BIRDIE: Natural Language-Driven Table Discovery Using Differentiable Search Index

Yuxiang Guo
Zhejiang University
guoyx@zju.edu.cn

Zhonghao Hu
Zhejiang University
zhonghao.hu@zju.edu.cn

Yuren Mao
Zhejiang University
yuren.mao@zju.edu.cn

Baihua Zheng
Singapore Management University
bhzheng@smu.edu.sg

Yunjun Gao
Zhejiang University
gaoyj@zju.edu.cn

Mingwei Zhou
Zhejiang Dahua Technology Co., Ltd
zhoumingwei_hz@163.com

ABSTRACT

Natural language (NL)-driven table discovery identifies relevant tables from large table repositories based on NL queries. While current deep-learning-based methods using the traditional dense vector search pipeline, i.e., *representation-index-search*, achieve remarkable accuracy, they face several limitations that impede further performance improvements: (i) the errors accumulated during the table representation and indexing phases affect the subsequent search accuracy; and (ii) insufficient query-table interaction hinders effective semantic alignment, impeding accuracy improvements. In this paper, we propose a novel framework BIRDIE, using a differentiable search index. It unifies the indexing and search into a single encoder-decoder language model, thus getting rid of error accumulations. BIRDIE first assigns each table a prefix-aware identifier and leverages a large language model-based query generator to create synthetic queries for each table. It then encodes the mapping between synthetic queries/tables and their corresponding table identifiers into the parameters of an encoder-decoder language model, enabling deep query-table interactions. During search, the trained model directly generates table identifiers for a given query. To accommodate the continual indexing of dynamic tables, we introduce an index update strategy via parameter isolation, which mitigates the issue of catastrophic forgetting. Extensive experiments demonstrate that BIRDIE outperforms state-of-the-art dense methods by 16.8% in accuracy, and reduces forgetting by over 90% compared to other continual learning approaches.

PVLDB Reference Format:
Yuxiang Guo, Zhonghao Hu, Yuren Mao, Baihua Zheng, Yunjun Gao, Mingwei Zhou. BIRDIE: Natural Language-Driven Table Discovery Using Differentiable Search Index. PVLDB, 18(X): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ZJU-DAILY/BIRDIE.

Proceedings of the VLDB Endowment, Vol. 18, No. 8 ISSN 2150-8097.
doi:XX.XX/XXX.XX

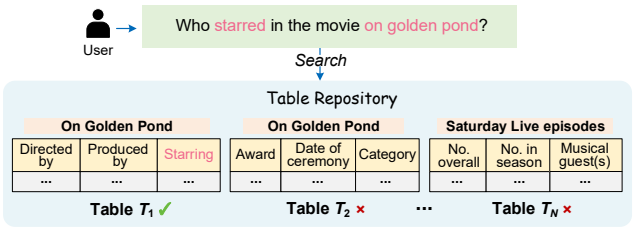


Figure 1: An example of NL-driven table discovery.

1 INTRODUCTION

Tables are a prevalent format for data storage across governmental institutions [39], businesses [34], and the Web [11]. They contain vast amounts of information that can drive decision-making [6]. However, the sheer volume of tabular data complicates the process for users to locate relevant tables in large repositories or data lakes [12]. In response, the data management community has developed table discovery methods that allow users to search for tables based on various query formats, such as keywords [3, 5], base tables [14, 16], and natural language queries [22, 52]. Natural language (NL) queries, in particular, are user-friendly and empower non-technical users to express their needs more precisely. As an example shown in Figure 1, assume that a user wants to know who starred in the movie "on golden pond". NL-driven table discovery aims to identify T_1 from the large table repository, as it contains a cell to answer this query. Once table T_1 is retrieved, tools like NL2SQL [58] or Large Language Models [60] can be used to formulate a response to the query.

The success of deep learning techniques in various fields has led to the emergence of the state-of-the-art (SOTA) NL-driven table discovery methods [22, 52], which typically follow the traditional dense vector search pipeline involving *representation*, *indexing* and *search*. As illustrated in Figure 2(a), a bi-encoder system is trained for representation, comprising a table encoder and a query encoder. The table encoder transforms each table into a fixed-dimensional embedding, and the indexes are constructed on these embeddings. During the search phase, the query embedding is generated using the query encoder, and the nearest neighbor search (NNS) or approximate NNS (ANNS) is conducted to locate table embeddings that closely resemble the query embedding, leveraging pre-constructed indexes. Although this paradigm has significantly outperformed sparse methods (e.g. BM25) [52], it faces two major limitations.

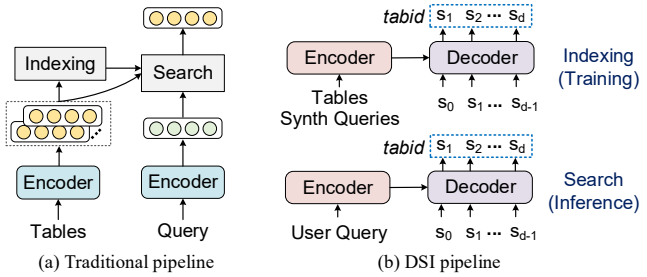


Figure 2: Traditional dense pipeline vs. DSI pipeline.

- **Cumulative Errors from the Multi-Stage Process**. The separation of representation, indexing, and search results in the accumulation of errors from one stage to the next. First, capturing the rich and complex information within tables using a single vector remains a challenge [4, 10]. Insufficient table representations degrade the effectiveness of search results. Second, the choice of indexing techniques can greatly impact the search results; for instance, discrepancies between the inverted index and ANNS index can negatively affect the accuracy [16].
- **Insufficient Query-Table Interactions**. Encoding the query-table pair using a cross-encoder enables deep query-table interactions, thus enhancing semantic alignment and improving accuracy [59]. However, it is computationally intensive. For online search efficiency, current dense search methods typically encode the query and table independently, and rely on basic similarity computations (e.g., cosine similarity) between their embeddings [22, 52], which inadequately capture the nuanced query-table interactions necessary for effective retrieval.

To overcome these limitations, this paper introduces Birdie, a novel framework for NL-driven table discovery using a Differentiable Search Index (DSI) [48]. Birdie unifies both indexing and search into an encoder-decoder Transformer [50] architecture. In this framework, indexing is integrated into the model training process, while search is conducted through model inference, as shown in Figure 2(b). Specifically, each table in the given table repository is assigned a unique table identifier (tabid), represented as a sequence $\mathbf{s} = (s_1, s_2, \ldots, s_d)$. During the training (indexing) phase, the model learns to map each table to its corresponding tabid, and associates generated synthetic queries with the tabids of relevant tables. This design encodes the table information within the model's parameters, ensuring an end-to-end differentiable training process. Additionally, training with synthetic queries and tabids fosters deep interactions between queries and tables through encoder-decoder attention [50]. During the inference (search) phase, the trained model receives a query and directly generates tabids token by token. Despite some successful attempts of DSI in certain applications [31, 45, 53], three primary challenges emerge in developing an effective table discovery approach based on DSI:

Challenge I: *How to design prefix-aware tabids to capture the complex semantics in tabular data?* The tabid generation during the search phase is autoregressive, as shown in Figure 2(b). Thus, it is crucial to design prefix-aware tabids so that tabids of similar tables share similar prefixes, thereby enhancing accuracy. However, existing ID generation methods [48, 57, 66] are designed for flattened, short documents and are not capable of capturing the complex, hierarchical semantics inherent in tabular data. To this end, we propose a two-view-based clustering algorithm, i.e., metadata-view and instance-data-view, to generate tabids for each table. Through inter-view and intra-view hierarchical modeling, we derive prefix-aware semantic tabids for each table.

Challenge II: *How to automatically collect NL queries tailored to tabular data for model training?* Collecting high-quality, diverse, and table-specific queries and associating them with tabids during the training (indexing) phase effectively simulates the subsequent search phase [66], thus improving search accuracy. However, collecting queries for large table repositories manually is intractable. Although some synthetic query generation methods have been proposed, they either focus on flattened text paragraphs [41, 53] or rely on a two-stage transformation (table-to-SQL and SQL-to-NL) [52]. The former overlooks the structured and non-continual semantics of tabular data, while the latter suffers from quality issues due to the error accumulations across two transformation stages. To tackle this, we train a query generator tailored to tabular data using powerful Large Language Models (LLMs), and design a table sampling strategy to generate diverse and high-quality NL queries.

Challenge III: *How to continually index new tables while alleviating the catastrophic forgetting?* When new tables are added to the repository, they require new tabids and updates to the parameters of the existing model to incorporate new information. Training the model from scratch using all the tables whenever the repository changes is resource-intensive, while updating model with only new tables can lead to catastrophic forgetting. Although recent studies [7, 37] suggest replaying some old data during continual training, we still observe several catastrophic forgetting (see experiments in Section 6.4). To tackle this, we design an incremental algorithm for efficient tabid assignment, and a parameter-isolation method to maintain trained model unchanged while training a memory unit for each new batch of tables.

Solution. Incorporating techniques that address these challenges, we present Birdie, a novel framework for natural language-driven table discovery via differentiable search index. The main contributions of this paper are summarized as follows:

- *Differentiable framework.* We propose Birdie, an end-to-end differentiable framework for NL-driven table discovery. To the best of our knowledge, it is the first attempt to perform table discovery using differentiable search index, which completely subverts the convention of previous representation-index-search methods.
- *Prefix-aware tabid construction.* We design a simple yet effective two-view-based clustering approach to model both explicit and implicit hierarchical information embedded in tabular data. Based on this, we assign a prefix-aware tabid to each table, which is well-suited for the autoregressive decoding.
- *LLM-powered query generator.* We continually refine an open-source LLM for better understanding of tabular data, and construct a tailored query generator with a table sampling strategy to create high-quality and diverse NL queries for model training.
- *Effective continual indexing.* We design an incremental method for tabid assignment that avoids re-clustering, and introduce a parameter isolation-based strategy to effectively index new tables while mitigating catastrophic forgetting.

- *Extensive experiments.* Our extensive experiments on three benchmark datasets demonstrate the superiority of Birdie, achieving significant accuracy improvements against SOTA methods.

The remainder of this paper is organized as follows. Section 2 provides the preliminaries related to our work. Section 3 presents the overview of Birdie. Section 4 and Section 5 introduce indexing from scratch and index update, respectively. Section 6 reports experimental results and our findings. Section 7 provides a case study, Section 8 reviews related works, and Section 9 concludes the paper, with directions for future work.

2 PRELIMINARIES

In this section, we first provide the problem statement, and then introduce the backgrounds of large language models and the low-rank adaptation technique.

Problem Statement. Let T be a relational table with a schema $S = \{A_1, A_2, \cdots, A_n\}$, where each A_i represents an attribute (column). The table may also have a caption (title) C, which is a brief description summarizing its content. T consists of m tuples, with each tuple $t_i \in T$ containing n cells, denoted as $e_{ij} = t_i[A_j]$. A table repository $D = \{T_1, T_2, \cdots, T_N\}$ is a collection of such tables.

Definition 1. *(NL-Driven Table Discovery). Given a natural language query q and a table repository D, NL-Driven Table Discovery aims to search for a table $T^* \in D$ that contains the answer to the query. The answer may be a specific cell within T^* or derived from the reasoning across multiple cells in T^*.*

The objective of NL-driven table discovery is to locate a table relevant to the given query. We leave the scenario where the answer spans multiple tables for future work. Another line of research focuses on ranking a small set of candidate tables based on the query using cross-encoders [51, 59], which typically serves as a post-retrieval step and can be time-consuming. While Solo [52] proposed a re-ranking model after retrieval, this paper focuses on the retrieval stage, allowing for any existing ranking techniques to be applied after Birdie returns a list of tables.

Definition 2. *(Differentiable Search Index). Given a query space Q and a table repository D, a differentiable search index is defined as a function $\mathcal{I}_\theta : Q \rightarrow D$, which is differentiable w.r.t. the parameters θ, allowing optimization via gradient-based methods.*

In traditional search indexes, the mapping $\mathcal{I}(q), q \in Q$, is typically based on predefined, discrete, and non-differentiable operations or structures (e.g., inverted index). In contrast, a Differentiable Search Index parameterizes $\mathcal{I}_\theta$ as a neural model, making the search index differentiable w.r.t. θ and enabling end-to-end optimization.

