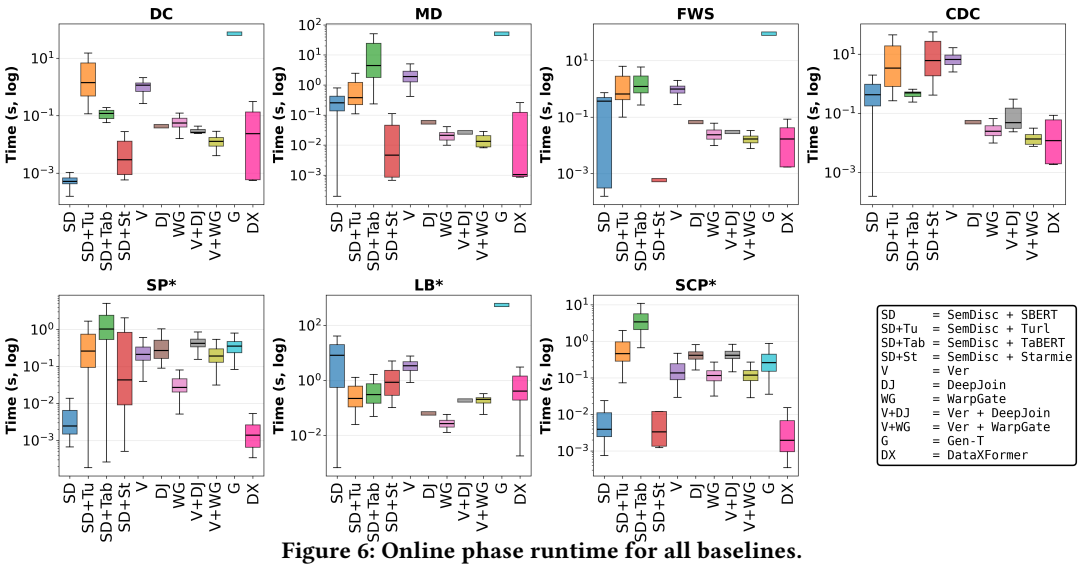

Figure 6: Online phase runtime for all baselines.

ID, causing incomplete tuple recovery. DataXFormer assumes all input columns reside within a single table and enforces one-to-one functional dependencies. Since real query tables may span multiple tables and require many-to-many joins, DataXFormer can only recover limited join paths. SemDisc removes these constraints and does not assume any initial order of tables in a target join path, and fetches the best join path that matches the query table from the join path index.

Comparison of Semantic-join Baselines. In Figure 5, competing semantic join baselines underperform even with the semantic join workloads compared to SemDisc primarily due to the poor initial selection of candidate tables. DeepJoin and WarpGate embed entire columns using fine-tuned language models, but query tables contain only a few example values (five), yielding weak context and poor candidate column alignment. Both systems also fail to verify tuple-level matches between query and join-path results, as both models operate at the column-level granularity. Ver+WarpGate and Ver+DeepJoin combine Ver's join-graph traversal with semantic joinability models, but still impose arbitrary table-pair orderings to find a join path, which lowers precision. In contrast, SemDisc explores join paths without a fixed ordering, allowing for optimal candidate table alignment and a higher number of semantically valid joins.

Comparison of Embedding Models. As shown in Figure 5, SemDisc +SBERT consistently outperforms SemDisc +TaBERT, SemDisc +TURL, and SemDisc +Starmie.

TaBERT underperforms because it was pretrained for text-to-SQL tasks, where it jointly encodes tables and natural-language utterances. In our setting, where the input is a query table rather than text, TaBERT lacks the contextual grounding its encoder expects, resulting in weaker representations for Query-by-Example (QbE) join discovery.

TURL performs better than TaBERT since its pretraining tasks—entity linking and type annotation—are closer to joinability reasoning [22]. However, it was not fine-tuned to perform join path discovery from input tables.

Starmie, in turn, produces column-level embeddings optimized for unionability rather than joinability. When used within SemDisc, Starmie's column-contrastive training yields noisier, less discriminative cell-level embeddings, making tuple-level semantic matching unreliable. Even adapting SemDisc to use Starmie's per-column vectors would still require cell-level embedding comparisons during materialization, negating efficiency benefits.

By contrast, SBERT combined with SimHash offers both semantic fidelity and scalability. SimHash (1) converts SBERT's semantic similarity into exact-value lookups, (2) supports standard cardinality estimation during join-path indexing by treating semantically similar pairs as equi-joinable (because they would have the same SimHash code), (3) enables MinHash-based column signatures for fast joinability estimation, and (4) simplifies path ranking under the Semantic Tuple Match criterion.

SemDisc leverages semantic types to narrow candidate columns, applies SimHash pruning for scalable matching, and prioritizes join paths that preserve query-table tuples. These design choices yield higher precision in semantic tuple matching than all competing baselines.

We omit Ver+DeepJoin results on the SP query set due to exceeding the 400 GB memory limit. Gen-T results on CDC and LB are also excluded as execution surpassed the 24-hour limit. Gen-T retrieves only one table per query, and DataXFormer guarantees that any retrieved path, if it exists, includes the tuples of the query tables. Therefore, we do not report P@K for K = 10, 15, 20, since these are identical to P@5 for both methods.

		DC	MD	FWS	CDC	SP	LB	SCP
ⓐ Offline Phase Time (minute)		1	36	105	87	73	41	212
ⓑ Ground Truth Parameter	θ	0.90	0.90	0.90	0.90	0.90	0.90	0.90
ⓒ SemDisc Parameter	θ_j	0.50	0.50	0.70	0.99	0.60	0.99	0.70
	λ	20	30	30	30	20	30	20
	b	18	16	20	16	18	20	18
ⓓ Total Join Paths		2.7K	3.8M	6.3M	2.7M	0.6M	9.6M	25.7M
ⓔ Satisfactory Join Path Per Query	mean	2.23	9.60	37.22	20.84	1.84	29.44	3.27
	std	1.74	11.45	31.16	18.37	0.88	28.02	0.01
ⓕ Before Prune Edge Count		4.1K	57K	640K	473K	23K	1.7M	0.8M
ⓖ After Prune Edge Count		**77**	**2.8K**	**5.4K**	**423**	**1.2K**	**3K**	**34K**

Table 4: Offline time, hyperparameters, average join path count per query and edge counts.

