Accelerating Relational Keyword Queries With Embedded Neural Networks

Limin Ma
Faculty of Science
Ontario Tech University
Oshawa, Canada
limin.ma@ontariotechu.net

Ken Q. Pu
Faculty of Science
Ontario Tech University
Oshawa, Canada
ken.pu@ontariotechu.net

Ying Zhu
Faculty of Business and IT
Ontario Tech University
Oshawa, Canada
ying.zhu@ontariotechu.net

***Abstract*—Relational keyword queries have proven to be highly effective for information retrieval. The challenge of evaluating keyword queries for relational databases is the performance bottleneck of fuzzy string matching when traditional full-text index structures. We propose a solution to overcome performance bottlenecks by incorporating horizontally partitioned full-text indexes. We rely on a neural network router to optimize the index lookup strategy to minimize index miss rate and thus maximize performance. Using textural features of the user queries, the neural network router supports fuzzy string matching. We evaluated different network architectural designs against real-world datasets. Our experiments demostrates that the neural network router can be self-trained and learn how to optimize index access effectively.**

***Index Terms*—keyword query, neural network, index structures**

I. Introduction

II. Partial tuple search

In this section, we formalize keyword queries as *partial tuple search*. Let R be a relational table. Recall that $\mathrm{attr}(R)$ is the attributes of R, and tuples in R are mappings from $\mathrm{attr}(R)$ to $\mathbf{Values}$.

Definition 1 (Labeled values and partial tuple): Let R be a relational table. A labeled value in R is a pair $(l : x)$ where $l \in \mathrm{attr}(R) \cup \{?\}$ and $x \in \mathbf{Values}$. A partial tuple $\vec{x}$ is a set of labeled values: $\vec{x} = \{(l_i, x_i) : i \in I\}$. We define the attributes of a partial tuple $\vec{x}$ as the labels of the partial tuple:

$$\mathrm{attr}(\vec{x}) = \{l_i\}_{i \in I}$$

. A partial tuple is considered ***complete*** if $\mathrm{attr}(\vec{x}) = \mathrm{attr}(R)$.

We will write $\vec{x}[l_i]$ to denote the corresponding value x_i.

Note that a partial tuple is a set of values and their respective attribute names from a relational table. However, we allow a special symbol "?" to be in place of the attribute name. The special attribute "?" indicates that the attribute name is unspecified (or unknown). The following example illustrates two partial tuples. The second partial tuple has a wild card "?" as its attribute name:

Example 1:

$$\begin{aligned} \vec{s}_1 &= \{\mathrm{name} : \mathrm{Einstein}\} \\ \vec{s}_2 &= \{\mathrm{name} : \mathrm{Einstein},\ ? : \mathrm{Professor}\} \end{aligned}$$

A keyword query is a partial tuple: $\vec{Q} = \{(l_i, q_i) : i \in I_Q\}$. The values q_i are keyword queries. We want to find a complete tuple $\vec{r} \in R$ that matches $\vec{Q}$ optimally. This requires us to define how to compare the partial tuple $\vec{Q}$ with a complete tuple $\vec{r}$. The guiding principle of comparing the two tuples is to match labeled values from $\vec{Q}$ with those from $\vec{r}$ according to the following:

- Match the labels if they are not the wild card "?".
- Match the values using a fuzzy string matching score.
- Optimize the sum of similarities between labeled values from $\vec{Q}$ and those from $\vec{r}$.

Definition 2 (Partial tuple search): Let $R_1, R_2, R_3, \ldots, R_K$ be K relations. Given a user query that is a partial tuple:

$$\vec{Q} = \{(l_i, q_i) : i \in I_Q\}$$

where $l_i \in attr(R) \cup \{?\}$ and q_i are keyword queries, we want to find a complete tuple $\vec{r} \in R_i,\ i \in \{1, 2, \ldots, K\}$, that maximizes the similarity score between the partial tuple $\vec{Q}$ and full relational query $\vec{r}$.

We use a neural network to optimize the query processing pipeline. Detailed description can be found in Section **??**. We define the neural network classifier as:

III. Partial tuple search using full-text search

Traditional full-text indexes support keyword queries over document collections using an inverted index data structure. First the full relational tuples are encoded as *documents* by some tokenizer. The tokenizer breaks the string values of partial tuples $\{q_i\}$ and full tuple $r \in R$ to *tokens*. It's the comparison between $\mathbf{tokens}(q_i)$ and $\mathbf{tokens}(r)$ that determines their similarity. To support approximate string matching, the standard approach [14], [16] is to break down strings into their n-grams. An example of a full-text encoding of a tuple is shown in Figure **??**.

IV. Fuzzy keyword search with partitioned indexes

Traditionally, a single full-text index is built on the tuples from all the relations. With the relations $\{R_1, R_2, \ldots, R_n\}$, the

$$\left[\begin{array}{rcl} \text{Name} & \mapsto & \text{\_\_J \_Ja Jac ack ck\_ k\_\_} \\ \text{Address} & \mapsto & \text{\_\_1 \_10 100 00\_ 0\_\_ \_\_S \_Si Sim imc mco coe oe\_ e\_\_ \_\_S \_St Str tre ree eet et\_ t\_\_} \\ \text{fulltext} & \mapsto & \text{\_\_J \_Ja Jac ack ck\_ k\_\_ \_\_1 \_10 100 00\_ 0\_\_ \_\_S \_Si Sim imc mco coe oe\_ e\_\_ \_\_S} \\ & & \text{\_St Str tre ree eet et\_ t\_\_} \end{array}\right]$$

Fig. 1: Full text encoding of a relational tuple with 3-gram tokenization

aggregated index is the full-text index built from the union of the tuples:

$$\mathbf{Index}_{\text{agg}} = \textsc{InvertedList}\left(\bigcup_{i=1}^{n} R_i\right)$$

Due to the inverted list architecture, the performance bottleneck comes from collisions of multiple documents containing the common tokens. With the total number of document is given by $\sum_{i=1}^{n} |R_i|$, the average number collision of the inverted lists in the aggregate index is estimated as:

$$\mathbf{Collision}_{\text{agg}} = \frac{\sum_{i=1}^{n} |R_i|}{|\text{vocab}|}$$

where $|\text{vocab}|$ is the size of the vocabulary of the distinct tokens. The query evaluate performance is known to be $\mathcal{O}(\mathbf{Collision})$.

When supporting fuzzy string matching, the tokenizer performs 3-gram tokenization, and thus $\text{vocab} = \text{all 3-grams} \in \mathcal{O}(1)$. Namely, with a fixed alphabet, the size of the vocabular is roughly a constant. This means the query evaluation time complexity is given by:

$$\mathcal{O}(\text{Lookup}(\mathbf{Index}_{\text{agg}}, q)) = \mathcal{O}\left(\sum_{i=1}^{n} |R_i|\right)$$

The query performance, thus, would degrade with increasing number of tuples, and increasing number of relations.

We propose the following strategies to overcome the performance bottleneck of the aggregated index.

- We partition $\mathbf{Index}_{\text{agg}}$ into multiple indexes $\{\mathbf{Index}_j : 1 \leq j \leq m\}$
- Given a query q, we scan through the partitioned indexes and evaluate $\text{Lookup}(\mathbf{Index}_j, q)$.

The advantages of the partitioned approach are:

- Each index access $\text{Lookup}(\mathbf{Index}_j, q))$ is more *efficient* due to the reduced number of collisions in the inverted index.
- We have the opportunity of *optimizing* the sequence of scanning the index partitions. If we can determine the likelihood of that the query q has high similarity with tuples in $\mathbf{Index}_j$, we want to access the partition j before the other partitions.

In Algorithm **??**, it evaluates the probability of if q is relevant to the tuples in the ith partition. The probability is computed as a binary classifier

$$\mathbf{classifier}(\mathbf{Index}, \mathbf{Query})$$

Algorithm 1 Accelerated index lookup

Require: q: Query, $partitions$: List of Partitions
1: $n \leftarrow$ length of $partitions$
2: p: List of float
3: **for** $i \in [0, n-1]$ **do**
4: $\quad p[i] \leftarrow \mathbf{classifier}(q, partition[i])$
5: **end for**
6: **for all** i in sorted(range(n), key=λk: $p[k]$, desc) **do**
7: $\quad results \leftarrow$ Lookup($partitions[i]$, q)
8: $\quad$ yield $results$
9: **end for**

V. Design and training of classifiers

Each index partition has an accompanying binary classifier $\mathbf{Classifier}_i : \mathbf{Query} \rightarrow [0, 1]$. We use a neural network to learn the classification function. Due to the application scenario, the design constraints are:

- The neural network must be compact in size so that they can be embedded in the database runtime system. Our goal is that the neural network is a small memory addition to the keyword query evaluation engine.
- The inference speed of the neural network incurs a minimal overhead. This means we prefer shallow networks over deeper architectures.

A. Vectorization of tokens and queries

Token representation of queries: given a query $\vec{q} = \{(l_i, x_i) : 1 \leq i \leq m\}$, we generate the text representation of the query by simply concatenating the text representation of labels and the tokenized query values.

$$\begin{aligned} \mathbf{tokens}(\vec{q}) \quad = \quad & \{l_1\} \oplus \mathbf{tokenize}(x_1) \\ & \oplus \{l_2\} \oplus \mathbf{tokenize}(x_2) \\ & \oplus \ldots \{l_m\} \cup \mathbf{tokenize}(x_m) \end{aligned}$$

where $\oplus$ is sequence concatenation.

Integer encoding of queries: next we encode $\mathbf{tokens}(\vec{q})$ using a universal vocabulary $\mathbf{vocab}$. The vocabulary consists of all known tokens, and their respective ordinal integer code. This vocabulary will be built using the existing relational tuples. The construction of the vocabulary is described in subsequent sections. The vocabulary is described as a function from tokens to integers:

$$\mathbf{vocab} : \mathbf{Token} \rightarrow \mathbb{N}$$

The integer sequence of a query is given by:

$$\mathbf{sequence}(\vec{q}) = \mathbf{vocab} \circ \mathbf{tokens}(\vec{q}) \in \mathbb{N}^*$$

Embedding vectors of queries: Using a standard embedding layer (Section **??**), embed the integer sequence representation of the query to a sequence of latent vectors.

$$\mathbf{vector}(\vec{q}) = \mathbf{Emb}(\mathbf{sequence}(\vec{q})) \in \mathbb{R}^{|q| \times d}$$

At this point, we have many options in mapping the vector sequence to the probability distribution in $\mathbb{R}^n$.

For the remainder of this chapter, we denote the vector sequence representation of $\vec{q}$ as:

$$\vec{x} = \mathbf{sequence}(\vec{q}) = (x_1, x_2, \ldots, x_{|q|})$$

where each $x_i \in \mathbb{R}^d$ is the embedding vector of the i-th token in a d-dimensional latent space.

B. Neural network architectures for query classification

MLP based classification.

Given the input of vectorized query representation $\vec{x}$, we first flatten it using global average over the entire sequence length. Then, we process it with a MLP with softmax activation function. The MLP must have n output neurons.

$$\begin{array}{lll} & \vec{x} & (|q|, d) \\ \rightarrow & \left(\frac{\sum_{i=1}^{|q|} x_i}{|q|}\right) & (d) \\ \rightarrow & \mathbf{MLP}(\text{output-dim} = n) & (n) \\ \rightarrow & \mathbf{softmax}(\cdot) & (n) \end{array}$$

Recurrent network architecture with LSTM cells.

Since recurrent neural networks (RNN) are specifically designed to process sequence inputs, we can utilize a RNN to map the input $\vec{x}$ to a flattened state vector, which can then be processed by a MLP to compute the output probability distribution.

The advantage of RNN is that it can learn inter-token dependencies in the query at the cost of a bigger model size and higher training cost.

$$\begin{array}{lll} & \vec{x} & (|q|, d) \\ \rightarrow & \mathbf{LSTM}(\text{output-state=True}) & (d) \\ \rightarrow & \mathbf{MLP}(\text{output-dim} = n) & (n) \\ \rightarrow & \mathbf{softmax}(\cdot) & (n) \end{array}$$

1D convolution architecture.

