SQL-ACT: Content-based and history-aware input prediction for non-trivial SQL queries

by

Eric R. Manzi

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

September 2017

© 2017 Eric R. Manzi. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Author:

Department of Electrical Engineering and Computer Science
September 7, 2017

Certified by:

Samuel R. Madden, Professor, Thesis Supervisor
September 7, 2017

Accepted by:

Christopher J. Terman, Chairman, Masters of Engineering Thesis Committee
Abstract

This thesis presents SqlAct, a SQL auto-completion system that uses content-based and history-aware input prediction to assist in the process of composing non-trivial queries. By offering the most relevant suggestions to complete the partially typed query first at the word-level and then at the statement-level, SqlAct hopes to help both novice and expert SQL developers to increase their productivity. Two approaches are explored: word-level suggestions are optimized based on the database’s schema and content statistics, and statement-level suggestions that rely on Long Short-term Memory (LSTM) Recurrent Neural Networks language models trained on historical queries. The word-level model is integrated in a responsive command-line interface database client which is evaluated quantitatively and qualitatively. Results shows SqlAct provides a highly-responsive interface that makes high quality suggestions to complete the currently typed query. Possible directions for integration with the word-level model in the command-line tool are explored as well as the planned evaluation techniques.
Acknowledgments

This thesis would not have been where it is today if not the help and guidance of a number of essential people.

First, I would like to thank my supervisor, Professor Samuel Madden who introduced me to the world of Database Systems through his class, 6.830, from which I learned the foundations of database systems and relational algebra. Professor Madden has been extremely patient when progress was slow, provided invaluable feedback, and pointed me towards useful resources. I have been extremely fortunate to be under his guidance.

I’d also like to thank my friend and classmate, Dalitso Banda, for his contribution in the early stages of this project. His insight and expertise are greatly appreciated.

Lastly, I’d like to thank my parents, without whose love, support, and sacrifice, I would not be at MIT.
# Contents

1 Introduction .......................................................... 16
   1.1 Motivation ....................................................... 16
   1.2 Features ......................................................... 17
   1.3 Outline .......................................................... 19

2 Related Work ......................................................... 20

3 Word-level completions ............................................. 22
   3.1 Overview ......................................................... 22
      3.1.1 Schema-based Predictions .............................. 23
      3.1.2 Keyword Predictions .................................... 24
      3.1.3 Parsing aliases .......................................... 24
      3.1.4 Join suggestions ........................................ 26
      3.1.5 Content-based Value Predictions ..................... 27

4 Evaluation .......................................................... 31
   4.1 Evaluation technique ........................................ 31
   4.2 Performance .................................................... 31
      4.2.1 Start-up Latency ........................................ 31
      4.2.2 Response Time .......................................... 32
      4.2.3 Memory and Storage efficiency ....................... 33
   4.3 Accuracy ....................................................... 33
      4.3.1 Projection clause ........................................ 34
4.3.2 FROM clause ......................................................... 34
4.3.3 JOIN clause ........................................................ 36
4.3.4 WHERE clause and Aliasing ..................................... 38
4.3.5 ORDER BY and GROUP BY clause ............................ 38

5 LSTM and N-gram Extensions ........................................ 42
  5.1 Statistical language models ...................................... 42
    5.1.1 N-gram models ............................................... 43
    5.1.2 Recurrent Neural Network Language Models ............. 43
  5.2 Network Architectures ........................................... 45
    5.2.1 Sequence-to-sequence ..................................... 45
    5.2.2 Sequence-to-token ......................................... 46
  5.3 The Dataset ...................................................... 47
    5.3.1 Query Log Exploratory Analysis ............................ 48
  5.4 Preprocessing ..................................................... 50
    5.4.1 Selection ..................................................... 51
    5.4.2 Tokenization ................................................ 52
    5.4.3 Dimensionality Reduction .................................. 53
    5.4.4 Case Folding ............................................... 54
    5.4.5 Real-valued Tokens ........................................ 55
    5.4.6 Data cleaning ................................................ 57
    5.4.7 Query validation ............................................ 60
    5.4.8 Padding ...................................................... 63
    5.4.9 One-hot encoding .......................................... 64

6 Experiments ............................................................ 66
  6.1 Training details ................................................... 66
    6.1.1 Sequence-to-sequence ..................................... 66
    6.1.2 Sequence-to-token ......................................... 66
    6.1.3 N-gram model ................................................ 67
  6.2 Experimental Results ............................................. 67
6.2.1 Training loss and over-fitting ........................................... 67
6.2.2 Memory and Compute Resources ................................. 68
6.3 Query Synthesis ................................................................. 71
6.3.1 Priming ................................................................. 71
6.3.2 Sampling ............................................................... 72
6.3.3 Validation ............................................................... 73
6.4 Model Evaluation ........................................................... 73
6.4.1 Sequence-to-token model ........................................... 73
6.4.2 Effect of using n-grams to assist in generation ............... 73
6.4.3 Sequence-to-sequence model .................................... 75
6.5 Proposed integration with the command-line tool ......... 75

7 Conclusion ........................................................................ 77
7.1 Contribution ................................................................. 77
7.2 Next steps ................................................................. 77
List of Figures

3-1 System architecture overview ................................................. 23
3-2 SQL keyword transition graph generated from SDSS query logs. Nodes
represent keywords. An edge from node A to node B indicates that the
SQL keyword A was observed to precede keyword B in the query logs
where all tokens expect those matching SQL keywords have been omit-
ted. An edge’s weight denotes the probability that keyword B follows
keyword A based on the observed precedence frequency. We should
note that this keyword transition graph is by no means complete. It
has been scaled down for visibility from 57 nodes and >280 edges to
33 nodes and <70 edges, so outgoing edge-weights in this graph may
not sum to unity. This graph is visualized using the GraphViz software 25
3-3 An example value completion. The number on the right of the name
shows the number of times that name occurs in the column.............. 28
3-4 Example of trie structure holding SQL keywords. A lookup on the key
's' would return the set ['select', 'set'] ...................................... 30
4-1 Normalized distribution of response times. The average response time
is 4.68ms. ................................................................. 33
4-2 Response time over session. A suggestion event is triggered at every
keypress ............................................................... 34
4-3 Suggestions in the projection clause ......................................... 35
4-4 Suggestions in the projection clause ......................................... 36
4-5 Suggestions in the FROM clause ........................................... 37
4-6 Suggestions in the JOIN clause .............................................. 38
4-7 Suggestions in the WHERE clause ......................................... 39
4-8 Suggestions in the GROUP BY and ORDER BY clauses ............ 40

5-1 A simple Recurrent Neural Net with 4 input units, 4 output units, and a recurrent hidden layer ............................................. 44
5-2 Long-Short Term Memory Cell. Source: deeplearning.net ........ 45
5-3 Sequence-to-sequence LSTM model with 5 units. Note that this is one of six input/output vectors that are created for the query shown. When training, we’ll first add padding to the front of the query as described in section 5.4.8 and move a context window of size \( n \) (\( n \) here is 5) across the sequence of tokens to generate the input/output pairs .... 46
5-4 Sequence-to-token LSTM model with 5 units ....................... 47
5-5 Zipfian distribution (frequency vs rank) of query tokens .......... 48
5-6 Top 15 keywords frequency distribution ............................... 49
5-7 An example of a complex nested query ................................. 51
5-8 Before (top) and After (bottom) preprocessing ...................... 54
5-9 In addition to the aforementioned preprocessing steps, ........... 57
5-10 Histogram showing distribution of percent composition of queries by UNK tokens. The x-axis shows the ‘x’ bins. The y-axis shows the probability that a randomly chosen variable falls within that the given bin. The closest fit Poisson distribution trace is plotted in each graph (dashed line). The data from the complete dataset (right) does not fit the Poisson distribution with \( \lambda = 7.7 \) where \( \lambda \) is the average % composition in the complete set suggesting that this data is skewed. On the other hand, the Poisson distribution with \( \lambda = 6.3 \) where \( \lambda \) is the average % composition in the distinct query set, is a good fit for this data .......................................................... 60
5-11 Box-plot distribution of percent composition of unknown tokens by query calculated over the complete dataset (a) and over the distinct dataset (b). ............................... 61

5-12 Box-plot distribution of query lengths (truncated) ....................... 64

5-13 One-hot encoding example .................................................. 65

6-1 Training loss vs epoch ....................................................... 68

6-2 % CPU usage measured during training ..................................... 70
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Some statistical data from pg_stats about the contents of the Yelp DB.</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>average start-up latency on a local and remote database</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>Equi-JOIN cardinalities by predicate. Corresponding JOIN clause on</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>predicate “table1.columnA = table2.columnB” take the form: table1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JOIN table2 ON table1.columnA = table2.columnB</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Example context-to-target mapping in 5-gram</td>
<td>43</td>
</tr>
<tr>
<td>5.2</td>
<td>Elements of the schema mentioned in the query logs</td>
<td>50</td>
</tr>
<tr>
<td>5.3</td>
<td>Top 5 most common queries in the dataset. Some parts of the query are</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>truncated for readability. Queries 2, 3 and 5 are essentially the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>same query with different JOIN syntaxes.</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>Vocabulary size and frequency cut-off</td>
<td>53</td>
</tr>
<tr>
<td>5.5</td>
<td>Summary statistics of the SqlLog dataset before and after preprocessing.</td>
<td>59</td>
</tr>
<tr>
<td>6.1</td>
<td>N-gram training details. Training time includes encoding phase. The</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>vocabulary size is the number of unique sequences with n words.</td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>Queries generated from different training epochs. Note that no special</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>keyword replacement is done at this stage</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

We live in a world increasingly driven by data. Data management and analysis systems have to make this data more accessible to a growing community of diverse database users ranging from domain experts to novice users. These users need systems that help them explore the data and allow them to find the relevant information without full knowledge of the content and relationships in a complex database. Novice and expert SQL developers alike are often required to manually type out entire queries which is often a boring and mechanical process which greatly slows down query writing and is often prone to errors.

The database community has been exploring ways to simplify the query composition process since the advent of SQL in the early 1970s. Several relational database tools today offer keyword, and table and column name suggestions based on the database schema, but few have leveraged the database content and query history to make non-trivial predictions about a user’s intended query.