Effectiveness of SimHash Approximation for Semantic Joins. We evaluate SimHash for detecting semantically joinable tuples

between column pairs. From each data lake, we sample 200 random column pairs and enumerate all semantically joinable tuple pairs using Semantic Match (Equation 1) and denote them as ground truth joinable tuple pairs. We then measure how well SimHash recovers these pairs. Using Equation 7, we compute F1 scores while varying the cosine similarity threshold θ in Equation 1, as shown in Figure 7(top row).

- **True Positive (TP):** Joinable tuple pairs identified by both SimHash and the ground truth.
- **False Positive (FP):** SimHash-only detected pairs.
- **False Negative (FN):** Ground truth-only detected pairs.

Each curve corresponds to a SimHash bit count b; we omit curves that are redundant or dominated. For $\theta = 0.9$—a value used in prior semantic-join work [22]—the best average F1 across data lakes is 0.96 for the b values in Table 4ⓒ, showing that SimHash accurately recovers semantically joinable rows.

Table 4ⓑ reports the cosine threshold θ used to construct the ground-truth join graph. Table 4ⓒ lists key SemDisc parameters: the approximate semantic-join threshold θ_j, the number of candidate columns per query column λ, and the SimHash bit count b. We use 128 MinHash functions (the *datasketch* default) [4]. For ground-truth join paths, we cap the path length $\mathcal{L}$ at 5. We also describe how to tune θ_j, b, and λ for new data lakes to improve join-path quality.

Uniqueness of Join Paths for Query Tables. In our experiments, the baseline methods aim to retrieve join paths for a given query table. Multiple join paths may satisfy this objective by containing the query table's columns and tuples after materialization. Table 4ⓓ reports the total number of join paths identified for each data lake, while Table 4ⓔ presents the average number and standard deviation (std) of join paths retrieved per query table. Data lakes and benchmarks derived from open data repositories (e.g., FWS, CDC, and LB) often contain tables with duplicate schemas or overlapping content, leading to higher average counts of join paths containing the query table's tuples and columns. SemDisc efficiently retrieves the required join paths, even in data lakes with join path counts in the millions.

Takeaways: (1) Because it is able to index high-quality join paths, the average precision of SemDisc is over 3× compared to state-of-the-art QbE and join discovery methods. (2) Table-aware embedding models fall short in tabular QbE tasks, where we have to go from a limited set of rows, and produce join paths that include hidden tables. (3) SimHash approximates semantic joins accurately, reaching an average F1-score of 0.96 in approximating semantic joins in all data lakes.

7.2 Efficiency of SemDisc

Offline and Online Phase Runtime. Table 4ⓐ reports the total build time of SemDisc across all data lakes. The offline cost, dominated by Join Path Index construction, can be reduced by (1) increasing pruning thresholds to remove weak edges and (2) limiting the maximum path length, both of which shrink the join graph and speed up indexing.

Figure 6 reports the runtime of all queries and baselines. Even though Ver does not detect semantic joins, it shows the worst query runtimes because it exhaustively searches for join paths between pairs of candidate tables using DFS during query time. On average, DataXFormer and WarpGate are the fastest. DataXFormer omits many-to-many joins while searching for join paths and only searches for join paths between the source columns and the target column, thereby significantly reducing runtime. WarpGate's SimHash encoding is generally faster than embedding extraction (DeepJoin variants), querying the Inverted Join Path Index and path ranking (SemDisc variants), or exhaustive path traversal (Ver variants).

Between the Ver variants, the base Ver suffers from high runtime due to a large amount of false positive candidate tables. Ver searches for columns that contain at least one query value as a substring, thereby expanding the search space. Ver+DeepJoin and Ver+WarpGate fetch the columns that contain all the example values semantically using embeddings and SimHash, achieving better accuracy and minimizing false positive matches.

Gen-T exhibits the highest runtime because, after query submission, it greedily performs a breadth-first search (BFS) to augment the ID column to all candidate tables lacking it and then materializes all these augmented tables to join them using the ID column, resulting in substantial runtime overhead.

Although the embedding models require comparable time for embedding extraction, the runtime differences among SemDisc and its variants primarily stem from variations in the constructed join graphs and the corresponding join path indexes. A larger join path index introduces a larger query runtime for these baseline variants.

Because SemDisc is an end-to-end system with multiple steps to answer queries (Section 6), its query runtime can sometimes be slightly worse than other baselines. However, as seen in the results presented in Section 7.1, this slightly added overhead results in join paths that are far better in quality to answer QbE queries.

Scalability of SemDisc. The offline indexing cost of SemDisc for any target data lake is primarily determined by the Join Path Index construction. As shown in Table 4ⓐ, the offline phase runtime generally increases with the total number of join paths across data lakes (Table 4ⓓ).

Effectiveness of Join Graph Pruning. Table 4ⓕⓖ reports the change in edge count in the join graph before and after pruning. As we can observe, pruning significantly reduces the number of edges in the join graph by orders of magnitude, which not only improves query runtime but also facilitates the return of high-quality join paths, as demonstrated in the previous experiments.

Takeaways: (1) SemDisc trades slightly longer query time for higher-quality results. (2) Our join graph pruning strategy significantly reduces the edge count of the join graph while retaining high-quality paths.

7.3 Hyperparameter Selection Experiments

Join Graph Approximation. To select hyperparameters for SemDisc on a new data lake, we first construct a ground-truth join graph using $\theta = 0.9$. We then build SemDisc 's join graph for $\theta_j \in [0.1, 0.9]$ and SimHash bit counts $b \in [8, 30]$, and compare against the ground truth.