Another standard technique to learn sequential features is to use 1D convolution. The 1D convolution layer (Section **??**) produces a sequence of *feature* vectors that capture the short-range token dependencies within the convolution window size. We then use global averaging to flatten the convolution features for further processing using a MLP.

$$\begin{array}{lll} & \vec{x} & (|q|, d) \\ \rightarrow & \mathbf{Conv1D} & (|q|, d) \\ \rightarrow & \mathbf{Global\ Average} & (d) \\ \rightarrow & \mathbf{MLP}(\text{output-dim} = n) & (n) \\ \rightarrow & \mathbf{softmax}(\cdot) & (n) \end{array}$$

Transformer and MLP Mixer

Transformers have shown to be a superior architecture for many tasks in the domain of MLP and sequence learning. The exact architecture of a transformer block is shown in the implementation section (See Figure **??**). Due to the nature of our problem, we chose to use only a single transformer block so that the model remains small enough to be embedded in the query processing pipeline.

A more recent MLP based architecture is MLP mixer which is a concatenation of two MLP layers separated by a matrix transpose operation. Details of the MLP mixer are shown in Figure **??**.

C. Unsupervised training of neural network classifiers

The classifier needs to be trained with data of the following form:

$$(\vec{q}, i)$$

where $\vec{q}$ is a sample query, and i is the relation R_i that contains the best matching tuple of $\vec{q}$.

The training data is generated directly from the relational tuples from the database. For each complete tuple $\vec{r} = \{(l_i, x_i) : i \in I\}$, we formed a query $\vec{q_r}$ by random sampling from the labeled values while masking the labels.

$$\vec{q_r} = \{(?, x_i) : i \in \mathbf{sample}(I)\}$$

Thus, given a database with n relations $R_1, R_2, \ldots, R_n$:

$$\mathbf{train} = \bigcup_{i=1}^{n} \{(\vec{q_r}, i) : r \in R_i\}$$

VI. Overall query processing pipeline

To summarize the proposed query processing pipeline for partial tuple search, we have the following stages:

1) A partial tuple is entered as user input.
2) The partial tuple is tokenized and converted to a structured keyword query.
3) The neural network classifier sorts the partitioned indexes based on the probability scores.
4) The partitioned indexes are scanned to find top-k complete tuple candidates from the database.
5) The candidates are ranked by max-matching based similarity scores.

Our pipeline requires that partitioned full-text indexes are built from tuples of each relation in offline mode. In addition, the neural network classifier should be trained using sampled tuples from each relation.

Dataset id	Dataset name
ds01	Labour Force Survey, April 2019 [Canada]
ds02	Crowdsourcing: Impacts of COVID-19 on Canadians' Experiences of Discrimination Public Use Microdata File
ds03	Crowdsourcing: Impacts of COVID-19 on Canadians Public Use Microdata File, [2020]
ds04	Crowdsourcing: Impacts of the COVID-19 on Canadians – Your Mental Health Public Use Microdata File, [2020]
ds05	Crowdsourcing: Impacts of COVID-19 on Canadians' Perception of Safety Public Use Microdata File, [2020]
ds06	Crowdsourcing: Impacts of the COVID-19 on Canadians – Trust in Others Public Use Microdata File, [2020]
ds07	Impacts of the COVID-19 pandemic on postsecondary students (ICPPS) 2020 Crowdsourcing Public use microdata file
ds08	Canadian Income Survey (CIS), 2017
ds09	Canadian Community Health Survey - Annual Component (CCHS) 2017-2018
ds10	Canadian Housing Survey, 2018

TABLE I: Dataset ids and names

Attribute	Description
survyear	Survey year
survmnth	Survey month
lfsstat	Labour force status
prov	Province
age_12	Five-year age group of respondent
educ	Highest education attainment
immig	Immigration
noc_10	2016 NOC (10 categories)
ftptmain	Full- or part-time status at main or only job
hrlyearn	Usual hourly wages, employees only

TABLE III: 10 attributes and their descriptions from the labour force survey dataset "ds01".

Dataset id	No. of attrs.	No. of tuples
ds01	60	100,885
ds02	129	36,662
ds03	47	242,512
ds04	43	45,989
ds05	23	43,631
ds06	47	36,538
ds07	42	101,902
ds08	194	92,286
ds09	1051	113,286
ds10	132	61,750

TABLE IV: Number of attributes and tuples in each dataset.

VII. Experimental Evaluation

In this section, we will describe the evaluation of the proposed solutions. We will verify the effectiveness and limitations of our solutions, and perform a comparative study of different network architectures. For each network architecture, we will present the benefits and drawbacks, and provide our understanding of the explanation of the observations.

A. Datasets

The datasets we have used to evaluate our system is a collection of survey data from Statistics Canada [2], [4], [6], [8], [5], [7], [9], [1], [3], [10], which includes one labour force survey, six COVID-19 surveys, one income survey, one community health survey, and one housing survey. They offer many real-world characteristics that pose as challenges to neural network classifiers. Given the intended application scenarios of our research, we felt that it was important to evaluate our work using real-world datasets. For easy reference in subsequent sections, we give each dataset an id, as shown in Table I.

These 10 datasets contain both numerical and textual data with different numbers of attributes and tuples. For example, the labour force survey contains data of the Canadian labour market. It has total 60 attributes related to the job market, such as employment status, industry, status of working full-time or part-time, hourly wage, etc. Table II shows 5 samples with a subset of 10 attributes. The descriptions of the selected 10 attributes are shown in table III. Table IV shows the number of attributes and tuples in each dataset.

survyear	survmnth	lfsstat	prov	age_12	educ	immig	noc_10	ftptmain	hrlyearn
2019	April	Employed, at work	Ontario	25 to 29 years	Postsecondary certificate or diploma	Non-immigrant	Natural resources, agriculture and related production occupations	Full-time	33.00
2019	April	Not in labour force	British Columbia	70 and over	Bachelors degree	Non-immigrant	Business, finance and administration occupations	Full-time	26.00
2019	April	Employed, at work	British Columbia	45 to 49 years	High school graduate	Non-immigrant	Health occupations	Full-time	31.00
2019	April	Employed, at work	Ontario	20 to 24 years	Postsecondary certificate or diploma	Non-immigrant	Natural resources, agriculture and related production occupations	Full-time	22.00
2019	April	Employed, at work	Ontario	70 and over	Some postsecondary	Non-immigrant, 10 years earlier	Occupations in art, culture, recreation and sport	Part-time	32.00

TABLE II: 5 samples from the labour force survey dataset "ds01", showing only 10 attributes.

Sometimes different datasets share common attributes. For example, the COVID-19 datasets have a common attribute about community size and metropolitan influence zones where the survey correspondents live in. By using multiple partial tuple completion, users can "jump" from one dataset to another. For example, users may find some data related to discrimination from the dataset of impacts of COVID-19 on Canadians' experiences of discrimination. Then they can use the results to search more information about people's perception of safety in the communities with the same sizes in the dataset of impacts of COVID-19 on Canadians' perception of safety. This will help users to gain more insights from the survey results.

VIII. Architecture and Software Stack

In order to thoroughly evaluate the end-to-end search system, we have developed a platform with the following technology stack. We use Lucene [23] as the full-text index. A customized search engine is developed on top of Lucene to support partial tuple search queries. This is necessary for us to have detailed instrumentation to collect performance metrics, and to gain control over the ordering of text index search. We have implemented all neural network based classifiers using TensorFlow [12]. The trained models are used to generate the ordering of Lucene index access. The evaluation of the neural network acceleration is based on the speed-up of query performance between the machine learning generated ordering and other alternatives. In addition, top-k accuracy is also used to evaluate models. The query workloads used in evaluation are generated from the datasets themselves. Figure 2 shows the high-level architectural overview of our implementation. More details about the training datasets and the query workload generation will be presented in subsequent sections.

A. Lucene and a customized search engine

We developed a customized search engine on top of Lucene. It has two main functions. The first is to index datasets to create Lucene indexes. When indexing datasets, each tuple

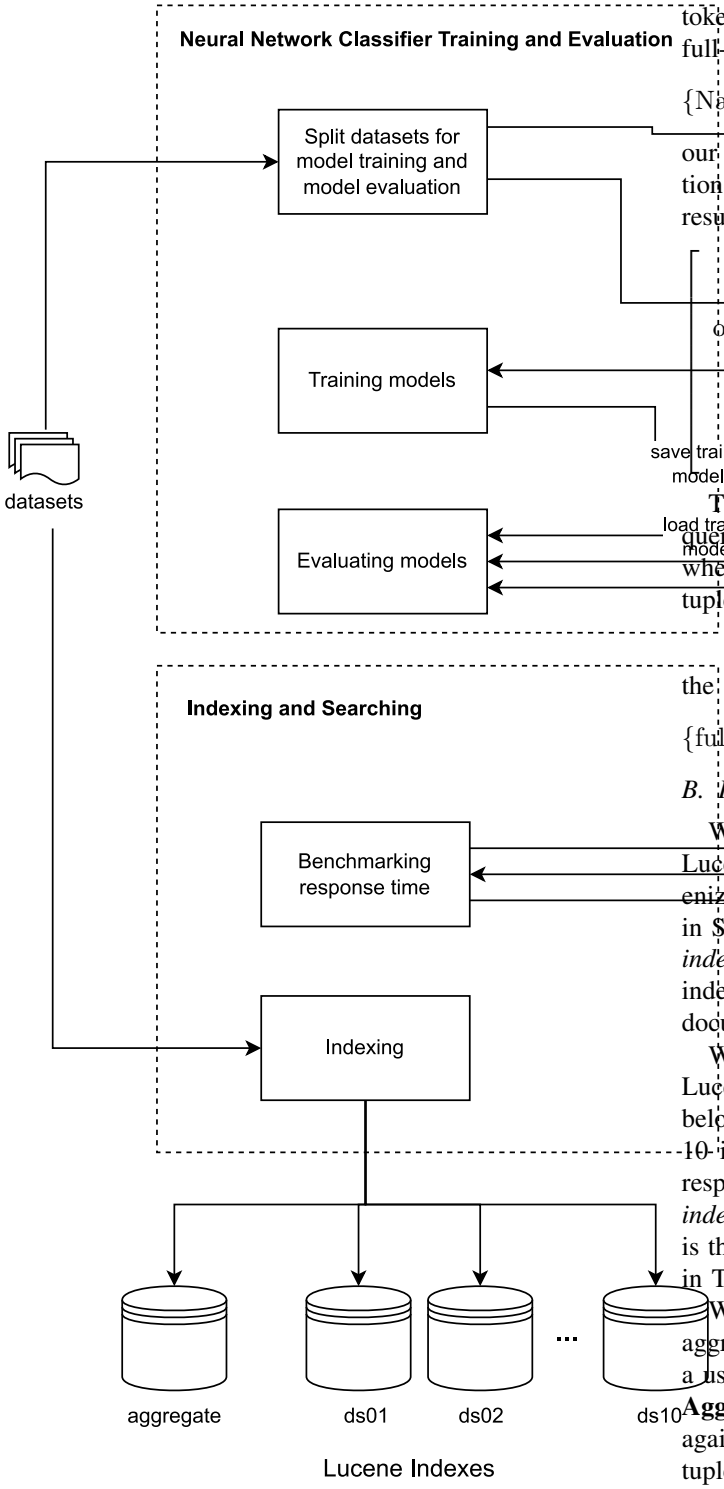


Fig. 2: High-level overview of architecture

is converted to a Lucene document, which contains a set of *(attribute name, attribute value)* pairs. We apply character-based 3-gram tokenization to attribute values. We use the special character "_" for padding. In addition, we add an extra field "fulltext" to the document, which contains all 3-gram tokens of all attribute values. This field is needed to support full-text search. For example, given a tuple t as,

$\{\text{Name} : Jack,\ \text{Gender} : Male,\ \text{Occupation} : \text{Software developer}\}$

our search engine will first apply Lucene's standard tokenization, and then our character-based 3-gram tokenization. The resulting Lucene document will be,

name : __j _ja jac ack ck_ k__
gender : __m _ma mal ale le_ e__
occupation : __s _so sof oft ftw twa war are re_ e__ __d _de
elo lop ope per er_ r__
fulltext : __j _ja jac ack ck_ k__ __m _ma mal ale le_ e__
oft ftw twa war are re_ e__ __d _de dev eve vel e
per er_ r__

The second main function is to support partial tuple search queries. We also use character-based 3-gram tokenization when constructing a search query. For example, given a partial tuple as:

$\{\text{Name} : Jack,\ \text{Gender} : Male\}$

the search query will be:

{fulltext : __j _ja jac ack ck_ k__ __m _ma mal ale le_ e__}

B. Partitioned indexes and an aggregate index

We have indexed the tuples from all datasets into a single Lucene index. The tuples are tokenized using the 3-gram tokenizer in order to support fuzzy string matching as described in Section **??**. This index will be referred to as the *aggregate index* since it contains all the tuples from 10 datasets in a single index. The aggregate index contains approximately 877,000 documents.