Large Language Models. Due to the outstanding ability to handle sequential data and capture complex dependencies, Transformer [50] has become the main backbone for most large language models (LLMs). It consists of an encoder and a decoder, with both components relying on layers of multi-head attention and feed-forward neural (FFN) networks. Transformer-based LLMs typically adopt one of three architectures: encoder-only, decoder-only, and encoder-decoder. Encoder-only models, such as BERT [13], utilize only the encoder part of the Transformer. These models focus on generating latent embeddings of input text and have found extensive applications in data management [18, 61]. Decoder-only models, such as GPT-4 [42] and Llama [49], excel in diverse generative tasks, including question answering [62] and NL2SQL [35]. In this paper, we choose Llama, a widely used open-source decoder-only model, as the base model for query generation. Finally, encoder-decoder models, also known as sequence-to-sequence models, such as BART [29] and T5 [44], use an encoder to create a latent representation of the input, which is then passed to the decoder to generate a new sequence. Birdie uses this encoder-decoder model to construct the differentiable search index, with the encoder processing tables and queries while the decoder generates the corresponding tabids, facilitating a sequence-to-sequence process.

Low-Rank Adaptation. As the parameter size of LLMs grows rapidly, full-parameter fine-tuning has become impractical due to the significant resource and time costs involved. To address this, parameter-efficient fine-tuning (PEFT) techniques have been proposed. Low-Rank Adaptation (LoRA) [24], one of the most recognized PEFT techniques, adjusts the weights of additional rank decomposition matrices and demonstrates comparable effectiveness across various downstream tasks. Specifically, for a pre-trained weight matrix $\mathbf{W}_0 \in \mathbb{R}^{d_1 \times d_2}$, LoRA constrains the update $\Delta\mathbf{W}$ by representing it as a low-rank decomposition $\mathbf{W}_0 + \Delta\mathbf{W} = \mathbf{W}_0 + \mathbf{BA}$, where $\mathbf{B} \in \mathbb{R}^{d_1 \times d_r}$ and $\mathbf{A} \in \mathbb{R}^{d_r \times d_2}$, with $d_r \ll min(d_1, d_2)$. Let $\mathbf{h} = \mathbf{W}_0\mathbf{x}$, then the modified forward pass is: $\mathbf{h}' = \mathbf{W}_0\mathbf{x} + \Delta\mathbf{W}\mathbf{x} = \mathbf{W}_0\mathbf{x} + \mathbf{BA}\mathbf{x}$. During training, the pre-trained $\mathbf{W}_0$ remains frozen, with only the low-rank matrices $\mathbf{B}$ and $\mathbf{A}$ being learnable. As $\mathbf{B}$ and $\mathbf{A}$ are much smaller than $\mathbf{W}_0$, the fine-tuning costs for LLMs are significantly reduced. The trained LoRA weights can either be merged with the pre-trained weights or used as a plug-and-play module during inference, without altering the original model.

3 OVERVIEW OF BIRDIE

The overview of Birdie is illustrated in Figure 3. Birdie supports both indexing from scratch and updating the index with newly added tables.

Given a table repository D, the two-view-based tabid assignment module generates a prefix-aware tabid for each table $T \in D$ by constructing a clustering tree. The tabid $\mathbf{s} = (s_1, s_2, \ldots, s_d)$ is a numerical sequence, ensuring that semantically similar tables share similar (or identical) prefixes. The tables in the repository are also taken by the LLM-based query generator to produce synthetic queries for each table. Both the generated queries and the serialized tables are input to the encoder-decoder architecture. The encoder transforms these inputs into latent representations, capturing the essential semantics and information of the sequences. The decoder then uses these latent representations, along with a special beginning-of-sequence token, to generate tabids token by token, until the end-of-sequence token is produced. The model is trained to align the predicted tabids with the true tabids for each table/query using a sequence-to-sequence language modeling loss. After training, we obtain a trained DSI model $\mathcal{M}$. During the search (inference) phase, the trained model takes a user's query as input and generates tabids token by token. To produce a list of probable tabids, Birdie adopts beam search [55], exploring multiple decoding paths by maintaining a set of the most promising partial sequences at each decoding step. Ultimately, the corresponding tables are returned to the user.

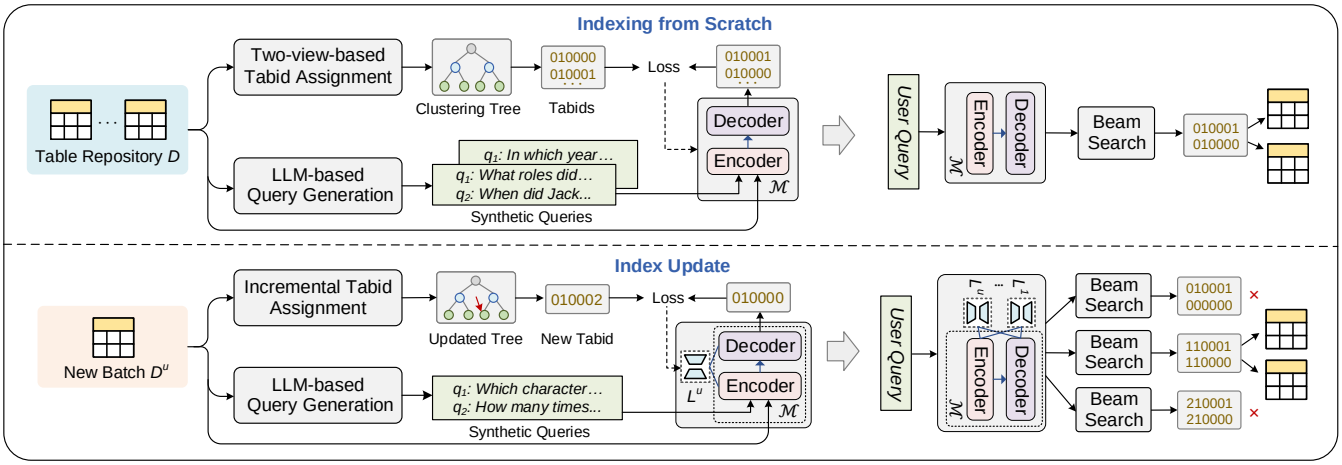


Figure 3: The framework of Birdie.

When a new batch of tables $D^u (u > 0)$ arrives, Birdie performs an index update. First, the incremental tabid assignment strategy assigns a tabid for each table $T_i^u \in D^u$, accounting for the semantics of all previous tables while preserving existing tabids. Next, the query generator generates queries for the new tables. Based on the model $\mathcal{M}$ trained from the original repository D, a memory unit L^u is trained on the new tables D^u and the generated queries, using LoRA techniques. During this training, the parameters of $\mathcal{M}$ remain frozen, while only the LoRA weights being trainable. During search, all existing models (including $\mathcal{M}$ and all the memory units) generate tabids in parallel. A query mapping strategy is then employed to select the final tables from the model outputs.

4 INDEXING FROM SCRATCH

In this section, we first present the two-view-based tabid assignment mechanism, which assigns a semantic and prefix-aware tabid to each table in the repository. Then, we introduce the LLM-based query generation method to generate synthetic NL queries for DSI training. Finally, we outline the training and inference processes.

4.1 Two-view-based Tabid Assignment

Given that a table typically contains many rows and columns, it is impractical for the DSI model to generate the contents of a table directly in response to a given input query. Instead, a practical strategy is to first assign a unique tabid to each table and guide the model to generate the corresponding tabid. The method of tabid assignment is crucial for effective model learning.

Tabids can take the form of numerical sequences or textual descriptions (e.g. a brief abstract of a table). While textual tabids align more closely with the Transformer's pre-training process, ensuring their uniqueness can be challenging, especially in large and dynamic table repositories or data lakes. Therefore, we choose numerical tabids to guarantee uniqueness. A straightforward approach is to assign each table a unique atomic identifier, such as $0, 1, \ldots, N-1$. However, this method results in tabids that lack semantic relationships, which limits the Transformer-based model's ability to fully leverage its semantic capture capabilities and complicates the indexing process. Additionally, since the tabid generation process of the decoder is autoregressive, each tabid is generated in multiple steps, with each step influenced by previously generated tokens. Considering these factors, we design a two-view-based tabid assignment strategy that ensures (i) tabids encapsulate the semantics of the corresponding tables, and (ii) tables with similar tabid prefixes exhibit higher semantic similarity, which enhances the autoregressive decoding process.

Two Views of Table Semantics. We extract two views of each table T. The first view $V_1(T)$ contains high-level semantics about the primary topic of the table. Since the metadata of a table often provides essential context or properties related to its subject or usage, we utilize this metadata as the first view. Specifically, we extract the table's caption C and schema $S = \{A_1, A_2, \ldots, A_n\}$, concatenating them to form a sequence: $V_1(T) := C, A_1, A_2, \ldots, A_n$. Note that metadata can sometimes be incomplete in real-world table repositories. To tackle this, one approach could be to train a metadata generation model to augment the tables with additional information. However, in this paper, we will simply skip any missing elements during serializing.

The second view $V_2(T)$ is derived by concatenating the cell values of each table, providing a more fine-grained perspective. To enhance the understanding of different columns, we also include the attribute names in the cell value sequence: $V_2(T) = A_1 : e_{11}, \ldots A_n : e_{1n} \ldots, A_1 : e_{m1}, \ldots, A_n : e_{mn}$.

Subsequently, we employ a pre-trained language model [40] to encode both $V_1(T)$ and $V_2(T)$, yielding the semantic embeddings $\mathbf{h}^1$ and $\mathbf{h}^2$ for each table.

Clustering Algorithm. Building on the two-view embeddings, we design the two-view-based clustering algorithm (TCA). The core idea is to perform clustering based on the semantic embeddings from both views. The first view captures high-level semantics, allowing us to group tables with related topics into the same cluster. We iteratively perform clustering based on $\{\mathbf{h}^1\}$ for l iterations. Then, we switch views, utilizing the fine-grained embeddings $\{\mathbf{h}^2\}$ to further differentiate tables that share similar topics but contain different contents. The pseudocode of TCA is presented in Algorithm 1. Given a collection of tables $\{T_i\}_{i=1}^N$ with their semantic embeddings $\{\mathbf{h}_i^1\}_{i=1}^N$ and $\{\mathbf{h}_i^2\}_{i=1}^N$, along with the desired number

Algorithm 1: Two-view-based Clustering Algorithm (TCA)

Input: the embeddings $\{\mathbf{h}_i^1\}_{i=1}^N$ and $\{\mathbf{h}_i^2\}_{i=1}^N$ of each table $\{T_i\}_{i=1}^N$, the number k of clusters, the maximum size c of leaf clusters, and the maximum depth l of the first view

Output: the root C_R of the clustering tree

1 $level \leftarrow 0$, $C_R \leftarrow \{\mathbf{h}_i^1\}_{i=1}^N$
2 HClus (C_R, k, c, $level$)
3 **Procedure** HClus(C, k, c, $level$)**:**
4 **if** $level = l$ **then** $C \leftarrow \{\mathbf{h}_i^2 | \mathbf{h}_i^1 \in C\}$ // view switching
5 $C_{1:k} \leftarrow k$-means(C, k), C.addChild$(C_{1:k})$
6 **foreach** $i \in \{1, \ldots, k\}$ **do**
7 **if** $|C_i| > c$ **then** HClus(C_i, k, c, $level + 1$)
8 **return** C_R

of clusters k, the maximum depth l for the first view, and the maximum size c for leaf clusters, TCA initializes the current level as 0, sets the root C_R to $\{\mathbf{h}_i^1\}_{i=1}^N$, and invokes the clustering procedure HClus (lines 1–2). Within HClus, the algorithm performs view switching to use the embeddings $\{\mathbf{h}_i^2\}$ from the second view if the current clustering level reaches l (line 4). Next, tables are grouped into k clusters $C_{1:k}$ based on their embeddings from the current view, which are then inserted into the child nodes C (line 5). For each child C_i, if it contains more than c tables, HClus is recursively applied within that cluster (lines 6–7). Finally, TCA returns the root of the clustering tree (line 8).

This approach enables TCA to construct a clustering tree (as shown in Figure 4) and captures hierarchical relations in two ways: firstly, through view switching, which explicitly incorporates hierarchical structures using metadata and instance data; and secondly, through recursive clustering within each view, which provides an implicit hierarchical structure that refines semantic representations from coarse to fine-grained levels.

Tabid Assignment. Based on the two-view clustering tree, we assign a unique tabid to each table. Specifically, we assign a unique number ranging from 0 to $k-1$ to each cluster at every level of the clustering tree. For each leaf node containing c tables or fewer, we assign each table within that cluster a unique number from 0 to $c-1$. Consequently, each table receives a tabid $\mathbf{s} = (s_1, \ldots s_d)$ that represents its unique path from the root to the leaf node in which it resides, where $s_i \in [0, k-1]$ for $i < d$ and $s_d \in [0, c-1]$.