We measure quality using the F1 score (Equation 7), where:

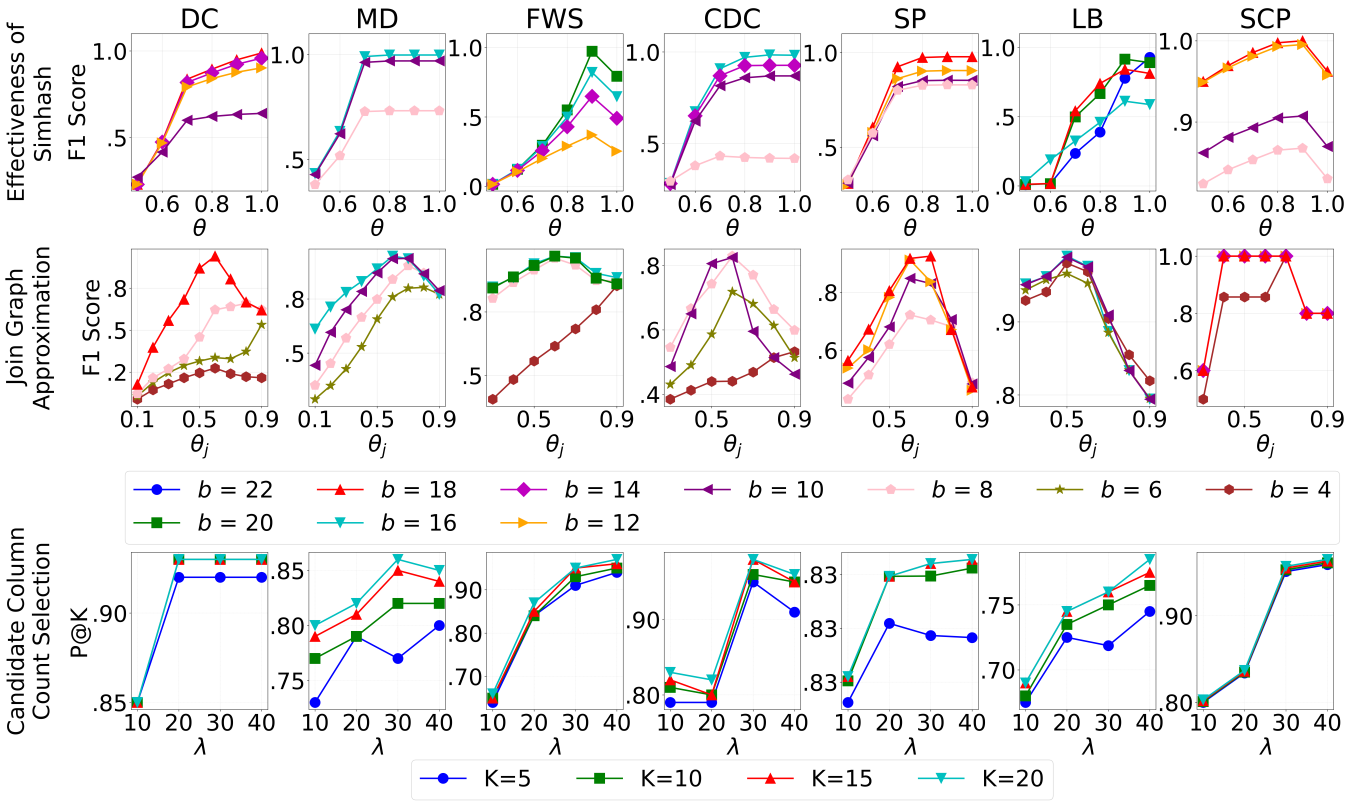

Figure 7: (top) SimHash effectiveness on detecting joinable tuples for varying θ, (middle) join graph approximation using SimHash and MinHash for varying θ_j, (bottom) P@K vs candidate column count (λ).

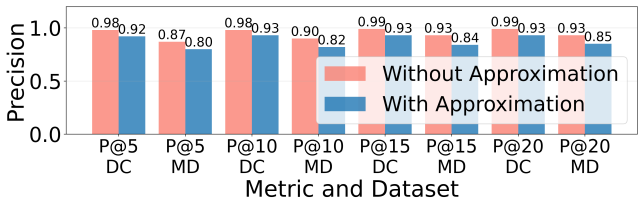

Figure 8: P@K for queries with and without approximation using SimHash and MinHash

- **True Positive (TP):** edges appear in both SemDisc 's graph and the ground-truth graph.
- **False Positive (FP):** edges appear in SemDisc 's graph but not in the ground truth.
- **False Negative (FN):** edges appear in the ground-truth graph but are missing from SemDisc 's graph.

Figure 7(middle row) shows the F1 scores across hyperparameter settings on multiple data lakes. The x-axis varies the approximate semantic joinability threshold θ_j; each line corresponds to a different SimHash bit count b. We omit curves that are strictly dominated.

SimHash Bit Count and Joinability Threshold. For each data lake, we first identify the curve that attains the highest F1 score and take its SimHash bit count b as the best setting. These values are reported in Table 4ⓒ.

Given b, we then choose the θ_j value on that curve that achieves the highest F1. From Figure 7(middle row), a threshold in the range $\theta_j \in [0.5, 0.6]$ is sufficient to recover the join graph with high accuracy. In our experiments we use slightly higher θ_j values (Table 4ⓒ) to prune more aggressively and reduce runtime in both the offline and online stages.

Candidate Column Count. At query time, SemDisc uses the HNSW index to retrieve the top λ candidate columns for each query column. Figure 7(bottom row) shows P@K as we vary λ. The curves have an elbow at some λ, after which gains are marginal. The values of λ used in our online phase are listed in Table 4ⓒ. As a rule of thumb, for data lakes with about 100 tables, we recommend $\lambda \in [10, 20]$. For larger data lakes with more than 200 tables, $\lambda = 30$ is adequate to surface joinable tables.

Effect of SimHash and MinHash on P@K. Using SimHash and MinHash (Section 4.2) introduces a small precision loss but yields a large efficiency gain compared to computing joinability directly from embeddings. In this experiment we disable sketching and compute pairwise column joinability using the raw embeddings (Equation 3) when building the join-path index and answering queries. Figure 8 reports the change in P@K. On average, sketching reduces precision by only 0.07, showing that SimHash and MinHash preserve joinability signals while enabling much faster indexing and query processing.

8 CONCLUSION AND FUTURE DIRECTIONS

In this paper, we introduced SemDisc, an end-to-end system for QbE join discovery from data lakes. SemDisc currently only supports joins. However, integrating other data discovery operators, such as unions, would also be useful, presenting a new set of technical challenges to answer qualitative QbE joinability and unionability

queries. As part of future work, we plan to extend SemDisc to support different join paths across tuple subsets of the same column. Additionally, supporting self-joins presents further challenges, such as managing repeated tables in join paths and ensuring consistent tuple validation. Extending SemDisc to address these aspects will improve its ability to discover more relational structures within large data lakes.