We also have an alternative way of indexing the tuples using Lucene. We partition the tuples based on which relation they belong to. Given that we have 10 datasets, we have created 10 individual indexes, each containing the tuples from their respective relation. We refer to the set of indexes as *partitioned indexes*. The number of documents in each partitioned index is the same as the number of tuples in each dataset, as shown in Table IV.

With the two types of indexing schemes (partitioned vs aggregate indexing), we have many alternatives in evaluating a user query.

Aggregate index lookup: we can simply issue the user query against the aggregate index as it contains all the available tuples.

- Pro: This is the traditional approach. It requires the least system design. Only a single thread is needed for each user query.
- Con: As we have indicated previously and will show in our experimental evaluation, this approach suffers from a performance bottleneck resulting from high token hashing collision rate.

Parallel lookup of partitioned indexes: we can issue the query against all individual index in the partitioned index

set. This will perform parallel index lookup using multiple concurrent threads.

- Pro: The result can be found quickly. The index with the *correct* tuple will contain much less documents compared to the aggregate index, and thus will respond with the search result faster. In fact, we argue that this is the optimal performance one can expect.
- Con: The concurrency will impose a higher CPU and disk IO demand as each user query will take up to n threads to process. In high traffic or low resource scenarios, this may be prohibitively expensive.

Predictive sequential lookup of partitioned indexes: we advocate to perform single threaded access of the partitioned indexes. But rather than simple or random sequential scan of the index set, we utilize predictive neural networks to generate an optimized access pattern, as described in Section **??**.

- Pro: This method enjoys the advantage of both single-threaded index lookup and low disk latency due to the potential early hit in a smaller partitioned index. So we argue that the predictive lookup will yield high query performance and low resource usage.
- Con: This method requires a self-supervised neural network to perform the prediction.

IX. Evaluation methodology

We evaluate five model architectures, including Multilayer Perceptron (MLP), Long Short-Term Memory (LSTM), 1D Convolution (Conv1D), Transformer and MLP-Mixer. We also exam how misspelled keywords affect the models' top-5 accuracies.

We use two metrics for model evaluation. The first metric is query processing time. We submit queries to partitioned indexes and the aggregate index, and record the response times as evaluation benchmark. More details are given in the subsection X-A.

The second metric is top-k accuracy. A model's prediction gives us the ordering of Lucene index access. If the Lucene index that a query belongs to is among the top-k of the model's prediction, we consider the prediction accurate. For example, for a query sampled from `ds02`, if the predicted access ordering is

```
idx_ds06, idx_ds10, idx_ds07, idx_ds05, idx_ds02, ...
```

we consider it an accurate top-5 prediction.

In the following subsections, we give more details about our training data, query workloads and experiments. We first describe how training data and query workloads are generated and tokenized in subsections IX-A and IX-B, respectively. After that, we describe our experiments in subsections starting from X-A to X-D.

A. Training data generation and tokenization

The raw datasets are in SAS7BDAT format, which is a binary database format. To make data processing easier for the downstream model training and evaluation, we convert all raw datasets to CSV files. Then we remove some unuseful attributes. For example, the labour force survey dataset contains the attribute *rec_num*, which indicates the record number in the file. This attribute is removed since it is not useful for our model training. After that the preprocessed CSV files are splitted into data for model training and data for model evaluation. Data for model evaluation are used to generate query workloads. More details about the query workloads generation will be presented in subsection IX-B.

	100% attrs.		**75%**	
	word-based	3-gram based	word-based	
vocabulary size	56,482	4,136	47,536	

TABLE V: Vocabulary size of six sets of training datasets.

Our next step is to generate training datasets from data reserved for model training. We randomly sample tuples from each CSV file without replacement. Then we normalize the sampled tuples to lowercase and remove special characters from them, which gives us word-based training data. In addition, we also apply character-based 3-gram tokenization to normalized tuples. We use "_" as the special character for padding in our implementation. This results in 3-gram based training data. For example, given a raw tuple:

```
(2019,April,"Employed, at work",Nova Scotia,Other
45 to 49 years,,Female,Married,Above bachelor''s d
"Single jobholder")
```

the normalized tuple will be:

```
(2019,april,employed at work,nova scotia,other cma
45 to 49 years,female,married,above bachelors degr
single jobholder)
```

and the 3-gram tokenized tuple will be:

```
(__2 _20 201 019 19_ 9__,__a _ap apr pri ril il_ l_
plo loy oye yed ed_ d__ __a _at at_ t__ __w _wo wo
_no nov ova va_ a__ __s _sc sco cot oti tia ia_ a__
her er_ r__ __c _cm cma ma_ a__ __o _or or_ r__ __
__c _cm cma ma_ a__,__4 _45 45_ 5__ __t _to to_ o_
__y _ye yea ear ars rs_ s__,__f _fe fem ema mal al
mar arr rri rie ied ed_ d__,__a _ab abo bov ove ve_
ach che hel elo lor ors rs_ s__ __d _de deg egr gr
_si sin ing ngl gle le_ e__ __j _jo job obh bho ho
r__)
```

Since we evaluate our models using partial tuple queries, a natural question is whether training models using partial tuples will have any impact on models' performance. Therefore, besides training datasets containing full tuples, we also create datasets containing partial tuples with 75% and 50% of attributes. Since each dataset will have an additional 3-gram tokenzed version, we have total six sets of training datasets. Table V shows their vocabulary sizes. In addition, we use dataset names as labels, so there is no need to manually label training tuples.

Workload name	Description
workload A	5 randomly sampled attributes
workload B	3 randomly sampled attributes

TABLE VI: Query workload names and descriptions.

B. Query workload generation and tokenization

Query workloads are sets of partial tuples. We first randomly sample tuples from data reserved for model evaluation. Then we apply the same normalization process to them as we do to training datasets. After that we convert them to partial tuples by randomly sampling some attributes from them. We create different workloads by varying the number of attributes sampled. Table VI shows two query workloads we used for model evaluation.

1) Converting partial tuples to Lucene queries for searching: We use Lucene's Query API to construct search queries from partial tuples. First we apply 3-gram tokenization to a partial tuple to get a list of tokens. Then we convert each token to a search term. After that we use the logical `OR` operator to combine all search terms to form a query. For example, given a partial tuple as the following:

```
no very easy no
```

by adding the character "_" for padding, the constructed query will be:

```
__n OR _no OR no_ OR o__ OR __v OR _ve OR ver OR ery OR ry_ OR y__ OR
__e OR _ea OR eas OR asy OR sy_ OR y__ OR __n OR _no OR no_ OR o__
```

X. Experiments and observations

In this section, we will enumerate over the experiments we conducted to evaluate our methodology. For each experiment, we will focus on the motivation, experimental setup, the observation and their respective conclusions.

A. Performance of aggregate vs optimal index lookup

Description:

Our first experiment tries to establish a baseline for the evaluation of our methodology using the query processing time. We measure three types of performance. The first is the optimal performance, which is achievable by parallel partitioned indexes lookup. The second is the performance of the aggregate index lookup. The last is the performance of sequential scan of all partitioned indexes. This will be the worst sequential scan if the predicted ordering of index access from our neural network classifiers puts the true partitioned index, i.e., the one contains the tuple, as the last one to scan.

To create the performance benchmark, we perform search against partitioned indexes and the aggregate index using the constructed partial tuple queries and log the response times. We repeat the process 10 times and compute the mean response times to be used as benchmark for model evaluation. Table VII and VIII show the mean response times in milliseconds (ms) of the first 20 queries of the workload A and B, respectively. Both of the workloads have 1,000 queries. In both tables, the first 10 columns are the results of partitioned indexes, and the last one is that of the aggregate index. Details about the query workloads are described in subsection IX-B.

	idx_ds01	idx_ds02	idx_ds03	idx_ds04	idx_ds05	idx_ds06	i
0	46.57	18.074	11.766	12.315	11.681	10.991	
1	9.785	95.191	9.112	3.577	5.211	5.076	
2	41.464	30.662	15.134	7.381	9.833	6.114	
3	81.665	71.147	10.45	11.374	13.238	62.585	
4	11.86	12.193	4.124	3.576	2.569	4.466	
5	111.691	35.233	23.712	14.259	37.553	32.163	
6	10.171	15.061	6.72	6.757	5.178	6.366	
7	8.565	17.936	3.222	11.276	3.91	9.101	
8	4.702	2.492	3.492	2.109	1.748	2.76	
9	29.805	25.247	4.121	3.972	3.618	8.858	
10	12.438	28.071	11.567	7.722	8.266	8.296	
11	78.894	199.486	36.34	16.409	59.3	19.544	
12	34.995	17.869	6.531	8.366	4.515	9.189	
13	27.613	291.843	36.575	18.002	16.23	21.494	
14	14.984	19.801	18.374	7.259	9.394	10.142	
15	13.587	7.268	3.283	4.145	2.499	3.895	
16	13.702	95.056	9.045	3.973	5.103	8.488	
17	56.258	33.413	15.297	20.917	77.721	36.136	
18	82.657	47.663	13.094	16.612	20.459	22.733	
19	54.616	101.808	19.846	10.232	17.782	10.651	

TABLE VII: Mean query processing time in milliseconds of the query workload A.

	idx_ds01	idx_ds02	idx_ds03	idx_ds04	idx_ds05	idx_ds06	i
0	98.113	57.467	10.403	37.902	12.051	62.483	
1	14.557	55.388	28.667	5.697	9.935	7.249	
2	8.631	3.937	3.175	2.545	5.582	2.101	
3	11.335	102.476	9.366	3.973	5.219	5.831	
4	5.308	3.488	1.841	1.794	1.599	1.831	
5	3.67	3.528	1.455	0.888	1.378	1.309	
6	91.594	71.66	13.706	19.588	13.286	26.387	
7	31.022	8.103	3.399	3.869	10.046	19.07	
8	12.886	2.355	1.921	1.164	1.289	1.857	
9	22.284	24.1	5.037	5.806	5.746	6.755	
10	15.28	100.405	18.486	22.014	10.342	14.922	
11	25.516	112.658	16.634	7.536	10.233	3.534	
12	10.205	14.475	2.911	8.536	14.028	10.525	
13	22.995	37.862	31.601	5.188	6.149	12.015	
14	19.333	17.295	22.824	9.389	7.053	8.066	
15	19.342	15.851	17.527	8.61	10.577	10.279	
16	14.672	6.689	3.557	3.29	5.204	5.284	
17	37.963	110.194	27.724	24.375	57.606	39.03	
18	53.005	31.016	6.474	14.265	10.915	30.343	
19	8.122	40.389	4.695	2.191	19.547	7.517	

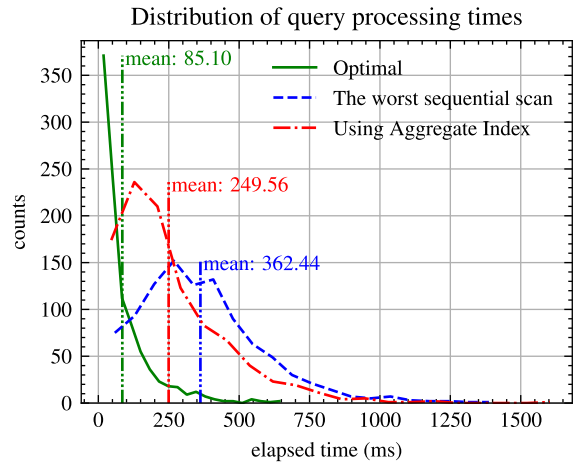
TABLE VIII: Mean query processing time in milliseconds of the query workload B.

