Bridging the Semantic Gap with SQL Query Logs in Natural Language Interfaces to Databases

Christopher Baik, H. V. Jagadish
University of Michigan
Ann Arbor, MI, USA
Email: {cjbaik, jag}@umich.edu

Yunyao Li
IBM Research - Almaden
San Jose, CA, USA
Email: yunyaoli@us.ibm.com

Abstract—

A critical challenge in constructing a *natural language interface to database (NLIDB)* is bridging the semantic gap between a natural language query (NLQ) and the underlying data. Two specific ways this challenge exhibits itself is through *keyword mapping* and *join path inference*. Keyword mapping is the task of mapping individual keywords in the original NLQ to database elements (such as relations, attributes or values). It is challenging due to the ambiguity in mapping the user's mental model and diction to the schema definition and contents of the underlying database. *Join path inference* is the process of selecting the relations and join conditions in the FROM clause of the final SQL query, and is difficult because NLIDB users lack the knowledge of the database schema or SQL and therefore cannot explicitly specify the intermediate tables and joins needed to construct a final SQL query. In this paper, we propose leveraging information from the SQL query log of a database to enhance the performance of existing NLIDBs with respect to these challenges. We present a system TEMPLAR that can be used to augment existing NLIDBs. Our extensive experimental evaluation demonstrates the effectiveness of our approach, leading up to 138% improvement in top-1 accuracy in existing NLIDBs by leveraging SQL query log information.

I. INTRODUCTION

From business intelligence to web design to answering everyday factual questions, today's technology-rich world is overwhelmingly dependent on database systems. End-users have various means to query these databases for information, whether through voice-based assistants such as Google Assistant[1], Apple Siri[2], or Amazon Echo[3]; queries in a structured language such as SQL or LINQ; or keyword searches on search engines such as Google or Bing.

While relational databases have been around for decades, query languages to access such databases, such as SQL, are unlikely to ever become common knowledge for the average end-user. Accordingly, a major goal in this area is to provide a means for users to query databases using everyday language via a *natural language interface to database (NLIDB)*.

The task of an NLIDB has been primarily modeled as the problem of translating a natural language query (NLQ) into a SQL query. State-of-the-art systems developed to solve this task take one of two architectural approaches: (1) the *pipeline approach* of converting an NLQ into intermediate representations then mapping these representations to SQL (e.g. [22], [32], [33], [41]), and (2) the *deep learning approach* of using an end-to-end neural network to perform the translation (e.g. [38], [40], [44]) .

However, as pointed out by [23], one fundamental challenge in supporting NLIDBs is *bridging the semantic gap* between a NLQ and the underlying data. When translating NLQ to SQL, this challenge arises in two specific problems: (1) *keyword mapping* and (2) *join path inference*. Keyword mapping is the task of mapping individual keywords in the original NLQ to database elements (such as relations, attributes or values). It is a challenging task because of the ambiguity in mapping the user's mental model and diction to the schema definition and contents of the database. *Join path inference* is the process of selecting the relations and join conditions in the FROM clause of the final SQL query, and is difficult because NLIDB users do not have a knowledge of the database schema or SQL and therefore cannot explicitly specify the intermediate tables and joins needed to construct a final SQL query.

Table I summarizes state-of-the-art systems and their strategy to handle each step of NLQ to SQL translation. The upper half lists pipeline-based systems, where each subproblem is explicitly handled, while the lower half are deep learning systems which implicitly tackle these challenges by the choice of input representation and network architecture. The *keyword mapping* task is split into the *Rel/Attr Mapping* and *Value Mapping* columns because some systems have independent procedures for handling each. Some common patterns emerge:

- For *keyword mapping*, the vast majority of systems make use of a lexical database such as WordNet [28] or a word embedding model [27], [31].
- *Join path inference* is primarily handled via user interaction [22] or heuristics such as selecting the shortest join path [38] or hand-written repair rules [41].

While each of these approaches works reasonably well, there is still significant room for improvement. For example:

Example 1. *John issues an NLQ: "Find papers in the Databases domain" on an academic database (Figure 1) using a pipeline NLIDB. John's intended SQL query is:*

```
SELECT p.title
FROM publication p, publication_keyword pk
```

System	NLQ Pre-processing	Rel/Attr Mapping	Value Mapping	Join Path Inference	SQL Post-processing
Precise [32]	Tokenizer + Charniak [10] parser	WordNet [28]	Same as rel/attr	Max-flow algorithm + User interaction	N/A
NaLIR [22]	Stanford Parser [11]	WordNet [28] + User interaction	Same as rel/attr	Preset path weights + User interaction	Query tree heuristics + User interaction
SQLizer [41]	Sempre [5]	word2vec [27]	Same as rel/attr	Hand-written repair rules	Hand-written repair rules
ATHENA [33]	Tokenizer	Synonym lexicon + Pre-defined ontology	Index with semantic variants	Pre-defined ontology	N/A
Seq2SQL [44]	Tokenizer + Stanford CoreNLP [26]	GloVe [31] + character n-grams [16]	Unsupported	N/A	N/A
SQLNet [40]	Tokenizer + Stanford CoreNLP [26]	GloVe [31]	Unsupported	N/A	N/A
DBPal [38]	Replace literals with placeholders	Unspecified	word2vec [27]	Select min-length path	SQL syntax repair + Fill placeholders

```
keyword k, domain_keyword dk, domain d
WHERE d.name = 'Databases'
 AND p.pid = pk.pid AND k.kid = pk.kid
 AND dk.kid = k.kid AND dk.did = d.did
```

*The NLIDB attempts **keyword mapping** by matching "papers" in the NLQ to either the relation* publication *or* journal, *and "Databases" to a value in the* domain *relation. It maps "papers" to* journal *because they have a high similarity score in the NLIDB's word embedding model. After this, the NLIDB performs **join path inference** by examining the schema graph and selects the shortest join path from* journal *to* domain *to form the (unintended) SQL query:*

```
SELECT j.name
FROM journal j, domain_journal o, domain d
WHERE d.name = 'Databases'
 AND j.jid = o.jid AND o.did = d.did
```

The example demonstrates how error in keyword mapping can propagate through the pipeline to produce an incorrect SQL query. Even when the keyword mapping is correct, however, the join path inference remains as a challenge:

Example 2. *In the keyword mapping process for John's NLQ, assume the NLIDB correctly matched "papers" to* publication. *The NLIDB examines the schema graph and its algorithm selects the shortest path from* publication *to* domain. *The returned SQL does not match John's intent:*

```
SELECT p.title
FROM publication p, conference c,
 domain_conference dc, domain d
WHERE d.name = 'Databases'
 AND p.pid = c.pid AND c.cid = dc.cid
 AND dc.did = d.did
```

While there is always inherent ambiguity introduced in NLQs that even humans have difficulty interpreting, our goal is to improve the accuracy of *keyword mapping* and *join path inference* in NLIDBs to better match the user's intent.
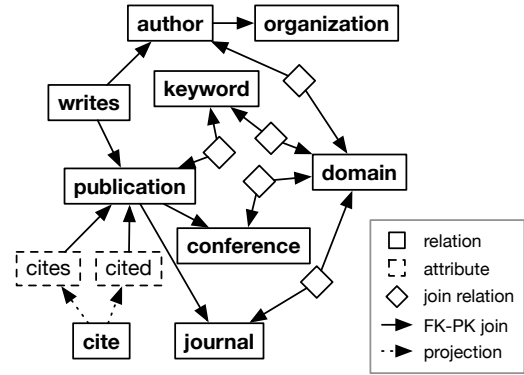

Fig. 1. A simplified version of the Microsoft Academic Search database's schema graph.