REFERENCES

[1] [n. d.]. The Home of the U.S. Government's Open Data. https://data.gov/.

[2] [n. d.]. MITDWH. https://ist.mit.edu/warehouse.

[3] 2002. DBXplorer: A System for Keyword-Based Search over Relational Databases. In *Proceedings 18th International Conference on Data Engineering (ICDE '02)*. USA, 5–16. https://doi.org/10.1109/ICDE.2002.994693

[4] 2025. datasketch Python library. https://github.com/ekzhu/datasketch.

[5] Ziawasch Abedjan, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, and Michael Stonebraker. 2016. DataXFormer: A robust transformation discovery system. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 1134–1145. https://doi.org/10.1109/ICDE.2016.7498319

[6] Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. 1999. Join synopses for approximate query answering. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*. 275–286.

[7] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[8] Yael Amsterdamer and Moran Cohen. 2021. Automated Selection of Multiple Datasets for Extension by Integration. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 27–36.

[9] Lennart Behme, Sainyam Galhotra, Kaustubh Beedkar, and Volker Markl. 2024. Fainder: A Fast and Accurate Index for Distribution-Aware Dataset Search. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3269–3282.

[10] Alex Bogatu, Alvaro AA Fernandes, Norman W Paton, and Nikolaos Konstantinou. 2020. Dataset discovery in data lakes. In *2020 ieee 36th international conference on data engineering (icde)*. IEEE, 709–720.

[11] A.Z. Broder. 1997. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*. 21–29. https://doi.org/10.1109/SEQUEN.1997.666900

[12] Sonia Castelo, Rémi Rampin, Aécio Santos, Aline Bessa, Fernando Chirigati, and Juliana Freire. 2021. Auctus: A dataset search engine for data discovery and augmentation. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2791–2794.

[13] Moses S Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. 380–388.

[14] S. Chaudhuri, V. Ganti, and R. Kaushik. 2006. A Primitive Operator for Similarity Joins in Data Cleaning. In *22nd International Conference on Data Engineering (ICDE'06)*. 5–5. https://doi.org/10.1109/ICDE.2006.9

[15] Lu Chen, Yunjun Gao, Baihua Zheng, Christian S. Jensen, Hanyu Yang, and Keyu Yang. 2017. Pivot-based metric indexing. *Proc. VLDB Endow.* 10, 10 (June 2017), 1058–1069. https://doi.org/10.14778/3115404.3115411

[16] Martin Pekár Christensen, Aristotelis Leventidis, Matteo Lissandrini, Laura Di Rocco, Renée J Miller, and Katja Hose. 2025. Fantastic Tables and Where to Find Them: Table Search in Semantic Data Lakes. (2025).

[17] Tianji Cong, James Gale, Jason Frantz, H. V. Jagadish, and Çagatay Demiralp. 2023. WarpGate: A Semantic Join Discovery System for Cloud Data Warehouses. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org.

[18] Xiang Deng, Huan Sun, Alyssa Lees, You Wu, and Cong Yu. 2020. TURL: table understanding through representation learning. *Proceedings of the VLDB Endowment* 14, 3 (2020), 307–319.

[19] Yuhao Deng, Chengliang Chai, Lei Cao, Qin Yuan, Siyuan Chen, Yanrui Yu, Zhaoze Sun, Junyi Wang, Jiajun Li, Ziqi Cao, et al. 2024. LakeBench: A Benchmark for Discovering Joinable and Unionable Tables in Data Lakes. *Proceedings of the VLDB Endowment* 17, 8 (2024), 1925–1938.

[20] Till Döhmen, Radu Geacu, Madelon Hulsebos, and Sebastian Schelter. 2024. SchemaPile: A Large Collection of Relational Database Schemas. *Proceedings of the ACM on Management of Data* 2, 3 (2024).

[21] Yuyang Dong, Kunihiro Takeoka, Chuan Xiao, and Masafumi Oyamada. 2021. Efficient joinable table discovery in data lakes: A high-dimensional similarity-based approach. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 456–467.

[22] Yuyang Dong, Chuan Xiao, Takuma Nozawa, Masafumi Enomoto, and Masafumi Oyamada. 2023. DeepJoin: Joinable Table Discovery with Pre-Trained Language Models. *Proc. VLDB Endow.* 16, 10 (June 2023), 2458–2470. https://doi.org/10.14778/3603581.3603587

[23] Mahdi Esmailoghli, Jorge-Arnulfo Quiané-Ruiz, and Ziawasch Abedjan. 2021. *Cocoa: Correlation coefficient-aware data augmentation*. Konstanz, Germany: OpenProceedings. org, University of Konstanz, University ....

[24] Mahdi Esmailoghli, Jorge-Arnulfo Quiané-Ruiz, and Ziawasch Abedjan. 2022. MATE: multi-attribute table extraction. *Proc. VLDB Endow.* 15, 8 (April 2022), 1684–1696. https://doi.org/10.14778/3529337.3529353

[25] Mahdi Esmailoghli, Christoph Schnell, Renee J. Miller, and Ziawasch Abedjan. 2025. BLEND: A Unified Data Discovery System . In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 737–750. https://doi.org/10.1109/ICDE65448.2025.00061

[26] Grace Fan, Roee Shraga, and Renee J. Miller. 2024. Gen-T: Table Reclamation in Data Lakes . In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 3532–3545. https://doi.org/10.1109/ICDE60146.2024.00272

[27] Grace Fan, Jin Wang, Yuliang Li, Dan Zhang, and Renée J. Miller. 2023. Semantics-Aware Dataset Discovery from Data Lakes with Contextualized Column-Based Representation Learning. *Proc. VLDB Endow.* 16, 7 (March 2023), 1726–1739. https://doi.org/10.14778/3587136.3587146

[28] Wenfei Fan, Ping Lu, Kehan Pang, Ruochun Jin, and Wenyuan Yu. 2024. Linking entities across relations and graphs. *ACM Transactions on Database Systems* 49, 1 (2024), 1–50.

[29] Anna Fariha and Alexandra Meliou. 2019. Example-Driven Query Intent Discovery: Abductive Reasoning using Semantic Similarity. *PVLDB* 12, 11 (2019).