Observations:

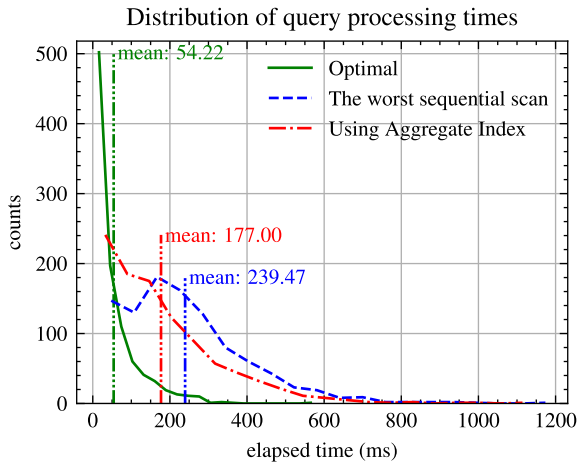
As shown in figures 3a and 3b, the mean values of processing times of the worst sequential scan are longer than that of searching the aggregate index. The worst sequential scan is slower than searching the aggregate index by 45% and 35%, respectively. The goal of our neural network classifiers is to improve the performance of sequential scan by generating an optimized access pattern.

B. Performance of optimal matching for partial tuple completion

In this experiment, we examine the performance of optimal matching in three different steps. In our first step, we set query



(a) Workload A



(b) Workload B

Fig. 3: The distribution of query processing times of workload A and B.

size to be 10, tuple size to be 200, and edge connectivity density to be 0.1. We compute the optimal matching for 100 times and record the response times. Figure 4 shows the histogram of the results. In our second step, we vary edge connectivity densities with the same fixed query size and tuple size as in step one. We repeat the process 100 times and compute the mean response times as results, which are shown in Figure 5. In our last step, we fix the query size and edge connectivity density to be the same as in step one, but change the tuple size so that it goes from 100 to 1,000. Again, we repeat the process for 100 times and compute the mean values. Figure 6 shows the results.

Our workloads A and B have query sizes of 3 and 5, respectively. The average number of attributes of the 10 datasets is less than 200. The experiment results from the above three steps indicate that the response time of optimal matching will be less than 30 milliseconds in our use case. The performance cost of optimal matching will not be as significant as that of full-text index lookup. Therefore, the majority of our

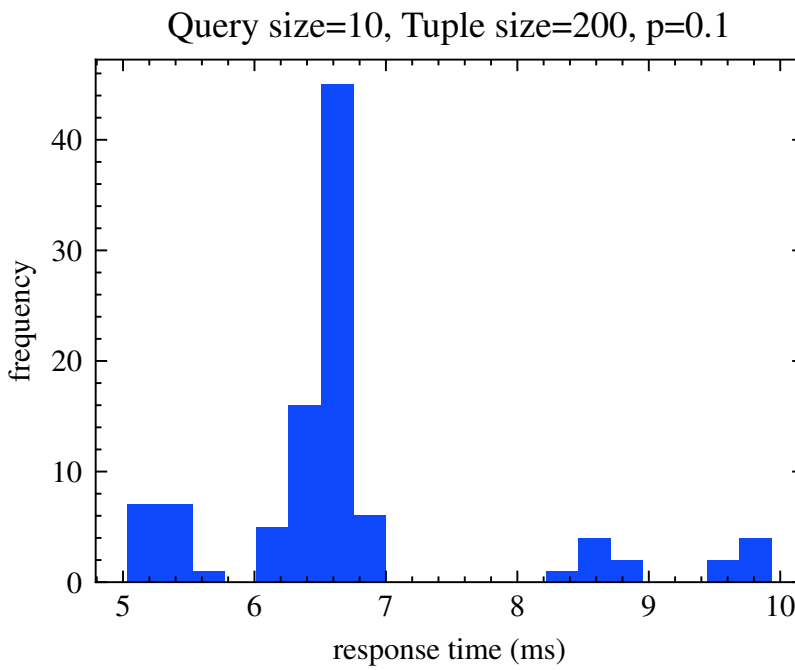


Fig. 4: Distribution of optimal matching response times (ms)

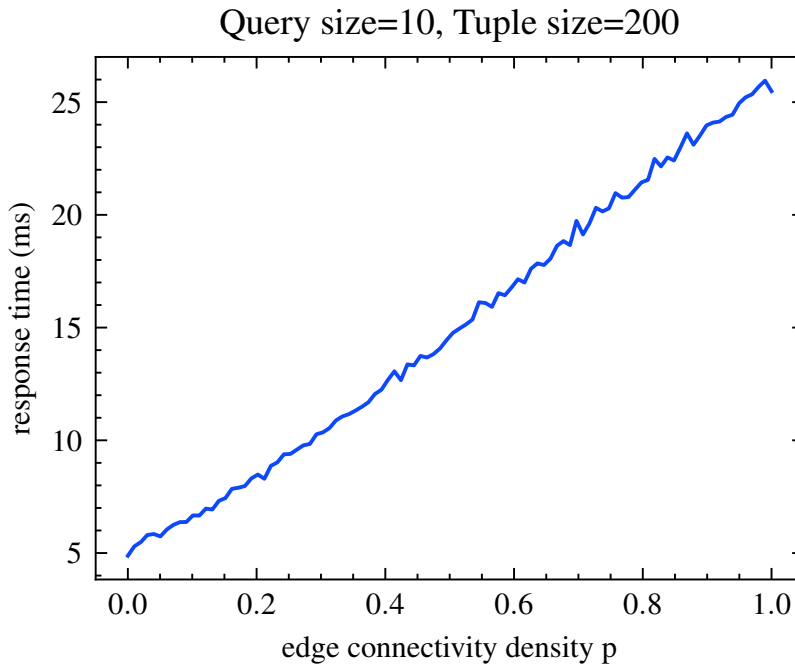


Fig. 5: Optimal matching response time (ms) w.r.t. varying edge connectivity densities

work focuses on how to optimize index lookup using neural networks.

C. Neural network based predictive access

We evaluate five neural network architectures including MLP, LSTM, Conv1D, Transformer and MLP-Mixer. While

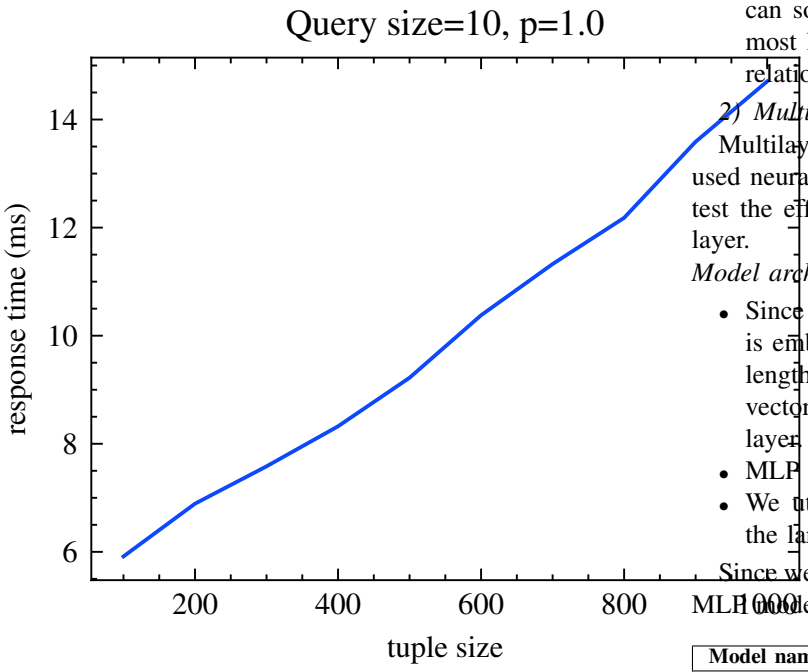


Fig. 6: Optimal matching response time (ms) w.r.t. varying tuple sizes

all of them can be made into deep networks, or in conjunction with other network architectures, our motivation is to make our models as small as possible so that they can be embedded in a search system. Therefore, we focus on the minimalist approach to network design.

1) Common experimental setup and evaluation metrics:
Evaluation scenarios: We evaluate models with the following combination of different scenarios in all experiments:

- We train the models using different sampling rates of attributes, as described in Section IX-A.
- We consider both word-based and 3-gram based training.
- We evaluate the performance of index scan using models' predictions and compare with the optimal and aggregate index lookup for each query in our query workloads.

Tokenization and embedding of texts: The tokens are either words or 3-grams of the values in the partial tuples. Let $\mathbf{Voc}$ be the vocabulary. The vocabulary size is determined by the attribute sampling rate during training. The vocabulary sizes are given in Table V. Each token is embedded into $\mathbb{R}^{64}$ by a simple embedding layer that has $|\mathbf{Voc}| \times 64$ parameters.
Evaluation metrics: We evaluate our models using the following metrics:

- Top-k classification accuracy: since we know the true relations that contain the partial tuples in our query workloads, we can evaluate the classification accuracy of our model. The top-k accuracy is defined as the percentage of the correct label among the top-k labels predicted.
- Index lookup response time using the predicted access pattern: using the likelihoods produced by our model, we can sort the indexes by their likelihoods and access the most likely index first. The scan continues until the true relation is reached.

*2) Multilayer perceptron (MLP): D*escription:
Multilayer perceptron (MLP) is probably the most widely used neural network architecture. In this experiment, we will test the effectiveness of MLP by itself with a single hidden layer.
Model architecture:

- Since each query is a sequence of tokens, each query is embedded into $\mathbb{R}^{L\times 64}$ where L is the token sequence length. We use global average pooling to generate a flat vector in $\mathbb{R}^{64}$, which is used as the input to the MLP layer.
- MLP has one hidden layer of size 100.
- We utilize one drop-out layer to prevent overfitting by the large embedding layer.

Since we have six sets of training datasets, we have 6 trained MLP models, as shown in Table IX.

Model name	Training dataset
mlp100	100% attrs., word-based
mlp100-3gram	100% attrs., 3-gram based
mlp75	75% attrs., word-based
mlp75-3gram	75% attrs., 3-gram based
mlp50	50% attrs., word-based
mlp50-3gram	50% attrs., 3-gram based

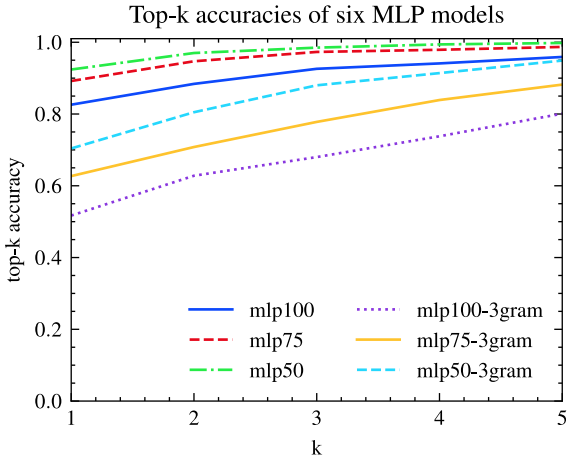
TABLE IX: MLP model names and corresponding training datasets.

*O*bservations:

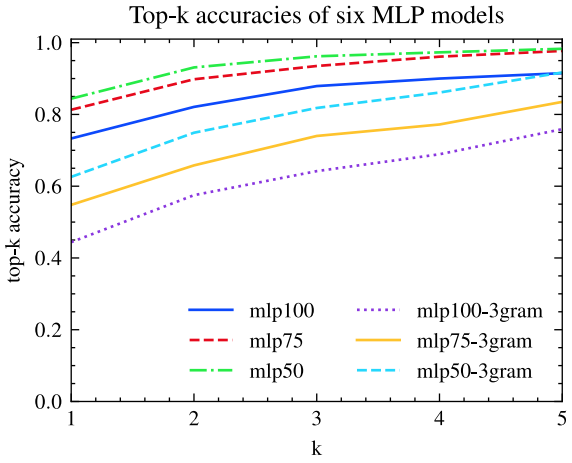
- The top-1 to top-5 accuracies for all variations of MLP models under the workloads A and B are shown in Figure 7. Word-based models perform better than 3-gram based models under both workloads, which do not contain misspelled and unknown keywords.
- Figure 8 and Figure 9 show the distribution of query processing time under the workloads A and B, respectively. Word-based models improve query processing more than 3-gram based models do. In addition, the models trained using partial tuples with less attributes perform better than those trained using partial tuples with more attributes.