Example 1. *Figure 4 illustrates an example of a clustering tree where $k = c = l = 2$. The blue nodes represent clusters obtained using the embeddings $\mathbf{h}_i^1$, while the green nodes represent cluster derived from $\mathbf{h}_i^2$. Each branch is assigned a number within the range $[0, k-1]$, while each table within a leaf node is assigned a number within the range $[0, c-1]$. Accordingly, T_2 is assigned the tabid* 01000*, while T_5 receives the tabid* 01001*.*

To optimize memory usage, we design a tree compression (TC) method that uses a compact structure to retain only the essential information for each cluster (node) in the clustering tree. Specifically, each node in the clustering tree maintains the cluster radius r, cluster cohesion δ, cluster center $\mathbf{c}$, and the view v (1 or 2) of the embeddings used to generate this cluster. The radius r and cohesion

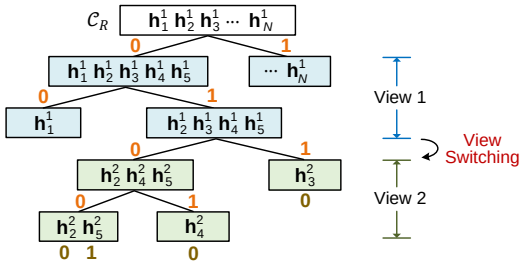


Figure 4: An example of a two-view-based clustering tree.

δ of a cluster C are defined as follows:

$$r(C) = \max_{\mathbf{h}_j \in C} \text{dist}\left(\mathbf{c}, \mathbf{h}_j\right), \delta(C) = \frac{1}{|C|} \sum_{j=1}^{|C|} \text{dist}\left(\mathbf{c}, \mathbf{h}_j\right) \quad (1)$$

where $\mathbf{c}$ is the center of the cluster, and dist$(\cdot)$ is a distance function. This approach allows us to manage the tabid assignments dynamically (to be detailed in Section 5.1) without the need to store all high-dimensional embeddings $\mathbf{h}^1$ and $\mathbf{h}^2$ for large table repositories.

4.2 LLM-based Query Generation

Unlike previous query generation methods that either focus on flattened text paragraphs [41] or rely on manually-designed SQL templates [52], we leverage the powerful generative capabilities of decoder-only LLMs to directly produce NL queries for tables. While powerful closed-source LLMs like GPT-4 [42] could be used, the costs of API calls and the privacy concerns [46] often restrict its usage. Consequently, we aim to train a local query generator based on the open-source LLMs.

Training TLLaMA3. We adopt the latest LLaMA3 8b [2] as our base model, as it outperforms other LLMs of similar or larger scales. However, since LLMs are mainly pre-trained on textual data, they may struggle to comprehend tabular structures [30]. To tackle this, we first fine-tune LLaMA3 on seven fundamental table-related tasks listed in Table 1 that range in difficulty. Specifically, we adopt LoRA technique to tune it with a mixture of data derived from these seven tasks. We then merge the trained LoRA module with the original pre-trained weights of LLaMA3 to create TLLaMA3. This process allows TLLaMA3 to develop a fundamental understanding of tables, enabling it to perform more complex and specific table tasks through further fine-tuning.

Query Generation. Building on TLLaMA3, we construct instruction data to train a query generator tailored to tabular data, as shown in Figure 5. The instruction requires the model to generate NL queries that can be answered by specific cells in the given table, or derived through reasoning with aggregation operators. It is important to note that some NL queries may lack sufficient specificity for table search tasks. For example, given the table T_1 titled "On Golden Ponds" (see Figure 1), the LLM might generate the query "Who directed this film?". While this is a common query for a closed-domain QA, it is ambiguous due to the phrase "this film", making it less effective for searching tables in repositories. Therefore, we prompt the LLM to generate more explicit queries by including necessary table information. The table is transformed into markdown format, and the output consists of a labeled query.

Using our constructed instruction data, we adopt LoRA to fine-tune TLLaMA3, resulting in a plug-and-play LoRA module that

Table 1: Table-related tasks used to continue to train LLaMA3

Task Name	Task Description	Datasets	Difficulty	Size
Table Size Recognition	Identify the number of rows and columns given a table	TSR [63]	Easy	1.5k
Table Cell Extraction	Extract the specific cell in a table given the row and column IDs	TCE [63]	Easy	1.5k
Table Row/Column Extraction	Extract the specific row (column) in a table given the row (column) ID	RCE [63]	Easy	1.5k
Table Cell Retrieval	Answer the row and column IDs of a given cell in a table	TCR [63]	Medium	1.5k
Table to Text	Generate a textual description of a given table	WikiBIO [28]	Medium	5k
Table Fact Verification	Verify whether a textual hypothesis holds given tabular data as evidence	TABFACT [8]	Hard	5k
Table Question Answering	Answer the question based on a given a table	WTQ [43]	Hard	5k

Instruction:
Generate the probable question a user would ask based on this table, ensuring the answer can be found in the table cells or derived using operators such as (*Max*, *Min*, *Avg*, *Count*). The question should contain the explicit information of this table (such as the information of the caption) so that given the question and a repository of tables, this table can be successfully found.

Input:
Caption: <Table Caption>
Table: <Markdown Format of Table>

Output:
Question: <NL Question>

Figure 5: Instruction data for query generator training.

Algorithm 2: Table Sampling Algorithm

Input: a table T_i, the number B of required queries, and the number b of generated queries per invocation
Output: the generated query set Q_i

1 $n_v \leftarrow |B|/|b|$, $r_s \leftarrow |T_i|/n_v$, $R_i \leftarrow T_i$, $Q_i \leftarrow \emptyset$
2 **while** $|Q_i| < B$ **do**
3 **if** $|R_i| < r_s$ **then** $R_i \leftarrow T_i$
4 $S(R_i) \leftarrow \mathsf{SampleRows}(R_i, r_s)$
5 $R_i \leftarrow$ remove $S(R_i)$ from R_i
6 $Q_{cand} \leftarrow \mathsf{Gen}(\mathcal{G}, S(R_i), b)$
7 $Q_i \leftarrow Q_i \cup \mathsf{Filter}(Q_{cand})$
8 **return** Q_i

can be integrated with the weights of TLLaMA3 to form our query generator $\mathcal{G}$. Subsequently, we use the trained query generator $\mathcal{G}$ to generate multiple NL queries Q_i for each table $T_i \in D$ using the same prompt shown in Figure 5. We restrict the maximum input length to 2048 tokens. To reduce query generation time, $\mathcal{G}$ is allowed to generate multiple queries per invocation; however, this can decrease the diversity of the generated queries. Specifically, we noticed that certain cells in the table were more frequently included in the generated queries, which may be attributed to word frequency bias [21] during the pre-training phase of LLMs.

To mitigate this issue, we design a table sampling algorithm (TSA) to enrich the generated queries and maximize the coverage of table contents. The key idea behind TSA is to divide the table into several sub-tables and invoke the generator $\mathcal{G}$ multiple times for each sub-table. This approach allows $\mathcal{G}$ to focus on each sub-table individually. The pseudo-code for TSA is presented in Algorithm 2. Given a table T_i, the number B of required synthetic queries, and the number b of queries generated per invocation of $\mathcal{G}$, TSA first computes the number r_s of rows for each sampling process, and initializes the row set R_i and query set Q_i (line 1). TSA then iteratively performs table sampling and query generation until B queries are collected. Specifically, TSA re-initializes R_i to include all rows from T_i if R_i contains fewer than r_s rows (line 3). It next randomly samples r_s rows from R_i and removes the sampled $S(R_i)$ from R_i (lines 4–5) to prevent repeated sampling, which increases the coverage of table information. TSA then uses the sampled $S(R_i)$ to invoke the query generator $\mathcal{G}$, generating candidate queries Q_{cand} (line 6). Given the potential for hallucinations in LLMs, candidate queries may exhibit quality issues. To address this, TSA implements quality checks on Q_{cand} via Filter function that establishes two rules: i) ensure the output format is consistent with "`Question: <NL Question>`" (as shown in Figure 5), as we found that many low-quality outputs exhibit formatting errors; and ii) check for duplication with existing queries Q_i, and perform deduplication if necessary. TSA adds the filtered Q_{cand} to the query set Q_i (line 7). Finally, TSA returns the query set Q_i (line 8).

4.3 Model Training and Inference

After generating the query set Q_i and tabid $\mathbf{s}_i$ for each table T_i, we outline the process of constructing differentiable search indexes through model training. We leverage an encoder-decoder model $\mathcal{M}$ as the backbone. The model is fine-tuned to associate each table T_i with its corresponding tabid $\mathbf{s}_i$, and generate the tabid $\mathbf{s}_i$ based on an input synthetic query derived from table T_i. The model is trained using a standard sequence-to-sequence objective that employs a teacher forcing policy [54] and minimizes the cross-entropy loss:

$$\mathcal{L} = \sum_{T_i \in D} \left(\log p\left(\mathbf{s}_i \mid \mathcal{M}_\theta\left(T_i\right)\right) + \sum_{q_{ij} \in Q_i} \log p\left(\mathbf{s}_i \mid \mathcal{M}_\theta(q_{ij})\right) \right) \quad (2)$$

Here, Q_i represents the set of queries relevant to table T_i generated by our query generator $\mathcal{G}$, and θ denotes the trainable parameters of model $\mathcal{M}$. For input representation, the table is serialized into a sequence by concatenating its caption, attributes, and cells.

In the search (inference) phase, given an input user query q, the fine-tuned model outputs the tabid token by token:

$$p\left(\mathbf{s} \mid q, \theta\right) = \prod_{j=1}^{d} p\left(s_j \mid \mathcal{M}_\theta(q, s_0, s_1, \ldots, s_{j-1})\right) \quad (3)$$

Here, $\mathbf{s}$ is the output tabid corresponding to the user query q, s_j represents the j-th token of tabid $\mathbf{s}$, s_0 is a special start token indicating the beginning of the decoding process, and θ denotes the fine-tuned parameters of model $\mathcal{M}$. We employ beam search [55] during the decoding process. This technique explores multiple decoding paths by maintaining a set of the most promising partial sequences at each decoding step, ultimately generating a list of probable tabids. Note that, beam search allows us to attach a probability to each generated tabid, serving as an effective ranking mechanism without incurring the additional computational cost of re-ranking. With the valid tabid set obtained through two-view-based tabid assignment, we can effectively filter out invalid output tabids during inference.

5 INDEX UPDATE

After training on the table repository D, we obtain a model $\mathcal{M}$ that encodes all the information of tables in D. Since table repositories are typically dynamic [20], it is crucial to continuously index new tables. We consider a scenario that Z batches of new tables $\{D^1, D^2, \ldots, D^Z\}$ arrive sequentially, with each D^i consisting of newly encountered tables $\{T_1^i, T_2^i, \ldots, T_{N_i}^i\}$. The original D is assumed to be significantly larger than the incoming batches, and the new batches generally align with the distribution of D. We employ a lazy update strategy, as used in previous studies [7, 37], where updates are triggered only when accumulated data exceeds a predefined threshold. For clarity, we refer to the original repository as D^0 and the model as $\mathcal{M}^0$.

In a dynamic table repository, tables are either inserted or removed. The modifications of tables are naturally handled as deletions followed by insertions. When a table is removed from the repository, its corresponding tabid is also removed from the valid tabid set. Consequently, any invalid tabids produced by the decoder can be easily filtered out during search. Thus, table deletion is a straightforward operation that does not impact model accuracy, unlike table insertion. Therefore, the remainder of this paper focuses on updating the index for table insertion. In the following, we present the incremental tabid assignment algorithm to attach tabids to new tables without affecting the previous tabids, and illustrate how to index new tables by parameter isolation and how to search for tables under this scenario.

5.1 Incremental Tabid Assignment

When a new batch $D^u (u > 0)$ of tables arrives, the first step is to assign tabids to each new table. ReIndex, a naive method, re-clusters all tables in $\bigcup_{i=0}^{u} D^i$ to account for table semantics and ensure unique tabids using Algorithm 1, and re-assigns tabids. However, ReIndex has two main drawbacks: (i) re-clustering is computationally expensive and time-consuming, and (ii) previously assigned tabids become invalid as clusters change, necessitating a complete retraining of the model from scratch to index all the tables in $\cup_{i=0}^{u} D_i$. To overcome these challenges, we propose an incremental algorithm that assigns tabids to new tables without affecting existing ones.

