7 Supplemental Material

7.1 Examples of SemQL Query

Figure 10 presents more examples of SemQL queries.

7.2 Inference of SQL Query

To infer a SQL query from a SemQL query, we traverse the tree-structured SemQL query in pre-order and map each tree node to the corresponding SQL query components according to the production rule applied to it.

The production rule applied to the *Z* node denotes whether the SQL query has one of the following components, `UNION`, `EXCEPT` and `INTERSECT`. The *R* node stands for the start of a single SQL query. The production rule applied to R denotes whether the SQL query has a `WHERE` clause and `ORDERBY` clause. The production rule applied to a *Select* node denotes how many columns does the `SELECT` clause has. Each *A* node denotes a column/aggregate function pair. Specifically, nodes under *A* denote the aggregate function, the column name and the table name of the column. The subtrees under nodes *Superlative* and *Order* are mapped to the `ORDERBY` clause in the SQL query. The production rules applied to *Filter* denote different condition operators in SQL query, e.g. `and`, `or`, `>`, `<`, `=`, `in`, `not in` and so on. If there is a *A* node under the *Filter* node and its aggregate function is not $None$, it will be filled in the `HAVING` clause, otherwise in the `WHERE` clause. If there is a *R* node under the *Filter* node, we will repeat the process recursively on the *R* node and return a nested SQL query. The `FROM` clause is generated from the selected tables in the SemQL query by identifying the shortest path that connects these tables in the schema (Database schema can be formulated as an undirected graph, where vertex are tables and edges are relations among tables). At last, if there exists an aggregate function applied on a column in the SemQL query, there should be `GROUPBY` clause in the SQL query. The column to be grouped by occurs in the `SELECT` clause in most cases, or it is the primary key of a table where an aggregate function is applied on one of its columns.

7.3 Transforming SQL to SemQL

To generate a SemQL query from a SQL query, we first initialize a *Z* node. If the SQL query has one of the components `UNION`, `EXCEPT` and `INTERSECT`, we attach the corresponding keywords and two *R* nodes under *Z*, otherwise a single *R* node. Then, we attach a *Select* node under *R*, and the number of columns in `SELECT` clause determines the number of *A* nodes under the *Select* node. If an `ORDERBY` clause in a SQL query contains a `LIMIT` keyword, it will be transformed into a *Superlative* node, otherwise a *Order* node. Next, the sub-tree of *Filter* node is determined by the condition in `WHERE` and `HAVING` clause. If it has a nested query in `WHERE` clause or `HAVING` clause, we process the subquery recursively. For each column in a SQL query, we attach its aggregate function node, a *C* node and a *T* node under *A*. Node *C* attaches the column name and node *T* attaches its table name. For the special column '$\star$', if there is only one table in the `FROM` clause that does not belongs to any column, we assign it the column '$\star$', otherwise, we label the table name of '$\star$' manually. If a table in `FROM` clause is not assigned to any column, it will be transformed into a subtree under a $Filter$ node with in condition. In this way, a SQL query can be successfully transformed into a SemQL query.

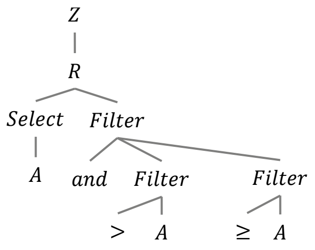


Figure 6: The skeleton of the SemQL query presented in Figure 3

7.4 Coarse-to-Fine Framework

The skeleton of a SemQL query is obtained by removing all nodes under each *A* node. Figure 6 shows the skeleton of the SemQL query presented in Figure 3.

Figure 7 depicts the coarse-to-fine framework to synthesize a SemQL query. In the first stage, a skeleton decoder outputs the skeleton of a SemQL query. Then, a detail decoder fills in the missing details in the skeleton by selecting columns and tables. The probability of generating a SemQL query y in the coarse-to-fine framework is formal-