[30] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A data discovery system. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, ICDE, ICDE, 1001–1012.

[31] Raul Castro Fernandez, Essam Mansour, Abdulhakim A Qahtan, Ahmed Elmagarmid, Ihab Ilyas, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2018. Seeping semantics: Linking datasets using word embeddings for data discovery. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, ICDE, ICDE, 989–1000.

[32] Raul Castro Fernandez, Jisoo Min, Demitri Nava, and Samuel Madden. 2019. Lazo: A cardinality-based method for coupled estimation of jaccard similarity and containment. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1190–1201.

[33] Benjamin Feuer, Yurong Liu, Chinmay Hegde, and Juliana Freire. 2024. ArcheType: A Novel Framework for Open-Source Column Type Annotation Using Large Language Models. *Proc. VLDB Endow.* 17, 9 (May 2024), 2279–2292. https://doi.org/10.14778/3665844.3665857

[34] Sainyam Galhotra, Yue Gong, and Raul Castro Fernandez. 2023. Metam: Goal-Oriented Data Discovery . In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 2780–2793. https://doi.org/10.1109/ICDE55515.2023.00213

[35] Sainyam Galhotra and Udayan Khurana. 2020. Semantic search over structured data. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 3381–3384.

[36] Yue Gong, Sainyam Galhotra, and Raul Castro Fernandez. 2024. Nexus: Correlation Discovery over Collections of Spatio-Temporal Tabular Data. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–28.

[37] Yue Gong, Zhiru Zhu, Sainyam Galhotra, and Raul Castro Fernandez. 2023. Ver: View discovery in the wild. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, ICDE, ICDE, 503–516.

[38] Yuxiang Guo, Yuren Mao, Zhonghao Hu, Lu Chen, and Yunjun Gao. 2025. Snoopy: Effective and Efficient Semantic Join Discovery via Proxy Columns . *IEEE Transactions on Knowledge & Data Engineering* 37, 05 (May 2025), 2971–2985. https://doi.org/10.1109/TKDE.2025.3545176

[39] Ahmed Helal, Mossad Helali, Khaled Ammar, and Essam Mansour. 2021. A demonstration of kglac: A data discovery and enrichment platform for data science. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2675–2678.

[40] Jonathan Herzig, Thomas Müller, Syrine Krichene, and Julian Eisenschlos. 2021. Open Domain Question Answering over Tables via Dense Retrieval. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou (Eds.). Association for Computational Linguistics, Online, 512–519. https://doi.org/10.18653/v1/2021.naacl-main.43

[41] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: learn from data, not from queries! *Proc. VLDB Endow.* 13, 7 (March 2020), 992–1005. https://doi.org/10.14778/3384345.3384349

[42] Richard M. Karp. 1972. *Reducibility among Combinatorial Problems*. Springer US, Boston, MA, 85–103. https://doi.org/10.1007/978-1-4684-2001-2_9

[43] Aamod Khatiwada, Grace Fan, Roee Shraga, Zixuan Chen, Wolfgang Gatterbauer, Renée J Miller, and Mirek Riedewald. 2023. Santos: Relationship-based semantic table union search. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–25.

[44] Aamod Khatiwada, Roee Shraga, Wolfgang Gatterbauer, and Renée J Miller. 2022. Integrating data lake tables. *Proceedings of the VLDB Endowment* 16, 4 (2022), 932–945.

[45] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. 2022. Learned cardinality estimation: An in-depth study. In *Proceedings of the 2022 international conference on management of data*. 1214–1227.

[46] Kyoungmin Kim, Sangoh Lee, Injung Kim, and Wook-Shin Han. 2024. Asm: Harmonizing autoregressive model, sampling, and multi-dimensional statistics merging for cardinality estimation. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–27.

[47] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf

[48] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling.. In *Cidr*.

[49] Hao Li, Chee-Yong Chan, and David Maier. 2015. Query from examples: an iterative, data-driven approach to query construction. *Proc. VLDB Endow.* 8, 13 (Sept. 2015), 2158–2169. https://doi.org/10.14778/2831360.2831369

[50] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, and Wang-Chiew Tan. 2020. Deep entity matching with pre-trained language models. *Proc. VLDB Endow.* 14, 1 (Sept. 2020), 50–60. https://doi.org/10.14778/3421424.3421431

[51] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems* 45 (2014), 61–68. https://doi.org/10.1016/j.is.2013.10.006

[52] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (April 2020), 824–836. https://doi.org/10.1109/TPAMI.2018.2889473

[53] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1301.3781

[54] Marius Muja and David G. Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 11 (2014), 2227–2240. https://doi.org/10.1109/TPAMI.2014.2321376

[55] Fatemeh Nargesian, Ken Q. Pu, Erkang Zhu, Bahar Ghadiri Bashardoost, and Renée J. Miller. 2020. Organizing Data Lakes for Navigation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1939–1950. https://doi.org/10.1145/3318464.3380605

[56] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. 2019. Data lake management: challenges and opportunities. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 1986–1989. https://doi.org/10.14778/3352063.3352116

[57] Paul Ouellette, Aidan Sciortino, Fatemeh Nargesian, Bahar Ghadiri Bashardoost, Erkang Zhu, Ken Q Pu, and Renée J Miller. 2021. RONIN: data lake exploration. *Proceedings of the VLDB Endowment* 14, 12 (2021).

[58] Fotis Psallidas, Bolin Ding, Kaushik Chakrabarti, and Surajit Chaudhuri. 2015. S4: Top-k Spreadsheet-Style Search for Query Discovery. In *SIGMOD*. 2001–2016.

[59] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, Hong Kong, China, 3982–3992. https://doi.org/10.18653/v1/D19-1410

[60] El Kindi Rezig, Anshul Bhandari, Anna Fariha, Benjamin Price, Allan Vanterpool, Vijay Gadepally, and Michael Stonebraker. 2021. DICE: data discovery by example. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2819–2822.

[61] Yanyan Shen, Kaushik Chakrabarti, Surajit Chaudhuri, Bolin Ding, and Lev Novik. 2014. Discovering Queries Based on Example Tuples. In *SIGMOD*. 493–504.