*3) Long short-term memory (LSTM): D*escription:
Long short-term memory (LSTM) is a neural network architecture for sequence learning. Even though attributes in a tuple is not considered as a sequence since they are not ordered, the attribute values can be considered as short sequences of tokens. We test how LSTM performs when dealing with relational data in this experiment. Again we apply the minimalist approach to its network design by a single LSTM layer.
Model architecture:

- Each query is embedded into $\mathbb{R}^{L\times 64}$ where L is the token sequence length, which is used as the input to the LSTM layer.
- We apply one drop-out layer to the output of the LSTM layer to prevent overfitting.



(a) Under the workload A.



(b) Under the workload B.

Fig. 7: Top-k accuracies of six MLP models under the workload A and B.

Model name	Training dataset
lstm100	100% attrs., word-based
lstm100-3gram	100% attrs., 3-gram based
lstm75	75% attrs., word-based
lstm75-3gram	75% attrs., 3-gram based
lstm50	50% attrs., word-based
lstm50-3gram	50% attrs., 3-gram based

TABLE X: LSTM model names and corresponding training datasets.

- The LSTM model has one hidden layer of size 100.

*O*bservations:

- The top-1 to top-5 accuracies for all variations of LSTM models under the workloads A and B are shown in Figure 10. Word-based models perform better than 3-gram based models. When compared to MLP models, LSTM models underperform under both workloads.
- Figure 11 and Figure 12 show the distribution of query processing time under the workloads A and B, respectively. Word-based models improve query processing more than 3-gram based models do. In addition, similar to MLP models, the models trained using partial tuples with less attributes perform better than those trained using partial tuples with more attributes.

*4) One dimensional convolution (Conv1D): D*escription:

Another way of doing sequence learning is 1-dimensional convolutional neural networks (Conv1D). Similar to what we do to LSTM, we want to see how Conv1D performs when processing relational data. We also minimize the network structure by using a single Conv1D layer.

Model architecture:

- Each query is embedded into $\mathbb{R}^{L\times 64}$ where L is the token sequence length, which is used as the input to the Conv1D layer.
- The Conv1D layer has 64 kernels with kernel size of 3.
- We apply global average pooling to the output of Conv1D layer to generate a flat vector in $\mathbb{R}^{64}$.
- We apply one drop-out layer to the flattened vector to prevent overfitting.

Model name	Training dataset
conv1d100	100% attrs., word-based
conv1d100-3gram	100% attrs., 3-gram based
conv1d75	75% attrs., word-based
conv1d75-3gram	75% attrs., 3-gram based
conv1d50	50% attrs., word-based
conv1d50-3gram	50% attrs., 3-gram based

TABLE XI: Conv1D model names and corresponding training datasets.

*O*bservations:

- The top-1 to top-5 accuracies for all variations of Conv1D models under the workloads A and B are shown in Figure 13. Word-based models perform better than 3-gram based models.
- Figure 14 and Figure 15 show the distribution of query processing time under the workloads A and B, respectively. Word-based models improve query processing more than 3-gram based models do. Under the workload A, the 3-gram based models trained using partial tuples with less attributes perform better than those trained using partial tuples with more attributes, but the word-based models show an opposite relationship. Under the workload B, both word-based and 3-gram based models do better when trained using partial tuples with less attributes.

*5) Single transformer block: D*escription:

The Transformer architecture introduced in the original Transformer paper [25] is one of the main advances in natural language processing. In this experiment, we evaluate Transformer architecture for relational data. We largely follow the encoder architecture of the vanilla Transformer presented in the paper [25]. However, since our data are relational, we remove positional encoding. In addition, we only create one Transformer block to minimize the model architecture.

Model architecture:

- Each query is embedded into $\mathbb{R}^{L\times 64}$ where L is the token sequence length, which is used as the input to the Transformer block.
- Inside the Transformer block, the attention layer has 4 attention heads. We add an extra drop-out layer after both the attention layer and the feed-forward network to prevent overfitting.
- The feed-forward network has a hidden layer of size 64.
- We apply global average pooling to the output of the Transformer block to generate a flat vector in $\mathbb{R}^{64}$.
- Then we apply a drop-out layer to the flattened vector.

Model name	Training dataset
transformer100	100% attrs., word-based
transformer100-3gram	100% attrs., 3-gram based
transformer75	75% attrs., word-based
transformer75-3gram	75% attrs., 3-gram based
transformer50	50% attrs., word-based
transformer50-3gram	50% attrs., 3-gram based

TABLE XII: Transformer model names and corresponding training datasets.

*O*bservations:

- The top-1 to top-5 accuracies for all variations of Transformer models under the workloads A and B are shown in Figure 16. Again, word-based models perform better than 3-gram based models.
- Figure 17 and Figure 18 show the distribution of query processing time under the workloads A and B, respectively. Word-based models improve query processing more than 3-gram based models do. In addition, the models trained using partial tuples with less attributes perform better than those trained using partial tuples with more attributes.

*6) MLP-Mixer: D*escription:

MLP-Mixer is an architecture based on MLPs only, which is proposed by Tolstikhin et al. [24]. It is intended for computer vision. However, we want to see whether we could use it for our use case. In this experiment, we modify its architecture for text processing. The same minimalist approach and evaluation scenarios are used as in other experiments.

Model architecture:

- Each query is embedded into $\mathbb{R}^{L\times 64}$ where L is the token sequence length, which is used as the input to the Mixer Layer.
- The Mixer Layer is implemented following the architecture presented in the paper [24].
- Layer normalization is applied to outputs from the Mixer Layer.
- We apply global max pooling to the outputs of layer normalization to generate a flat vector in $\mathbb{R}^{64}$, which is used as input to the output layer.

*O*bservations:

- The top-1 to top-5 accuracies for all variations of MLP-Mixer models under the workloads A and B are shown in Figure 19.

Model name	Training dataset
mlpmixer100	100% attrs., word-based
mlpmixer100-3gram	100% attrs., 3-gram based
mlpmixer75	75% attrs., word-based
mlpmixer75-3gram	75% attrs., 3-gram based
mlpmixer50	50% attrs., word-based
mlpmixer50-3gram	50% attrs., 3-gram based

TABLE XIII: MLP-Mixer model names and corresponding training datasets.

- Figure 20 and Figure 21 show the distribution of query processing time under the workloads A and B, respectively. The word-based models perform slightly better than 3-gram based models, with the exception of two models $mlpmixer75$ and $mlpmixer75\text{-}3gram$. In addition, the models trained using partial tuples with less attributes perform better than those trained using partial tuples with more attributes, with the exception of the model $mlpmixer75$ under the workload B.

7) Comparison of different models: Based on the observations from Section X-C2 to Section X-C6, we have compiled the following table to compare their relative performances with respect to the optimal and the aggregate index lookup under the workload A.

	Word tokens			
Model	parameters	$\frac{\text{model}}{\text{optimal}}$	$\frac{\text{model}}{\text{aggr}}$	pa
MLP	3.02M	**1.10**	0.37	27
LSTM	3.05M	1.52	0.52	30
Conv1D	3.03M	**1.12**	0.38	27
Transformer	3.09M	**1.12**	0.38	34
MLP Mixer	3.04M	1.65	0.56	28

TABLE XIV: Comparison of models under the workload A with the top-3 measurements highlighted.

The observation supports our proposal of utilizing neural networks to accelerate index lookup. As shown in Table XIV, many of the models perform well compared to the optimal index lookup, and significantly outperform the aggregate index lookup.

For word based tokens, MLP is only 10% slower than the optimal index lookup, and outperforms the aggregate index by almost 3 times. The Conv1D and Transformer are only slightly worse than MLP. But the transformer models have larger model sizes. Both LSTM and MLP Mixer do not perform that well when compared to the other three networks.

We realize that we only use one transformer block and one MLP Mixer Layer, respectively. In our context, we are interested in embedded networks as part of the query processor. Therefore, our focus is limited to small network architectures.

Regarding the model sizes, the main source of parameters is the embedding layer. Word based tokens produce far more bigger vocabulary, which then requires many more embedding vectors as model parameters. The vocabulary of 3-grams is much smaller, and thus produces much more compact models.

Since 3-gram tokens individually capture less information than word-based tokens, we expect the observed performance degradation when using the 3-gram tokens. With that being

said, MLP has shown to outperform the aggregate index with twice the performance even for 3-grams.

The value of 3-gram vocabularies becomes more apparent in scenarios with noisy queries. When query strings contain spelling and other noises at a sub-word level, the two methods of tokenization, i.e., word-based and 3-gram based, behave quite differently. For word-based tokenization, noisy words will generate out-of-vocabulary (OOV) tokens which do not contribute to the classification of the network. However, the 3-gram tokenization can still produce *some* 3-gram tokens even for misspelled and unknown words in a query string. The next section is dedicated to evaluate how well word-based and 3-gram based networks behave in the presence of noisy queries.

D. Impact of noisy queries on models' performance

*D*escriptions:

In this experiment, we investigate the impact of misspelled words in queries on the models' performance of top-5 accuracy. We simulate the scenario by replacing a randomly selected character in a word with the special character "_". For example, the city name `toronto` becomes `to_onto` after mutation. Its 3-gram tokens will be `[__t, _to, to_, o_o, _on, ont, nto, to_, o__]`.

Observations:

The impact of noisy queries on models' performance of top-5 accuracy is shown in Figure 22 and 23 under the workload A and B, respectively. The performance degradation of word-based models is much faster than that of 3-gram based models. This clearly shows that 3-gram based models are more resilient to query noises.

XI. Related Work

Early papers: Hristidis and Papakonstantinou [13] presented DISCOVER, a system that allowed users to submit keyword queries to relational databases without the knowledge of the underlying database schema. DISCOVER processes keyword queries to generate candidate networks of relations and create execution plans to be submitted to RDBMS, which will return search results. Liu et al. [17] proposed an information retrieval (IR) ranking strategy for effective keyword search related to relational databases. The ranking strategy used four normalization factors for computing ranking scores. All answers of a query were ranked based on computed ranking scores, and the top-k answers were returned as results.

Extension to include frequent co-occurring term (FCT): Tao and Yu [22] proposed an operator called frequent co-occurring term (FCT) search, and an algorithm that could solve FCT search effectively without using the conventional keyword search methods. The purpose of FCT search is to extract the terms that most accurately represent a set of keywords, i.e., to discover the concepts closely related to the keywords set.

Survey of keyword search: A survey about research on keyword search in relational databases was done by Park and Lee [20]. First they listed fundamental characteristics of keyword search in relational databases: an indexing structure, being able to formalize internal queries based on query keywords, being able to correctly constructing candidate answers, and an answer ranking strategy. Then they investigated five research dimensions, including data representation, ranking, efficient processing, query representation, and result presentation. At the end, they pointed out some promising research directions. One of them is efficient top-k query processing. The authors believed that keyword search in relational databases would benefit from advance in top-k query processing techniques.

Interpretation of keywords in query: Zeng et al. [26] presented a framework based on keyword query interpretations. It incorporated human feedback to remove keyword vaguenesses and use a ranking model for query interpretation evaluation afterwards.

Top-k recommendation: Meng et al. [19] presented an approach to solve typical and semantically related queries to a given query. This can help users to explore their query intentions and improve their query formulation.

Index optimization: Ding et al. [11] presented an updatable learned index called ALEX, an in-memory index structure, for index optimization. ALEX addressed practical issues related to various types of workloads with dynamic updates. RadixSpline [15], another learned index, tackled the issue of index implementation difficulty. RadixSpline offered quick build using a single pass over data while achieving competitive performance.

Query optimization: Bao [18] is a learned query optimization system using reinforcement learning. It can learn from mistakes and adapt to dynamic workloads, data, and schema, thus is capable of applying per-query optimization hints.

Cost models for query processing: Siddiqui1 et al. [21] investigated how to learn cost models from cloud workloads for big data systems. The learned cost models could be integrated with existing query optimizers. Such a query optimizer could make accurate cost predictions, which can improve resource efficiency in big data systems.

References

[1] Statistics Canada. Canadian Income Survey (CIS), 2017, 2019.

[2] Statistics Canada. Labour Force Survey, April 2019 [Canada], 2019.

[3] Statistics Canada. Canadian Community Health Survey - Annual Component (CCHS) 2017-2018, 2020.

[4] Statistics Canada. Crowdsourcing: Impacts of COVID-19 on Canadians' Experiences of Discrimination Public Use Microdata File, 2020.

[5] Statistics Canada. Crowdsourcing: Impacts of COVID-19 on Canadians' Perception of Safety Public Use Microdata File, [2020], 2020.