Auto-completion is quickly becoming a must-have feature for any code editor. A significant amount of work has been put into implementing and improving auto-completion in web-based search interfaces. Similarly, several techniques have been developed to speed up the editing process for code editors that offer keyword completion for different languages. The most popular techniques for input prediction use
frequency and probability tables, syntactic prediction using grammars and semantic prediction models such as Trie-based language models. [1]

For SQL, most existing approaches to code completion are based solely on the database schema and only offer keyword, table name and column name suggestions. There is limited focus on leveraging the information available about the content of the database. The majority of these approaches do not employ the above techniques to make more accurate predictions about future queries, and those that do are only able to capture shallow semantic information.

1.2 Features

To address these limitations, this paper proposes SqlAct, a dynamic SQL input prediction tool that offers the most relevant suggestions to complete the currently typed query. To provide this functionality, SqlAct uses several techniques:

- When statistical information about a database’s contents is publicly available, it is utilized to inform predictions in queries where table values are referenced. When this information is not available, a random sample of the content of the database is used to estimate these statistics. We hypothesize that by restricting suggestions to values that occur more often, the probability of suggesting irrelevant completions is minimized;

- SqlAct suggests JOIN predicates that are based on measured JOIN cardinalities. Predicates with non-zero cardinalities are suggested, in descending order of cardinality. We estimate that JOIN predicates with higher cardinalities are more likely to be chosen;

- SqlAct logs all queries issued through it to a database. Past queries are used to inform future predictions;

- A prefix tree structure is used to store and retrieve suggestions matching the prefix (currently typed query) in order of the suggestion rank. This helps reduce the interface response time.
The above features are implemented in the SqlAct command-line tool and provide the key auto-completion capabilities. We explore an extension to SqlAct that aims to extend its predictive capabilities to the statement level using Long-Short Term Memory (LSTM) Recurrent Neural Networks. Recent neural network research has yielded models that can learn to generate new token streams that are syntactically similar to the real-world data and have achieved state of the art performance on natural language modeling tasks. LSTM Recurrent Neural Networks trained on a dataset of historical queries to learn a word-level model of the language which is then used to generate new queries conditioned on the currently typed query. LSTMs are able to remember arbitrary amounts of context so the generated sequences have long-range structure. It is vital that the model is able to find complex patterns since we need to capture dependencies that exist across arbitrarily long sequences in queries. This is of particular interest in query languages since words that appear early on in the query are more likely to reappear at a later point. This same pattern is usually observed in natural languages; N-grams are count-based language models which were previously considered state of the art in word prediction until recently, and they can still outperform LSTMs in capturing local dependencies. Bigram, trigram and pentagram models are trained on the historical query set and used to assist LSTMs in synthesizing completions.

To summarize, the contributions of this thesis are as follows:

- A command-line tool utilizing the database statistics, schema and content to provide auto-completion capabilities to speed up the query composition process.

- An exploration of different LSTM recurrent neural network language models that learn how to synthesize valid SQL conditioned on the typed prefix, assisted by N-gram language models.
1.3 Outline

Chapter 2 provides a brief comparison to other SQL auto-completion tools and a discussion of some code recurrent neural network based code synthesis.

Chapter 3 presents an overview of the framework and describes the features provided in the SqlAct command-line tool. It further describes how the database schema and content statistics are leveraged to provide word-level predictions and discusses techniques used to improve performance.

Chapter 4 evaluates the performance and accuracy of the SqlAct tool.

Chapter 5 introduces the LSTM RNN extensions, describes the preprocessing steps taken to prepare the historical query dataset, and the different models architectures that train on this data.

Chapter 6 discusses training details and resource limitations as well as preliminary training results. It further discusses how this extension could be integrated into the SqlAct command-line tool to assist with auto-completion.

Finally, chapter 7 presents an overall summary of the contributions of this work and planned improvements.
Chapter 2

Related Work

In this chapter, we discuss existing research in query auto-completion. The existing work in this field is fairly limited, and to the extent of our research there is no existing tool that attempts a neural network based approach to query prediction. The traditional approaches include: suggesting SQL keywords, and providing quick access to past queries. Tools like PgAdmin [4], a leading graphical developer tool for PostgreSQL and the IntelliJ IDEA editor [5] use these approaches to speed up the query editing process. However, the query history, which both of these tools present to the user, is not taken into account when suggestions are made and none of these approaches takes full advantage of the database content to make smarter suggestions.

Another common approach to query auto-completion is suggesting column and table names based on the database schema. Tools like pgcli [6] and Mysql workbench [7] provide word-level auto-completion by leveraging the database schema. Pgcli, is a command line interface postgres client, is effective at suggesting SQL commands as they are typed but does a poor job at ranking suggestions, which are presented in alphabetical order. After the user types the "FROM" keyword, one would expect the tables in the database to appear as top suggestions instead of all postgres literals in alphabetical order. While Mysql workbench does a better job of suggesting column names from the schema after a use types in a keyword, it was found to be slow when returning suggestions; it takes a few seconds for suggestions to show.

Recent research has demonstrated the usefulness of applying deep learning tech-
niques to software corpora to make code suggestions. [8] Statistical language models based on recurrent neural networks have been found to outperform most existing solutions, even in simple configurations. [10]

While historical query data sets have been used in query synthesis applications [13], and in context-aware probabilist query recommender algorithms [14], to the extent of our research no existing tool attempts to apply LSTM recurrent neural nets to query data sets for auto-completion.
Chapter 3

Word-level completions

3.1 Overview

Our implementation of this SQL autocomplete tool is based on the postgres variant of the SQL standard. Without loss of generality, we focus on supporting typical queries to retrieve data (projections, joins, aggregation, and selections). Support for table creation, user access management and database management that use keywords such as GRANT are more SQL variant specific. Our aim was to build a tool that can intelligently offer and rank suggestions of words to the user in real time as the query is being typed.

SqlAct was first built as a nodejs web application installable over a pre-existing postgres database server which used the node-postgres API to interact with the postgres server. Queries could be typed through the web frontend to be passed to the system which is made up of a parser and suggestion engine that work to retrieve suggestions for the next word from the database. The second version of SqlAct is a python-based command-line application which follows the same modular design but interacts with the database through a python-postgres library [32].
3.1.1 Schema-based Predictions

The SQL relational data model allow for data manipulation on the column and table levels in most cases. As such almost every keyword in SQL performs some action on a table or column and is followed by a set of one of the two in some form. Our design for word-level prediction uses a state-machine that keeps track of the currently typed query attributes such as the last keyword, the last word at the cursor, the projection, selection and nesting level. It uses this information to offer ranked suggestions for the next set of characters to complete the current word being typed or to suggest the next word if no word is currently being typed. In our design, we group SQL keywords that operate on tables as tableVerbs (e.g INSERT, FROM) and those that operate on columns as columnVerbs (e.g SELECT, WHERE, GROUP BY). This creates 5 major cases in which we grow the markov chain.

1. $tableVerb \to \text{table name}$
   
   The next word to follow a tableVerb keyword is a table name via definition

2. $columnVerb \to \text{table name}$
   
   Similarly, the next word to follow a columnVerb keyword is a column name via
definition

3. table name → table name

This case occurs when the user enters a table name followed by a comma (e.g. FROM users,). In this case the user is inputting a set of tables so the next word to follow is a table names (e.g. FROM users, business)

4. column name → column name

Similarly, this case occurs when the user enters a column name followed by a comma

5. columnVerb → record value

This case occurs when user types in a predicate. (e.g WHERE name='bob')

When the user enters a column name followed by a comma, we know that the next word is likely to be a column name. Suggestions are then obtained from other columns in the same table, then from columns in other tables, followed by aggregation functions and other SQL literals.

3.1.2 Keyword Predictions

We prime our suggestion engine with keyword co-occurrence statistics collected from SDSS query logs described in section 5.3. This is used to infer which clause is currently being typed and which clause will be entered next. We hypothesize that using this next-keyword model is transferable between databases with different schema and perhaps even different SQL dialects.

3.1.3 Parsing aliases

There are several rules that the parser runs to clean update and detect what stage of input in the query the user is at. The first rule of the parser is to delete comments from the query. Next, the parser scans for aliases in the query string. Aliases are of the form name AS username. We can identify aliases and then replace them with their original names. For example in the query
Figure 3-2: SQL keyword transition graph generated from SDSS query logs. Nodes represent keywords. An edge from node A to node B indicates that the SQL keyword A was observed to precede keyword B in the query logs where all tokens expect those matching SQL keywords have been omitted. An edge’s weight denotes the probability that keyword B follows keyword A based on the observed precedence frequency. We should note that this keyword transition graph is by no means complete. It has been scaled down for visibility from 57 nodes and >280 edges to 33 nodes and <70 edges, so outgoing edge-weights in this graph may not sum to unity. This graph is visualized using the GraphViz software.

SELECT name AS username WHERE username

is transformed to

SELECT name WHERE name |

This design simplifies how we handle aliases. Alias mappings are stored in a state object in memory.

After aliases and comments have been handled, the parser searches for unmatched parenthesis. The query is then truncated from the earliest unmatched parenthesis. By doing this, we reset the state to the earliest unmatched opening parenthesis. This allows SQL-ACT to treat sub-queries in parentheses as their own queries.

The parser’s state also keeps track of which columns are in the projection (between
SELECT and FROM) and the corresponding tables to help inform its predictions.
To grow the markov chain, the parser keeps track of the last word and the word that is currently being typed.

### 3.1.4 Join suggestions

Typing JOIN queries is one of the more difficult tasks for SQL developers as these can be quite long. SQL-ACT attempts to reduce this burden by suggesting a series of possible queries that might be typed given the current input. There are a number of ways two tables can be JOINed (inner, left, outer) and a number of attributes on which they can be JOINed. This get even more complex as more tables are added to the JOIN. For the case of simplicity, we limit our scope to suggesting only EQUIJOINs on two tables. As soon as the "FROM" keyword is received, the Parser extracts all the columns specified in the projection clause. Using the table schema, SQL-ACT finds all the tables containing the projected columns for each column and computes the intersection of these tables.