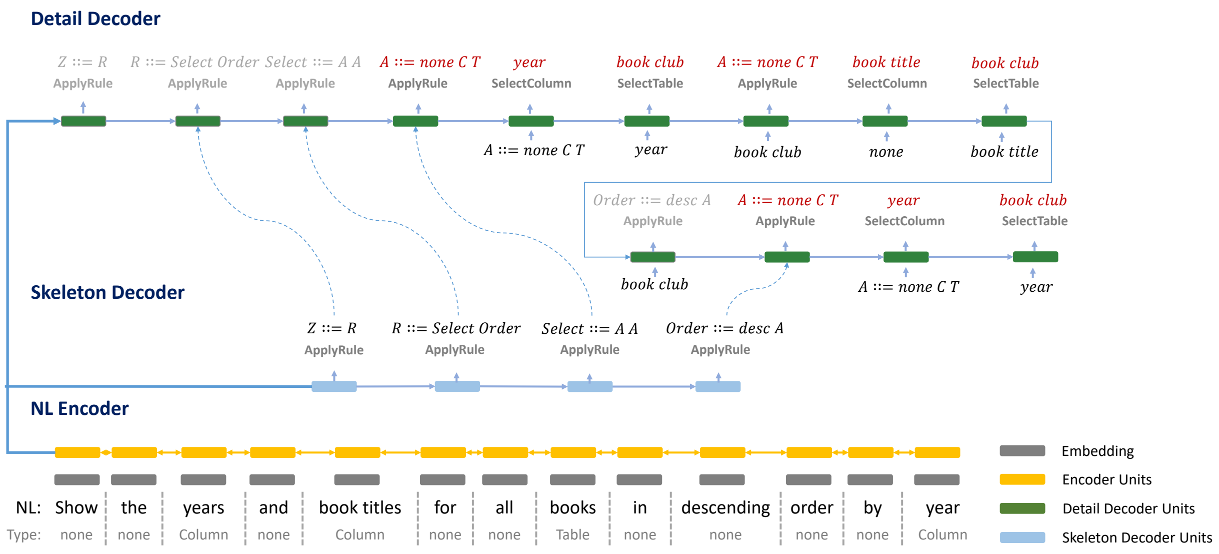


Figure 7: An overview of the coarse-to-fine framework to synthesize SemQL queries.

ized as follows.

$$
\begin{aligned}
p(y|x,s) &= p(q|x,s)p(y|x,s,q) \\
p(q|x,s) &= \prod_{i=1}^{T_s} p(a_i = \textsc{ApplyRule}[r]|x,s,a_{<i}) \\
p(y|x,s,q) &= \prod_{i=1}^{T_c}[\lambda_i p(a_i = \textsc{SelectColumn}[c]|x,s,q,a_{<i}) \\
&\quad + (1-\lambda_i)p(a_i = \textsc{SelectTable}[t]|x,s,q,a_{<i})]
\end{aligned}
$$

where q denotes the skeleton. $\lambda_i = 1$ when the ith action type is SelectColumn, otherwise $\lambda_i = 0$.

At training time, our model is optimized by maximizing the log-likelihood of the ground true action sequences:

$$
max \sum_{(x,s,q,y)\in\mathcal{D}} \log p(y|x,s,q) + \gamma \log p(q|x,s)
$$

where $\mathcal{D}$ denotes the training data and γ represents the scale between $\log p(y|x,s,q)$ and $\log p(q|x,s)$. γ is set to 1 in our experiment.

7.5 BERT

Figure 8 depicts the architecture of the BERT encoder.

7.6 Analysis on the Performance Gap between the Development set and the Test set

To test our hypothesis that the performance gap is caused by the different distribution of the SQL queries in Hard and Extra Hard level, we first construct a pseudo test set from the official training set of Spider benchmark. Then, we conduct further experiment on the pseudo test set and the official development set. Specifically, we sample

Dataset	Easy	Medium	Hard	Extra Hard
Pseudo Test A	24.2%	44.5%	14.4%	16.9%
Pseudo Test B	22.7%	44.1%	16.7%	16.5%
Pseudo Test C	24.7%	37.1%	22.9%	15.3%
Development	**24.1%**	**42.5%**	**16.8%**	**16.4%**

Table 5: The hardness distribution of the pseudo test A, the pseudo test B, the pseudo test C, and the development set.

20 databases from the training set to construct a pseudo test set, which has the same hardness distributions with the development set. Then, we train IRNet on the remaining training set, and evaluate it on the development set and the pseudo test set, respectively. We sample the pseudo test set from the training set for three times and obtain three pseudo test sets, namely, pseudo test A, pseudo test B and pseudo test C. They contain 1134, 1000, and 955 test data respectively.

Table 5 presents the hardness distribution of the three pseudo test sets and the official development set. Figure 9 presents the exact matching accuracy of SQL on the development set and three pseudo tests set after each epoch during training. IRNet performs competitively on the development set and the pseudo set C (Figure 9c), but there exists a clear performance gap on the pseudo test A and B (Figure 9a and Figure 9b). Although the hardness distributions among the development set and the three pseudo sets are nearly the same, the data distribution still has some difference, which results in the performance gap.