[6] Statistics Canada. Crowdsourcing: Impacts of COVID-19 on Canadians Public Use Microdata File, [2020], 2020.

[7] Statistics Canada. Crowdsourcing: Impacts of the COVID-19 on Canadians – Trust in Others Public Use Microdata File [2020], 2020.

[8] Statistics Canada. Crowdsourcing: Impacts of the COVID-19 on Canadians – Your Mental Health Public Use Microdata File, [2020], 2020.

[9] Statistics Canada. Impacts of the COVID-19 pandemic on postsecondary students (ICPPS) 2020, Crowdsource file, Public use microdata file, 2020.

[10] Statistics Canada. Canadian Housing Survey, 2018, 2021.

[11] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. Alex: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 969–984, 2020.

[12] Google LLC. Tensorflow.

[13] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 670–681. Elsevier, 2002.

[14] M Kim, K Whang, and J Lee. n-gram/2l-approximation: a two-level n-gram inverted index structure for approximate string matching. *Computer Systems Science and Engineering*, 22(6):365, 2007.

[15] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Radixspline: a single-pass learned index. In *Proceedings of the third international workshop on exploiting artificial intelligence techniques for data management*, pages 1–5, 2020.

[16] Grzegorz Kondrak. N-gram similarity and distance. In *String Processing and Information Retrieval: 12th International Conference, SPIRE 2005, Buenos Aires, Argentina, November 2-4, 2005. Proceedings 12*, pages 115–126. Springer, 2005.

[17] Fang Liu, Clement Yu, Weiyi Meng, and Abdur Chowdhury. Effective keyword search in relational databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 563–574, 2006.

[18] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. *ACM SIGMOD Record*, 51(1):6–13, 2022.

[19] Xiangfu Meng, Longbing Cao, Xiaoyan Zhang, and Jingyu Shao. Top-k coupled keyword recommendation for relational keyword queries. *Knowledge and Information Systems*, 50:883–916, 2017.

[20] Jaehui Park and Sang-goo Lee. Keyword search in relational databases. *Knowledge and Information Systems*, 26:175–193, 2011.

[21] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. Cost models for big data query processing: Learning, retrofitting, and our findings. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 99–113, 2020.

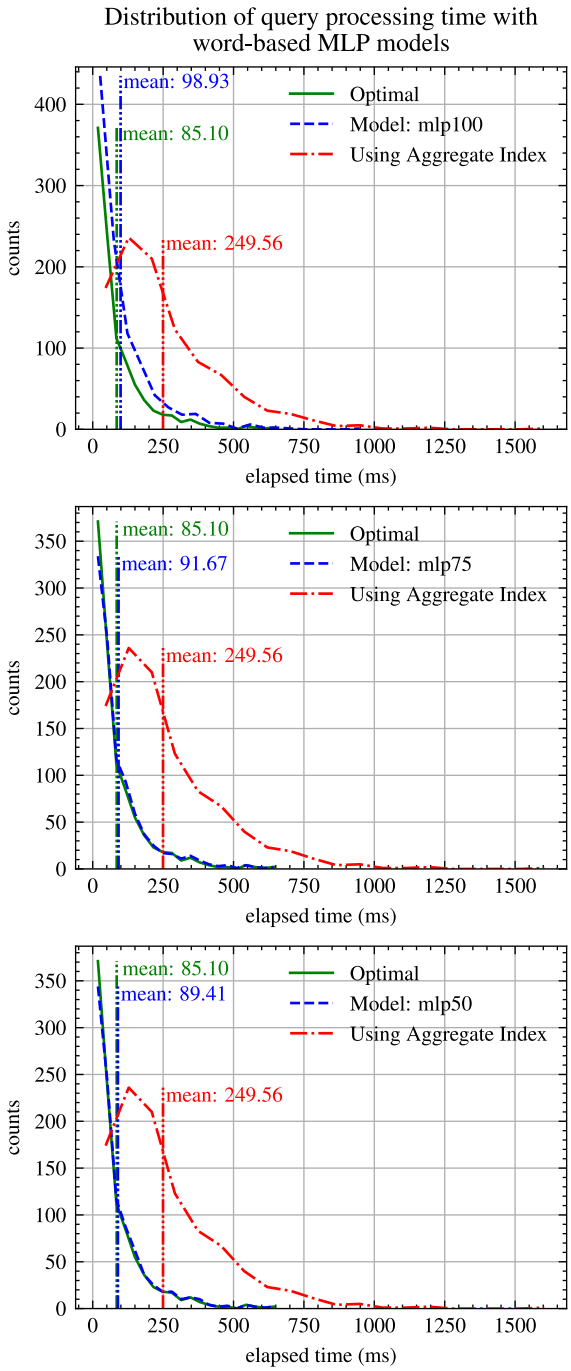
[22] Yufei Tao and Jeffrey Xu Yu. Finding frequent co-occurring terms in relational keyword search. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 839–850, 2009.

[23] The Apache Software Foundation. Apache lucene.

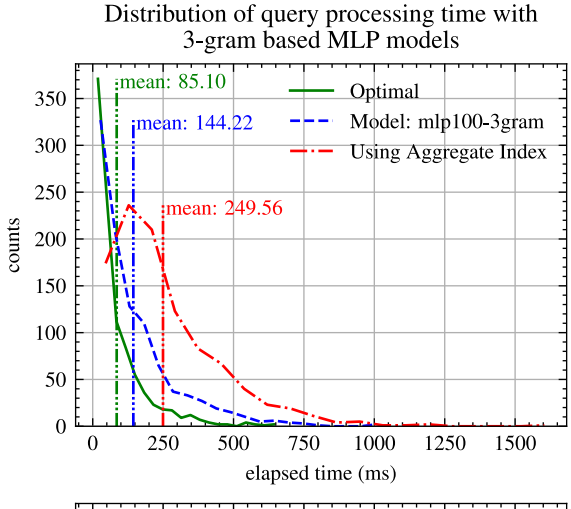
[24] Ilya O. Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, and Alexey Dosovitskiy. Mlp-mixer: An all-mlp architecture for vision. *CoRR*, abs/2105.01601, 2021.

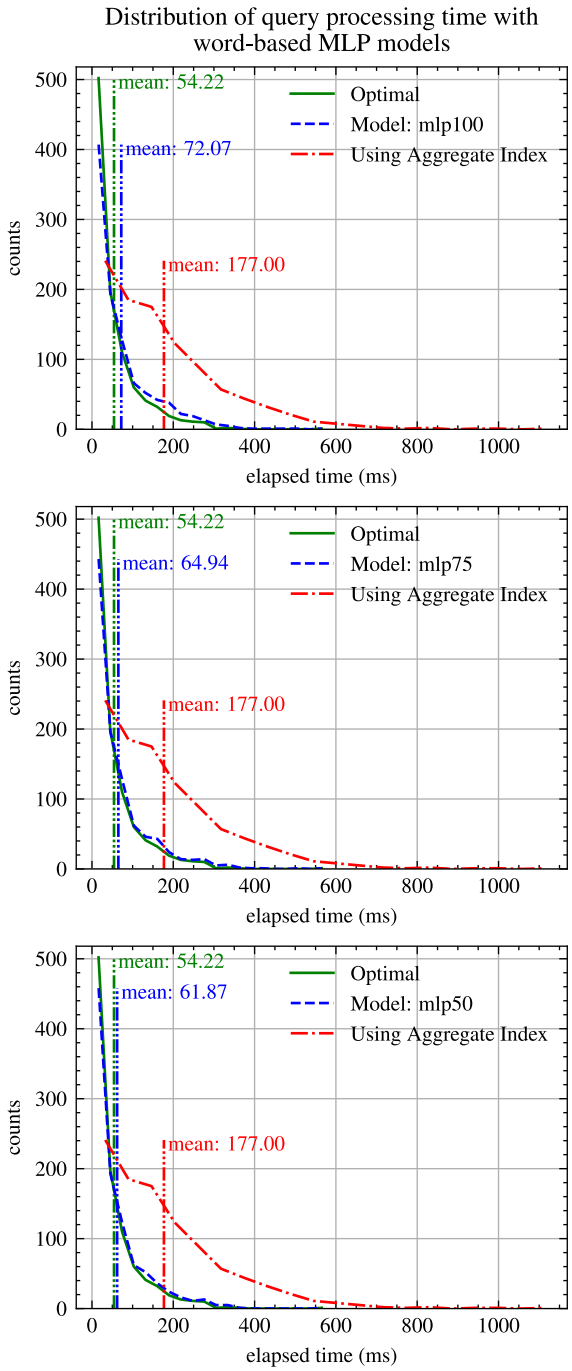
[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[26] Zhong Zeng, Zhifeng Bao, Tok Wang Ling, and Mong Li Lee. isearch: an interpretation based framework for keyword search in relational databases. In *Proceedings of the Third International Workshop on Keyword Search on Structured Data*, pages 3–10, 2012.

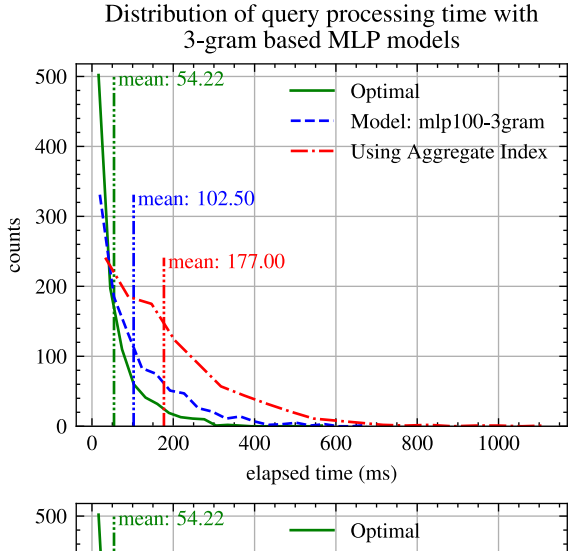


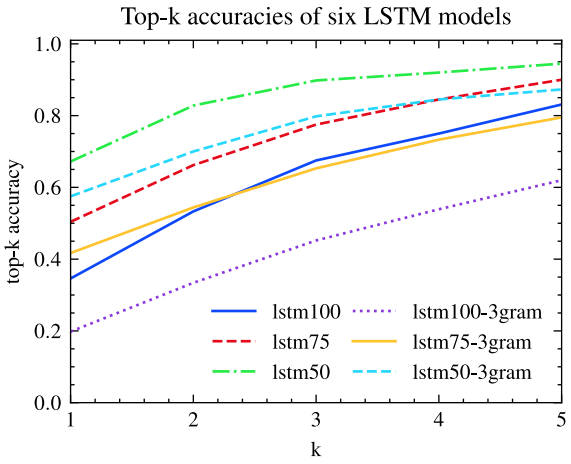
(a) Word-based MLP models



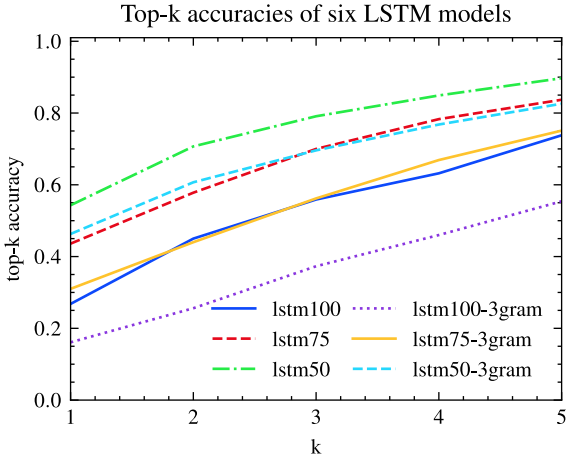


(a) Word-based MLP models



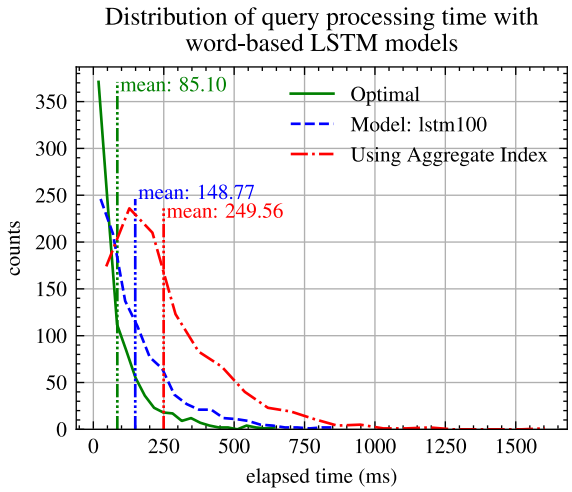


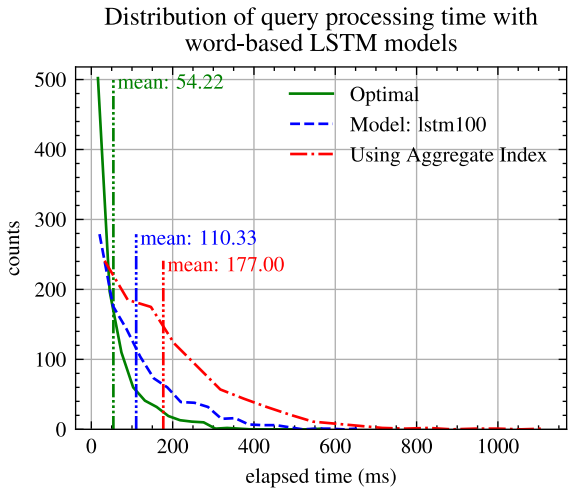
(a) Under the workload A.