[62] Ji Sun and Guoliang Li. 2019. An end-to-end learning-based cost estimator. *Proc. VLDB Endow.* 13, 3 (Nov. 2019), 307–319. https://doi.org/10.14778/3368289.3368296

[63] Yushi Sun, Hao Xin, and Lei Chen. 2023. RECA: Related Tables Enhanced Column Semantic Type Annotation Framework. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1319–1331. https://doi.org/10.14778/3583140.3583149

[64] Oleg Ursu, Jayme Holmes, Cristian G Bologa, Jeremy J Yang, Stephen L Mathias, Vasileios Stathias, Dac-Trung Nguyen, Stephan Schürer, and Tudor Oprea. 2019. DrugCentral 2018: an update. *Nucleic acids research* 47, D1 (2019), D963–D970.

[65] Jin Wang, Chunbin Lin, and Carlo Zaniolo. 2019. Mf-join: Efficient fuzzy string similarity join with multi-level filtering. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 386–397.

[66] Qiming Wang and Raul Castro Fernandez. 2023. Solo: Data Discovery Using Natural Language Questions Via A Self-Supervised Approach. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–27.

[67] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are we ready for learned cardinality estimation? *Proc. VLDB Endow.* 14, 9 (May 2021), 1640–1654. https://doi.org/10.14778/3461535.3461552

[68] Yubo Wang, Hao Xin, and Lei Chen. 2024. KGLink: A Column Type Annotation Method that Combines Knowledge Graph and Pre-Trained Language Model . In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1023–1035. https://doi.org/10.1109/ICDE60146.2024.00083

[69] Jiacheng Wu, Yong Zhang, Jin Wang, Chunbin Lin, Yingjia Fu, and Chunxiao Xing. 2019. Scalable metric similarity join using mapreduce. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1662–1665.

[70] Peizhi Wu and Gao Cong. 2021. A unified deep model of learning from both data and queries for cardinality estimation. In *Proceedings of the 2021 International Conference on Management of Data*. 2009–2022.

[71] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: one cardinality estimator for all tables. *Proc. VLDB Endow.* 14, 1 (Sept. 2020), 61–73. https://doi.org/10.14778/3421424.3421432

[72] Pengcheng Yin, Graham Neubig, Wen tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data. In *Annual Conference of the Association for Computational Linguistics (ACL)*.

[73] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (Eds.). Association for Computational Linguistics, Brussels, Belgium, 3911–3921. https://doi.org/10.18653/v1/D18-1425

[74] Shuo Zhang and Krisztian Balog. 2018. Ad hoc table retrieval using semantic similarity. In *Proceedings of the 2018 world wide web conference*. 1553–1562.

[75] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J Miller. 2019. Josie: Overlap set similarity search for finding joinable tables in data lakes. In *Proceedings of the 2019 International Conference on Management of Data*. 847–864.

[76] Erkang Zhu, Fatemeh Nargesian, Ken Q. Pu, and Renée J. Miller. 2016. LSH ensemble: internet-scale domain search. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 1185–1196. https://doi.org/10.14778/2994509.2994534

[77] Moshe M. Zloof. 1977. Query-by-example: A data base language. *IBM systems Journal* 16, 4 (1977), 324–343.

APPENDIX

A NP-HARDNESS PROOFS

A.1 Proofs for Problem 1

Let $G = (V, E)$ be the join graph, where each vertex represents a table in the data lake and each edge $e \in E$ represents a possible join between two tables, with an associated joinability score $weight(e) \in [0, 1]$. For a query table Q, let $T_{set}(Q) \subseteq V$ be the set of tables that contain columns semantically joinable with the columns of Q. For any join tree P in G, we write $V(P)$ and $E(P)$ for its set of vertices and edges, respectively. The goal is to find a join tree $P^*(Q)$ that maximizes the total weight of selected edges while satisfying the following constraints: (1) P^* maximizes the sum of edge weights; (2) P^* includes all tables in $T_{set}(Q)$; (3) P^* may include other (hidden) tables that are needed to join the tables in $T_{set}(Q)$; and (4) $|V(P^*)| \leq \mathcal{L}$, a maximum table count budget (we want to bound how many tables we can have in a join tree).

A.1.1 Unbudgeted Weight-Optimal Join Tree for Query Table Problem. We first focus on Conditions (1)–(3) to establish NP-hardness via a reduction from the *Minimum-Weight Steiner Tree* problem. Subsequently, we show—by proof of restriction—that introducing the table budget constraint in Condition (4) preserves NP-hardness (Appendix A.1.2).

We show that this problem is NP-hard by a reduction from the *Minimum-weight Steiner Tree* problem, which is known to be NP-hard [42].

Reduction. Given an instance of the *Steiner Tree* problem defined by $G' = (V', E')$, a positive edge weight function $w : E' \to \mathbb{R}_{>0}$, a set of terminal vertices $R \subseteq V'$ (vertices that have to included in the Steiner tree), we construct an equivalent instance of our problem as follows:

(1) Let the join graph $G = (V, E)$ be identical to G', i.e., $V = V'$ and $E = E'$.
(2) For each edge $e \in E$, set its join weight as $weight(e) = -w(e)$. This preserves the relative ordering of solutions while converting the minimization objective of the Steiner Tree problem into a maximization objective.
(3) Set $T_{set}(Q) = R$, i.e., the required tables in the query correspond to the Steiner terminals.

Correctness. We now show that there exists an optimal join tree P^* of total weight W^* for the constructed instance if and only if there exists a minimum-weight Steiner tree of total weight C^* for the original instance.

($\Rightarrow$) Suppose there exists a Minimum-weight Steiner tree $H = (V_H, E_H)$ such that $R \subseteq V_H$ and $\sum_{e \in E_H} w(e) = C^*$. Since H is connected, we can define a corresponding join tree P consisting of the same edges. The total weight of P is:

$$\sum_{e \in E(P)} weight(e) = \sum_{e \in E_H} (-w(e)) = -C^*.$$

Because H connects all vertices in $R = T_{set}(Q)$ and includes only edges in G, P satisfies all join tree constraints. Thus, P is optimal with weight value $-C^*$.