Recent end-to-end deep learning systems [38], [40], [44] show the great promise of learning from large volumes of NLQ-SQL pairs. However, manually creating labeled NLQ-SQL pairs is costly and time-consuming. Despite recent efforts to synthesize NLQ-SQL pairs [18], [38] or derive them from user descriptions of SQL queries [9], obtaining realistic labeled data remains an open research challenge. As a result, state-of-the-art deep learning systems [40], [44] have thus far only been tested on datasets of simple NLQs requiring no join.

a) Our Approach: While NLQ-SQL pairs are rarely available in large quantities for a given schema, large SQL query logs are more readily available given that NLIDBs are often built not for freshly instantiated databases, but for existing production databases [1], [2]. Although the SQL query log is not a typical supervised learning training set of input-output pairs, an output set of rich data like the SQL query log can still provide value in translation, akin to the way that one could infer much about what is being communicated and what should be spoken next even by listening to only one end of a phone conversation. Our approach is to use the information in the SQL query log of a database to select more likely keyword mappings and join paths for SQL translations of NLQs.

We propose a system TEMPLAR, which augments existing pipeline-based NLIDBs such as [22], [32], [33], [41] with SQL query log information. While it is also possible to augment end-to-end deep learning NLIDBs, this would require additional pre- or post-processing, and we leave it for future work. Consider the user of TEMPLAR with our running example:

Example 3. *John issues the NLQ from Example 1 on a NLIDB augmented with* TEMPLAR. *The NLIDB defers the **keyword mapping** to* TEMPLAR, *which uses information in the SQL query log to determine* `publication` *as the most likely mapping. The NLIDB receives this information, performs any necessary processing, and then defers **join path inference** to* TEMPLAR *by passing the mapped relations and attributes to it.* TEMPLAR *takes the input and again uses the SQL query log to conclude that the most likely join path involves connecting* `publication` *to* `domain` *via the* `keyword` *relation. This join path is passed back to the NLIDB, which constructs the final SQL query matching John's intent.*

b) Technical Challenges: Unlike traditional learning tasks where full input-output pairs (i.e. NLQ-SQL pairs) are used to train a model, we use only output logs (i.e. SQL queries). Consequently, the information in the SQL query log does not directly map to the translation task. Furthermore, even with large query logs, it is likely that most queries are not exact repeats of queries previously issued. Finally, our goal is to augment, rather than replace, NLIDBs. So we need TEMPLAR to be able to assist multiple NLIDBs through a simple common interface. In short, the challenges are to (1) *selectively activate information in the SQL log* for NLQ-SQL translation, (2) *allow the generation of new SQL queries* not in the log, and (3) *gracefully integrate log information with existing techniques* in NLIDBs.

c) Contributions: Our main contributions are as follows:

- We propose the *query fragment* as an atomic building block for SQL, providing a fine-grained view of a SQL log to allow *selective activation of information in the log*. Query fragments can be mixed and matched to *allow the generation of new SQL queries* not yet observed in the query log.
- We propose the *Query Fragment Graph* as a novel abstraction to enhance the accuracy of keyword mapping and join path inference in NLIDBs by modeling the co-occurrence of query fragments from a SQL query log, and *gracefully integrating this with existing techniques* to improve the accuracy of keyword mapping and join path inference in NLIDBs.
- We introduce a prototype system TEMPLAR, which augments existing NLIDBs without altering their internal architecture.
- We demonstrate by an extensive evaluation on how TEMPLAR can improve the top-1 accuracy of state-of-the-art NLIDBs by up to 138% on our benchmarks.

d) Organization: We discuss related work (Section II), then present the architecture of TEMPLAR and formal problem definitions (Section III), before introducing the query fragment and Query Fragment Graph to model the SQL query log (Section IV). We then explain our algorithms for improving the accuracy of keyword mapping and join path inference by leveraging the Query Fragment Graph (Sections V-VI). We present our experimental evaluation of NLIDBs augmented with TEMPLAR (Section VII), and conclude (Section VIII).

II. RELATED WORK

Our work builds upon multiple streams of prior work:

a) Natural language interfaces to databases (NLIDB): Research on NLIDBs extends as far back to the sixties and seventies [4], when interfaces were focused on solutions tailored to a specific domain. Early approaches depended on grammars that were manually-specified [4] or learned from database-specific training examples [14], [35], making it difficult to scale them across different database schemas.

Since then, advances in deep learning have inspired efforts to build an end-to-end deep learning framework to handle natural language queries [12], [25], [42]. The limiting factor for such systems is the need for a large set of NLQ to SQL pairs for each schema, and consequently some work focuses on the challenge of synthesizing and collecting NLQ-SQL pairs [9], [18], [38], [43] to be able to train these systems. The most recent deep learning-based end-to-end systems [38], [40], [44] make use of the sequence-to-sequence architecture, and these systems can benefit from the enhancements TEMPLAR provides to keyword mapping. On the other hand, the current scope of these systems is confined to single-table schemas which do not require joins, and therefore the join path inference techniques described in TEMPLAR provide no benefit to these systems as yet.

An alternative approach has been to combine techniques from the natural language processing and database communities to construct pipeline-based NLIDBs. Such systems often utilize intermediate representations in the NLQ to SQL translation process, such as a parse tree [22], query sketch [37], [41], or an ontology [33]. They also ensure reliability by doing at least one of the following: explicitly defining their semantic coverage [22], [24], [32], allowing the user to correct ambiguities [22], asking the user to provide a mapping from a database schema to an ontology [33], or by engaging in an automated query repair process [41]. TEMPLAR can enhance the performance of these NLIDBs by leveraging query logs as an additional data source to increase accuracy.

b) Keyword search: Keyword search interfaces [3], [6]–[8], [17], [36] emulate web search engines by allowing users to type in keywords to retrieve information. These keyword search interfaces often face the keyword mapping and join path inference problems that were described in our work, but TEMPLAR is the first to make use of the SQL query log to address these issues.

c) Using query logs: Previous work used SQL query logs to autocomplete SQL queries [19], proposing a similar abstraction to query fragments for a different purpose. QueRIE [13] and qunits [29] organized the query log in a similar fashion

to the Query Fragment Graph, but for the purposes of query recommendations and keyword queries, respectively.

III. Overview

A. Preliminaries

We first introduce some preliminary definitions.

The schema graph depicts the relations and their connections in a relational database:

Definition 1. *A **schema graph** is a directed graph $G_s = (V, E, w)$ for a database D with the following properties:*

- *V consists of two types of vertices:*
  - *Relation vertices $V_R \subseteq V$, each corresponding to a relation in D.*
  - *Attribute vertices $V_\alpha \subset V$, each corresponding to an attribute in D.*
- *E consists of two types of edges:*
  - *Projection edges $E_\pi \subseteq E$, each extending from a given relation vertex to each of its corresponding attribute vertices.*
  - *FK-PK join edges $E_\bowtie \subset E$, each extending from each foreign key attribute vertex to its corresponding primary key attribute vertex.*
- *$w : V \times V \to [0, 1]$ is a function that assigns a weight to each pair of vertices which have an edge in E.*

A join path is a specific type of tree within the schema graph, which can be represented by a combination of relations and join conditions in a SQL query:

Definition 2. *Given a schema graph G_s and a bag of relations B_R, a **join path** (V_j, E_j, V_t) is a tree of vertices $V_j \subset G_s$ and edges $E_j \subset G_s$ spanning all terminal vertices $V_t \subset G_s$, where each relation instance in B_R is represented by a terminal vertex $v_R \in V_t$.*

B. Definitions

As we will discuss in detail in Sec. IV, a complete SQL query is too large and too specific a unit of data to be able to use it effectively to represent a SQL log. Instead, we use *query fragments*, which are pieces of SQL queries:

Definition 3. *A **query fragment** $c = (\chi, \tau)$ is a pair of:*

- *χ: a SQL expression or non-join condition predicate;*
- *τ: the context clause in which χ resides.*

For example, in the SQL query:

```
SELECT t.a FROM table1 t, table2 u
WHERE t.b = 15 AND t.id = u.id
```

The query fragments are (`t.a`, `SELECT`), (`table1`, `FROM`), (`table2`, `FROM`), (`t.b = 15`, `WHERE`).