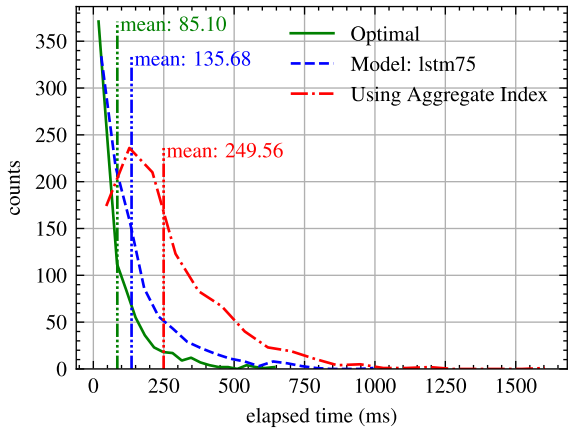


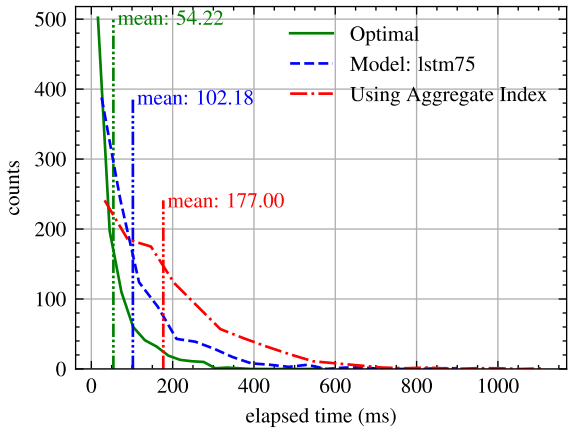
(b) Under the workload B.

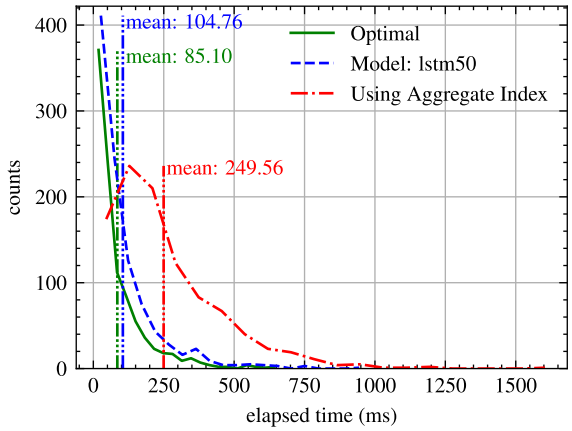
Fig. 10: Top-k accuracies of six LSTM models under the workload A and B.

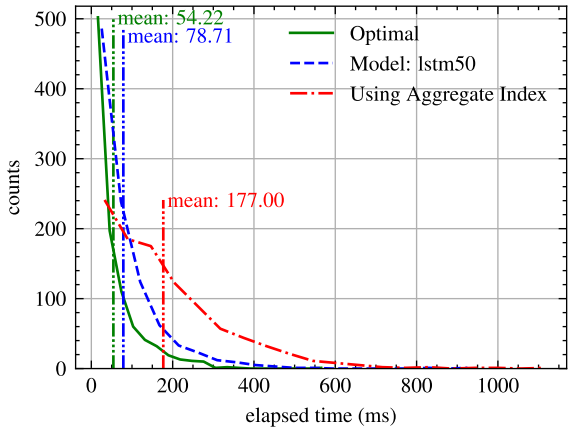






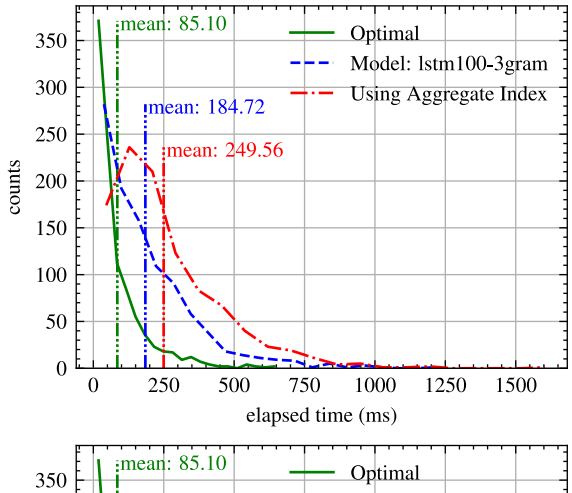


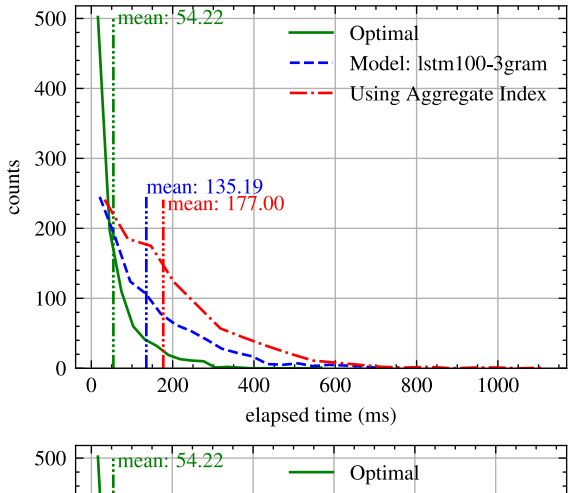


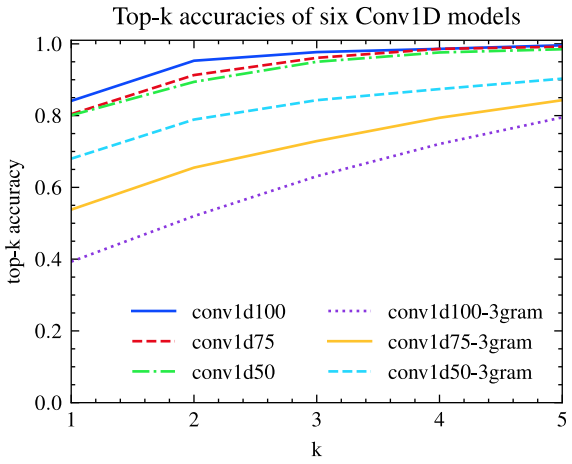


(a) Word-based LSTM models

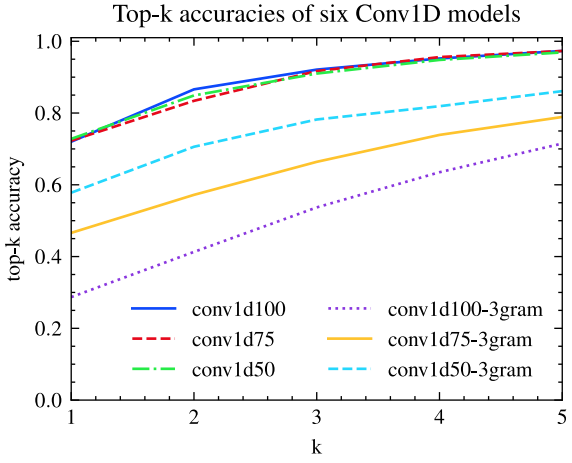
(a) Word-based LSTM models





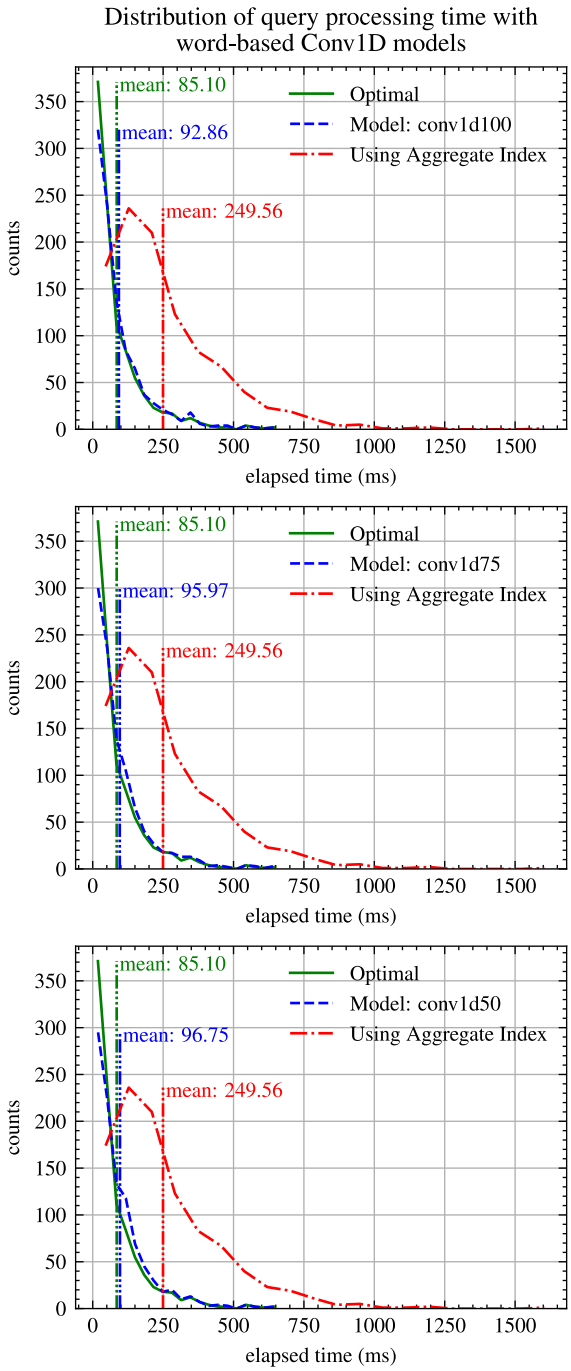


(a) Under the workload A.