If this intersection is non-empty, then there exists a table, or a set of tables containing all the projected columns. This set of tables is consequently suggested for completion. Otherwise, if this intersection is empty, that means that barring any typing errors by the user, the projected columns come from different tables. This is a good indicator that the intended query contains a JOIN. A list of all columns shared by these two tables is then computed and a list of possible JOIN queries is constructed and returned for suggestion.

Join Cardinality is then used to rank these suggestions. A previous iteration of this feature used frequency histograms available in the statistics table to estimate join selectivity and cardinality, but the current iteration executes all possible join combinations in the background and stores actual cardinalities on persistent disk to be used later. This approach introduces a relatively heavy workload to the database and is the most expensive query issued by SqlAct. To reduce the computational complexity, cardinality is only computed for columns of the same type to reduce the computational complexity. Queries of the form: "SELECT COUNT(*) FROM join_clause" are
prepared for each candidate JOIN clause and issued to the database in the background while the user composes their query. The Yelp database used for testing has 5 tables and 22 columns, \( x \) possible joins are run 33 candidate 2-way joins in which the columns in the JOIN predicate are of the same datatype are selected from a total 176 possible joins. On a local instance of the database, these complete in 18.1 seconds on average, and 15.1 seconds on the remote database (It’s almost surprising that the average latency was consistently lower on the remote database than on the local one, but Amazon RDS \([33]\) instances have several performance optimizations which may offset the additional network overhead.)

### 3.1.5 Content-based Value Predictions

Consider the case where the following query is being typed:

```
SELECT name FROM users WHERE name=|
```

Upon parsing the `WHERE` keyword, SqlAct’s state machine enters the *Typing Predicate* state. First, the parser extracts the column name “name”, the operator “=”", the typed value which in this case is the empty string “\(|\)”, and the table name “users”. SqlAct then proceeds to retrieve value suggestions for the column `users.name`.

The PostgreSQL catalog, `pg_statistic` \([16]\) stores statistical data about the contents of the database. `pg_statistic` is not be readable by the public, but exposes information about public tables in via a publicly readable view: `pg_stats` \([15]\).

When available, interesting table statistics that are available include:

- **n-distinct**: A measure of uniqueness of the values in a column. “An n-distinct value of -1 indicates a unique column in which the number of distinct values is the same as the number of rows. If greater than zero, the estimated number of distinct values in the column. If less than zero, the negative of the number of distinct values divided by the number of rows.” \([15]\) For instance, in table 3.1 the business_id column is unique in the business table so n-distinct is -1, but in the reviews table, business_id n-distinct is 3583 meaning that there are 3583 unique business_id’s in the 111542 rows of reviews. In the tips table, -0.15
means business_id is repeated a number of times.

- most_common_vals: This lists the most common values in the column.

- most_common_freqs: This lists the frequencies of the most common values in the column.

The values that are suggested for value comparisons are from most_common_vals in the pg_stats view. The number of entries in most-common-vals and most-common-freqs is configurable, but is usually set to 100. When the currently typed query is ends in a comparison operator preceded by a column, SqlAct may suggest values to complete the predicate clause if the column has an n-distinct is in the range $-0.5 \leq n_{\text{distinct}} < 0$ or $1 \leq n_{\text{distinct}} < 500$. So, in the yelp DB database, value suggestions would be offered for the following columns: business.state, business.stars, reviews.stars, tips.business_id, tips.user_id, tips.date, users.name, users.fans, and users.yelping_since. Intuitively, these columns are most likely to be queried on with a value comparison predicate because they contain re-occurring values. Figure 3.1.5 shows how SqlAct’s suggested completions for the example above.

```
SELECT name FROM users WHERE name=|
   | 389
John | 337
Michael | 317
David | 289
Mike | 237
Chris | 223
Matt |
```

Figure 3-3: An example value completion. The number on the right of the name shows the number of times that name occurs in the column.

**Prefix tree optimization**

A key optimization used to boost performance is the use of a prefix tree/trie structure to store retrieved suggestions in order to prevent additional queries to the database.
<table>
<thead>
<tr>
<th>column name</th>
<th>avg-width</th>
<th>n-distinct</th>
</tr>
</thead>
<tbody>
<tr>
<td>business.business_id</td>
<td>23</td>
<td>-1</td>
</tr>
<tr>
<td>business.name</td>
<td>18</td>
<td>-0.881057</td>
</tr>
<tr>
<td>business.city</td>
<td>10</td>
<td>65</td>
</tr>
<tr>
<td>business.full_address</td>
<td>47</td>
<td>-0.861723</td>
</tr>
<tr>
<td>business.state</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>business.latitude</td>
<td>8</td>
<td>-0.908713</td>
</tr>
<tr>
<td>business.longitude</td>
<td>8</td>
<td>-0.909202</td>
</tr>
<tr>
<td>business.stars</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>reviews.review_id</td>
<td>23</td>
<td>-1</td>
</tr>
<tr>
<td>reviews.business_id</td>
<td>23</td>
<td>3583</td>
</tr>
<tr>
<td>reviews.user_id</td>
<td>23</td>
<td>-0.165175</td>
</tr>
<tr>
<td>reviews.stars</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>reviews.text</td>
<td>597</td>
<td>-0.99679</td>
</tr>
<tr>
<td>reviews.date</td>
<td>4</td>
<td>3086</td>
</tr>
<tr>
<td>tips.business_id</td>
<td>23</td>
<td>-0.15252</td>
</tr>
<tr>
<td>tips.user_id</td>
<td>23</td>
<td>-0.271462</td>
</tr>
<tr>
<td>tips.date</td>
<td>4</td>
<td>-0.123187</td>
</tr>
<tr>
<td>tips.text</td>
<td>61</td>
<td>-0.979854</td>
</tr>
<tr>
<td>users.name</td>
<td>6</td>
<td>-0.194311</td>
</tr>
<tr>
<td>users.fans</td>
<td>4</td>
<td>130</td>
</tr>
<tr>
<td>users.yelping_since</td>
<td>4</td>
<td>139</td>
</tr>
<tr>
<td>users.user_id</td>
<td>23</td>
<td>-1</td>
</tr>
</tbody>
</table>

Table 3.1: Some statistical data from pg_stats about the contents of the Yelp DB.

When the user starts typing a query, a set of suggested completions is retrieved. As they continue typing, several of the suggestions still match the prefix so it is not necessary to retrieve new results from the database. Prefix trees provide the most efficient support for this functionality. When the results are fetched, they are inserted into the prefix tree and each time a new key is typed, instead of issuing the query to the database which adds an additional delay, SqlAct performs a prefix-tree lookup for the typed prefix and if it the lookup returns more than 10 results, then these are returned as the new set of suggestions. When the lookup returns fewer than 10 results, these are sent for auto-completion, while new suggestions are fetched and inserted into the trie. The old contents of the trie are then discarded.

A prefix lookup in a Trie takes time $O(n)$ where $n$ is the number of letters in the prefix. Since the prefix tree is stored in memory, on the same machine as the user,
Figure 3-4: Example of trie structure holding SQL keywords. A lookup on the key ‘s’ would return the set ['select', 'set']

these lookups are able to return results almost instantaneously.

Performance Evaluation of w
Chapter 4

Evaluation

4.1 Evaluation technique

This tool is evaluated along 2 metrics: performance and accuracy. Start-up latency and response time are used to evaluate the tool’s performance. A qualitative analysis of the tool’s accuracy is then presented based on suggestion quality at different stages of query composition. Finally, the storage overhead induced by the tool is discussed. Testing is done with local and remote (hosted on Amazon RDS [33]) PostgreSQL instances of the Yelp database [35]. Note that the Yelp database is has a relatively small schema (5 tables and 22 columns in total). Measurements described in this section may be not representative of SqlAct’s behaviour on a more complicated database.

4.2 Performance

4.2.1 Start-up Latency

When SqlAct is first launched, several start-up routines are issued. The most significant of these are:

- Reading configuration files on disk (this includes keyword literals, keyword cooccurrence values, and join cardinality values when available)
- Establishing a connection to the database
• Reading the database schema

• Collecting table statistics

Table 4.1 compares the average start-up latency on a local and remote database. Latency resulting from collecting JOIN cardinalities is not included since it is run in the background. JOIN cardinality statistics, however, contribute to the start-up latency along with other configuration files when are read off disk.

In general, we found that SqlAct introduces 38% increase in start-up latency when connecting to a database client.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Latency on local DB (s)</th>
<th>Latency on remote DB (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Establishing DB connection</td>
<td>0.0363</td>
<td>1.27</td>
</tr>
<tr>
<td>Reading configuration files</td>
<td>0.0403</td>
<td>0.0383</td>
</tr>
<tr>
<td>Loading schema</td>
<td>0.00716</td>
<td>0.186</td>
</tr>
<tr>
<td>Collecting table statistics</td>
<td>0.00418</td>
<td>0.269</td>
</tr>
<tr>
<td>Total start-up time</td>
<td>0.0879</td>
<td>1.76</td>
</tr>
</tbody>
</table>

Table 4.1: average start-up latency on a local and remote database

4.2.2 Response Time

It is established that 100 ms is the maximum response time limit for an interactive application to appear to a user as if it is reacting instantaneously [34]. Suggestions are emitted at every key-press by the \texttt{emit\_suggestion()} function. We collect the execution times for this function during a 60 second-long query composition session. Results are plotted in figures 4.2.2 and 4.2.2

The average response time is the same for a local database as for a remote database because the Completer module stores information required for suggestions in-memory. The average response time for this session is 4.68ms. (The same session averages 3.9ms response time on the remote database.) One observation from figure 4.2.2 is that the response time tends to increase slightly as the query typed gets longer. This might be explained by the increased complexity of parsing the typed prefix.
The maximum recorded latency was 12.8 ms so the 100 ms interactivity threshold is met.

4.2.3 Memory and Storage efficiency

Memory usage statistics were collected during the execution of the SqlAct command-line process. The memory usage of sqlact-cli ranges from 8 to 12 MB, with 11.3 MB being the average. The sqlact CLI tool itself is 2.5 MB in uncompressed format and 26 KB with lzma compression, with the next-keyword statistics file making the largest contribution to storage.