Keyword phrases in a NLQ are mapped to query fragments by NLIDBs to form *query fragment mappings*:

Definition 4. *A **query fragment mapping** $m = (s, c, \sigma)$ is a triple of a keyword s, a query fragment c, and a similarity score σ between the keyword and query fragment.*
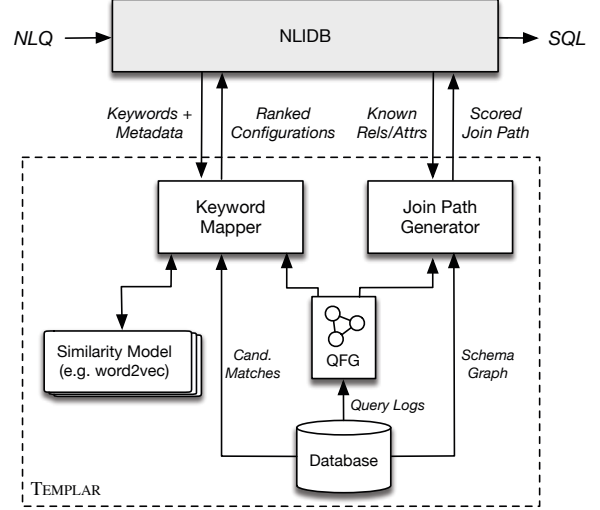

Fig. 2. The overall architecture of an NLIDB augmented with Templar.

A selection of mappings for an NLQ form a *configuration*:

Definition 5. *A **configuration** $\phi(S)$ of a set of keywords S is a selection of exactly one query fragment mapping (s_k, c_k, σ_k) for each keyword $s_k \in S$, where c_k is a query fragment, and σ_k is the associated similarity score for the keyword and fragment.*

C. Problem Definitions

We now present a formal definition for the *keyword mapping* and *join path inference* problems.

1) Keyword Mapping: The keyword mapping problem is described by the function:

$$\Phi = \text{MapKeywords}(D, S, M)$$

The input to the problem is a database D, a set of keywords representing an NLQ, $S = \{s_1, s_2, ..., s_n\}$, where each keyword $s_k \in S$ can be comprised of multiple words or tokens in natural language; and a set of metadata annotations, M, where each element $M_k = (\tau_k, \omega_k, \mathcal{F}_k, g_k)$ of M includes parser metadata about s_k: the context τ_k of the query fragment that should be mapped to s_k, an optional predicate comparison operator ω_k, an optional ordered list of aggregation functions $\mathcal{F}_k$, and a boolean g_k which if true, indicates that the resulting mapping of s_k should be grouped. The goal of the problem is to return a list of *configurations* Φ ordered by likelihood.

2) Join Path Inference: The join path inference problem is described by the function:

$$J = \text{InferJoins}(G_s, B_D)$$

The input is a schema graph G_s, a bag (i.e. a multiset) of attributes and relations B_D that are known to be part of the SQL query. The goal is to return a list of join paths J on G_s ranked from most to least likely.

D. Architecture

TEMPLAR's architecture is shown in Figure 2. It interfaces with the NLIDB it is augmenting on two fronts: one for keyword mapping, and the other for join path inference.

The **Keyword Mapper** carries out the execution of MAP-KEYWORDS, and uses a word similarity model such as word2vec [27] or GloVe [31], the query fragment graph (QFG) which stores the SQL query log information, and the database itself to retrieve candidate matches. The **Join Path Generator** executes INFERJOINS, and it utilizes the QFG and the schema graph of the database to infer join paths.

E. NLIDB Prerequisites

An NLIDB to which we can apply our approach is responsible for the following:

- It must be able to parse the NLQ into keywords, which may require recognition of multi-word entities. Each keyword should have associated metadata (query fragment type, predicate operator, aggregation functions, and presence of a group-by) for the keyword mapping problem.
- It is responsible for constructing a SQL query given the keyword mappings and join paths provided by TEMPLAR.

The categories of metadata we expect as input in MAP-KEYWORDS are all obtainable using existing parser technology [20], [30], [34] by existing NLIDBs [22], [41].

Since the two main interface calls of keyword mapping and join path inference are independent of one another in our approach, we do not enforce any ordering of when and how these calls should be made within the NLIDB. However, in every currently known system in Table I, the keyword mapping step precedes the join path inference step.

The interface to pipeline-based NLIDBs such as [22], [32], [33], [41] is transparent, as most already support the above requirements or can be easily modified to do so. Integrating TEMPLAR into an end-to-end deep learning NLIDB is possible by integrating the information from the SQL query log into the input representation or by performing some pre-processing and/or post-processing, but we leave this for future work.

F. Example Execution

In this section, we describe an example execution of a generic pipeline-based NLIDB augmented with TEMPLAR. Consider the architecture in Figure 2 and the following example NLQ from the Microsoft Academic Search (MAS) dataset [22] with schema in Figure 1:

Example 4. *Return the papers after 2000.*

First, the NLIDB parses the NLQ to return the keywords which map to elements in the database and corresponding parser metadata. In Example 4, the keywords emitted by the NLIDB would be *papers* and *after 2000*. NLIDBs have various techniques of producing the metadata, whether through semantic parsing [41] or a designated lexicon of keywords [22]. The NLIDB in [22] would return that *papers* is in the SELECT context because it is a direct child of the keyword *Return* in the parse tree, and *after 2000* would be in the WHERE

context because *after* is a reserved keyword corresponding to the predicate comparison operator $>$ in the NLIDB's lexicon.

The keywords are passed to the **Keyword Mapper**, which maps each keyword to candidate query fragments using the keyword metadata and information about the database schema and contents. These candidate query fragment mappings are individually scored using a similarity model (such as word2vec [27]) and information from the *Query Fragment Graph (QFG)*. For Example 4, the candidate mappings for *papers* includes (journal.name, SELECT) and (publication.title, SELECT), and *after 2000* is mapped to (publication.year > 2000, WHERE).

A configuration is generated by selecting one candidate mapping per keyword. The top-κ most likely candidate configurations are returned by the **Keyword Mapper**. Example 4 produces at least two candidate configurations, whose mapped query fragments, respectively, are:

- [(journal.name, SELECT);
  (publication.year > 2000, WHERE)]
- [(publication.title, SELECT);
  (publication.year > 2000, WHERE)]

These configurations are then sent back to the NLIDB, which can augment the ranked configurations with other information such as domain-specific knowledge.

After processing the configurations, the NLIDB sends known relations for each candidate SQL translation to the **Join Path Generator**, which identifies the most likely join path and returns it along with an associated score.

For the schema graph shown in Figure 1 and continuing with Example 4, this step will produce the join path journal-publication for our first configuration, and the single relation publication for the second.