($\Leftarrow$) Conversely, suppose there exists a feasible join tree P in the constructed instance with total weight $W^* = \sum_{e \in E(P)} weight(e)$. Let H be the undirected subgraph of G' induced by the edges in $E(P)$. By construction, H is connected, spans all vertices in $T_{set}(Q) = R$, and has weight

$$\sum_{e \in E_H} w(e) = -\sum_{e \in E(P)} weight(e) = -W^*.$$

Hence, if P is optimal for our problem, then H is a minimum-weight Steiner tree for the original instance.

A.1.2 Budgeted Weight-Optimal Join Tree for Query Table. Now we add a table budget: find P^* as above (in the unbudgeted weight-optimal join tree problem) with the additional constraint $|V(P^*)| \leq \mathcal{L}$ (we can only use up to $\mathcal{L}$ tables in the join tree).

Proposition 3. *The Budgeted Weight-Optimal Join Tree problem is NP-hard (by restriction).*

Proof (by restriction). Every instance of the unbudgeted problem is a special case of the budgeted problem obtained by setting $\mathcal{L} := |V|$, which makes the budget non-restrictive. Therefore, if the budgeted problem were solvable in polynomial time, so would be the unbudgeted problem. Since the unbudgeted problem is NP-hard, the budgeted problem is also NP-Hard. □

A.2 Proof for Problem 2

We prove by restriction that Problem 1 is a special case of Problem 2, and therefore, Problem 2 is at least as hard as Problem 1.

Proof (by restriction). We can reduce any instance I of Problem 1 to the Problem 2 instance I' with the same data and parameters, and either (a) an empty example set in Q (so $STM(Q, T(P^*))$ holds vacuously), or (b) similarity threshold $\theta = 0$ so $M^s_\theta(\cdot, \cdot) = 1$ and $STM(Q, T(P^*))$ always holds. Under either choice, the feasible solutions of I' are exactly those of I, and the primary objective (weight optimality) is identical. The secondary tie-break (maximize $|T(P^*)|$ among weight-optimal trees) does not change NP-hardness. Hence, Problem 1 is a special case of Problem 2, implying Problem 2 is NP-Hard. □

B EXAMPLE PROMPT FOR SEMANTIC TYPE EXTRACTION

A complete example prompt for the LLM for a table from the LakeBench (Open Data dataset) benchmark is given below, where SemDisc extracts semantic types of three sample columns `iPlayerStartupDelay24hr` and `NetflixHD24hr` with 10 example values:

```
Below is a list of column ids and a sample of their
values seperated by commas from a table. Give detailed
elaborate semantic type for each column in the
following format:

[{"id": columnid1, "semantictype":typename1},
{"id": columnid2, "semantictype":typename2}, ...]
```

```
input:

id: iPlayerStartupDelay24hr
sample values: 904.85, 740.89, 485.23, 551.3, 421.49,
557.25, 522.2, 601.27,

id: NetflixHD24hr
sample values: 0.00%, 100.00%, 0.28%, 0.29%, 0.30%,
0.85%, 0.57%, 2.27%, 0.73%, 0.31%,
```

The output of the LLM is as follows:

```
[
    {
        "id": "iPlayerStartupDelay24hr",
        "semantictype": "24-Hour iPlayer Startup Delay
        ↪ (Milliseconds)"
    },
    {
        "id": "NetflixHD24hr",
        "semantictype": "24-Hour Netflix HD Streaming Success
        ↪ Rate Percentage"
    }
]
```

C EXAMPLES OF REAL WORLD QUERY TABLES AND JOIN PATHS WITH HIDDEN TABLES

We provide examples of query tables that require join paths with hidden tables from the data lakes in Figure 9. Each example query table requires hidden tables in order to connect the candidate tables and create a join path. For brevity, we show a sample of 2-3 query rows for each query table.

D TIME COMPLEXITY ANALYSIS OF ALGORITHM 2

Building the HNSW index requires $O(|C|\log|C|)$, where C is the set of all data lake columns. Search using the HNSW index after embedding extraction requires $O(l \cdot \log|C|)$, where l is the query column count. The time complexity of filtering out candidate columns based on SimHash values is $O(l \cdot \lambda \cdot r)$- where λ is the count of data lake columns selected by semantic type comparison and r is the count of rows in the query table Q.

E EFFECTIVENESS OF SEMANTIC TYPES AND EDGE PRUNING

In the following experiment, we evaluate the benefit of semantic type detection in SemDisc and how it affects precision.

Contribution of Semantic Types to Join Path Quality. We manually annotate 50 column pairs per data lake to determine *joinability* based on semantic type similarity and value overlap. For instance, in the MD data lake, columns such as `(is course taken)` and `(is a senior)` exhibit high value overlap but differ semantically, and are thus non-joinable. In contrast, `(is course taken)` and `(is enrolled in course)` share similar semantics and are joinable.

We compare two filtering techniques before computing value overlap:

(1) Embedding similarity between default column headers, and
(2) Embedding similarity between LLM-extracted semantic types.

A column pair is considered for value overlap if its cosine similarity exceeds the threshold θ.

We evaluate both techniques using the following metrics:

$$\text{Accuracy (ACC)} = \frac{TP + TN}{TP + TN + FP + FN}$$

$$\text{Precision (PR)} = \frac{TP}{TP + FP}$$

$$\text{Recall (R)} = \frac{TP}{TP + FN}$$

$$\text{F1-score (F1)} = 2 \times \frac{PR \times R}{PR + R}$$

Here,

- TP denotes *true positives* (joinable pairs correctly predicted as joinable),
- TN denotes *true negatives* (non-joinable pairs correctly predicted as non-joinable),
- FP denotes *false positives* (non-joinable pairs incorrectly predicted as joinable), and
- FN denotes *false negatives* (joinable pairs incorrectly predicted as non-joinable).

Figure 10 shows that detecting joinable column pairs using semantic types outperforms using the column headers as semantic types in the data lakes. Detecting joinable columns based on column header similarity gives numerous false positives. For example, in the DrugCentral data lake, two 'ID' columns from the 'drugs' and 'pharmacies' tables have the same column headers and contain values with high overlap. However, the two columns indicate 'Drug ID' and 'Pharmacy ID', which are unsuitable for joining. Comparing column header similarity would mark this column pair as joinable, whereas using semantic types alongside semantic joinability would detect the subtle semantic type difference between the columns. The semantic types are extracted with prompts to the LLM containing the context of the entire table; therefore, they are rich in context compared to the default column headers, which is why semantic types are better at detecting joinable column pairs.