4.3 Accuracy

We now present a qualitative analysis of the tool’s accuracy based on suggestion relevance. We look at its ability to suggest relevant columns in the SELECT, FROM, ORDER BY and GROUP BY clauses based on the columns in the projection. We also look at how tables in the WHERE and JOIN clauses are suggested.
4.3.1 Projection clause

Figures 4-3, 4-5, 4-7 and 4-8 show SqlAct in action. Screenshots of the command-line interface are taken to demonstrate some example completions suggested by SqlAct. Figures 4-3(a)-(d) show suggestions in the projection clause. We show in 4-4(a) and (b) how the order of columns are suggested for the next column in the projection changes based on the previous column to one that gives preference to columns in the same table first. Figures 4-3(c) shows an example of how suggestions are filtered by typed prefix. Also notice that matching functions are suggested after all possible columns have been suggested. Figure 4-3(d) shows that columns can be suggested as function arguments. Finally, we see that SqlAct is able to switch from suggesting columns in figures 4-4(a) and (b) to suggesting the next keyword in 4-3(b)

4.3.2 FROM clause

Figures 4-5(a) and (b) show that tables suggested in the FROM-clause depend on columns specified in the projection. In (a), the projected column, ‘fans’, is only found in the ‘users’ table, so this is the only suggested table while in (b), all available tables are suggested. Figure 4-5(d) shows an example of SqlAct recognizes the nested sub-
query syntax and suggests the SELECT keyword instead of WHERE or JOIN which usually follow the FROM keyword. We see in figure 4-5(c) that the most common keywords after FROM will be suggested after when SqlAct believes that the user has finished typing all the tables in the selection. One limitation of SqlAct is observed here - the keywords, BETWEEN and AND, which are supposed to appear in the predicate, are suggested here. Even more worrisome is that they appear before valid following keywords like ORDER, GROUP, INNER... . This suggests that SqlAct’s reliance on the next-keyword statistics collected from the SDSS query logs comes.
Figure 4-4: Suggestions in the projection clause.

4.3.3 JOIN clause

Figure 4-6(a) and (b) show suggested equi-JOIN clauses after the JOIN keyword is typed. We see in (a) that the suggestions are only ordered by JOIN cardinality, which is shown in table 4.2. However, in (b), which has the columns ‘stars’ and ‘yelping-since’ specified in the projection, the top JOINs suggested are with the ‘users’ table since it has ‘yelping-since’ as an attribute. The JOIN predicate which makes the most sense to JOIN the ‘reviews’ and ‘users’ tables on is the second suggestion shown. Ranking JOIN predicates by cardinality alone doesn’t always make sense. The top ranked JOIN predicate is ‘reviews.stars = business.stars’ which has a cardinality of 59,483,141 but it is highly unlikely that this will ever be chosen. However, this approach eliminates a large number of non-sensible JOIN predicates that would have...
been wasteful to consider. Furthermore, out of the 22 JOIN predicates with non-zero cardinalities, there are 7 sensible predicates, and these are among the top 11. This could be improved on, but is good enough for our purposes since it’s merely an estimation. An interesting direction might involve consideration of the uniqueness of the involved columns and from the n-distinct statistic, and row counts for the involved tables. We reiterate that the Yelp DB with which these measurements are done has a relatively simple schema and may not represent how well this approach SqlAct would fair with complex schemata.

Figure 4-5: Suggestions in the FROM clause.
(a) JOIN clause when projected columns are unique to alternate table

(b) No columns projected

Figure 4-6: Suggestions in the JOIN clause.

4.3.4 WHERE clause and Aliasing

Figures 4-7(a)-(d) show suggestions made in the WHERE clause. Columns in the predicate are suggested with preference given to columns belonging to the table in the FROM clause as shown in (a). When column aliases are used, they are used substituted in for the corresponding column in future suggestions as shown in (b). Figure 4-7(c) show value predictions in the predicate with the “=” and LIKE comparison operators. While the

4.3.5 ORDER BY and GROUP BY clause

In figure 4-8(a) we again see the next-keyword prediction in action. Figures 4-8(b) and (c) show that columns suggested in the GROUP BY and ORDER BY clause are ranked higher if they appear in the projection. Suggestions in the GROUP BY clause however, have ‘stars’ as the top suggestion although the second option, ‘name’ is the more likely choice.
(a) Column suggestions in the WHERE clause

(b) Column suggestions with Aliasing

(c) Value in equality predicate

(d) Value in predicate with LIKE operator

Figure 4-7: Suggestions in the WHERE clause.
(a) ORDER followed by BY keyword

(b) Preferred GROUP BY columns appear in projection

(c) Preferred ORDER BY columns appear in projection

(d) Next keyword after ORDER BY

Figure 4-8: Suggestions in the GROUP BY and ORDER BY clauses.
Table 4.2: Equi-JOIN cardinalities by predicate. Corresponding JOIN clause on predicate “table1.columnA = table2.columnB” take the form: table1 JOIN table2 ON table1.columnA = table2.columnB
Chapter 5

LSTM and N-gram Extensions

The system described above can make current-word, next-word, and multi-word suggestions to complete the partially typed query. In this chapter, we describe an extension to the system that extends its predictive capabilities to the statement level. We motivate the use of a recurrent neural network (RNN) based language model enhanced with n-gram models, describe the design and architecture of these models, and the preprocessing steps taken to prepare the models for training on historical queries.

5.1 Statistical language models

If you want to predict the next word in a sentence you better know which words came before it. Statistical language models can be used to build probabilistic predictive distributions over the next word conditioned on the preceding context. Query statements contain recurrent themes at varying intervals. For instance, a column mentioned in the projection typically reappears in the selection clause. So, to model a query language, we need a model that is able to capture these long-range dependencies, as well as the local semantic dependencies.
5.1.1 N-gram models

The most widely used frequency-based technique for word prediction is based on n-gram statistics, which are basically word co-occurrence frequencies collected from sequences of words in a corpus. N-gram models are probabilistic language models and build a probabilistic distribution over the next word given a sequence of \( n - 1 \) words, so n-grams are considered Markov models. These models are relatively fast to train but quickly grow in complexity as the size of the context \( (n) \) grows. Complexity reduction schemes have been studied. [1]

A key limitation of these models is that they lack the ability to capture long-range regularities since they are limited to the size of the context. Despite significant progress, existing techniques cannot synthesize usable code beyond simple sequences, although they have been shown [17] to be complementary to neural net models.

Table 5.1 shows how the query below is recorded in the 5-gram. (Note that this representation does not show the encoding that we apply to the query.)

\[
\text{SELECT description FROM dbobjects WHERE name = 'DataConstants'}
\]

<table>
<thead>
<tr>
<th>Context</th>
<th>Target</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SELECT, description, FROM, dbobjects)</td>
<td>WHERE</td>
<td>87147</td>
</tr>
<tr>
<td>(description, FROM, dbobjects, WHERE)</td>
<td>name</td>
<td>87147</td>
</tr>
<tr>
<td>(FROM, dbobjects, WHERE, name)</td>
<td>=</td>
<td>175576</td>
</tr>
<tr>
<td>(dbobjects, WHERE, name, =)</td>
<td>'DataConstants'</td>
<td>939</td>
</tr>
</tbody>
</table>

Table 5.1: Example context-to-target mapping in 5-gram

5.1.2 Recurrent Neural Network Language Models

Recurrent Neural Networks (RNNs) are a type of Deep Neural Nets that encode behaviour over sequences of inputs. They can be trained to generate sequences with the same long-range structure as real-world data. Connections between RNN units can form cycles which allows them to learn local and long-term dependencies of data which means they can capture phrase-level interactions in training examples.

Contrary to n-grams, which count exact matches in the training set, RNNs do
not use exact templates from the training data to make predictions, but rather, use their internal representation to perform a “high-dimensional interpolation" between training examples. Because recurrent neural networks can preserve sequential information in their hidden state, they are said to have ‘memory’ which spans several time steps and affects the way future examples are learned.

RNNs typically consist of an embedding input layer, one or more hidden layers, and a densely connected output layer. Our design uses softmax layer to convert outputs to probabilities. Input sequences are presented to the input layer, and propagated through the hidden layers to the output layer. This process is known as the forward pass of the network. The backward pass involves minimizing the loss function using gradient descent.

![Figure 5-1: A simple Recurrent Neural Net with 4 input units, 4 output units, and a recurrent hidden layer](image)

Standard RNNs generate new sequences based on the previous input one step at a time by sampling from the previous output and including that sample in the input to the next generation step. The problem with this is that if the input was a short sequence, and the RNN makes a bad prediction at one previous step, it’s difficult to recover from the mistake.

Long-Short Term Memory (LSTM) is an RNN architecture that is built to be better than standard RNNs at learning and exploiting long-term dependencies in the data by remembering information for long periods of time. Having longer term
memory helps resolve this because if the network cannot make sense of the recent context, it is able to look back further in time when making predictions. [18]

LSTMs also help resolve a common RNN problem known as the vanishing gradient problem, [11] which is caused by an unstable relationship between hidden state parameters. LSTMs avoid this problem by introducing an intermediate type of storage via a memory cell, and attaching feedback links to different layers of the network to preserve the error which is back-propagated through time to give them an internal recurrence (or self-loop). Figure 5.1.2 shows the structure of a basic LSTM cell. It has a weight matrix that connects input to hidden state and a weight matrix that connects hidden state to hidden state at previous time step. [19]

Figure 5-2: Long-Short Term Memory Cell. Source: deeplearning.net

5.2 Network Architectures

Both recurrent neural networks are built in tensorflow. [20] with LSTM cells instead of standard RNN cells. We test 2 different architectures: sequence-to-sequence (or many to many) and sequence-to-token (many to one).

5.2.1 Sequence-to-sequence

In this model, we train the network by feeding it with an input sequence input and an output sequence. The input and output layers are 100 units long. Each unit There is a single LSTM hidden layer with 100 hidden units to unfold. We use softmax
activation and categorical cross-entropy loss when training. As a simple example, take the query: SELECT * FROM specobjall

Figure 5.2.1 shows the equivalent sequence to sequence mapping that is fed to the network with only 5 units for visibility. Note that this representation omits the one-hot encoding that is first applied to the sequence, which is described in section 6.1.