We further study the performance gap of Syn-

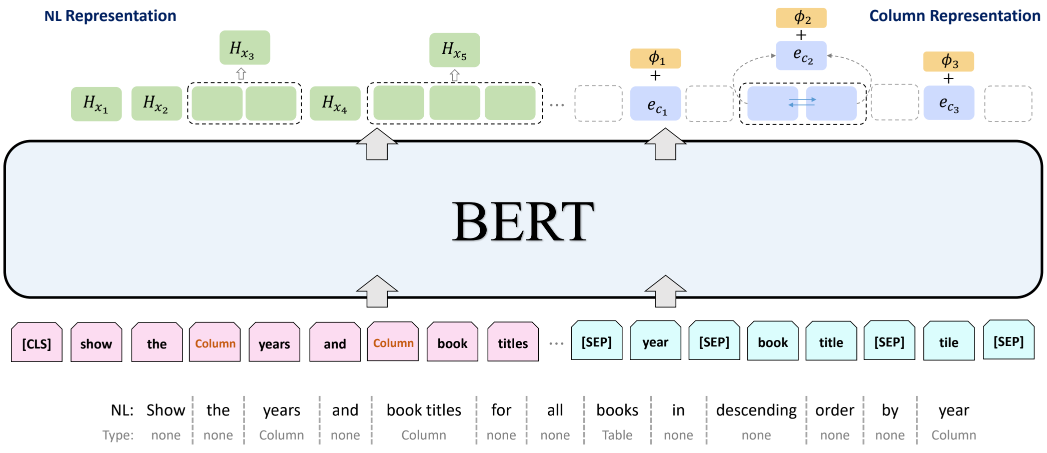
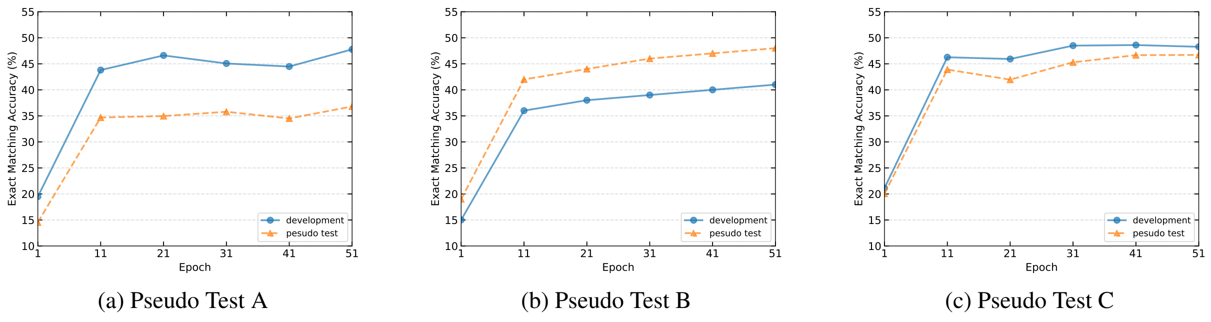

NL Representation
Column Representation
BERT
[CLS]
show
the
Column
years
and
Column
book
titles
[SEP]
year
[SEP]
book
title
[SEP]
tile
[SEP]
NL: Show the years and book titles for all books in descending order by year
Type: none none Column none Column none none Table none none none none Column

Figure 8: Encoding a question and column names with BERT.

Approach	**Dev**			**Pseudo Test A**		
	All	**Hard**	**Extra Hard**	**All**	**Hard**	**Extra Hard**
SyntaxSQLNet	17.4%	15.5%	2.9%	16.9%	11.0%	2.6%
+BERT +SemQL	34.5%	30.5%	17.6%	30.2%	24.5%	15.7%

Table 6: Exact matching accuracy on the development set and the pseudo test A set.



(a) Pseudo Test A (b) Pseudo Test B (c) Pseudo Test C

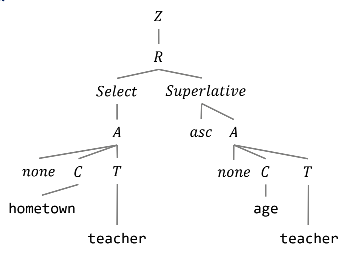
Figure 9: Exact matching accuracy of IRNet on the development set and the pseudo test sets.

taxSQLNet on the development set and the pseudo test A. As shown in Table 6, SyntaxSQLNet achieves 16.9% on the development set and 17.4% on the pseudo test A. When incorporating BERT and learning to synthesizing SemQL, SyntaxSQL-Net(BERT,SemQL) achieves 34.5% on the development set and 30.2% on the pseudo test A, exhibiting a clear performance gap (4.3%). SyntaxSQLNet(BERT, SemQL) significantly outperforms SyntaxSQLNet in the Hard and Extra Hard level. The experimental results show that when SyntaxSQLNet performs better in the Hard and Extra Hard level, the performance gap will be larger, since that the performance gap is caused by the different data distributions.