We assume that the existing clustering tree is $\mathbb{T}$, and we illustrate the insertion of one new table with embedding $\mathbf{h}_{new}$. First, we find the child node C^* of the root of $\mathbb{T}$ that is closest to $\mathbf{h}_{new}$:

$$C^* = \text{argmin}_C \, \text{dist}(C.\mathbf{c}, \mathbf{h}_{new}) \tag{4}$$

where C represents the child node of the root of $\mathbb{T}$, $\text{dist}(\cdot)$ is a distance metric implemented by Euclidean distance, and $C.\mathbf{c}$ denotes the center $\mathbf{c}$ of cluster C. We then compute the distance d from $\mathbf{h}_{new}$ to the center $C^*.\mathbf{c}$. For simplicity, we denote the radius, cohesion, and center of cluster C as r, δ, and $\mathbf{c}$, respectively. We elaborate on the following three cases when inserting $\mathbf{h}_{new}$ into $\mathbb{T}$.

Case I: Inserting without updates. If h_{new} is sufficiently close to cluster C^*, its inclusion has a negligible impact on the properties of C^*.To enhance the efficiency of incremental tabid assignment, updates to the properties can be omitted when $d \leq \delta$, i.e., the distance from h_{new} to the cluster center does not exceed the average distance from existing embeddings to the center. In such cases, we directly insert $\mathbf{h}_{new}$ to cluster C^* without updates.

Case II: Inserting with updates. When $\delta < d \leq r$, it suggests that $\mathbf{h}_{new}$ belongs to C^* but is farther from the cluster center than average. In this case, we insert $\mathbf{h}_{new}$ into C^* and update the center $\mathbf{c}$, cohesion δ, and radius r of C^*. We denote the updated values as $\mathbf{c}'$, δ' and r', with $\mathbf{c}'$ and δ' calculated as follows:

$$\mathbf{c}' = \frac{|C^*| \cdot \mathbf{c} + \mathbf{h}_{new}}{|C^*| + 1}, \qquad \delta' = \frac{|C^*| \cdot \delta + d}{|C^*| + 1} \tag{5}$$

where $|C^*|$ denotes the size of C^* after the insertion of $\mathbf{h}_{new}$. However, computing the radius r' by calculating the distance between the new center and all table embeddings is not feasible, as the clustering tree does not retain table embeddings to save storage, as stated in Section 4.1. Therefore, we present the following lemma to estimate the radius r':

Lemma 1. *The upper and lower bounds of the radius r' are given by $r + \text{dist}(\mathbf{c}, \mathbf{c}')$ and $\text{dist}(\mathbf{h}_{new}, \mathbf{c}')$, respectively.*

The proof can be found in Appendix A. Accordingly, we can estimate r' by sampling uniformly between its lower and upper bounds:

$$r' = \text{Uniform}(\text{dist}(\mathbf{h}_{new}, \mathbf{c}'), r + \text{dist}(\mathbf{c}, \mathbf{c}')) \tag{6}$$

Case III: Constructing a new cluster. The final case occurs when $\mathbf{h}_{new}$ is too distant from the center of the nearest cluster, indicating that $\mathbf{h}_{new}$ should be assigned to a new cluster. Specifically, if $d > r$, we create a new cluster with $\mathbf{h}_{new}$ as its center, initializing its radius and cohesion to the average values of all child nodes of the root, ensuring consistency with the scale and density of existing clusters at the same hierarchical level. While splitting an existing cluster and inserting the new embedding is a viable alternative, it incurs significantly higher computational costs, making it less preferable.

After inserting the new embedding $\mathbf{h}_{new}$ into an existing node or a newly created node, the insertion process is recursively applied until a leaf node is reached, at which point a tabid is assigned to the new table. This process is repeated for each new table. The detailed incremental tabid assignment procedure (ITA) is outlined in Algorithm 3. If the current node is a leaf, ITA generates and returns the tabid (lines 1–2). Otherwise, ITA continues with embedding insertion, determining which embedding view to use (lines 3–4), identifying the closest cluster C^* using Eq. (4) (line 5), and calculating the distance d from the new embedding to the center of cluster C^* (line 6). Based on the value of d, one of the following actions is taken: i) if d does not exceed the cohesion of C^*, the embedding is inserted directly into C^* and the insertion process continues recursively (line 7); ii) if d falls between the cohesion and the radius of C^*, the embedding is inserted into C^* with cluster information updated, followed by recursive insertion (lines 9–11); and iii) otherwise (d exceeds r), a new cluster is created and the embedding is inserted (lines 13–14).

5.2 Continual Indexing via Parameter Isolation

When tabids of new tables in D^u are obtained, a straightforward method to update indexes is to use new tables or combine them with key tables from previous batches to continually train the model [7]. However, this strategy can lead to catastrophic forgetting, where the model loses information about older tables that are not included in the continual training process. To tackle this issue, we propose a method that isolates the parameters of the model $\mathcal{M}^0$ trained on D^0 from those learned continually.

Algorithm 3: Incremental Tabid Assignment

Input: the current node C, the embeddings $\mathbf{h}^1_{new}$ and $\mathbf{h}^2_{new}$ of a new arrival table T_{new}
Output: tadid of T_{new}

if *C is a leaf node* **then**
  tabid ← path from the root to C, **return** tabid
if $C.child.v = 1$ **then** $\mathbf{h}_{new} \leftarrow \mathbf{h}^1_{new}$
else $\mathbf{h}_{new} \leftarrow \mathbf{h}^2_{new}$
find the child node C^* of C that is closest to $\mathbf{h}_{new}$ // Eq. (4)
$d \leftarrow \text{dist}(C^*.\mathbf{c}, \mathbf{h}_{new})$
if $d \leq C^*.\delta$ **then** insert $\mathbf{h}_{new}$ to C^*, $\text{ITA}(C^*, \mathbf{h}^1_{new}, \mathbf{h}^2_{new})$
else if $C^*.\delta < d \leq C^*.r$ **then**
  insert $\mathbf{h}_{new}$ to C^*
  update the information of C^* // Eq.(5) and Eq.(6)
  $\text{ITA}(C^*, \mathbf{h}^1_{new}, \mathbf{h}^2_{new})$
else
  create a new child node C_{new} under C centered at $\mathbf{h}_{new}$
  insert $\mathbf{h}_{new}$ to C_{new}

Construction of Memory Units. For each batch of tables $D^u (u > 0)$, we train a LoRA module L^u to index the new tables based on the existing model $\mathcal{M}^0$. The intuition behind this approach is that $\mathcal{M}^0$ has already been trained on D^0, which we assume to be a relatively large repository compared to the incoming batches. As a result, $\mathcal{M}^0$ has developed the capability to understand, index, and search for tables effectively. Therefore, we only need to train a LoRA module L^u as a memory unit based on $\mathcal{M}^0$, which is more efficient than performing full fine-tuning. Unlike previous studies [33, 58] that typically add LoRA matrices to the weights of self-attention layers in each Transformer block to acquire new capabilities, we argue that this is not optimal for our context. Our goal is to encode information of new tables into the LoRA modules rather than to acquire new abilities. Inspired by the earlier research [17] suggesting that the FFN in Transformer operates as a memory, we add LoRA modules to FFN in each Transformer block. This allows each batch of new tables $D^u (u > 0)$ to have its own dedicated LoRA memory unit L^u.

We store all these units along with the initial model $\mathcal{M}^0$ in a memory hub. These model parameters remain isolated, preventing them from affecting one another. After training the LoRA module L^u on the newly arrived tables $D^u (u > 0)$, the model hub contains the following models $\mathcal{M}^i$, where each $\mathcal{M}^i$ indexes tables from a sub-repository $D^i (0 \leq i \leq u)$ and $\bigoplus$ denotes the plug-and-play of L^i during inference.

$$\mathcal{M}^i = \begin{cases} \mathcal{M}^0 & \text{if } i = 0 \\ \mathcal{M}^0 \bigoplus L^i & \text{if } 1 \leq i \leq u \end{cases}$$

Search Stage. During the search phase, each model $\mathcal{M}^i$ takes a user query q as input and performs inference using beam search either serially or in parallel, depending on available memory resources. After that, each model obtains the candidate tables $\mathcal{T}^i = \{T^i_1, \ldots, T^i_K\} \subset D^i$, with K a user-defined parameter that controls the number of tables returned. However, not all results are equally reliable, as relevant tables for the query q may reside in a few sub-repositories. To effectively select candidate tables from $\{\mathcal{T}^0, \ldots, \mathcal{T}^u\}$, we implement a simple yet effective mapping strategy. For each $\mathcal{T}^i$, we first compile the synthetic queries generated during training into a query set $Q^i = Q^i_1 \cup \cdots \cup Q^i_K$, where Q^i_j represents the synthetic queries for table $T^i_j \in D^i$. These query sets collectively form a small query pool $Q^0 \cup \ldots Q^u$. Next, we encode all queries in the query pool, along with the user query q, using a pre-trained semantic-aware encoder [1]. We then identify the top-n_q queries of q in the query pool that are most similar to q. Finally, we select the query set Q^z that includes the largest number of queries found among the top-n_q similar queries, and random selection is used as the tiebreaker. The corresponding candidate tables $\mathcal{T}^z \subset D^z$ are then returned as the results for table discovery. An extremely small n_q may lead to incorrect selection due to the randomness of the synthetic queries, while a large n_q will consider the contributions of many less relevant queries. Therefore, we set n_q to $\lfloor \frac{B}{4} \rfloor$ in our experiments, where B is the number of synthetic queries generated for each table T^i_j, as specified in Algorithm 2.

Discussion. Current mainstream libraries (e.g., PEFT [36]) typically merge each LoRA module individually into the base model before inference, leading to multiple full-model copies being loaded into GPU memory during parallel inference. One potential optimization is to load a single base model alongside all LoRA modules. During inference, for a given input $\mathbf{x}$, the output of the base model $\mathcal{M}^0(\mathbf{x})$ and the output of each LoRA module $L^i(\mathbf{x}) = \mathbf{B}^i\mathbf{A}^i\mathbf{x}$ are computed in parallel. The output of each LoRA module is subsequently added to $\mathcal{M}^0(\mathbf{x})$ to get $\mathcal{M}^i(\mathbf{x})$. A recent study [47] explores this optimization for parallel inference with multiple LoRAs, utilizing tensor parallelism strategy and highly optimized custom CUDA kernels to efficiently support inference of thousands of LoRA modules on a single GPU. However, it is currently limited to decoder-only architectures and does not support our encoder-decoder DSI model. Extending it to encoder-decoder architectures is outside the scope of this paper but remains an avenue for future work.

6 EXPERIMENTS

In this section, we conduct extensive experiments to demonstrate the efficacy of our proposed Birdie.

6.1 Experimental Setup

Datasets. The experiments were conducted on three real-world benchmark datasets, with statistics summarized in Table 2. (i) **NQ-Tables** [22] is a widely used benchmark for table retrieval. Each table in NQ-Tables includes a brief caption, though many tables with the same caption differ in schema and content. The dataset contains some inconsistencies, making it relatively noisy. Following the previous work [52], we use only the 952 test queries that have a single ground truth table. (ii) **FetaQA** [38] contains clean tables and test NL queries designed for free-form table question answering. Each test query is paired with a unique ground truth table containing the answer. (iii) **OpenWikiTable** [27] is built upon two closed-domain table QA datasets WikiSQL and WikiTableQuestions. Table descriptions (i.e. page title, section title, caption) are manually re-annotated and then added to the original queries, ensuring this dataset can be applicable to table discovery task.

Baselines. To demonstrate the efficacy of our proposed Birdie, we compare it with the following baseline methods. 1) BM25 is a widely

Table 2: Statistics of the datasets used in experiments.

Dataset	# Tables	Quality	# Train	# Test
NQ-Tables	169,898	dirty	9,534	952
FetaQA	10,330	clean	7,326	2,003
OpenWikiTable	24,680	clean	53,819	6,602

used sparse retrieval method. 2) DPR (Dense Passage Retriever [25]) is the SOTA dense retrieval method for open-domain QA of passages. We serialize each table to text and use bert-base-uncased [13] for both query and text encoders, training them with in-batch negatives on the training splits. Mean pooling is adopted to obtain both the query and table embeddings. 3) DPR-T is an extension of DPR to incorporate table structure information. Following [27], we add special tokens like [Caption], [Header], [Rows] during table serialization, using the embedding of the first [CLS] token as the table embedding. 4) OpenDTR [22] is a learning-based table discovery method using TAPAS [23] for both query and text encoders. 5) Solo-Ret is the first-stage retrieval module of Solo [52], which is a SOTA learning-based NL-driven table discovery system comprising two stages: retrieval and ranking. To ensure a fair comparison, we focus solely on the retrieval module Solo-Ret. 6) Solo [52] reranks results from Solo-Ret. We follow its original implementation to retrieve the top-250 triplets, then apply second-stage ranking.