Finally, it is the NLIDB's responsibility to construct the final SQL query and return it. Any post-processing, such as the hand-written repair rules in [41] or soliciting additional user interaction as in [22] may also be performed at this point. For our running example, the final SQL queries returned by the NLIDB for each candidate configuration would be:

- SELECT j.name
  FROM journal j, publication p
  WHERE p.year > 2000 AND j.jid = p.jid
- SELECT title FROM publication
  WHERE year > 2000

IV. QUERY LOG MODEL

In this section, we explore how to model information in the SQL query log to aid in NLQ to SQL translation. Consider the SQL query log in Figure 3a and the example task:

Example 5. *The task is:*

- **NLQ**: *Select all papers from TKDE after 1995.*
- **SQL**: SELECT p.title
  FROM journal j, publication p
  WHERE j.name = 'TKDE' AND
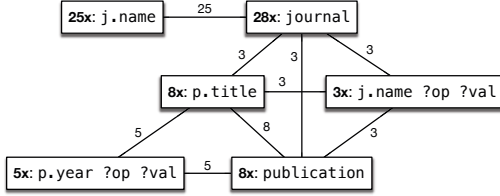 p.year > 1995 AND j.jid = p.jid

```
25x: SELECT j.name FROM journal j
5x: SELECT p.title FROM publication p WHERE p.year > 2003
3x: SELECT p.title FROM journal j, publication p WHERE j.name = 'TMC'
     AND p.pid = j.pid
```

(a) Example query log.

| 25x: j.name | 28x: journal | 5x: p.year ?op ?val |
| 8x: j.name | 8x: publication | 3x: j.name ?op ?val |

(b) Query fragment occurrences.

(c) Query fragment graph.

Fig. 3. Storing query log information in the QFG.

First, we want *to generate queries not yet observed in the SQL log*—i.e. not be constrained to only translate to queries already in the log. In Example 5, the NLQ has the keyword *papers* which might map to `publication.title` or `journal.name`. If we are limited to selecting existing SQL queries in the log to translate to, the NLQ could erroneously be translated to `SELECT j.name FROM journal j`.

To avoid this, we break down SQL queries into query fragments which can be mixed and matched to form new SQL queries, and count occurrences of each query fragment in the log as in Figure 3b.

Now, consider that we boost the scores of commonly-occurring query fragments in the SQL log. Unfortunately, there is still a high chance that "papers" will be mapped to `journal` because of its high frequency in the log.

Consequently, we want to *selectively activate information in the log* only when helpful for the NLQ at hand. The intuition is that the full NLQ provides context for each individual keyword, and this should be leveraged to illuminate what queries in the SQL log are relevant to the NLQ. In Example 5, the keywords are *papers*, *TKDE*, and *after 1995*. A human expert would that *TKDE* is referring to a journal and *after 1995* refers to a year, and can conclude that *papers* isn't referring to `journal.name` because the NLQ would be redundantly asking for "all journals from a journal".

Finally, we want to *maximize the semantic information in the SQL query log*. We observe a distinction between the more abstract semantic information and the specific value instances in a query. For Example 5, we can replace specific values in the NLQ with placeholders: *Select all papers from (journal) after (year)*, preserving the semantic structure while obscuring exact values. Similarly, we can put placeholders in the SQL:

```
SELECT p.title
FROM journal j, publication p
WHERE j.name ?op ?val AND p.year ?op ?val
 AND j.jid = p.jid
```

Using such placeholders allows us to focus on the recurrence of semantic contexts without being distracted by specific values. Consequently, it allows us to make more extensive use of the data in the SQL query log as more query fragments in the log are likely to match any given keyword in a NLQ.

We implement three levels of *obscurity* for query fragments. The first level, *Full*, retains all values in the original query. The second, *NoConst*, replaces literal constants with a placeholder to convert a fragment `p.year > 2000` into `p.year > ?val`. Finally, we further obscure comparison operators in *NoConstOp* to make the fragment `p.year ?op ?val`.

A. Query Fragment Graph

While automated NLIDBs don't have the benefit of human logic, the SQL query log can play a similar role by using the full context of a NLQ to revise individual keyword mappings. Previous user queries in the log in Figure 3a show that years are often queried in the context of `publication.title`, and similarly, when a specific journal name such as *TMC* is a predicate in a query, the user is often querying `publication.title`. This observation leads us to desire not only the occurrences of individual query fragments in the SQL log, but also the co-occurrences of query fragments—in other words, given the information in the SQL log, when one query fragment appears in a query, how likely is it that another query fragment is present in the query?

Given the intuition above, we introduce the ***Query Fragment Graph (QFG)*** as a data structure to store the information in a SQL query log.

Definition 6. *A **query fragment graph** for database D and SQL query log L is a graph $G_f = (V_f, E_f, n_v, n_e)$ where:*

1) *each vertex $v \in V_f$ represents a query fragment in L;*
2) *each edge $e \in E_f$ exists if and only if two query fragments co-occur in L;*
3) *$n_v : V_f \to \mathbb{Z}_{\geq 0}$ is a function which maps V_f to the number of occurrences in L of the query fragment represented by each $v \in V_f$;*
4) *$n_e : V_f \times V_f \to \mathbb{Z}_{\geq 0}$ is a function which maps each pair of vertices to the co-occurrence frequency in L of the two query fragments represented by the vertices.*

In short, the QFG stores information on *query fragment occurrences* (n_v) in the log, as well as *co-occurrence relationships* (n_e) between each pair of query fragments.

V. KEYWORD MAPPING

In this section, we explain the keyword mapping procedure. While many techniques described here are already applied in existing work, we explain each step in detail to keep this work self-contained, and to clearly show how our novel approach of using SQL query log information comes into play.

Mapping keywords involves three steps: (1) retrieving candidate keyword to query fragment mappings, (2) scoring and retaining the top-κ candidates, and (3) generating and scoring configurations. Information from the query fragment graph is

Algorithm 1 Mapping Keywords

```
1: function MAPKEYWORDS(D,S,M)
2:      R ← {}
3:      for k ← 1, . . . , |S| do
4:          (τ_k, ω_k, F_k, g_k) ← M_k
5:          C_k ← KEYWORDCANDS(D, s_k, τ_k, ω_k, F_k, g_k)
6:          R_k ← SCOREANDPRUNE(s_k, C_k, κ)
7:          R.add(R_k)
8:      Φ = genAndScoreConfigs(R)
9:      return Φ
```

Algorithm 2 Retrieve Candidate Keyword Mappings

```
1: function KEYWORDCANDS(D,s,τ,ω,F,g)
2:      C ← {}
3:      if containsNumber(s) then
4:          s_num ← extractNumber(s)
5:          β ← findNumericAttrs(s_num, ω)
6:          for b ∈ β do
7:              C.add((Pred(b, ω, s_num), WHERE))
8:      else
9:          if τ = FROM then
10:             for r ∈ getRelations(D) do
11:                 C.add((r, τ))
12:         else if τ = SELECT then
13:             for α ∈ getAttributes(D) do
14:                 C.add((Attr(α, F, g), τ))
15:         else
16:             for t ∈ findTextAttrs(s) do
17:                 C.add((Pred(t, =, s), WHERE))
18:     return C
```

Algorithm 3 Score and Prune Keyword Mappings

```
1: function SCOREANDPRUNE(s,C,κ)
2:      R ← {}
3:      for c ∈ C do
4:          if containsNumber(s) then
5:              s_num ← extractNumber(s)
6:              s_text ← s − s_num
7:              σ ← sim_num(s_text, c)
8:          else
9:              σ ← sim_text(s, c)
10:         R.add((s, c, σ))
11:     sort R by descending σ
12:     return PRUNE(R, κ)
```

used in the final step to score configurations according to the evidence in the SQL query log.

We now describe our algorithm for the MAPKEYWORDS function, shown in Algorithm 1. We loop through all the keywords $s_k \in S$ with their corresponding metadata, then combine and rank them to form our output configurations.