Figure 5-3: Sequence-to-sequence LSTM model with 5 units. Note that this is one of six input/output vectors that are created for the query shown. When training, we’ll first add padding to the front of the query as described in section 5.4.8 and move a context window of size $n$ ($n$ here is 5) across the sequence of tokens to generate the input/output pairs.

## 5.2.2 Sequence-to-token

In this model, we model next-word prediction as a sequence classification problem where we are given a sequence of inputs and tasked to predict a classification for the sequence. For our purposes, this sequence is a length 20 word vector and this classification is the word following this sequence. This architecture has been used in sentiment analysis models [31] and somewhat resembles that of a 21-gram model with a context size of 20. The input layer of the network has 20 units which take in a fixed length history of 20 word vectors. The output layer is a densely connected single neuron that output the 21st token in the sequence. The model is trained with softsign.
activation and mean-squared error loss (RMSProp [30]). Softsign activation is chosen over softmax or tanh since it’s faster and less prone to saturation in LSTMs. [9] Figure 5.2.2 shows the equivalent sequence to sequence mapping that is fed to the network with only 5 inputs units for visibility.

Figure 5-4: Sequence-to-token LSTM model with 5 units.

5.3 The Dataset

We now describe the historical query dataset. The Sloan Digital Sky Survey (SDSS) is a large digital astronomy archive made publicly accessible via several means including HTTP and SQL. The Catalog Archive Server provides query interfaces for accessing the SDSS data via a collection of SQL Server databases. All SQL requests issued to database are recorded and made publicly available [21] in the SqlLog tables along with query information including, but not to limited to, the number of rows returned, the timestamp, and whether or not the request returned with an error.
5.3.1 Query Log Exploratory Analysis

Term frequency

Counting the terms present in the query logs alone gives interesting information. The dialect used by the SDSS query tool is syntactically most similar to SQL Server. SQL (which stands for Structured Query Language) is a formal, structured language so one might not expect to see a Zipfian distribution of term frequency that is usually found in natural languages. But indeed that is what we see. Figure 5.3.1a, shows the power law relationship (note the long-tailed distribution) between the frequency and rank calculated over the complete dataset. In figure 5.3.1b, log(frequency) is plotted against log(rank) to demonstrate the Zipfian-like behavior with the corresponding line of best fit of gradient -1.7.

The step-like effect in Figure 5.3.1 (a) differentiates this distribution from that of natural languages. Figure 5.3.1 shows the frequency distribution of the top 15 most frequent SQL keywords found in the dataset. Terms that commonly appear together like (select, from), (join, on), (order, by) have very similar frequencies which explains this step-like effect.

We conclude from this that we can treat this historical query dataset as a natural
language corpus and apply to it language models that are commonly applied to natural languages.

**Query coverage**

Table 5.2 shows the query set’s coverage of the schema. The most commonly mentioned view in the sdss schema is `photoprimary` which is queried against approximately 2.6M times. For tables, this is `photoobjall` at 449k mentions, and for functions, `fgetnearbyobjeq`, which is called 2.4M times. The most popular column, `objID`, is mentioned over 5.2M times which seems high at first glance, but is not surprising considering that at least 28 tables and 12 views have `objID` as an attribute. We have about 6.5M queries in the dataset, so the query logs seem to be heavily skewed towards a few of the tables which might lead to a skewed model.

One step taken to prevent reduce the effect of this heavy skew in the dataset is to include all distinct queries in the training set (with the exception of a few samples used for testing later). Since the training set taken from a random sample of the complete dataset, it’s very likely that some of the less frequent syntax will never be seen during training and won’t be learned by the model. Table 5.3 shows the
Table 5.2: Elements of the schema mentioned in the query logs

<table>
<thead>
<tr>
<th>Type</th>
<th># in schema</th>
<th># mentioned in the query logs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tables</td>
<td>124</td>
<td>115</td>
</tr>
<tr>
<td>Views</td>
<td>59</td>
<td>51</td>
</tr>
<tr>
<td>Functions</td>
<td>246</td>
<td>205</td>
</tr>
<tr>
<td>Columns</td>
<td>5246</td>
<td>853</td>
</tr>
<tr>
<td>Constants</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 5.3: Top 5 most common queries in the dataset. Some parts of the query are truncated for readability. Queries 2, 3 and 5 are essentially the same query with different JOIN syntaxes.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Query statement</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SELECT TOP 1 clean FROM photoobj</td>
<td>1195221</td>
</tr>
<tr>
<td>2</td>
<td>SELECT TOP 1 CAST ( p.objid AS varchar ( 20 ) ) AS objid, p.run, p.rerun, p.camcol ... FROM fgetnearbyobjeq ( 195, 2.5, 0.5 ) n , photoprimary p WHERE n.objid = p.objid</td>
<td>1120197</td>
</tr>
<tr>
<td>3</td>
<td>SELECT TOP 1 p.objid , p.run , p.rerun , p.camcol , p.field , ... FROM fgetnearbyobjeq ( 195, 2.5, 0.5 ) n , photoprimary p WHERE n.objid = p.objid</td>
<td>421812</td>
</tr>
<tr>
<td>4</td>
<td>SELECT TOP 1 clean FROM star</td>
<td>304401</td>
</tr>
<tr>
<td>5</td>
<td>SELECT TOP 1 CAST ( p.objid AS varchar ( 20 ) ) AS objid , p.run , p.rerun , p.camcol ... FROM fgetnearbyobjeq ( 195, 2.5, 0.5 ) n JOIN photoprimary p ON n.objid = p.objid</td>
<td>289313</td>
</tr>
</tbody>
</table>

Non-trivial queries

While the majority of queries in the dataset are short queries and have basic syntax, we still find some complex queries including nested sub-queries such as the one shown in figure 5-7 and multi-way joins (up to 11-way joins!)

5.4 Preprocessing

To train our model, the data first needs to be preprocessed to get it into the right format. Preprocessing is vital step when preparing data for analysis and involves a
Figure 5-7: An example of a complex nested query

series of steps through which the data is transformed to a form that is more suitable for analysis. These steps include data cleaning and selection to eliminate invalid or troublesome elements of the dataset, dimensionality reduction, which aims to minimize unnecessary memory, storage, or computational complexity by reducing the feature space. Preprocessing also involves text normalization, a process that transforms non-standard variations of the words into a standard form. This makes the input more consistent which facilitates operations on the data.

5.4.1 Selection

For this study, 10 million valid SQL queries were collected from the Query Logs. A query was considered valid if it returned without an error (that is, when $SqlLog.error = 0$). Of these 10M queries, 7M returned at least 1 row ($SqlLog.rows > 1$). The logs were further cleaned to eliminate records which contained illegal characters or
encoding errors, leaving 6,572,559 records.

5.4.2 Tokenization

The purpose of tokenization is to produce a dataset which when split by line then by single whitespace, produces a list of token sequences, where each sequence represents a query statement and each token is a stand-alone element of the query. The following pre-processing steps are taken:

- Multi-line query statements are reduced to single lines (for the sake of simplicity, paragraphs marks are not preserved);
- Multiple-space/tab characters are collapsed to single space;
- We then prepend and append <SOQ> (start-of-query) and <EOQ> (end-of-query) tokens to denote the start and end of a query, respectively. This allows us to learn which tokens typically appear at the start of a query statement. Since statements in our dataset do not end in a semi-colon, the standard statement terminator, this allows us to keep track of where a query ends;

Further care is taken to handle punctuation and special characters.

- The special wildcard character ‘*’ is replaced with the token <W> to differentiate it from the multiplication operator.
- Whitespace is prepended and appended to commas, parentheses and comparison operators, if none exists. What this means is that punctuation marks are treated as separate tokens. While this increases the number of tokens per query, it allows us to tokenize every clause in each query in a standardized fashion. SQL syntax doesn’t require whitespace before or after a comma, parenthesis, or operator which might make it difficult to separate functions from their parameters, columns from other columns in a projection or predicate clause.
5.4.3 Dimensionality Reduction

After the above steps, our dataset has a 238,694,701 words of which 1,398,311 are distinct. The set of all distinct words in our dataset is hereafter referred to as the Vocabulary. Training neural networks is computationally expensive so it is a good idea to limit the of size our Vocabulary. At 1.4M vocabulary elements, the model will be very slow to train, if at all the input data can fit in memory.

The majority of words in our dataset never appear more than once. Vocabulary truncation is a common preprocessing pattern in which infrequent tokens are merged into one class that represents all infrequent tokens. [12] Replacing all infrequent words with the $<$UNK$>$ (for “unknown”) token shrinks the size of our Vocabulary from 1.4M to 9.5K. Table 5.4 shows the vocabulary size computed at different minimum frequency cut-offs.

<table>
<thead>
<tr>
<th>Frequency cut-off</th>
<th>Vocabulary size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (original)</td>
<td>1398311</td>
</tr>
<tr>
<td>10</td>
<td>28387</td>
</tr>
<tr>
<td>20</td>
<td>9481</td>
</tr>
<tr>
<td>30</td>
<td>7294</td>
</tr>
<tr>
<td>40</td>
<td>6069</td>
</tr>
<tr>
<td>50</td>
<td>5332</td>
</tr>
<tr>
<td>60</td>
<td>4472</td>
</tr>
<tr>
<td>70</td>
<td>4050</td>
</tr>
<tr>
<td>80</td>
<td>3791</td>
</tr>
<tr>
<td>90</td>
<td>3584</td>
</tr>
</tbody>
</table>

Table 5.4: Vocabulary size and frequency cut-off

A minimum frequency of 20 is chosen as the frequency threshold for infrequent words. 2,506,518 words (approximately 1% of words in the original dataset) are replaced with the $<$UNK$>$ token. We believe this small loss in accuracy is a small price to pay for the 98% reduction in the Vocabulary size which a very significant speedup. This technique is somewhat similar to how humans learn - before adding a word to one’s vocabulary, one first needs to hear it used a number of times and in different contexts.