Evaluation metrics. To evaluate the effectiveness, we follow the previous study [52] to adopt the precision-at-K ($P@K$) metric, aggregated over the test queries. $P@K$ measures the proportion of test queries for which the ground truth table appears among the top-K results returned by each method. Since each query in our benchmark dataset has a single ground truth table, precision alone suffices for evaluating the effectiveness [52]. In addition, we measure runtime to assess efficiency.

Implementation details. Birdie is implemented in PyTorch. For tabid assignment, we use the encoder of pre-trained Sentence-T5 [40] to obtain the embeddings, and fit as much of the second view as possible within the default 512-token limits. The number k of clusters and the maximum size c of leaf clusters are set to 32 for NQ-Tables and 20 for FetaQA and OpenWikiTable, while the depth l of the first view is set to 2 for all three datasets. For training the query generator, we fine-tune TLLaMA3 using LoRA on the training split of each dataset, adding LoRA to the attention layers with a rank d_r of 8. We generate $B = 20$ queries per table by default. We use mT5-base [56] as the backbone for Birdie, and set the maximum length of the input data to 64 tokens. The batch size is set to 64, and the learning rate is set to 5e-4. For training the memory units, we add LoRA to FFNs of both the encoder and decoder with $d_r = 8$. During inference (search), we adopt beam search with a beam size of 20. All experiments were conducted on a server with an Intel(R) Xeon(R) Silver 4316 CPU (2.30GHz), 6 NVIDIA 4090 GPU (24G each), and 256GB of RAM. All programs were implemented in Python.

6.2 Evaluation of Indexing from Scratch

Table 3 summarizes the $P@1$, $P@5$, and the search time (Time) for Birdie and the baseline methods, averaged across all test queries.

Effectiveness. Several key observations can be made as follows. (i) BM25 exhibits poor accuracy on NQ-Tables but performs notably better on FetaQA and OpenWikTable. This disparity arises because

Table 3: Performance of Birdie compared to other baselines.

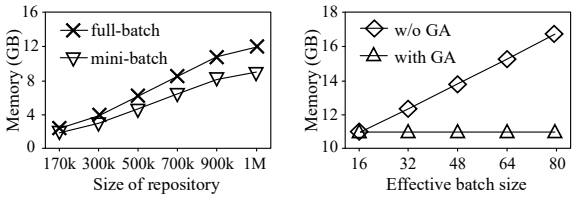
Methods		NQ-Tables			FetaQA			OpenWikiTable		
		$P@1$	$P@5$	T(s)	$P@1$	$P@5$	T(s)	$P@1$	$P@5$	T(s)
Sparse	BM25	17.31	31.07	**0.053**	45.38	70.59	0.034	49.88	75.39	**0.039**
Dense	DPR	23.15	51.30	0.296	49.93	73.69	0.024	64.31	87.59	0.046
	DPR-T	13.24	36.50	0.297	40.89	68.20	0.022	57.16	84.22	0.045
	OpenDTR	37.37	64.28	0.296	54.22	76.93	**0.020**	90.90	98.90	0.102
	Solo-Ret	36.18	53.70	3.532	79.28	88.72	1.151	90.37	94.38	1.724
Rerank	Solo	41.92	70.49	4.398	81.43	91.51	1.857	94.71	96.59	2.541
Ours	Birdie	**46.64**	**73.21**	0.097	**86.27**	**92.56**	0.145	**97.20**	**99.06**	0.105

Table 4: Memory/storage usage (GB) on NQ-Tables.

Phases	DPR	OpenDTR	Solo-Ret	Solo	Birdie
Training (GPU Memory)	5.83	5.86	–	7.61	10.83
Indexing (Disk Storage)	0.49	0.49	16.10	16.10	2.30
Inference (GPU Memory)	0.97	2.18	1.46	7.27	2.33

the queries tested in NQ-Tables are brief, providing insufficient context for effective sparse retrieval. In contrast, queries in the other two datasets, especially OpenWikiTable, are more detailed, which benefits the performance of sparse methods. (ii) OpenDTR outperforms DPR and DPR-T. Interestingly, DPR-T, which incorporates special tokens to denote different table structures, performs worse than DPR, suggesting that simply adding special tokens does not enhance the encoder's understanding of tabular data. (iii) The multi-vector dense method (Solo-Ret) does not consistently outperform single-vector dense methods. For instance, Solo-Ret surpasses OpenDTR on FetaQA but underperforms it on NQ-Tables. This behavior can be explained by Solo-Ret's fine-grained representation of cell-attributes-cell triplets, which tends to overemphasize local information of some specific triplets, but sometimes overlooks the important global semantics of the table. (iv) Birdie outperforms all existing dense methods. For example, Birdie achieves an average improvement of 16.8% in $P@1$ over the best-performing dense method on NQ-Tables and FetaQA. This demonstrates the effectiveness of Birdie's differentiable search index, which addresses the limitations of the existing dense methods using traditional representation-index-search pipeline. (v) Birdie even outperforms the SOTA retrieval-rerank method Solo. This underscores Birdie's superiority as an end-to-end solution, where beam search serves as an effective ranking mechanism. (vi) All methods perform better on OpenWikiTable than the other two datasets. This can be attributed to: 1) the queries of OpenWikiTable contain detailed title-relevant information of ground truth tables, making it easier to locate the relevant table; and 2) a remarkable 99.8% (24,634 out of 24,680) of tables in OpenWikiTable have unique titles, simplifying the task of identifying the correct table using title information alone.

Efficiency. In terms of online search efficiency, BM25 demonstrates short time across all the datasets, as shown in Table 3. However, for dense methods, search time increases with the size of the table repository. The difference in search time is less pronounced for Solo-Ret, as its efficiency depends more on the number of triplets extracted from tables rather than the number of tables themselves. Solo requires the most time due to its reranking process. Birdie shows an average of 20× and 27× shorter search time on the three datasets compared with Solo-Ret and Solo, respectively. Note that, Birdie maintains stable efficiency across varying sizes of table repositories, presenting a significant advantage.



(a) CPU memory for tabid assignment (b) GPU memory for training

Figure 6: Scalability of tabid assignment and model training.

Memory/Storage Usage. Table 4 presents the memory/storage usage during the key phases. For fairness, we set the batch size to 1 for all methods during training and inference. Birdie requires 10.83 GB of memory for training, more than baseline methods due to its larger encoder-decoder model. Solo-Ret, using a pre-trained model for retrieval, requires no extra training. In contrast, Solo requires training the re-ranker. For indexing, dense methods require storing the table/triplet embeddings on disk, while Birdie requires only 2.3 GB to store the DSI model. For inference, Birdie uses just 2.33 GB of memory, making it highly cost-efficient for deployment.

Scalability. We evaluate the scalability of clustering-based tabid assignment and DSI model training. For clustering, we implement mini-batch k-means with a batch size of 1k to construct the semantic tree for large repositories. By expanding the NQ-Tables dataset through table duplication up to 1 million, we analyze memory usage with or without the mini-batch optimization, as shown in Figure 6(a). The results indicate that memory usage scales linearly with the repository size, and mini-batch k-means effectively reduces memory requirements. For model training, memory requirements are directly influenced by batch size and can be reduced using gradient accumulation (GA) [32]. Figure 6(b) illustrates the GPU memory usage for varying batch sizes, including a scenario with a per-device effective batch size of 16 and GA applied.

6.3 Ablation Study

We conduct an ablation study with results presented in Table 5.

Table ID Assignment. First, we replace our semantic tabids with simple atom IDs $(0, 1, \ldots, N-1)$ for each table. This change results in significant accuracy drop on NQ-Tables. This is attributed to the atom IDs' lack of semantic context, which complicates the training (indexing) process. The smaller performance drops on FetaQA and OpenWikiTable can be attributed to (i) their smaller repository sizes compared to NQ-Tables (170k), which reduce the indexing burden on the DSI model; and (ii) the test queries being more specific and semantically rich, making it easier to generate correct IDs and narrowing the performance gap between atom and semantic IDs. Next, we remove the instance view (view2) used in tabid generation, resulting in a decrease in accuracy across three datasets. This underscores the importance of instance data in capturing essential table information. Finally, we remove the metadata view (view1), leaving only the instance data for tabid generation. This change also leads to a decline in accuracy, confirming the critical role of metadata in understanding tabular data.

Query Generation. First, we replace our LLM-based query generator with the method proposed by Solo [52], which generates SQL queries based on pre-defined SQL templates and subsequently converts them to NL queries using the pre-trained SQL2Text [64] model.

Table 5: P@1 and P@5 of Birdie and its variants.

Stages	Strategies	NQ-Tables		FetaQA		OpenWikiTable	
		$P@1$	$P@5$	$P@1$	$P@5$	$P@1$	$P@5$
–	Birdie	**46.64**	**73.21**	**86.27**	**92.56**	**97.20**	**99.06**
Tabid Assignment	Atom ID	17.75	42.54	83.58	90.81	94.29	98.53
	w/o view2	44.12	70.48	86.07	92.41	94.33	97.01
	w/o view1	45.16	70.59	84.62	91.91	96.91	98.99
Query Generation	Solo-based	16.60	31.83	24.06	36.10	93.80	97.82
	Textual-based	30.15	62.19	72.78	87.52	95.12	98.30
	w/o TS	33.19	63.34	58.76	74.34	95.75	98.23

This method results in significant accuracy drops on NQ-Tables and FetaQA. The decline is likely due to error accumulation across the two phases of query generation and the lack of fine-tuning of the SQL2Text model for the specific domain of our datasets. Then, we apply the textual-based method docT5query [41] to train a query generator using training splits, treating tables as flattened text. This approach results in an average decrease of 26% in $P@1$ and 10% in $P@5$ on NQ-Tables and FetaQA, further confirming the superiority of our LLM-based query generator specifically tailored to tabular data. Finally, removing the table sampling (TS) module from the LLM-based query generation process reduces accuracy across datasets, underscoring the importance of TS in generating high-quality synthetic queries. Note that the performance drops on OpenWikiTable with these ablations are small due to the nature of its test queries. As long as the generated synthetic queries include title information, the accuracy remains high.

6.4 Evaluation of Index Update

Setting. To simulate a dynamic scenario, we randomly sample 70% of the tables from the original dataset to create the initial repository D^0. We then randomly sample 10% of the remaining tables to form a new batch, repeating this process three times to generate batches D^1, D^2, and D^3. For each dataset D^i ($i \in [0, 3]$), we construct the query set $Test^i$ by filtering queries from the original testing queries whose ground truth tables are included in D^i.

Competitors. We adapt two existing continual learning methods for document retrieval via DSI: CLEVER [7] and DSI++ [37], for table discovery. CLEVER samples old tables with similar tabids, and combines them with the new batch D^u to continually train the DSI model. In contrast, DSI++ randomly samples an equal number of old tables as CLEVER. Additionally, we compare a naive yet time-consuming solution, Full, which uses all tables in $\cup_{i=0}^{u} D^i$ during continual training. All three methods adopt our incremental tabid assignment strategy for new tables, and continually train the model from the last checkpoint. Lastly, we include ReIndex, which re-clusters all tables in $\cup_{i=0}^{u} D^i$ to reassign tabids, and re-trains the model from scratch, serving as a theoretical upper bound.

Metrics. Following previous studies [7, 37], we employ three metrics: (i) average performance ($AP@K$) to measure the average performance on all existing tables, (ii) forgetting ($FT@K$) to measure the performance decline on old tables after learning from a new batch D^u; and (iii) learning performance ($LP@K$) to measure the ability to learn after indexing a new batch of tables D^u. Formally, we denote $P@K$ on D^w (tested by $Test^w$) after indexing the new batch D^u as $P_{u,w}@K$, and $P@K$ on $\cup_{w=0}^{u} D^w$ (tested by $\cup_{w=0}^{u} Test^w$) after indexing D^u as $AP@K$. The other two metrics

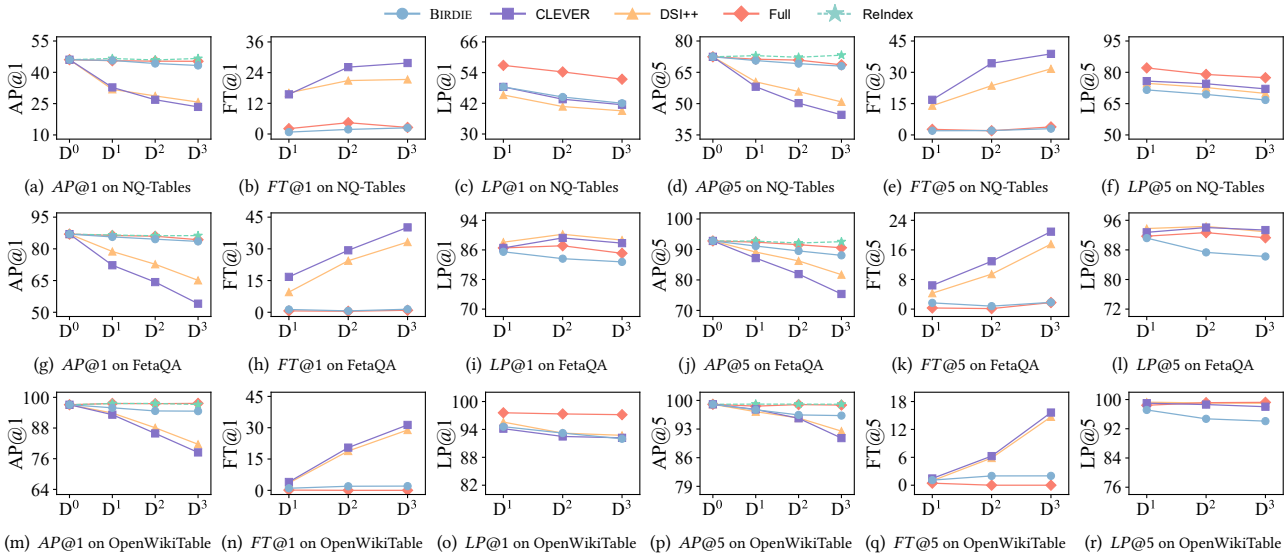


Figure 7: Performance of index update under dynamic scenario.