NL: What is the hometown of the youngest teacher?

SQL:
```
SELECT hometown FROM teacher
ORDER BY age ASC LIMIT 1
```

SemQL:

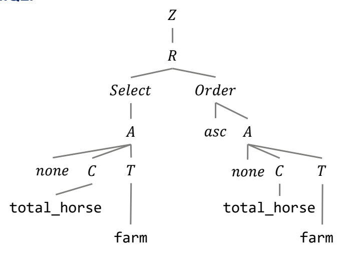


(a) Example 1.

NL: List the total number of horses on farms in ascending order.

SQL:
```
SELECT total_horses FROM farm
ORDER BY total_horses ASC
```

SemQL:

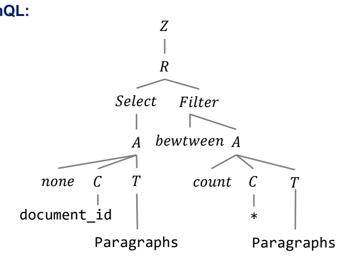


(b) Example 2.

NL: Gives the ids of documents that have between one and two paragraphs.

SQL:
```
SELECT document_id FROM Paragraphs GROUP BY
document_id HAVING count(*) BETWEEN 1 AND 2
```

SemQL:

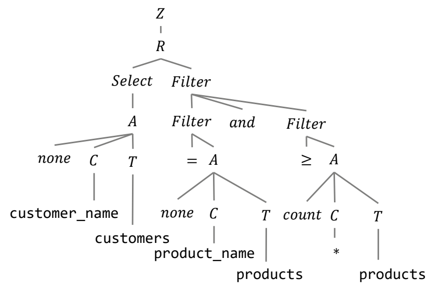


(c) Example 3.

NL: List the names of the customers who have once bought product "food".

SQL:
```
SELECT T1.customer_name FROM customers AS T1 JOIN
orders AS T2 JOIN order_items AS T3 JOIN products AS
T4 WHERE T4.product_name = "food" GROUP BY
T1.customer_id HAVING count(*) >= 1
```

SemQL:

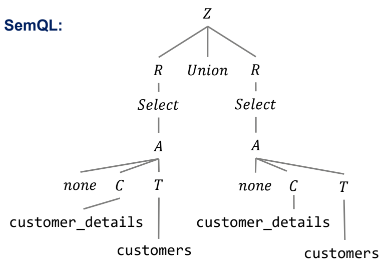


(d) Example 4.

NL: Find the names of all the customers and staff members.

SQL:
```
SELECT customer_details FROM customers
UNION SELECT staff_details FROM staff
```

SemQL:

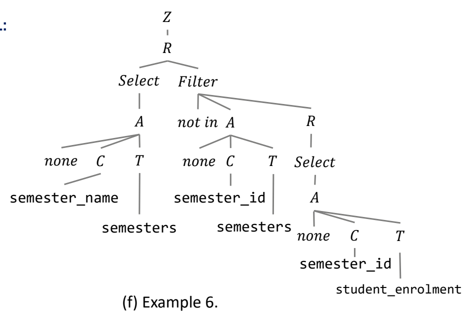


(e) Example 5.

NL: Which semesters do not have any student enrolled? List the semester name.

SQL:
```
SELECT semester_name FROM Semesters WHERE semester_id
NOT IN (SELECT semester_id FROM Student_Enrolment)
```

SemQL:

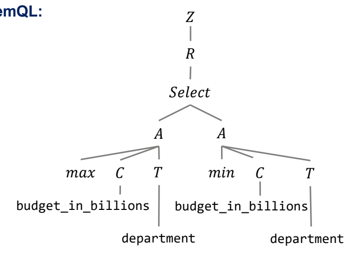


(f) Example 6.

NL: What are the maximum and minimum budget of the departments?

SQL:
```
SELECT max(budget_in_billions),
min(budget_in_billions) FROM department
```

SemQL:

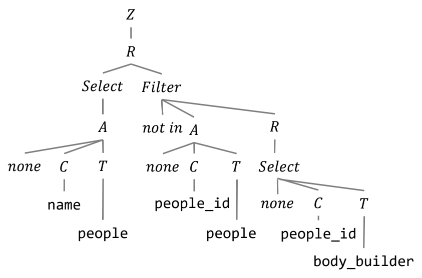


(g) Example 7.

NL: What are the names of body builders?

SQL:
```
SELECT T2.Name FROM body_builder AS T1 JOIN people AS
T2 ON T1.People_ID = T2.People_ID
```

SemQL:



(h) Example 8.

Figure 10: Examples of SemQL Query.