A. Retrieving Candidate Mappings

The function KEYWORDCANDS in Algorithm 2 maps a keyword s, along with its associated metadata $(τ, ω, F, g)$, to its candidate mappings C by querying the database D.

First, we evaluate whether s contains a number (Line 3), such as in the keyword *after 2000*. If so, we return all numeric attributes in the database that match a predicate formed by the number extracted from s with the operator $ω$ for s (Line 5). For the keyword *after 2000*, we return all attributes containing at least one value that satisfies the predicate `?attr > 2000`. Predicates are constructed from matching attributes and added to the candidate set C.

If s does not contain a number, we have three different cases. In the first two cases, where the context $τ$ of the query fragment is FROM or SELECT, we simply add either all the

relations or all the attributes (along with relevant metadata) of D to the candidate set C.

For the final case covering all other structures, we first run a full-text search with every Porter-stemmed [39] whitespace-separated token in s to retrieve all matching text attributes T in D (*findTextAttrs* in Line 16). For example, for the keyword *restaurant businesses*, the stemming procedure would result in the tokens *restaur busi*, and we run the following SQL query, replacing `?attr` with each text attribute in D:

```
SELECT DISTINCT(?attr) FROM ?rel
WHERE MATCH(?attr)
 AGAINST ('+restaur* +busi*'
  IN BOOLEAN MODE)
```

If any of the stemmed tokens from s exactly match the stemmed attribute or relation names of a candidate query fragment, we remove them so as not to unnecessarily constrain our search. For example, if the keyword is *movie Saving Private Ryan* and a candidate query fragment mapping is an attribute from the `movie` relation, we remove the token *movie* from our full-text search query when searching on that attribute. For each matching text attribute, we then construct a predicate for the WHERE context.

B. Scoring and Pruning

Our next step is to retain only the top-$κ$ most likely mappings from C with the function SCOREANDPRUNE.

We calculate a score $σ$ for each keyword mapping in the range $[0, 1]$. For comparing keywords with purely text tokens against relation and attribute names and text predicates, we can use a similarity function sim_{text} (Line 9) through a word embedding model such as word2vec [27] or GloVe [31]. For keywords including numeric tokens, we execute (i.e. $exec(c)$) the candidate predicate on the database, then evaluate the similarity of only the text tokens if the predicate returns a non-empty set, and return a small $ε$ value otherwise:

$$sim_{num}(s_{text}, c) = \begin{cases} sim_{text}(s_{text}, c), & \text{if } exec(c) \nrightarrow \varnothing \\ ε, & \text{otherwise} \end{cases}$$

$σ$ is then combined into a tuple with the original keyword s and candidate mapping c and added to the result set R,

which is finally sorted by descending σ score. We then prune R to prevent a combinatorial explosion when generating configurations, using the following PRUNE procedure (Line 12):

- If there are any candidates in R that are exact matches ($\sigma \geq 1 - \epsilon$ for a small ϵ), we prune away all remaining non-exact candidates.
- Otherwise, we prune R to the top-κ results, including any results that have a non-zero σ value that is equal to the σ of the candidate at the κ-th place.

C. Ranking Configurations

At this point, we have a set of candidate mappings for each keyword $s_k \in S$. We combine and score them (Line 8 of Algorithm 1) to form candidate configurations for S. We first describe a standard way of scoring configurations, then show how we can apply the SQL query log to improve scoring.

1) Word Similarity-Based Score: A naïve scoring function for configurations selects the best mapping for each keyword independently. We can take the geometric mean of the scores of all mappings to accomplish this:

$$Score_\sigma(\phi) = [\prod_{(s_k, c_k, \sigma_k) \in \phi} \sigma_k]^{\frac{1}{|\phi|}}$$

We prefer the geometric mean over the arithmetic mean, as in [41], to mitigate the impact of the variation in ranges of values for each keyword's candidate mapping scores.

2) Query Log-Driven Score: Since we have the query log information available to us via the Query Fragment Graph, we leverage this information to derive an improved scoring function contextualized for our specific database schema.

While word similarity-based scoring considers each mapping independently, we now consider the *collective score* of each configuration of mappings. Previous work such as [22] attempts a collective scoring approach based on *mutual relevance* which considers the proximity of keywords in the natural language dependency tree in relation to the edge weights connecting the candidate query fragments within the schema graph. Unfortunately, these schema graph edge weights are assigned manually without justification.

In contrast, the intuition behind our collective scoring mechanism is to give a higher score to configurations containing query fragments that frequently co-occur in queries in the SQL query log. Instead of relying on the system administrator's ability to preset the schema graph edge weights to match an anticipated workload, we derive our scoring directly from previous users' queries in the SQL query log.

To accomplish this, we calculate a metric for the co-occurrence of pairs of query fragments in the QFG, then aggregate this metric, along with the previously-computed similarity scores, over all query fragments in the configuration to derive a final score. We use the Dice similarity coefficient [15] to reflect the co-occurrence of two query fragments c_1 and c_2 in the QFG, defined as follows:

$$Dice(c_1, c_2) = \frac{2 \times n_e(c_1, c_2)}{n_v(c_1) + n_v(c_2)}$$

We accumulate *Dice* for every pair of non-relation (i.e. not in the FROM context) fragments $(c_1, c_2) \in \phi_{\tau \neq \text{FROM}} \times \phi_{\tau \neq \text{FROM}}$:

$$Score_{QFG}(\phi) = [\prod_{(c_1, c_2) \in \phi^2_{\tau \neq \text{FROM}}} Dice(c_1, c_2)]^{\frac{1}{|\phi|}}$$

The query fragments in the FROM context are excluded because involving relations can add information skewing the aggregate score—e.g. if `journal.name` is in a SQL query, then the relation `journal` is required to be by the rules of SQL, adding unnecessary redundancy to the aggregated Dice score. In addition, relations in the FROM clause are explicitly handled by our join path inference procedure, so we defer the evaluation of these query fragments for later.

Finally, we perform a linear combination (governed by a parameter $\lambda \in [0, 1]$) of $Score_\sigma$ and the query log-driven score $Score_{QFG}$ to produce a final configuration score:

$$Score(\phi) = \lambda Score_\sigma(\phi) + (1 - \lambda) Score_{QFG}(\phi)$$

We can also replace this means of combining evidence from multiple sources with other approaches, such as the Dempster Shafer Theory in [6]. We opt for a linear combination due to its simplicity and because it works sufficiently well in practice.

All configurations are now scored using $Score(\phi)$, ranked by descending score, and returned by MAPKEYWORDS.

VI. JOIN PATH INFERENCE

In this section, we describe how we generate join paths for a set of attributes and relations selected to be part of the final SQL query by the keyword mapping procedure, and show how we use the SQL query log to improve this process.

Example 6. *Consider that the NLIDB selected the following query fragments to be part of a SQL query of the schema given in Figure 1:*

- *(publication.title, SELECT)*
- *(domain.name = 'Databases', WHERE)*

INFERJOINS *should output the desired join path:* `publication–publication_keyword–keyword–domain_keyword–domain`.

A. Generating Join Paths

The process of generating the set of optimal join paths from a set of known relations B_R and a schema graph G_s has previously been modeled as the Steiner tree problem [21], where the goal is to find a tree on a graph that spans a given set of vertices with minimal total weight on its edges.

The expected input to the ***Join Path Generator*** is a bag of the attributes and relations B_D already known to be in the desired SQL translation. B_D can be converted to the bag of known relations B_R simply by replacing each attribute with its parent relation in G_s.