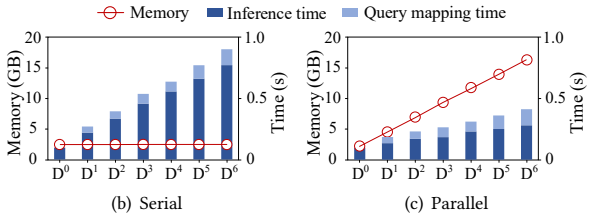


Figure 8: Runtime and memory usage of online search.

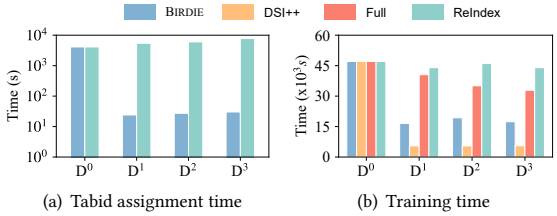


Figure 9: Runtime of index update on NQ-Tables.

are defined as follows: $LP@K = \frac{1}{u}\sum_{w=1}^{u} P_{w,w}@K$, and $FT@K = \frac{1}{u}\sum_{w=0}^{u-1} \max_{w' \in \{0,\cdots,u-1\}} \left(P_{w',w}@K - P_{u,w}@K\right)$.

Note that, since ReIndex re-trains the model from scratch instead of continual learning, we only report its *AP* values.

Effectiveness. The results are illustrated in Figure 7. In terms of *AP* (the higher the better), BIRDIE outperforms both CLEVER and DSI++, approaching the performance of Full and ReIndex, which use all previous tables. Interestingly, DSI++, which employs random sampling of old tables during continual learning, surpasses CLEVER, which utilizes a similarity-based sampling approach. Regarding *FT* (the lower the better), BIRDIE reduces forgetting by over 90% compared to CLEVER and DSI++. Additionally, as new batches are introduced, forgetting intensifies for CLEVER and DSI++, since each model update can lead to the forgetting of some old tables, accumulating with ongoing parameter updates. In contrast, BIRDIE implements parameter isolation, which keeps memory units independent, resulting in reduced forgetting. For *LP* (the higher the better), BIRDIE demonstrates comparable *LP*@1 and slightly lower *LP*@5 compared to other continual-learning-based methods. The higher *LP* of CLEVER and DSI++ indicates that replaying only a portion of old tables during continual learning tends to favor new table retention. This prioritization of new tables leads to increased forgetting, ultimately resulting in poorer average performance. In contrast, BIRDIE strikes a promising balance between *FT* and *LP*, yielding higher *AP*. Note that *LP* measures accuracy in an extreme scenario where the workload consists solely of new memory retrieval, implying that the older data may have become outdated. In Appendix C.2, we present a simple extension to improve BIRDIE's performance as old memories become obsolete.

Table 6: Memory usage (GB) on NQ-Tables for index update.

Phases	D^0	D^1	D^2	D^3
ITA (CPU Memory)	1.31	0.46	0.48	0.51
ITA w/o TC (CPU Memory)	1.31	2.16	2.52	2.74
Training (GPU Memory)	15.19	8.04	8.04	8.04
Inference-Serial (GPU Memory)	2.33	2.33	2.33	2.33
Inference-Parallel (GPU Memory)	2.33	4.67	7.00	9.47

Runtime. Figure 9 illustrates the time taken for index update across sequential batches D^0 to D^3 on NQ-Tables. Runtime for other datasets can be found in Appendix C.1. For tabid assignment, CLEVER, DSI++, and Full follow BIRDIE, which explains why we only report the time taken by BIRDIE and ReIndex. It is observed that BIRDIE's running time is 1-2 orders of magnitude shorter than that of ReIndex. For training time, DSI++ is the fastest (the runtime of CLEVER is similar and omitted), utilizing less data than Full and ReIndex and fine-tuning the model from the last checkpoint, though its effectiveness is limited. BIRDIE demonstrates greater efficiency than Full and ReIndex, while achieving comparable average performance.

Memory Usage. Table 6 reports memory usage on NQ-Tables. ITA w/o TC refers to ITA without TC optimization presented in Section 4.1. Results show that our TC optimization effectively reduces

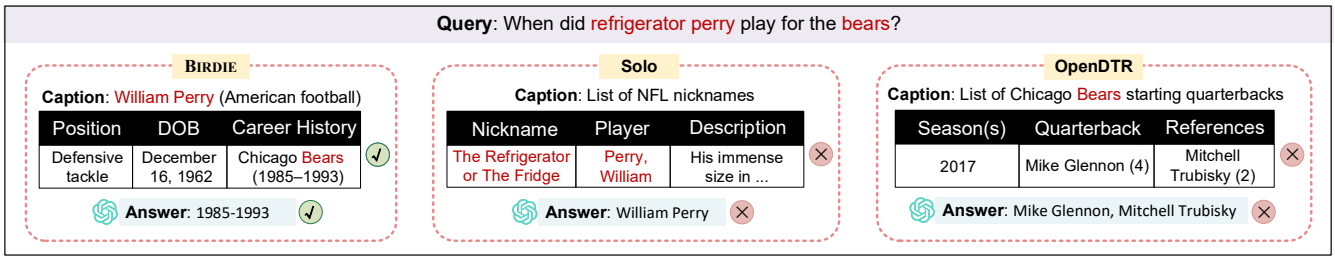


Figure 10: A case of NL-driven table discovery and the results of table QA using GPT-4o.

memory usage by 5×. For training, we evaluate memory usage with the default batch size of 64. The training process on D^1–D^3 is memory-efficient with LoRA techniques. Depending on available memory resources, Birdie supports two inference paradigms: the serial paradigm, which requires constant memory, and the parallel paradigm, which scales linearly with the number of updates.

Scalability. To evaluate the scalability during online search, we use the NQ-Tables dataset to create six incremental batches. Specifically, we sample 40% of the tables from NQ-Tables as D^0, and the remaining 60% tables are evenly divided into six batches $\{D_i\}_{i=1}^{6}$. Figure 8 illustrates the memory usage and runtime. In low-resource scenarios with serial inference, memory usage remains constant, while runtime scales linearly with the number of models. Conversely, in rich-resource scenarios with parallel inference, runtime remains relatively stable, while memory usage scales linearly with the number of models. This is because multiple copies of the base model are loaded into memory simultaneously, as discussed in Section 5.2. The number of synthetic queries per candidate table is a small constant B (e.g., 20 in our implementation), and each model generates only a limited number of candidate tables, making query mapping process both time and memory efficient.

7 CASE STUDY

We present a case of using Birdie, Solo, and OpenDTR to retrieve the top table from NQ-Tables in response to a given NL query. GPT-4o is then employed as a reasoning tool through the OpenAI API to answer the query based on the contents in the retrieved table. Figure 10 illustrates the case. Note that we only show the row relevant to the answer due to space limitation.

It is observed that only Birdie successfully retrieves the correct table, and GPT-4o provides the correct answer only when it receives the correct table. Solo encodes each cell-attribute-cell triplet as an embedding and measures similarity between the query and triplet embeddings. In this instance, the triplet in the table "List of NFL nicknames" exhibits high similarity to the query. However, it lacks time information, rendering it incapable of answering the user's query. This highlights the limitation of dense methods that do not facilitate deep query-table interactions. OpenDTR, which encodes the entire table as a single vector, struggles to capture detailed table information. As a result, the table "Chicago Bears" shows a higher similarity to the query than the correct table. In contrast, Birdie optimizes both the indexing and search processes jointly and engages in deep query-table interactions during model training. Therefore, it comprehensively understands the given NL query and successfully identifies the correct table.

8 RELATED WORK

Table Discovery. Table discovery has been extensively researched within the data management community [15, 20]. A prevalent line of table discovery is query-driven discovery, which includes: (i) keyword-based table search [3, 5] that aims to identify web tables related to specified keywords, utilizing metadata such as table headers and column names; (ii) table-driven search which locates target tables within a large data lake that can be joined [14, 19, 65] or unioned [16, 26, 39] with a given query table; and (iii) NL-query-driven table search [22, 27, 52].

NL-driven table discovery offers a user-friendly interface that allows users to express their needs more precisely. Existing NL-driven table discovery methods [22, 27, 52] typically follow a traditional representation-index-search pipeline. The encoder in the representation phase plays a crucial role in search accuracy. For instance, OpenDTR [22] uses TAPAS [23] as the backbone for its bi-encoder. OpenWikiTable [27] offers various options for query and table encoders. However, representing a table as a single vector can sometimes be insufficiently expressive. To address this, Solo [52] encodes each cell-attributes-cell triplet within the table into a fixed-dimensional embedding and retrieves similar triplet embeddings to the query embedding, followed by aggregation of triplets-to-table. However, the lack of deep query-table interactions during retrieval hinders further performance improvements.

Another line of NL-based table search literature focuses on the re-ranking [9, 51, 59]. Utilizing a cross-encoder, they input both the query and candidate table to obtain embeddings for each query-table pair. This process enhances accuracy due to the deep query-table interactions but lacks of scalability for first-stage retrieval.

Differentiable Search Index. Differentiable search index (DSI) [48] sparks a novel search paradigm that unifies the indexing and search within a single Transformer architecture. It was initially proposed for document retrieval [48, 53, 66] and has been applied in scenarios like retrieval-augmented generation (RAG) [31], recommendation systems [45], etc. To the best of our knowledge, Birdie is the first attempt to perform table discovery using DSI, taking into account the unique properties of tabular data to automate the collection of training data. Real-world applications often involve dynamically changing corpora. However, in DSI, which encodes all corpus information into model parameters, indexing new corpora inevitably leads to the forgetting of old ones. To mitigate catastrophic forgetting, some recent studies [7, 37] propose replay-based solutions that sample some old data and combine it with new data for continual learning. However, these methods often struggle to balance indexing new data and retaining old memories, resulting in suboptimal average performance. In contrast, Birdie designs a parameter isolation method that ensures the independence of each memory unit, thus achieving a promising average performance.

9 CONCLUSIONS

We present Birdie, an effective NL-driven table discovery framework using a differentiable search index. We first introduce a two-view-based tabid assignment method to assign a unique table identifier to each table, considering the semantics of both the metadata and instance-data of tables. Then, we propose a LLM-based query generation method tailored for tabular data to construct synthetic NL queries for DSI model training. To accommodate the continual indexing of dynamic tables, we design an index update strategy via parameter isolation. Comprehensive experiments confirm that Birdie significantly outperforms the SOTA methods, and our parameter isolation strategy alleviates catastrophic forgetting and achieves better average performance than competitors. In the future, we plan to develop acceleration methods to speed up the offline training phase of Birdie and reduce costs.

REFERENCES

[1] 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. https://huggingface.co/sentence-transformers/stsb-roberta-base.

[2] 2024. Introducing Meta Llama 3: The most capable openly available LLM to date. https://ai.meta.com/blog/meta-llama-3.

[3] Marco D. Adelfio and Hanan Samet. 2013. Schema Extraction for Tabular Data on the Web. *Proc. VLDB Endow.* 6, 6 (2013), 421–432.

[4] Gilbert Badaro, Mohammed Saeed, and Paolo Papotti. 2023. Transformers for Tabular Data Representation: A survey of models and applications. *TACL* 11 (2023), 227–249.