Figure 5-8 shows a sample raw query before and after the above preprocessing steps
are applied. Notice how multiple lines are collapsed to a single line while tabs and multiple spaces are collapsed to a single space. Commas, parentheses and comparison operators are padded with whitespace. A <SOQ> token is prepended to the start of the query to denote its start and an <EOQ> token is appended to denote its end. The parameters of the `dbo.fGetObjFromRectEq()` function are replaced with the <UNK> token to denote an infrequent token.

```
SELECT p.ra, p.dec, p.raErr, p.decErr, p.objId, p.run, p.rerun, p.camcol, p.field,
       dbo.fPhotoModeN(mode) AS mode, nChild,
       dbo.fPhotoTypeN(p.type) AS type, clean,
       flags, psfMag_u, psfMag_g, psfMag_r
FROM PhotoObj as p
JOIN dbo.fGetObjFromRectEq(358.881,-20.168,358.899,-20.152)
AS R
ON P.objID=R.objID;
```

Figure 5-8: Before (top) and After (bottom) preprocessing

5.4.4 Case Folding

It is often convenient to lowercase every word in the dataset because it makes terms more likely to match. Case Folding is a form of text normalization that transforms all tokens belonging to the same set into a single uniform case. [22] The query language used in SDSS is not case sensitive [23], so SQL keywords, columns, tables, views, functions, aliases and other elements of the query retain the same meaning after they
are lower cased. This also applies to queries where the predicate is conditioned on a pattern matching with a string value. For instance,

```sql
SELECT name, type FROM DBObjects WHERE name LIKE '%Constants\%
OR name LIKE '%Defs\%' ORDER BY name
```

produces the same result as

```sql
select name, type from dbobjects where name like '%constants\%
or name like '%defs\%' order by name
```

Due to the relaxed specifications of the language, the same token may be typed in several variations that differ only in casing and this happens quite often, in fact. In order to disambiguate these different forms and normalize the vocabulary, we lowercase every word in the dataset, with the exception of SQL keywords, which are all upper-cased for readability and for error correction. Some queries are corrupted during preprocessing causing whitespace omissions which merge words that were originally separate. Upper-casing all SQL keywords simplifies the detection and resolution of such corruptions because it makes it possible to distinguish concatenations with keywords from valid tokens. For instance, if the keyword `ON` was merged with a preceding word, a regex pattern such as `^[a-zA-Z]ON ` would highlight this as a corruption while the word "description", which is a valid column, would be ignored.

Case folding further reduces our vocabulary size from 9481 to 9186.

5.4.5 Real-valued Tokens

Due to 'bot' queries, simply filtering by frequency is bound to give a skewed representation of the query set. Take for instance, the truncated query below. This same query pattern is repeated several thousand times in our dataset with each occurrence varying from the last one by at most 2 tokens as in the trace below:

```sql
...fGetNearbyObjEq(198.106,61.705,22.522) WHERE h.z>0.20699999 and h.z<0.25700000;
...fGetNearbyObjEq(211.078,58.826,21.232) WHERE h.z>0.22599998 and h.z<0.27599999;
```
...fGetNearbyObjEq(235.790,49.928,16.142) WHERE h.z>0.34799999 and h.z<0.39800000;
...fGetNearbyObjEq(157.811,60.596,21.069) WHERE h.z>0.22863001 and h.z<0.27863002;
...fGetNearbyObjEq(186.963,63.384,32.741) WHERE h.z>0.12051000 and h.z<0.17051001;
...fGetNearbyObjEq(155.836,59.810,25.372) WHERE h.z>0.17388999 and h.z<0.22389001;
...fGetNearbyObjEq(177.705,62.330,16.971) WHERE h.z>0.32031000 and h.z<0.37031001;
...fGetNearbyObjEq(186.521,62.540,28.676) WHERE h.z>0.14576000 and h.z<0.19576001;
...fGetNearbyObjEq(154.678,58.975,22.822) WHERE h.z>0.20299999 and h.z<0.25299999;
...fGetNearbyObjEq(176.607,62.024,16.336) WHERE h.z>0.34112999 and h.z<0.39113000;

It’s highly unlikely that each one of these queries was typed out by a human. This trace was most likely generated by an automated process doing some sort of range search over the the photoz table. Queries such as these tend to dominate the query logs in terms of frequency and are less likely to be useful when training a model whose goal is to predict completions to human-written queries.

Another common of a bot-like query is of the form:

```
SELECT * FROM specobjall WHERE specobjid = 2000831126674892800
```

where the prefix with all tokens up to and including the ‘=’ operator appears 19775 times in the logs. This exact query appears 26 times so the value 2000831126674892800 would not be filtered out of the vocabulary and replaced with the <UNK> token with the frequency cut-off of 20.

To minimize the skew effect from such queries on the trained model, it would be advantageous to distinguish automated queries from human-issued queries. This is an involved task which has been explored in previous studies [24]

This research takes a simplified approach: Integer values with 3 or more digits, float values with 3 or more decimal places and hexadecimal values matched by the regex pattern, 0x[0-9a-z]{16,}, are replaced with the <VAL> token if found in a predicate condition of the form expression OP value, where OP is one of the comparison operators {<, <=, = , >, >=, >}, (excluding the LIKE operator) and expression is a column name, or if the value appears in a BETWEEN construct (as
in BETWEEN 261.042 AND 261.143). If such values are found within a function (as arguments), they are replaced with the <ARG> token.

Below is a sample query before and after applying this preprocessing step:

```
SELECT p.objID, psfMagErr_u, psfMagErr_g,
    psfMagErr_r, psfMagErr_i, psfMagErr_z
FROM PhotoObjAll p, photoz h,
dbo.fGetNearbyObjEq(245.36224,42.761299,29.060511) AS x
WHERE x.objID=p.objID
    and p.objID=h.objID
    and h.z>0.14299999
    and h.z<0.19300000
```

Figure 5-9: In addition to the aforementioned preprocessing steps, .

This preprocessing step reduces the size of our vocabulary from 9186 to 3553. This is a significant speedup and allows us to train on 2.6 times as many samples as described in the next section.

### 5.4.6 Data cleaning

The final preprocessing step is data cleaning. This involves detecting and removing (or correcting) invalid queries from our dataset. After applying the above preprocessing steps, some queries were found to contain syntax errors when inspecting the logs.
Unbalanced Parentheses

One simple test for syntax correctness is balanced parentheses. All queries containing unbalanced parentheses are eliminated from the dataset. Such queries were also found to have a relatively high \(<\text{UNK}\>\) token composition (that is, the percentage of a query’s tokens that are \(<\text{UNK}\>\) tokens.)

This step eliminates 260 queries and reduces our vocabulary size from 3553 to 3198. The average length of queries that were eliminated due to unbalanced parentheses is 557.9 tokens which is significantly longer than the average query. This suggests that long queries may have been subject to truncation during the query log collection phase. Furthermore, the contribution of these queries to the vocabulary was highly disproportionate, given that there were 260 queries (0.004% of the dataset) contributing about 10% of the vocabulary words.

Repeating \(<\text{UNK}\>\) Sequences

After replacing infrequent tokens with the \(<\text{UNK}\>\) token, the majority of queries that contained sequences of repeated \(<\text{UNK}\>\)’s were also found to be syntactically incorrect. So another technique that was used to identify invalid queries in the pre-processed dataset was to search for repeated \(<\text{UNK}\>\) tokens that are not separated by a non-\(<\text{UNK}\>\) token such as a comma or parenthesis. Such records are removed from the dataset. One explanation for this pattern is whitespace omission: When tokens that were separated by whitespace are merged, it’s unlikely that their combination will match another an existing vocabulary word, so they typically form a new or infrequent token which is a likely target for \(<\text{UNK}\>\) replacement. Repeating sequences of such tokens would be highly indicative of incorrect syntax.

Percent Composition of \(<\text{UNK}\>\)’s in Query

This next cleaning step was motivated by preliminary training results discussed in the next chapter which indicated a heavy bias towards the \(<\text{UNK}\>\) token in the network. We use a basic univariate anomaly detection technique to find queries with
an abnormally high composition of <UNK> tokens. Percent composition of <UNK> tokens is found by dividing the number of <UNK> tokens in a query by the total number of tokens in the query multiplied by 100%.

Most univariate methods for outlier detection rely on the assumption of a known underlying non-skewed distribution of the data. While the distribution of <UNK> token composition in the set of distinct queries can be approximated with a Poisson distribution (see figure 5.4.6 (b)), the complete set’s distribution is largely skewed and could not be closely fit with any discrete probability distribution as shown in figure 5.4.6 (a). Figure 5.4.6 depicts a box-plot distribution the percent composition of <UNK> tokens by query for the complete query set. The upper adjacent value, typically computed as (Q3 + 1.5 * IQR), where Q3 is the 75th percentile value, and IQR is the inter-quartile range, is 19.3%. In general, values that fall outside of the adjacent value region are deemed potential outliers. The highest observation still inside this region gives us an upper bound on acceptable <UNK> token compositions. Queries in which <UNK> tokens make up more than 19.3% of the query’s tokens (more than 1 in 5) are eliminated. For the distinct set, the upper adjacent value is 15.2%. This is presumably a more accurate threshold for detecting outliers, but to be conservative, 19.3% was chosen since it eliminates fewer queries (924) than the 15.2% threshold which eliminates 47,087.

After the 924 queries are purged, our vocabulary is reduced from 3198 to 3024.

Table 5.5 compares summary statistics of the original dataset to those of the preprocessed dataset. Note the significant reduction in vocabulary size from 1,398,311 to 3024. Only 1551 queries are eliminated.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Raw</th>
<th>Preprocessed</th>
</tr>
</thead>
<tbody>
<tr>
<td># of queries</td>
<td>6,572,559</td>
<td>6,571,008</td>
</tr>
<tr>
<td># of unique queries</td>
<td>989,112</td>
<td>12,243</td>
</tr>
<tr>
<td># of tokens</td>
<td>238,679,251</td>
<td>238,715,168</td>
</tr>
<tr>
<td># of unique tokens</td>
<td>1,398,311</td>
<td>3024</td>
</tr>
<tr>
<td>Average # of tokens per query</td>
<td>23.3</td>
<td>36.3</td>
</tr>
<tr>
<td>Maximum # of tokens per query</td>
<td>1578</td>
<td>1237</td>
</tr>
<tr>
<td>Minimum # of tokens per query</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 5.5: Summary statistics of the SqlLog dataset before and after preprocessing.
Figure 5-10: Histogram showing distribution of percent composition of queries by UNK tokens. The x-axis shows the ‘x’ bins. The y-axis shows the probability that a randomly chosen variable falls within that the given bin. The closest fit Poisson distribution trace is plotted in each graph (dashed line). The data from the complete dataset (right) does not fit the Poisson distribution with $\lambda = 7.7$ where $\lambda$ is the average % composition in the complete set suggesting that this data is skewed. On the other hand, the Poisson distribution with $\lambda = 6.3$ where $\lambda$ is the average % composition in the distinct query set, is a good fit for this data.