We use a known algorithm [21] for solving Steiner trees to find the set of optimal join paths for any given configuration. These optimal join paths, however, change depending on how weights are assigned to edges in the schema graph. We outline two ways to do this, first without information from the query log, and then adding in query log information.

1) Default Edge Weights: The default weight function w for edges in the schema graph is to assign every edge a weight of 1. If we solve the Steiner tree problem with this weight function, we are essentially finding join paths with the minimal number of join edges that span all the known relations.

For Example 6, this approach will produce the shortest join path between `publication` and `domain`, which is either:

- `publication–conference–domain_conference–domain`
- `publication–journal–domain_journal–domain`

Neither of these join paths are the one desired by the user.

2) Query Log-Driven Edge Weights: We look to the query log to provide some grounding for generating join paths. In contrast to previous work which depends on the system administrator to set schema graph edge weights [22], on hand-written repair rules [41], or a predefined ontology [33], query log information is driven by actual user queries executed on the system. Query log information allows us to prefer commonly queried join paths, even if they are longer, and also mitigates the number of situations where there are identical scores given to equal-length join paths.

We leverage the co-occurrence values of relations in the QFG to adjust the weights on the schema graph. Given any two vertices $(v_1, v_2) \in G_s$, and the function $q : V \to V_{QF}$ which maps a vertex in the schema graph G_s to its corresponding vertex in the QFG, the new weight function is:

$$w_L(v_1, v_2) = \begin{cases} 1 - Dice(q(v_1), q(v_2)) & \text{if } v_1 \in V_R \wedge v_2 \in V_R \\ 1, & \text{otherwise} \end{cases}$$

This query log-based weight function w_L returns a lower value for join edges that frequently occur in the query log.

B. Scoring Join Paths

The final score for any join path j we return is derived from the weights of the edges within the join path:

$$Score_j(j) = \frac{1}{|E_j|^2} \sum_{(v_1, v_2) \in E_j} w(v_1, v_2)$$

We divide by $|E_j|^2$ to normalize the score in a [0,1] range and also to prefer simpler join paths over more complex ones. This is based on the observations regarding *semantic relevance* [6], [22] that the closer two relations are in the schema, the likelier it is that they are semantically related.

C. Self-Joins

A challenge arises during join path inference when an attribute is included multiple times in the bag B_D. We present a novel approach to handling such situations to still produce valid results from the Steiner tree algorithm.

Due to the peculiarities of SQL, these situations require that our resulting join path include multiple instances of the same relation, resulting in a *self-join*. For example:

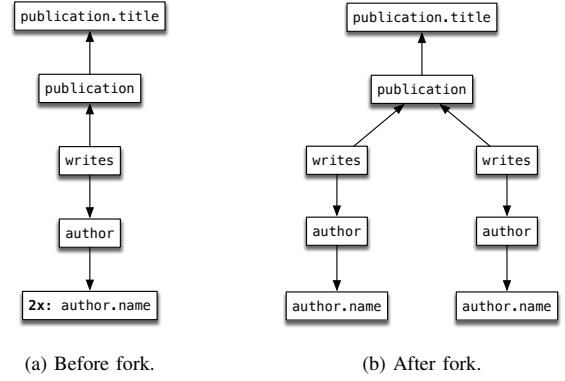**Example 7.** *In an NLQ for the academic database, "Find papers written by both John and Jane", "John" and "Jane"*

(a) Before fork.   (b) After fork.

Fig. 4.   A simplified overview of a schema graph fork for self-joins.

Algorithm 4 Forking Schema Graph for Self-Joins

1: **function** FORK(G_s,v)
2:     $stack_{old} \leftarrow$ **new** Stack()
3:     $stack_{new} \leftarrow$ **new** Stack()
4:     $stack_{old}.push(v)$
5:     $stack_{new}.push(G_s.clone(v))$
6:     $visited \leftarrow \{\}$
7:     **while** $stack_{old} \neq \varnothing$ **do**
8:         $v_{old} \leftarrow stack_{old}.pop()$
9:         $v_{new} \leftarrow stack_{new}.pop()$
10:        $visited \leftarrow visited \cup v_{old}$
11:        **for all** v_{conn} connected to v_{old} **do**
12:            **if** $v_{conn} \in visited$ **then continue**
13:            **if** $(v_{old}, v_{conn}) \in E_{\bowtie}$ of G_s **then**
14:                **add edge** (v_{new}, v_{conn}) to G_s
15:            **else**
16:                $v_{cloned} \leftarrow G_s.clone(v_{conn})$
17:                $dir \leftarrow$ direction of (v_{old}, v_{conn}) $\triangleright \leftarrow$ or $\rightarrow$
18:                **add edge** $(v_{new}, v_{cloned}, dir)$ to G_s
19:            $stack_{old}.push(v_{conn})$
20:            $stack_{new}.push(v_{cloned})$

---

both refer to attribute `author.name`. *The correct SQL output for this NLQ is:*

```
SELECT p.title
FROM author a1, author a2, publication p,
  writes w1, writes w2
WHERE a1.name = 'John' AND a2.name = 'Jane'
  AND a1.aid = w1.aid AND a2.aid = w2.aid
  AND p.pid = w1.pid AND p.pid = w2.pid
```

For these situations, we "fork" the schema graph, as shown (with some attribute vertices and edges removed for simplicity) in Figure 4, in order to account for the necessary vertices for a join path containing a self-join.

Algorithm 4 describes the process of forking the schema graph G_s in more detail, given an attribute vertex v that has been referenced multiple times. Two mirrored stacks v_{old} and v_{new} are used to track progress for the original graph and the new fork of the graph, respectively. We first clone the attribute

382

vertex v and add it to G_s (Line 5). We repeatedly pop the top of each stack, and find all vertices v_{conn} that are connected to the current existing vertex v_{old}. We clone each v_{conn} and the edge connecting it to v_{old}, then add both to the schema graph and continue traversal (Lines 16-20). We terminate the forking process when we reach a FK-PK join edge in the direction *from v_{old} to v_{conn}* (Line 13). For d duplicate references to an attribute vertex v, FORK is executed $(d-1)$ times to create a fork for each duplicate reference.

VII. EVALUATION

We performed an experimental evaluation of our system, TEMPLAR, to test whether we can use the SQL log to improve the accuracy of NLQ to SQL translation.

A. Experimental Setting

1) Machine Specifications: All our evaluations were performed on a computer with an 3.1 GHz Intel Core i7 processor and 16 GB RAM, running Mac OS Sierra.

2) Compared Systems: We enhanced two different NLIDB systems, NaLIR [22] and Pipeline, with TEMPLAR, and executed them on our benchmarks. The augmented versions are denoted NaLIR+ and Pipeline+ respectively.

The first system we augmented is NaLIR [22], a state-of-the-art pipeline-based NLIDB. We evaluated the system in its non-interactive setting because its application of user interaction is orthogonal to our approach.

We contacted authors of a few other existing NLIDBs but were not granted access to their systems. As a result, we built an NLIDB named Pipeline, which is an implementation of the keyword mapping and join path inference steps from the state-of-the-art approach in [41], excluding the hand-written repair rules. Pipeline was implemented using word2vec [27] for keyword mapping, with the default Google News corpus for calculating word similarity. While the default similarity value produced from word2vec is a cosine similarity value in the range [-1, 1], Pipeline normalizes these values to fall in the range [0, 1]. Pipeline also always selects the minimum-length join paths for join path inference. Our implementation of Pipeline was written in Java. We used MySQL Server 5.7.18 as our relational database.