[5] Dan Brickley, Matthew Burgess, and Natasha F. Noy. 2019. Google Dataset Search: Building a search engine for datasets in an open Web ecosystem. In *WWW*. 1365–1375.

[6] Shaofeng Cai, Kaiping Zheng, Gang Chen, H. V. Jagadish, Beng Chin Ooi, and Meihui Zhang. 2021. ARM-Net: Adaptive Relation Modeling Network for Structured Data. In *SIGMOD*. 207–220.

[7] Jiangui Chen, Ruqing Zhang, Jiafeng Guo, Maarten de Rijke, Wei Chen, Yixing Fan, and Xueqi Cheng. 2023. Continual Learning for Generative Retrieval over Dynamic Corpora. In *CIKM*. 306–315.

[8] Wenhu Chen, Hongmin Wang, Jianshu Chen, Yunkai Zhang, Hong Wang, Shiyang Li, Xiyou Zhou, and William Yang Wang. 2020. TabFact: A Large-scale Dataset for Table-based Fact Verification. In *ICLR*.

[9] Zhiyu Chen, Mohamed Trabelsi, Jeff Heflin, Yinan Xu, and Brian D. Davison. 2020. Table Search Using a Deep Contextualized Language Model. In *SIGIR*. 589–598.

[10] Tianji Cong, Madelon Hulsebos, Zhenjie Sun, Paul Groth, and H. V. Jagadish. 2023. Observatory: Characterizing Embeddings of Relational Tables. *Proc. VLDB Endow.* 17, 4 (2023), 849–862.

[11] Xiang Deng, Huan Sun, Alyssa Lees, You Wu, and Cong Yu. 2020. TURL: Table Understanding through Representation Learning. *Proc. VLDB Endow.* 14, 3 (2020), 307–319.

[12] Yuhao Deng, Chengliang Chai, Lei Cao, Qin Yuan, Siyuan Chen, Yanrui Yu, Zhaoze Sun, Junyi Wang, Jiajun Li, Ziqi Cao, Kaisen Jin, Chi Zhang, Yuqing Jiang, Yuanfang Zhang, Yuping Wang, Ye Yuan, Guoren Wang, and Nan Tang. 2024. LakeBench: A Benchmark for Discovering Joinable and Unionable Tables in Data Lakes. *Proc. VLDB Endow.* 17, 8 (2024), 1925–1938.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[14] 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 (2023), 2458–2470.

[15] Grace Fan, Jin Wang, Yuliang Li, and Renée J. Miller. 2023. Table Discovery in Data Lakes: State-of-the-art and Future Directions. In *SIGMOD*. 69–75.

[16] 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 (2023), 1726–1739.

[17] Mor Geva, Roei Schuster, Jonathan Berant, and Omer Levy. 2021. Transformer Feed-Forward Layers Are Key-Value Memories. In *EMNLP*. 5484–5495.

[18] Yuxiang Guo, Lu Chen, Zhengjie Zhou, Baihua Zheng, Ziquan Fang, Zhikun Zhang, Yuren Mao, and Yunjun Gao. 2023. CampER: An Effective Framework for Privacy-Aware Deep Entity Resolution. In *SIGKDD*. 626–637.

[19] Yuxiang Guo, Yuren Mao, Zhonghao Hu, Lu Chen, and Yunjun Gao. 2025. Snoopy: Effective and Efficient Semantic Join Discovery via Proxy Columns. *arXiv preprint arXiv:2502.16813* (2025).

[20] Rihan Hai, Christos Koutras, Christoph Quix, and Matthias Jarke. 2023. Data Lakes: A Survey of Functions and Systems. *IEEE Trans. Knowl. Data Eng.* 35, 12 (2023), 12571–12590.

[21] Tianxing He, Jingzhao Zhang, Zhiming Zhou, and James R. Glass. 2019. Quantifying Exposure Bias for Neural Language Generation. *CoRR* abs/1905.10617 (2019).

[22] Jonathan Herzig, Thomas Müller, Syrine Krichene, and Julian Martin Eisenschlos. 2021. Open Domain Question Answering over Tables via Dense Retrieval. In *NAACL-HLT*. 512–519.

[23] Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Martin Eisenschlos. 2020. TaPas: Weakly Supervised Table Parsing via Pre-training. In *ACL*. 4320–4333.

[24] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *ICLR*.

[25] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *EMNLP*. 6769–6781.

[26] Aamod Khatiwada, Grace Fan, Roee Shraga, Zixuan Chen, Wolfgang Gatterbauer, Renée J. Miller, and Mirek Riedewald. 2023. SANTOS: Relationship-based Semantic Table Union Search. *Proc. ACM Manag. Data* 1, 1 (2023), 9:1–9:25.

[27] Sunjun Kweon, Yeonsu Kwon, Seonhee Cho, Yohan Jo, and Edward Choi. 2023. Open-WikiTable : Dataset for Open Domain Question Answering with Complex Reasoning over Table. In *ACL (Findings)*. 8285–8297.

[28] Rémi Lebret, David Grangier, and Michael Auli. 2016. Neural Text Generation from Structured Data with Application to the Biography Domain. In *EMNLP*. 1203–1213.

[29] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461* (2019).

[30] Peng Li, Yeye He, Dror Yashar, Weiwei Cui, Song Ge, Haidong Zhang, Danielle Rifinski Fainman, Dongmei Zhang, and Surajit Chaudhuri. 2024. Table-GPT: Table Fine-tuned GPT for Diverse Table Tasks. *Proc. ACM Manag. Data* 2, 3 (2024), 176.

[31] Xiaoxi Li, Zhicheng Dou, Yujia Zhou, and Fangchao Liu. 2024. CorpusLM: Towards a Unified Language Model on Corpus for Knowledge-Intensive Tasks. In *SIGIR*. 26–37.

[32] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joey Gonzalez. 2020. Train big, then compress: Rethinking model size for efficient training and inference of transformers. In *ICML*. 5958–5968.

[33] Zibo Liang, Xu Chen, Yuyang Xia, Runfan Ye, Haitian Chen, Jiandong Xie, and Kai Zheng. 2024. DACE: A Database-Agnostic Cost Estimator. In *ICDE*. 4925–4937.

[34] Yiming Lin, Yeye He, and Surajit Chaudhuri. 2023. Auto-BI: Automatically Build BI-Models Leveraging Local Join Prediction and Global Schema Graph. *Proc. VLDB Endow.* 16, 10 (2023), 2578–2590.

[35] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2024. A Survey of NL2SQL with Large Language Models: Where are we, and where are we going? *arXiv preprint arXiv:2408.05109* (2024).

[36] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. 2022. PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods. https://github.com/huggingface/peft.

[37] Sanket Vaibhav Mehta, Jai Gupta, Yi Tay, Mostafa Dehghani, Vinh Q. Tran, Jinfeng Rao, Marc Najork, Emma Strubell, and Donald Metzler. 2023. DSI++: Updating Transformer Memory with New Documents. In *EMNLP*. 8198–8213.

[38] Linyong Nan, Chiachun Hsieh, Ziming Mao, Xi Victoria Lin, Neha Verma, Rui Zhang, Wojciech Kryściński, Hailey Schoelkopf, Riley Kong, Xiangru Tang, et al. 2022. FeTaQA: Free-form table question answering. *TACL* 10 (2022), 35–49.

[39] Fatemeh Nargesian, Erkang Zhu, Ken Q. Pu, and Renée J. Miller. 2018. Table Union Search on Open Data. *Proc. VLDB Endow.* 11, 7 (2018), 813–825.

[40] Jianmo Ni, Gustavo Hernandez Abrego, Noah Constant, Ji Ma, Keith Hall, Daniel Cer, and Yinfei Yang. 2022. Sentence-T5: Scalable Sentence Encoders from Pretrained Text-to-Text Models. In *ACL Findings*. 1864–1874.

[41] Rodrigo Nogueira, Jimmy Lin, and AI Epistemic. 2019. From doc2query to docTTTTTquery. *Online preprint* 6, 2 (2019).

[42] OpenAI. 2023. GPT-4 Technical Report. https://arxiv.org/abs/2303.08774.

[43] Panupong Pasupat and Percy Liang. 2015. Compositional Semantic Parsing on Semi-Structured Tables. In *ACL*. 1470–1480.

[44] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *JMLR* 21, 140 (2020), 1–67.

[45] Shashank Rajput, Nikhil Mehta, Anima Singh, Raghunandan Keshavan, Trung Vu, Lukasz Heidt, Lichan Hong, Yi Tay, Vinh Q Tran, Jonah Samost, et al. 2023. Recommender systems with generative retrieval. In *NeurIPS*. 10299–10315.

[46] General Data Protection Regulation. 2016. Regulation EU 2016/679 of the European Parliament and of the Council of 27 April 2016. *Official Journal of the*

European Union (2016).

[47] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph Gonzalez, and Ion Stoica. 2024. SLoRA: Scalable Serving of Thousands of LoRA Adapters. In *MLSys*.

[48] Yi Tay, Vinh Q Tran, Mostafa Dehghani, Jianmo Ni, Dara Bahri, Harsh Mehta, Zhen Qin, Kai Hui, Zhe Zhao, Jai Gupta, et al. 2022. Transformer memory as a differentiable search index. In *NeurIPS*. 21831–21843.

[49] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NIPS*. 5998–6008.

[51] Fei Wang, Kexuan Sun, Muhao Chen, Jay Pujara, and Pedro A. Szekely. 2021. Retrieving Complex Tables with Multi-Granular Graph Representation Learning. In *SIGIR*. 1472–1482.

[52] Qiming Wang and Raul Castro Fernandez. 2023. Solo: Data Discovery Using Natural Language Questions Via A Self-Supervised Approach. *Proc. ACM Manag. Data* 1, 4 (2023), 262:1–262:27.

[53] Yujing Wang, Yingyan Hou, Haonan Wang, Ziming Miao, Shibin Wu, Hao Sun, Qi Chen, Yuqing Xia, Chengmin Chi, Guoshuai Zhao, et al. 2022. A neural corpus indexer for document retrieval. In *NeurIPS*. 25600–25614.

[54] Ronald J Williams and David Zipser. 1989. A learning algorithm for continually running fully recurrent neural networks. *Neural computation* 1, 2 (1989), 270–280.

[55] Sam Wiseman and Alexander M Rush. 2016. Sequence-to-sequence learning as beam-search optimization. *arXiv preprint arXiv:1606.02960* (2016).

[56] Linting Xue, Noah Constant, Adam Roberts, Mihir Kale, Rami Al-Rfou, Aditya Siddhant, Aditya Barua, and Colin Raffel. 2021. mT5: A Massively Multilingual Pre-trained Text-to-Text Transformer. In *NAACL-HLT*. 483–498.

[57] Hansi Zeng, Chen Luo, Bowen Jin, Sheikh Muhammad Sarwar, Tianxin Wei, and Hamed Zamani. 2024. Scalable and Effective Generative Information Retrieval. In *WWW*. 1441–1452.

[58] Chao Zhang, Yuren Mao, Yijiang Fan, Yu Mi, Yunjun Gao, Lu Chen, Dongfang Lou, and Jinshu Lin. 2024. FinSQL: Model-Agnostic LLMs-based Text-to-SQL Framework for Financial Analysis. In *SIGMOD*. 93–105.

[59] Shuo Zhang and Krisztian Balog. 2018. Ad Hoc Table Retrieval using Semantic Similarity. In *WWW*. 1553–1562.

[60] Yunjia Zhang, Jordan Henkel, Avrilia Floratou, Joyce Cahoon, Shaleen Deep, and Jignesh M. Patel. 2024. ReAcTable: Enhancing ReAct for Table Question Answering. *Proc. VLDB Endow.* 17, 8 (2024), 1981–1994.

[61] Zhengxuan Zhang, Weixing Mai, Haoliang Xiong, Chuhan Wu, and Yun Xue. 2023. A Token-wise Graph-based Framework for Multimodal Named Entity Recognition. In *ICME*. 2153–2158.

[62] Zhengxuan Zhang, Yin Wu, Yuyu Luo, and Nan Tang. 2024. MAR: Matching-Augmented Reasoning for Enhancing Visual-based Entity Question Answering. In *EMNLP*. 1520–1530.

[63] Mingyu Zheng, Xinwei Feng, Qingyi Si, Qiaoqiao She, Zheng Lin, Wenbin Jiang, and Weiping Wang. 2024. Multimodal Table Understanding. *CoRR* abs/2406.08100 (2024).

[64] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103* (2017).

[65] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In *SIGMOD*. 847–864.