Takeaways: Semantic types provide the best accuracy in finding suitable join paths across multiple tables than using default column headers.

		DC	MD	FWS	CDC	SP	LB
ⓐ Candidate Table to Path (fraction of queries w/ correct path)	SD	**0.93**	**1.00**	**1.00**	**0.96**	**0.90**	**0.83**
	V	0.00	0.07	0.00	0.00	0.72	0.33
	DJ	0.10	0.06	0.33	0.06	0.72	0.78
	WG	0.14	0.03	0.35	0.21	0.86	0.49

Table 5: Fraction of queries with correct join paths detected by baselines only from candidate tables (SD=SemDisc, V=Ver, DJ=DeepJoin, WG=WarpGate)

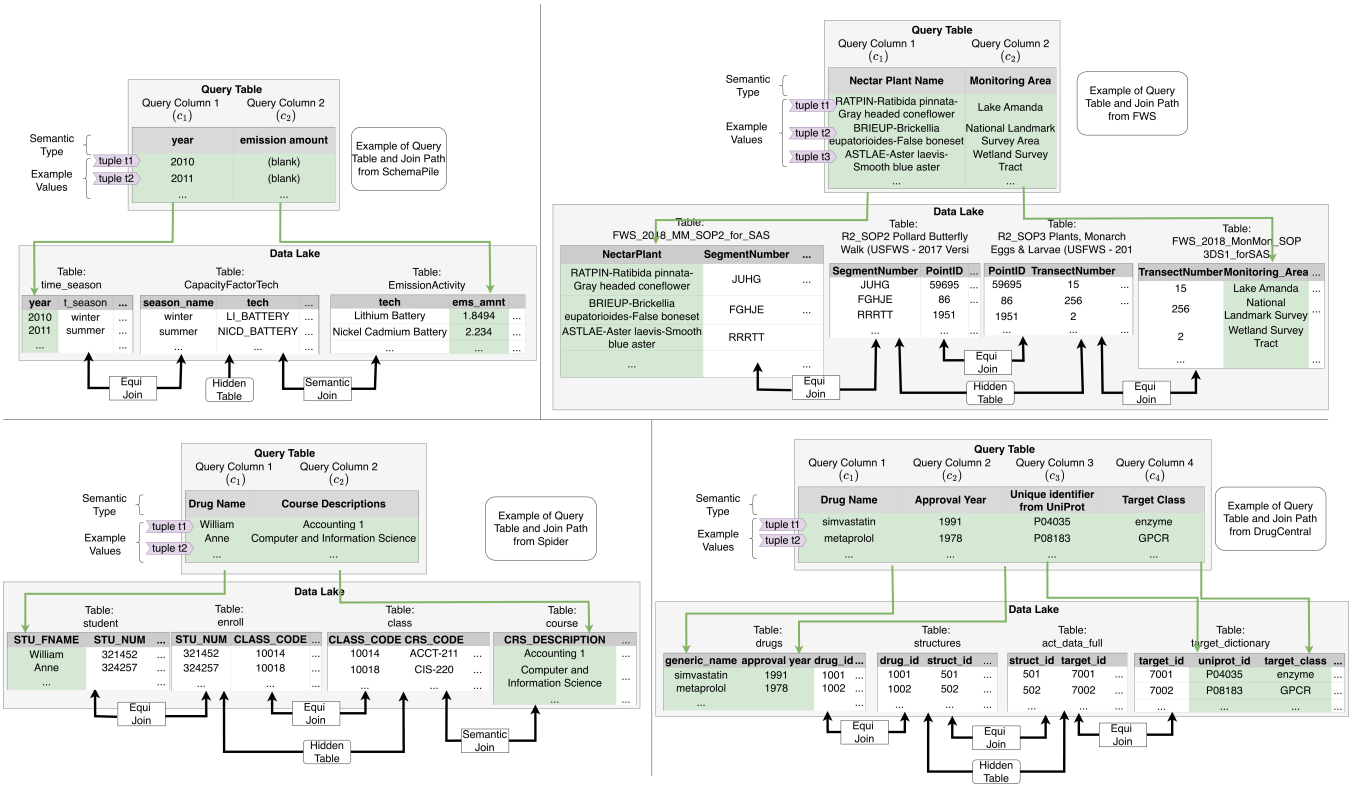

Figure 9: Examples of query tables and their corresponding join paths with hidden tables

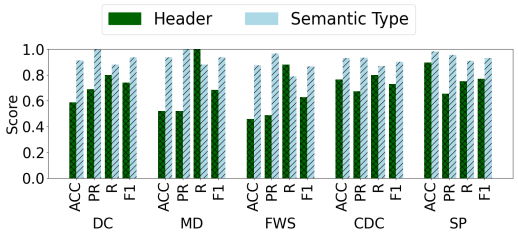

Figure 10: Metrics of joinable column detection by headers vs. semantic types

F CANDIDATE TABLE TO JOIN PATH DETECTION

In the end-to-end evaluation (Section 7.1), the baselines translate the query columns into candidate tables, then find the join paths between them, and report on whether the semantic tuple match condition holds in the top K returned join paths. In this experiment, we evaluate SemDisc and the baselines on their ability to find join paths only from the ground-truth candidate tables, thereby removing the query column to candidate column mapping and the semantic tuple match evaluation. Instead of giving the query tables as input, we directly give the baselines the candidate columns and report the effectiveness of finding the join path containing the candidate tables. Table 5 reports the fraction of queries for which the baselines found a path that contains all the candidate tables (1 means all candidate tables were found).

DeepJoin and WarpGate generally have a low fraction of found join paths because they find joins between the candidate tables directly without considering any hidden tables that might be needed but were not mentioned in the query table. However, for SP, all competing baselines are able to find over 70% of the target join paths. This is because SP has a small number of tables per database, and the number of rows per table is small. Ver arbitrarily orders the candidate tables first, and then finds join paths between pairs of candidate tables. This arbitrary ordering of the candidate tables does not guarantee finding the best join path for a given query table. SemDisc indexes all possible join paths in the Inverted Join Path Index, making it possible to find the best join path without making any assumption on the ordering between candidate tables.