3) Assumptions: We assume TEMPLAR is applied in a setting where *queries in the SQL query log are representative of the SQL queries issued by users via natural language.* While this assumption does not hold true for all databases, we believe TEMPLAR is applicable for databases which already implement user-friendly interfaces such as forms or keyword search where the pattern of users' information need is likely to be similar to that of natural language interfaces.

4) Dataset: We tested each system by evaluating its ability to translate NLQs accurately to SQL on three benchmarks: the Microsoft Academic Search (MAS) database used in [22], and two additional databases from [41] regarding business reviews from Yelp and movie information from IMDB. Table II provides some statistics on each of these benchmark datasets.

TABLE II
STATISTICS OF EACH BENCHMARK DATASET.

Dataset	Size	Rels	Attrs	FK-PK	Queries
MAS	3.2 GB	17	53	19	194
Yelp	2.0 GB	7	38	7	127
IMDB	1.3 GB	16	65	20	128

TABLE III
KEYWORD MAPPING (KW) AND FULL QUERY (FQ) RESULTS.

Dataset	System	KW (%)	FQ (%)
MAS	NaLIR	43.3	33.0
	NaLIR+	45.4	40.2
	Pipeline	39.7	32.0
	Pipeline+	**77.8**	**76.3**
Yelp	NaLIR	52.8	47.2
	NaLIR+	59.8	52.8
	Pipeline	56.7	54.3
	Pipeline+	**85.0**	**85.0**
IMDB	NaLIR	40.6	38.3
	NaLIR+	57.8	50.0
	Pipeline	32.0	27.3
	Pipeline+	**67.2**	**64.8**

We annotated each NLQ with SQL translation by hand as the original benchmarks did not include the translated SQL queries. We removed 2 queries from MAS, 1 query from Yelp, and 3 queries from IMDB because they were overly complex (i.e. contained correlated nested subqueries) or ambiguous, even for a human annotator.

We used a cross-validation method to ensure that the test queries were not part of the SQL query log used to perform the NLQ to SQL translation. Specifically, we randomly split the full dataset into 4 equally-sized folds, and performed 4 trials (one for each fold), where in each trial, the training set is comprised of 3 of the folds and the test set was the remaining fold held out of the training process. Our displayed results for all experiments are aggregated from the 4 trials.

For Pipeline and Pipeline+, we hand-parsed each NLQ into keywords and metadata to avoid any parser-related performance issues outside the scope of our work, while we passed the whole NLQ as input to NaLIR and NaLIR+ to make use of the authors' original system. For fairer comparison, we rewrote some NLQs with *wh-words* such as *who*, *what*, etc. to enable NaLIR/NaLIR+'s parser to process them correctly.

5) Evaluation Metrics: We measured accuracy by checking the top-ranked SQL query returned by each system by hand. For Pipeline and Pipeline+, since it was possible to return multiple queries tied for the top spot, we considered the resulting queries incorrect if there were any tie for first place.

B. Effectiveness of TEMPLAR Augmentation

In Table III, we present the overall performance of each system. Pipeline+ and NaLIR+ were both executed with obscurity *NoConstOp*, $\kappa = 5$, and $\lambda = 0.8$. While all obscurity levels, including *Full* and *NoConst*, consistently improved on the baseline systems, we only show results for the best-performing obscurity level *NoConstOp* for space reasons.

TABLE IV
IMPROVEMENT FROM ACTIVATING LOG-BASED JOINS IN PIPELINE+.

Dataset	LogJoin	FQ (%)
MAS	N	68.6
	Y	**76.3**
Yelp	N	68.5
	Y	**85.0**
IMDB	N	60.9
	Y	**64.8**

1) Full Query: The full query (FQ) was considered correct if the NLIDB ultimately produced the correct SQL query. Pipeline+ achieves 76.3% accuracy on MAS, 85.0% accuracy on Yelp, and 64.8% accuracy on IMDB. Compared to the vanilla Pipeline system, this was a 138%, 57%, and 137% increase in accuracy, respectively. NaLIR+ improved on NaLIR by more modest margins, with a 22% increase for MAS, 12% for Yelp, and 31% for IMDB.

2) Keyword Mapping: For keyword mapping (KW), we considered the mapping correct if and only if all non-relation keywords were mapped correctly by the system. Pipeline's performance improved with TEMPLAR most notably for KW, with a 96%, 50%, and 110% increase for MAS, Yelp, and IMDB respectively. The improvement on NaLIR was 5% for MAS, 13% for Yelp, and 42% for IMDB.

3) Join Path Inference: In Table IV, we investigate the effect of the Join Path Generator. We focus on Pipeline+ for space reasons, and because improvement was not as drastically evident in NaLIR for reasons described in Section VII-C.

Activating the Join Path Generator (LogJoin "Y") increased accuracy by 11% for MAS, 24% for Yelp, and 6% for IMDB. The combined effect of this with the Keyword Mapper enabled the overall improvement through TEMPLAR.

C. Error Analysis

Augmenting Pipeline with TEMPLAR had a more dramatic effect than with NaLIR because it was given perfectly parsed keywords and metadata as input. Pipeline consequently had a much higher ceiling for improvement compared to NaLIR. While NaLIR is designed to be able to return the relevant metadata, in practice, the system's parser had trouble digesting the correct metadata from NLQs with explicit relation references, such as the token *papers* in *Return the authors who have papers in Conference X* for MAS, or other NLQs which resulted in nested subqueries. Our takeaway from this is that NLIDBs with better parsers will reap greater benefits from TEMPLAR, and are hopeful as off-the-shelf parsers have drastically improved since NaLIR's original release.

D. Impact of Parameters

In addition to the system options, there are two parameters that are required to be set in TEMPLAR: κ and λ. κ is the number of top candidate keyword mappings to return before generating configurations, and λ is the weight given to the word similarity score as opposed to the log-driven score. We observed the effects of these parameters on Pipeline+.
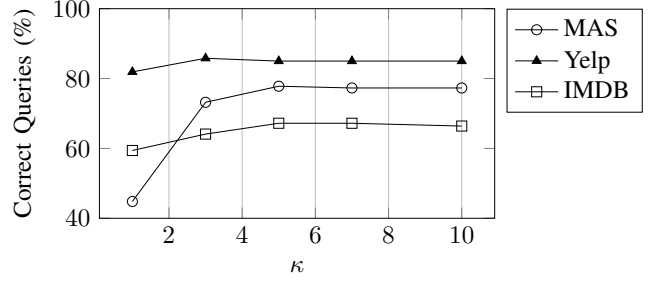

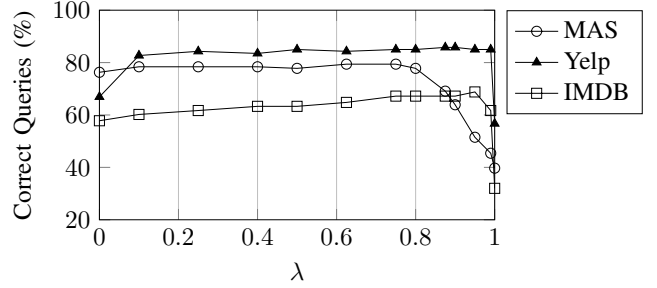Fig. 5. Accuracy of Pipeline+ on each benchmark given a value of κ, with λ fixed at 0.8.

Fig. 6. Accuracy of Pipeline+ on each benchmark given a value of λ, with κ fixed at 5.

Figure 5 shows that any $\kappa \geq 5$ yields more or less consistent performance. Consequently, we chose $\kappa = 5$ as a cutoff for all our benchmarks because it reflected optimal performance and queries were also evaluated in a timely manner.