[66] Shengyao Zhuang, Houxing Ren, Linjun Shou, Jian Pei, Ming Gong, Guido Zuccon, and Daxin Jiang. 2022. Bridging the Gap Between Indexing and Retrieval for Differentiable Search Index with Query Generation. *CoRR* abs/2206.10128 (2022).

APPENDIX

A PROOF OF LEMMA

Lemma 1. *The upper and lower bounds of the cluster radius r' after the insertion of $\mathbf{h}_{new}$ are given by $r + \text{dist}(\mathbf{c}, \mathbf{c}')$ and $\text{dist}(\mathbf{h}_{new}, \mathbf{c}')$, respectively.*

Proof. After inserting $\mathbf{h}_{new}$ into the cluster, the farthest embedding from the new center $\mathbf{c}'$ could either be an old embedding $\mathbf{h}_{old}$ within this cluster, or the newly inserted $\mathbf{h}_{new}$. In the first case, applying the triangle inequality yields $r' = \text{dist}(\mathbf{h}_{old}, \mathbf{c}') \leq \text{dist}(\mathbf{h}_{old}, \mathbf{c}) + \text{dist}(\mathbf{c}, \mathbf{c}') \leq r + \text{dist}(\mathbf{c}, \mathbf{c}')$. In the second case, we can also infer that $r' = \text{dist}(\mathbf{h}_{new}, \mathbf{c}') \leq \text{dist}(\mathbf{h}_{new}, \mathbf{c}) + \text{dist}(\mathbf{c}, \mathbf{c}') = d + \text{dist}(\mathbf{c}, \mathbf{c}')$. Since $\delta < d \leq r$, we have $r' \leq r + \text{dist}(\mathbf{c}, \mathbf{c}')$.

The distance from the newly inserted $\mathbf{h}_{new}$ to the new center $\mathbf{c}'$ provides a minimum bound for r'. □

B SENSITIVITY STUDY

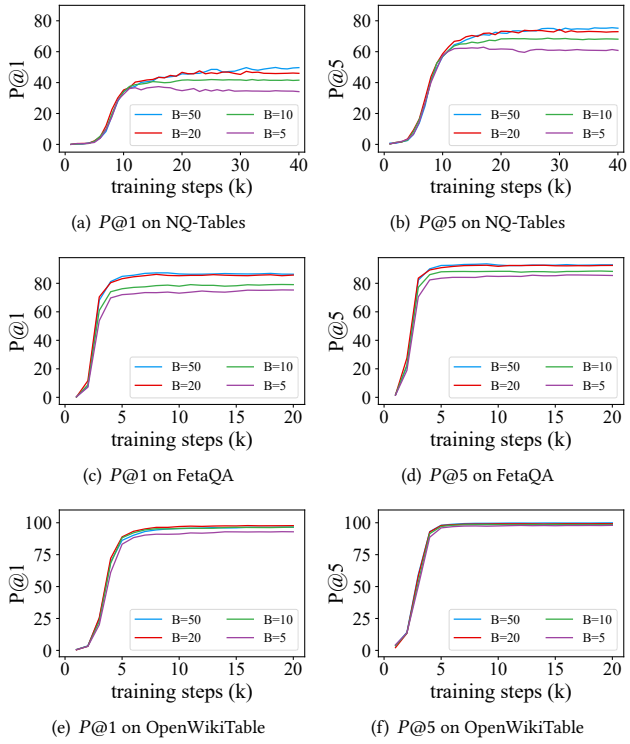
We provide some practical guidelines for the key hyperparameters and perform a sensitivity study.

For model training, most hyperparameters are set based on either default recommended values or commonly used configurations from previous studies. For example, the rank d_r of LoRA is set to 8 following the default setting in PEFT library [36]; the learning rate and beam size are chosen based on previous studies [48, 66]. For query generation, the size B of queries for each table is a tunable hyperparameter. We explore the impact of the number B of generated queries per table on Birdie's performance by varying B in the range of {5, 10, 20, 50}. Figure 11 illustrates the performance changes over the training steps. As expected, generating more queries generally improves accuracy by providing more comprehensive coverage of table information. However, performance does not continuously increase with the number of queries; for instance, generating 20 queries yields similar results to generating 50. Thus, we generate 20 queries for each table by default.

During tabid assignment, key hyperparameters include the number k of clusters, the maximum size c of leaf clusters, and the depth l of the first view. Recall that each tabid is represented as $\mathbf{s} = (s_1, \ldots s_d)$, where $s_i \in [0, k-1]$ for $i < d$ and $s_d \in [0, c-1]$. Thus, k and c determine the value ranges for $s_1..s_{d-1}$ and s_d, respectively. We recommend setting $k = c$ to ensure the consistency in the range of each token within tabids. To configure l and k, a larger l allows finer differentiation of first-view embeddings during the clustering. Preliminary experiments suggest that $l = 2$ is generally sufficient. The clustering tree is expected to have a depth of $2l$ to ensure balanced clustering for both views, resulting in a maximum tabid length $d = 2l + 1$. In a perfect k-ary clustering tree, where each leaf node contains k embedding vectors, the maximum number of tables N it can represent is k^{2l+1} if both views consist of l levels, or k^{l+1} if only the first view is considered. To ensure balanced clustering across two views, we recommend setting k within the range $\left(N^{\frac{1}{2l+1}}, N^{\frac{1}{l+1}}\right)$. Additionally, we perform a sensitivity study to explore impact of varying k within this range. As shown in Table 7, search accuracy remains stable, demonstrating Birdie's robust performance.

Table 7: The accuracy of Birdie under different k within the recommended range.

NQ-Tables			FetaQA			OpenWikiTable		
k	P@1	P@5	k	P@1	P@5	k	P@1	P@5
16	45.10	70.23	8	84.77	90.86	12	95.71	97.88
32	46.64	73.21	14	86.32	92.76	20	97.20	99.06
48	45.06	71.22	20	86.27	92.56	28	96.32	98.11



(a) P@1 on NQ-Tables (b) P@5 on NQ-Tables

(c) P@1 on FetaQA (d) P@5 on FetaQA

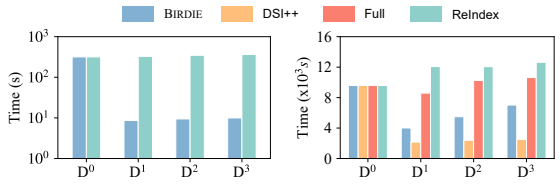
(e) P@1 on OpenWikiTable (f) P@5 on OpenWikiTable

Figure 11: Performance with different # of generated queries.

C SUPPLEMENTARY EXPERIMENTAL RESULTS

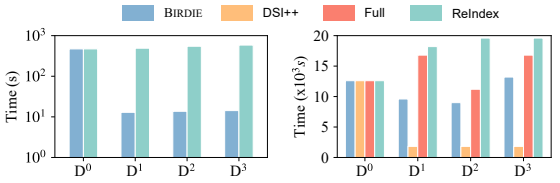
C.1 Runtime of index update

Figure 12 and Figure 13 illustrate the time taken for index update across sequential batches D^0 to D^3 on FetaQA and OpenWikiTable. The observation is consistent across different datasets. Specifically, for tabid assignment, Birdie's running time is 1-2 orders of magnitude shorter than that of ReIndex. This is because ReIndex needs to assign ids for all the tables from scratch, while Birdie only assigns tabids to new tables without affecting the ids of old tables. For training time, DSI++ is the fastest (the runtime of CLEVER is similar and omitted), utilizing less data than Full and ReIndex and fine-tuning the model from the last checkpoint, though its effectiveness is limited. Birdie demonstrates greater efficiency than Full and ReIndex, while achieving comparable average performance.



(a) Tabid assignment time (b) Training time

Figure 12: Runtime of index update on FetaQA.



(a) Tabid assignment time (b) Training time

Figure 13: Runtime of index update on OpenWikiTable.

Table 8: *LP*@1 results of Birdie with new memory prioritization and baseline methods.

Methods	NQ-Tables			FetaQA			OpenWikiTable		
	D^1	D^2	D^3	D^1	D^2	D^3	D^1	D^2	D^3
DSI++	45.26	40.74	39.06	88.08	**90.23**	88.65	95.58	93.20	92.73
CLEVER	48.42	43.60	41.45	86.53	89.24	87.83	94.16	92.48	92.2
Birdie	**53.68**	**49.69**	**47.41**	**89.12**	89.48	**88.98**	**98.43**	**97.69**	**97.36**

Table 9: *LP*@5 results of Birdie with new memory prioritization and baseline methods.

Methods	NQ-Tables			FetaQA			OpenWikiTable		
	D^1	D^2	D^3	D^1	D^2	D^3	D^1	D^2	D^3
DSI++	74.73	72.71	69.90	93.78	**94.34**	92.90	99.28	99.05	98.99
CLEVER	75.79	74.53	72.07	92.75	94.05	93.36	99.00	98.63	98.04
Birdie	**84.21**	**80.90**	**77.74**	**95.34**	93.86	**93.40**	**99.90**	**99.34**	**98.87**

C.2 Effectiveness in the absence of old retrieval

In the extreme case where old memory has expired, Birdie can easily adapt by adjusting the selection probability of candidate tables $\{\mathcal{T}^0, \ldots, \mathcal{T}^u\}$ to improve its performance, where $\mathcal{T}^i = \{T_1^i, \ldots, T_K^i\} \subset D^i$ represents tables generated by model $\mathcal{M}^i$ given an input query. Specifically, when old memory has been rarely retrieved over a period, Birdie can increase the probability of selecting $\mathcal{T}^w \subset D^w$ as the final result, where D^w is a new batch. To demonstrate the effectiveness of this adjustment, we have implemented a straightforward strategy. After query mapping, we define Q^z as the set with the largest number of queries among the top-n_q similar queries, and Q^w as the second largest. We prioritize $\mathcal{T}^w$ over $\mathcal{T}^z$ if D^w is a new batch and D^z is an old sub-repository. The results, presented in Table 8 and Table 9, demonstrate that this simple extension allows Birdie to outperform baselines in most cases, highlighting its robustness in extreme scenarios.

D SUPPLEMENTARY CASE STUDY

D.1 Examples of query-table pairs from OpenWikiTable

All methods perform better on OpenWikiTable compared to the other two datasets. This can be attributed to: (i) the queries in OpenWikiTable benchmark contain detailed descriptions of the ground truth table (i.e. page title, section title, caption), making it easier to locate the relevant table using textual similarity, even using a dense passage retriever; and (ii) a remarkable 99.8% (24,634 out of 24,680) of tables in OpenWikiTable have unique titles, simplifying the task of identifying the correct table using title information alone. While reasoning across multiple rows and columns in OpenWikiTable is challenging for QA tasks, this does not make the table discovery process equivalently difficult. Instead, table discovery is relatively straightforward for OpenWikiTable due to the substantial overlap between the queries and the titles of ground truth tables. Table 10 presents examples of query-table pairs from OpenWikiTable, highlighting why table discovery in this dataset is simpler.

D.2 Cases of table discovery and table QA

We include additional two representative cases to highlight the advantages of Birdie, illustrated in Figure 14. In Case 2, Birdie successfully captures the key intent of the query rather than blindly matching overlapping terms (e.g., “army") between the query and table attributes. In Case 3, despite the presence of a specific sports term (e.g., “NBA Finals”) in the query, identifying the correct table remains challenging, as multiple tables contain identical or similar titles with this jargon. Nevertheless, only Birdie identifies the correct table. While the table QA result from the table retrieved by Solo is correct, it is coincidental. Specifically, Solo retrieves a table showing the total number of wins for each team in the 2000 NBA Finals but fails to specify the winner of individual games, such as Game 4. The correct answer is coincidental because the team with 4 total wins happened to win Game 4.

Table 10: Examples of query and its corresponding table's title information in OpenWikiTable

Query	Title of the corresponding table
What is the pinnacle height for Metcalf, Georgia among the list of tallest structures in the world (past or present) 600 m and taller (1,969 ft) ?	`Page Title:` List of tallest structures in the world `Caption:` Structures (past or present) 600 m and taller (1,969 ft)
From the 2009 Supersport World Championship season , if the Autodromo Nazionale Monza is the circuit, what's the report?	`Page Title:` 2009 Supersport World Championship season `Caption:` Race calendar and results
What was the outcome of the district represented by Thomas R. Gold in the United States House of Representatives elections of 1812 ?	`Page Title:` United States House of Representatives elections, 1812 `Caption:` New York
What is the type of electronics with the Gamecube platform under Harry Potter and the Chamber of Secrets ?	`Page Title:` Harry Potter and the Chamber of Secrets `Caption:` Video game

Figure 14: NL-driven table discovery and QA results using GPT-4o, showing only the row serving as evidence for the QA result.