5.4.7 Query validation

The above cleaning steps remove queries that would introduce issues in the training phase. However, there are still a number of records in our training set that are syntactically incorrect but are harder to spot. We describe below how such queries are identified. Note that this step was not performed on the training set used in this research due to time limitations but will be applied in future studies.

Parsing SQL is a notably difficult task and The first approach that was considered for query validation was to query every log record against the public SDSS database and check for errors. This approach, however, has several shortcomings: It is computationally expensive since it would require each query to be issued to the database and its response returned before validating the query; It would be difficult to keep
Figure 5-11: Box-plot distribution of percent composition of unknown tokens by query calculated over the complete dataset (a) and over the distinct dataset (b).

track of the validation progress since the query response times and result size vary by query; and although this is a minor concern, this approach would potentially skew the Query Log data by introducing duplicate records.

The chosen approach uses an offline SQL syntax checker to validate queries without connecting to a database. This functionality is provided by the "General SQL Parser" tool [25] which supports several SQL dialects including SQL Server. The syntax checker takes a query string as its input and returns a binary value to indicate whether a syntax error was detected (1) or not (0). As far as I can tell, this tool provides no information about the syntax error such as the offending token or the starting position of the illegal syntax in the query string.

While the query validation step can be applied before preprocessing and potential allow some preprocessing steps to be skipped, preference is given to applying it as a last step since preprocessing introduces invalid queries of its own. Since the syntax checker does not recognize the helper tokens (<SOQ>, <EOQ>, <UNK>...) that
are added during preprocessing, these tokens are substituted with placeholder values that preserve query validity.

The syntax checker has a high validation throughput of over 12,000 queries/second which is a significant performance improvement over the database-querying approach. 107,677 queries in our dataset are marked as invalid, shrinking the vocabulary of the valid queries to 2856 (from 3024). One such query is shown below (post-preprocessing):

```
<SOQ> SELECT TOP 1000 objid , modelmag_u , modelmag_g , modelmag_r , modelmag_i , modelmag_z , z FROM SpecPhoto WHERE ( specclass = 3 OR specclass = 4 ) AND zconf > <VAL> modelmag_i >= 20 AND modelmag_z - modelmag_i <= <VAL> modelmag_u >= 20 <EOQ>
```

At first glance, it’s not obvious that this query is invalid but upon closer inspection, one finds that there is a missing token towards the end of the query that should have been between "<VAL>" and "modelmag_u". "AND" and "OR" are possible candidates. This type of syntax error would not have been found by any of the above preprocessing steps.

On the other hand, this validator is prone to false positive errors: Take, for instance the 2 queries below in which the period is separated from the neighboring tokens by whitespace

```
<SOQ> SELECT <W> FROM dbo . fdoccolumns ( 'specobj' ) <EOQ>
```

```
<SOQ> SELECT field.fieldid FROM dbo . fgetobjfromrect ( <ARG> , <ARG> , <ARG> ) p , field g WHERE g . run = p . run AND g . rerun = p . rerun AND g . field = p . field AND g . camcol = p . camcol <EOQ>
```

This query form is allowed by the SDSS SQL grammar but not by the validator. A trivial solution to this is to collapse the whitespace between the period and the neighboring tokens to merge such patterns into a single token.

It’s interesting to inspect the invalid query set since several patterns are repeated in these statements and could provide a basis for additional preprocessing steps. Some of the techniques that were noted down include:

- Purging queries with unbalanced quotation marks;
• Queries containing the BY keyword that is not preceded by ORDER or GROUP contain syntax errors and can be corrected by pre-pending it with ORDER or GROUP as appropriate;

• Queries in which the FROM keyword is missing but not the SELECT keyword should be checked for the pattern "<UNK> tablename". If this pattern exists, the <UNK> token was formed from merging an element in the projection with the FROM keyword. This can be resolved by changing the <UNK> token to FROM, the preceding token may be ignored;

5.4.8 Padding

Queries vary in length (refer to Figure 5.4.8 for a box-plot distribution of query length). The penultimate preparation step is to pad the training data accordingly for each model.

Choice of padding token

The average query is approximately 36.3 tokens long. Adding a pad token introduces significant storage overhead to the data. The pad token is chosen to be a single character "?", that doesn’t exist in our Vocabulary. This approach was found to be 4GB cheaper than using "<PAD>" as the pad token when padding data for the sequence-to-sequence model.

Sequence-to-sequence

Since each word-vector in our training set needs to be of the same length, we need pad all statements to the same number of tokens. We eliminate queries that have 100 or more tokens and append padding tokens to the remaining queries to make them all 100 tokens long. Over 98.4% of the queries in our dataset have fewer than 100 tokens.
Sequence-to-token

The input vector to this model is 20 tokens long, but around half the queries in our training set are fewer than 20 tokens long so taking the same approach as in the previous model would eliminate too many samples. Instead, we pre-pend each query with exactly 20 pad tokens. We’d initially considered padding only the queries that have fewer than 20 tokens. Even though the model would learn that pad tokens carry no information, this approach might have affected the way the model treats starting sequences.

5.4.9 One-hot encoding

Machine learning algorithms typically deal with real-valued/numerical data. However, textual data is discrete, so our final preprocessing step is to transform our input into equivalent numeral values before presenting it to the neural network. A common way
of doing this is via ‘one-hot’ encoding. Every word from the Vocabulary is associated
with a vector whose length is equal to the size of our vocabulary. All entries are set
to 0, except for one entry that corresponds to the index of the word in the vocabulary
(sorted by decreasing order of word frequency).

So for instance, <SOQ>, <EOQ>, SELECT are transformed as follows:

\[
\begin{bmatrix}
1 \\
0 \\
0 \\
\vdots \\
0 \\
0
\end{bmatrix} \rightarrow \begin{bmatrix}
0 \\
1 \\
0 \\
\vdots \\
0 \\
0
\end{bmatrix} \rightarrow \begin{bmatrix}
0 \\
0 \\
1 \\
\vdots \\
0 \\
0
\end{bmatrix}
\]

Figure 5-13: One-hot encoding example
Chapter 6

Experiments

6.1 Training details

Both RNN models are trained on Linux virtual machine instances on Compute Engine using the Google Cloud Platform. Each machine is provisioned with 8 virtual CPUs and 52 GB of RAM. We train both networks for 10 epochs and save checkpoints of the updated model after each epoch. For computational reasons, the networks, data is processed in batches of size $b$. This means that the network is passed $b$ input vectors at each step, but each CPU is sent $\frac{b}{8}$ inputs for parallel computation.

6.1.1 Sequence-to-sequence

This model is trained on 2,000,000 samples (that is, 2M input vectors) with a batch size of 2048 (so each CPU processes 256 input vectors at a time). The average training time per batch is 86.8s and we process 977 batches in each epoch. Each epoch takes about 23.5 hours so training took about 10 days with this implementation.

6.1.2 Sequence-to-token

We train this model on 58,232 samples (which corresponds to 2,148,531 input vectors) with a batch size of 1024 (so each CPU processes 128 input vectors at a time). The average training time per batch is 49.5s and we process close to 2100 batches in each
epoch. Each epoch takes about 28.8 hours so training took a little over 12 days with this implementation.

### 6.1.3 N-gram model

We train 2, 3 and 5-gram models on the complete data set but skip the real-value replacement step during preprocessing. Table [6.1] shows the training time, model size (in MB) and vocabulary size for each model. We use a sparse matrix representation for the bigram to minimize storage overhead, but use regular python dictionaries for the rest. The model and vocabulary objects are serialized with the python pickle module [26] and compiled into binary format. The Lempel-Ziv-Markov chain (LZMA) compression algorithm [27] is then used to compress the binary file before storing it. LZMA is chosen as this was shown to give the highest compression ratio among the other tested algorithms for these pickle files. Note that we don’t one-hot encode the input when passing it to the n-gram model. Instead, each word is encoded as an integer.

<table>
<thead>
<tr>
<th>n</th>
<th>Training time (s)</th>
<th>Model size (MB)</th>
<th>Vocabulary size</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>312</td>
<td>0.78</td>
<td>8655</td>
</tr>
<tr>
<td>3</td>
<td>405</td>
<td>1.31</td>
<td>27156</td>
</tr>
<tr>
<td>5</td>
<td>505</td>
<td>3.15</td>
<td>74277</td>
</tr>
</tbody>
</table>

Table 6.1: N-gram training details. Training time includes encoding phase. The vocabulary size is the number of unique sequences with n words.

### 6.2 Experimental Results

#### 6.2.1 Training loss and over-fitting

Recall that we use categorical cross-entropy error in the sequence-to-sequence model, and root-mean-squared error for the sequence-to-token model. To evaluate the quality of the neural network, we plot the training loss over the training epochs for both networks in figure [6.2.1] LSTM networks generally have the problem of over-fitting, which is indicated by the loss flattening out and starting to increase. The sequence-
The to-sequence model starts with a relatively high cross-entropy error in the first epoch of 7.48, which quickly drops to in 1.36 the first epoch, and continues decreasing over the next few epochs until it flattens out. At epoch 7, there is a small increase in loss, which suggests that the model could be over-fitting, but this starts to drop again after epoch 8. The sequence-to-token model, however, has only one significant drop in its training loss after the first epoch and quickly flattens out at around 0.3. No increase in loss was observed.

### 6.2.2 Memory and Compute Resources

The memory and compute resources that are required to train these models warrant some discussion.