In addition, we evaluated the end-to-end performance of Pipeline+ with varying values of λ and find similar performance across all benchmarks for $0.1 \leq \lambda \leq 0.8$. For the Yelp benchmark, accuracy falls when λ is 0 because the word similarity scores are necessary when ranking configurations, while for the other benchmarks, the pruning procedure for candidate mappings is sufficient to retain and distinguish the correct mappings. Accuracy gradually drops on the MAS and IMDB benchmarks for $\lambda > 0.8$, and sharply on all benchmarks as λ approaches 1, suggesting that the log information is crucial for most queries.

VIII. CONCLUSION

In this paper, we have described TEMPLAR, a system that enhances the performance of existing NLIDBs using SQL query logs. We model the information in the SQL query log in a data structure called the Query Fragment Graph, and use this information to improve the ability of existing NLIDBs to perform keyword mapping and join path inference. We demonstrated a significant improvement in accuracy when augmenting existing pipeline NLIDBs using log information with TEMPLAR. Possible future work includes exploring the influence of user sessions in the SQL query log, as well as finding ways to improve existing deep learning-based end-to-end NLIDB systems with information from the SQL log.

IX. ACKNOWLEDGMENTS

REFERENCES

[1] Friendlydata. https://friendlydata.io/. Accessed: 2018-03-28.

[2] Q&a in power bi service and power bi desktop. https://docs.microsoft.com/en-us/power-bi/power-bi-q-and-a. Accessed: 2018-03-28.

[3] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: a system for keyword-based search over relational databases. In *Proceedings 18th International Conference on Data Engineering*, pages 5–16, 2002.

[4] I. Androutsopoulos, G. Ritchie, and P. Thanisch. Natural language interfaces to databases an introduction. *Natural Language Engineering*, 1(1):2981, 1995.

[5] J. Berant, A. Chou, R. Frostig, and P. Liang. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544, 2013.

[6] S. Bergamaschi, F. Guerra, M. Interlandi, R. Trillo-Lado, and Y. Velegrakis. Combining user and database perspective for solving keyword queries over relational databases. *Inf. Syst.*, 55(C):1–19, Jan. 2016.

[7] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Proceedings 18th International Conference on Data Engineering*, pages 431–440, 2002.

[8] L. Blunschi, C. Jossen, D. Kossmann, M. Mori, and K. Stockinger. Soda: Generating sql for business users. *Proceedings of the VLDB Endowment*, 5(10):932–943, 2012.

[9] F. Brad, R. Iacob, I. Hosu, and T. Rebedea. Dataset for a neural natural language interface for databases (NNLIDB). *CoRR*, abs/1707.03172, 2017.

[10] E. Charniak. A maximum-entropy-inspired parser. In *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*, pages 132–139. Association for Computational Linguistics, 2000.

[11] M.-C. de Marneffe, B. MacCartney, and C. D. Manning. Generating typed dependency parses from phrase structure parses. In *LREC*, 2006.

[12] L. Dong and M. Lapata. Language to logical form with neural attention. *CoRR*, abs/1601.01280, 2016.

[13] M. Eirinaki, S. Abraham, N. Polyzotis, and N. Shaikh. Querie: Collaborative database exploration. *IEEE Transactions on Knowledge and Data Engineering*, 26(7):1778–1790, July 2014.

[14] R. Ge and R. J. Mooney. A statistical semantic parser that integrates syntax and semantics. In *Proceedings of the Ninth Conference on Computational Natural Language Learning*, CONLL '05, pages 9–16, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics.

[15] W. H. Gomaa and A. A. Fahmy. Article: A survey of text similarity approaches. *International Journal of Computer Applications*, 68(13):13–18, April 2013. Full text available.

[16] K. Hashimoto, C. Xiong, Y. Tsuruoka, and R. Socher. A joint many-task model: Growing a neural network for multiple nlp tasks. *arXiv preprint arXiv:1611.01587*, 2016.

[17] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 670–681. VLDB Endowment, 2002.

[18] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer. Learning a neural semantic parser from user feedback. *CoRR*, abs/1704.08760, 2017.

[19] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. Snipsuggest: Context-aware autocompletion for sql. *Proc. VLDB Endow.*, 4(1):22–33, Oct. 2010.

[20] D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 423–430. Association for Computational Linguistics, 2003.

[21] L. Kou, G. Markowsky, and L. Berman. A fast algorithm for steiner trees. *Acta informatica*, 15(2):141–145, 1981.

[22] F. Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *Proc. VLDB Endow.*, 8(1):73–84, Sept. 2014.

[23] Y. Li and D. Rafiei. Natural language data management and interfaces: Recent development and open challenges. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1765–1770, New York, NY, USA, 2017. ACM.

[24] Y. Li, H. Yang, and H. V. Jagadish. Nalix: An interactive natural language interface for querying xml. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 900–902, New York, NY, USA, 2005. ACM.

[25] P. Liang. Learning executable semantic parsers for natural language understanding. *Commun. ACM*, 59(9):68–76, Aug. 2016.

[26] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60, 2014.

[27] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

[28] G. A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, Nov. 1995.

[29] A. Nandi and H. V. Jagadish. Qunits: queried units in database search. *CoRR*, abs/0909.1765, 2009.

[30] J. Nivre, M.-C. de Marneffe, F. Ginter, Y. Goldberg, J. Hajic, C. D. Manning, R. T. McDonald, S. Petrov, S. Pyysalo, N. Silveira, et al. Universal dependencies v1: A multilingual treebank collection. In *LREC*, 2016.

[31] J. Pennington, R. Socher, and C. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[32] A.-M. Popescu, O. Etzioni, and H. Kautz. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th International Conference on Intelligent User Interfaces*, IUI '03, pages 149–157, New York, NY, USA, 2003. ACM.

[33] D. Saha, A. Floratou, K. Sankaranarayanan, U. F. Minhas, A. R. Mittal, and F. Özcan. Athena: An ontology-driven system for natural language querying over relational data stores. *Proceedings of the VLDB Endowment*, 9(12):1209–1220, 2016.

[34] S. Schuster and C. D. Manning. Enhanced english universal dependencies: An improved representation for natural language understanding tasks. In *LREC*, 2016.

[35] L. R. Tang and R. J. Mooney. Using multiple clause constructors in inductive logic programming for semantic parsing. In *Proceedings of the 12th European Conference on Machine Learning*, EMCL '01, pages 466–477, London, UK, UK, 2001. Springer-Verlag.

[36] S. Tata and G. M. Lohman. Sqak: Doing more with keywords. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 889–902, New York, NY, USA, 2008. ACM.

[37] C. Unger, L. Bühmann, J. Lehmann, A.-C. Ngonga Ngomo, D. Gerber, and P. Cimiano. Template-based question answering over rdf data. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 639–648, New York, NY, USA, 2012. ACM.

[38] P. Utama, N. Weir, F. Basik, C. Binnig, U. Cetintemel, B. Hättasch, A. Ilkhechi, S. Ramaswamy, and A. Usta. An End-to-end Neural Natural Language Interface for Databases. *ArXiv e-prints*, Apr. 2018.

[39] C. Van Rijsbergen, S. Robertson, and M. Porter. *New Models in Probabilistic Information Retrieval*. British Library research & development report. Computer Laboratory, University of Cambridge, 1980.

[40] X. Xu, C. Liu, and D. Song. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*, 2017.

[41] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig. Sqlizer: Query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA):63:1–63:26, Oct. 2017.

[42] P. Yin, Z. Lu, H. Li, and B. Kao. Neural enquirer: Learning to query tables in natural language. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, pages 2308–2314. AAAI Press, 2016.

[43] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. R. Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *CoRR*, abs/1809.08887, 2018.

[44] V. Zhong, C. Xiong, and R. Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*, 2017.