Recall that our VM instances are provisioned with 8 virtual CPUs and 52 GB
of physical memory (that is, 6.50GB of RAM per virtual CPU). Each virtual CPU is implemented as a single hardware hyper-thread on a 2.2 GHz Intel Xeon E5 v4. [28] By default, Compute Engine instances do not have any swap space so 200GB of persistent disk is allocated to be used for swapping. Each machine has a total of approximately 250GB of virtual memory available before training. A significant number of in-memory pages used by the training process during its startup phase is only used for initialization when building the model and then never used again. Adding swap spaces allows the system to swap out such pages to disk and free a significant amount of memory for other processes. On the other hand, using swap space significantly hinders performance since it is much more efficient to read from and write to memory than persistent disk. The maximum sustained read/write throughput for standard persistent disk on these Compute Engine instances is 1.2 MB/s while the theoretical maximum memory bandwidth is over 20 GB/s. [28] To enhance read/write performance, we configured the Linux kernel’s swappiness parameter to 1 (out of 100) so that swap is only used when physical memory (RAM) is full.

Neural networks typically require huge amounts of memory during training in order to store multi-dimensional input data, weights and activations as the input propagates through the network.

During training, the sequence-to-sequence RNN model utilizes 11.3 GB of Virtual Memory on average. That is, about 22% of physical Memory. Almost none of the swap space is used. The sequence-to-token model, on the other hand, utilizes 130 GB of Virtual Memory on average. That is, close to 100% of physical Memory and 80% of swap space. What this means is that this model takes significantly longer to train since the training process spends most of its time waiting for data to be read from swap memory. CPU usage is measured at 5 second intervals while the training process runs and the results are plotted in figure 6-2.

It’s clear from the bursty nature of this trace that the CPUs are idle for the majority of the execution time in the sequence-to-token model (dotted line). Compare this with the measured CPU usage when training the sequence-to-sequence model (solid line). It’s clear that the computational resources are under-used when training the
sequence-to-token model, so memory optimizations that trade compute resources for memory would be very advantageous. One such optimization introduced by Gruslys, Audrunas et al. [29] is to discard and re-compute retained activations during the backwards pass when they are needed again. This has not been implemented in this research but would be an interesting approach to consider in future implementations. Another memory optimization used was forcing garbage collection on a regular basis to ensure that dropped python objects are collected to free up memory. Since the compute resources are under-used the performance hit that this introduces is negligible.

In summary, the sequence-to-token model is slow to train and requires an exorbitant amount of memory to train on only 1% of the dataset. The sequence-to-sequence model, on the other hand, is reasonably memory-efficient and trains with 30% of the dataset.

Figure 6-2: % CPU usage measured during training
6.3 Query Synthesis

6.3.1 Priming

We generate new sequences by priming the models with some starting sequences. The preprocessing steps that were applied to the dataset in preparation for each network are re-applied to the samples before they are fed into the network. For the sequence-to-token model, for instance, an input sequence is padded with $20 - n$ pad tokens, where $n$ is the length of the sequence (after adding the $<$SOQ$>$ token).

Word-completions revisited

Before priming the network, we perform a prefix search in the Vocabulary for the last word in the typed query. The Vocabulary is loaded into into a prefix tree (trie) to optimize this search.

- If a complete match exists, we move on to the next step, no completion is needed.

- If one or more partial matches exist, we use the n-gram model with the context as the preceding 4 tokens (5-gram) and with the target as all partial matches and find the top 4 most popular of the partial matches. If the typed query has fewer than 5 tokens, or if there are fewer than 4 matching 5-gram sequences, the (3-gram) is used with just the last 2 tokens, and then the bigram. We stop when we have 4 possible word-completions, in which case we pass the 4 sequences (sorted by popularity) on to the sampling stage, or when all the (5, 3, 2)-grams have been exhausted.

- If no match was found, we treat the word as an out-of-vocabulary word as described below.

Out-of-Vocabulary (OOV) words

When a word in the typed prefix doesn’t have an exact or prefix match in our vocabulary of words, we replace it with the $<$UNK$>$ token before priming the model. When
the reconstruction step in the next section is applied to the generated sequences, the `<UNK>` token is again replaced with the word that was typed. An alternative approach that was considered was to make correction suggestions when this case was encountered. However, since only about 16% of columns in the SDSS schema are every mentioned in the query logs, it would not be safe to assume that OOV words are mistyped.

6.3.2 Sampling

Beam search

We perform a beam search with width 4 to the explore the prediction graph and stop when a maximum depth of 160 (This is the upper adjacent value in the box-plot distribution of query lengths as shown in figure ??) is reached or when we encounter an `<EOQ>` token. Beam search expands the most promising node in a limited set according to some heuristic, which in our case is the probability next-token probability. It is often chosen for use in neural networks because it minimizes memory requirements.

Reconstruction

The sequences generated by the network contain tokens that were added during preprocessing, which need to be replaced for the query to be valid. Pad tokens are truncated from the end of the query for the sequence-to-sequence model, and from the beginning of the query for the sequence-to-token model.

Special token replacements

When an `<UNK>`, `<VAL>` or `<ARG>` token is seen in the generated sequence, and was not part of the priming sequence (this is checked by the token’s index), we use the (5, 3, 2)-gram models to find the most popular replacements. Recall that the n-grams were trained on the dataset before these special token replacements were made, so these will not be returned by the n-gram. Note that after these replacements...
are made, the new sequence that is passed back into the network to generate more sequences will add these special tokens back again since they do not exist in the vocabulary.

6.3.3 Validation

Finally, after reconstruction, the queries are checked for syntax validity using the offline SQL syntax checker [25] described in the previous chapter and invalid queries are eliminated.

6.4 Model Evaluation

6.4.1 Sequence-to-token model

We now present a qualitative analysis of the sequence-to-token model.

Effect of the number of training epochs

We save model weights after each epoch and test the model at different epochs by priming it with the same primer:

<SOQ> SELECT s.ra, s.dec, s.modelmag_r FROM star AS s

Table 6.2 shows sample queries generated from different training epochs.

6.4.2 Effect of using n-grams to assist in generation

We used n-grams models to replace <VAL>, <UNK> and <ARG> tokens as new sequences are generated, with other non-special tokens that were observed with the preceding context. We found that this resulted in more “correct” structure in most cases. For instance, when primed with the sequence:

<SOQ> SELECT COUNT ( <w> ) , MIN

The model at epoch 1 generated the following:

| <SOQ> SELECT COUNT ( <w> ) , MIN , <UNK> , <UNK> , <UNK>, <UNK>, <UNK>, <UNK> , <UNK> , <UNK> , <UNK> , <UNK> , <UNK>, |
Table 6.2: Queries generated from different training epochs. Note that no special keyword replacement is done at this stage

However, the n-gram assisted model at the same epoch generated an entirely different sequence:

Although this is not a valid query either, it more closely resembles normal queries in structure.

In later epochs, however, the generated sequences are quite similar, and only differ by special tokens. With n-grams:
Wihtout n-grams:

\[
\begin{align*}
\text{SELECT COUNT} & \left( <W> \right), \text{MIN} \left( z \right), \text{AVG} \left( z \right) \\
\text{FROM specobj} & \text{WHERE} \left( \text{specclass} = 3 \text{ OR specclass} = 4 \right) \\
& \text{AND } z > 0 \text{ AND zconf} > .5
\end{align*}
\]

6.4.3 Sequence-to-sequence model

As of this writing, the above post-processing steps have not been implemented in the sequence-to-sequence model, so an in-depth evaluation was not possible. However, we list below a few samples generated with the primer:

\[
\begin{align*}
\text{SELECT TOP} & \left( 1 \right) \text{CAST} \left( p.objid \text{ AS varchar } \left( 20 \right) \right) \text{ AS objid,} \\
p.run, & \text{p.rerun, p.camcol, p.field, p.obj, p.type, p.ra,} \\
p.dec, & \text{p.u, p.g, p.r, p.i, p.z, p.err_u, p.err_g, p.err_r,} \\
p.err_i, & \text{p.err_z FROM fgetnearbyobjeq } \left( 195, 2.5, 0.5 \right) \text{ n,} \\
\text{photoprimary} & \text{p WHERE n.objid} = p.objid
\end{align*}
\]

Note that these are among the top 5 most common queries in the dataset in table 5.3

6.5 Proposed integration with the command-line tool

We have implemented and experimented with the above architecture, and have found that the models are able to generate valid queries based on the current query input. However, there is of course still much room for future improvements and re-
search remains to be done to evaluate how well this architecture performs as an auto-completion tool.

One simple approach to integration with the command-line tool is to have model-based predictions turned off in the early stages of query composition and only enabled at a later stage to predict the remainder of the query. An evaluation of this approach would not be complete without a user study. Such a study could focus on how well a user feels that the model predicted their intended query and comparing how long the query composition process takes compared to a model without these capabilities.
Chapter 7

Conclusion

7.1 Contribution

In this thesis, we have explored the problem of composing non-trivial queries. We have implemented our solution, SqlAct, and the experimental results show that our method achieves high performance and enhanced usability, at the cost of some extra storage cost which is configurable. This tool is database independent and can be seamlessly incorporated in a wide variety of database configurations and the LSTM language model can be trained on different historical queries logs. By providing current word-level and statement-level predictions, we believe SqlAct is a very handy tool that simplifies query composition, improves overall typing speed, accuracy and developer productivity.

7.2 Next steps

The next step of this research is to conduct a user study, to evaluate the recommendation quality. The experiment will collect usage statistics such as the rank of the picked suggestion, and the interface response time. The experiment will also ask respondents to rate the quality of the suggestions that were provided and how closely these matched their intended queries. This thesis only explored a small set of possible training parameter combinations. There almost certainly is a more effi-
cient configurations than what was used here, so several different configurations still remain to be tested, after applying additional preprocessing steps that were outlined in 5.4.7 but not implemented. A common metric used to evaluate the performance of the neural models is **accuracy**. This is the average number of predictions where the highest probability next word matches the actual next word. This was not computed in this research but can provide some interesting insight on the models. Finally, a user study is planned to help evaluate the proposed integration of the LSTM model with the command-line tool.
Bibliography