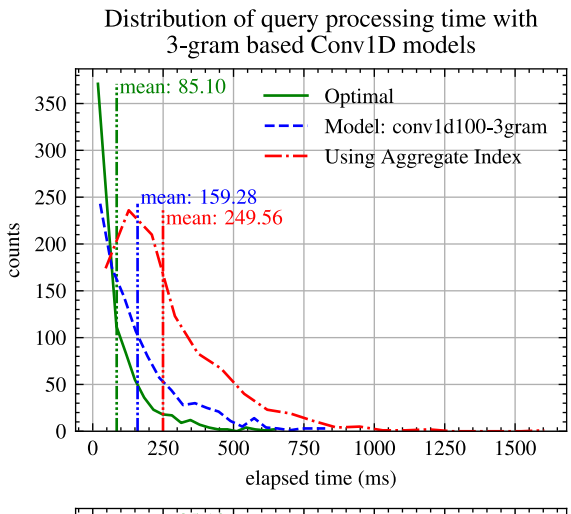


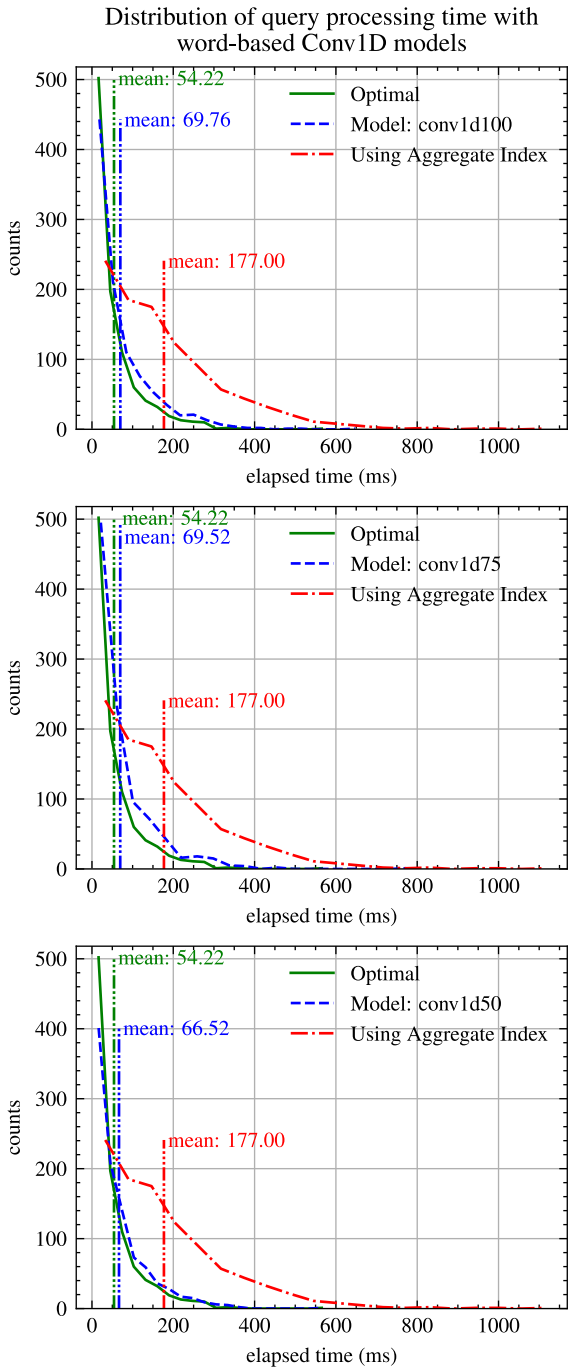
(b) Under the workload B.

Fig. 13: Top-k accuracies of six Conv1D models under the workload A and B.

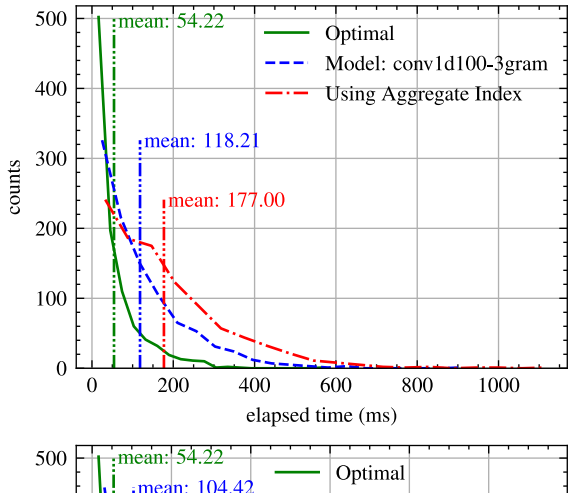


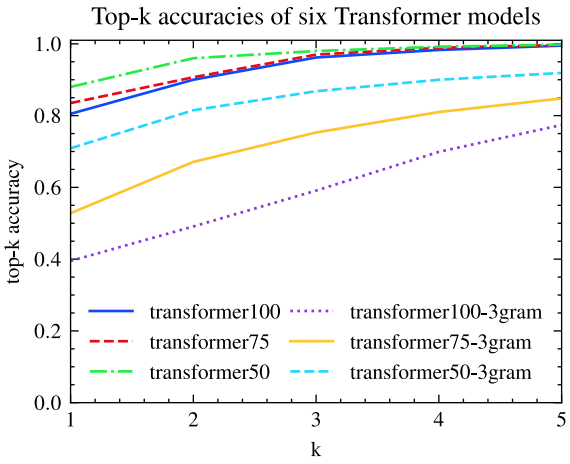
(a) Word-based Conv1D models



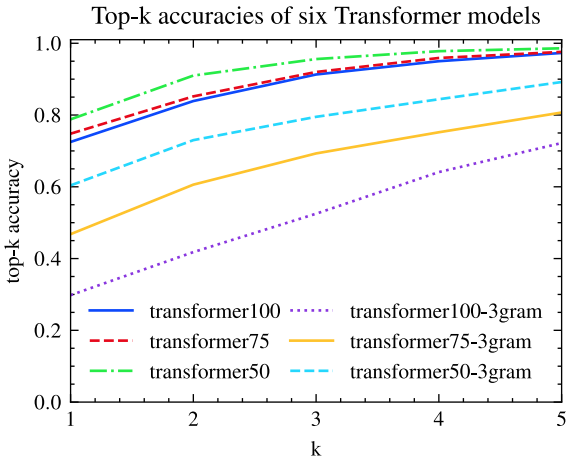


(a) Word-based Conv1D models



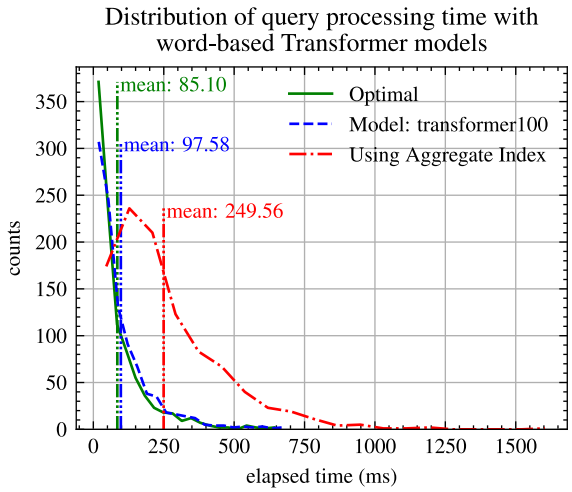


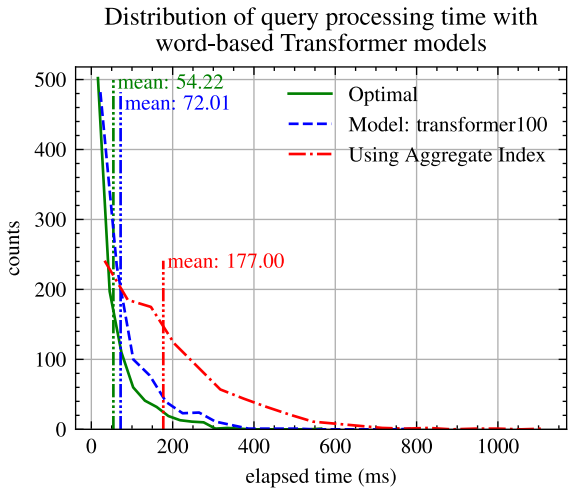
(a) Under the workload A.

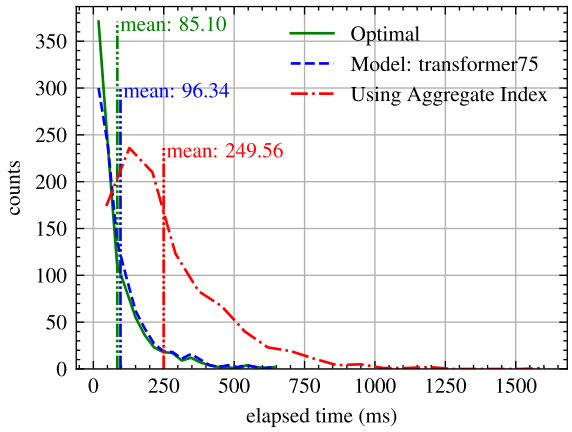


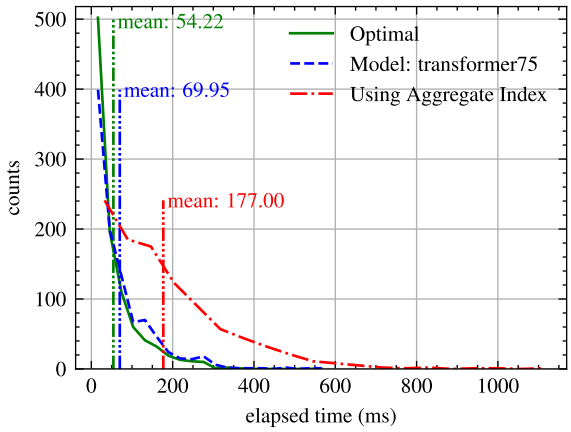
(b) Under the workload B.

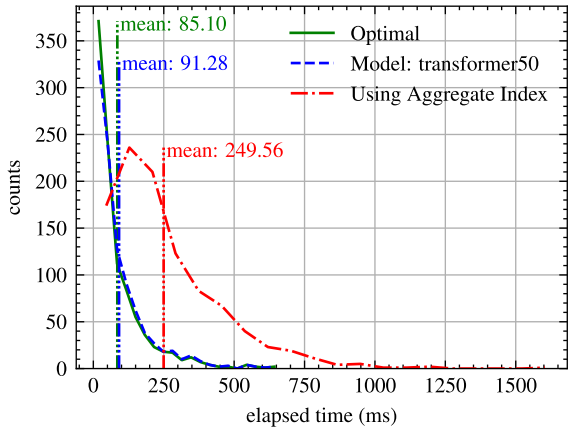
Fig. 16: Top-k accuracies of six Transformer models under the workload A and B.

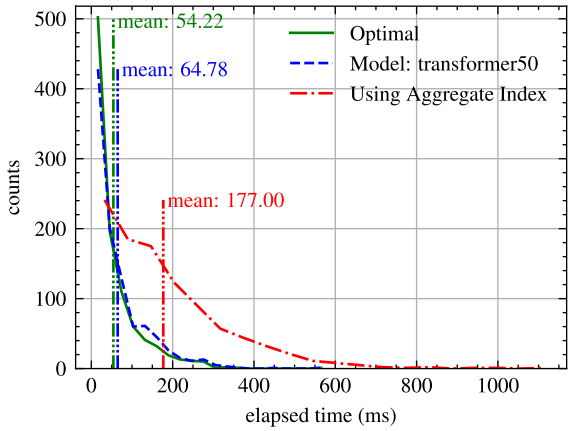






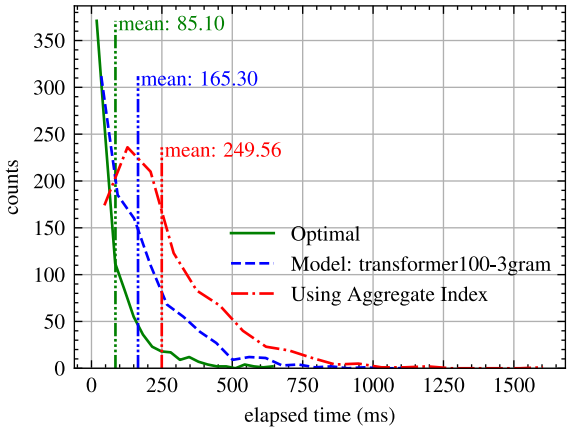


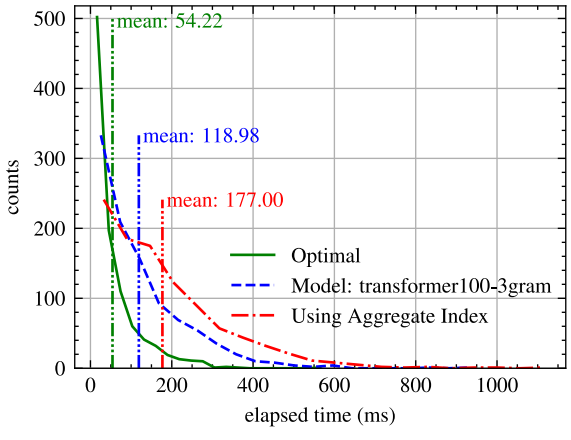


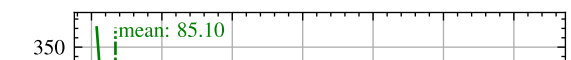


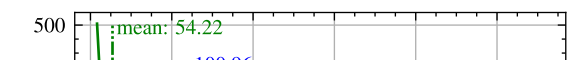
(a) Word-based Transformer models

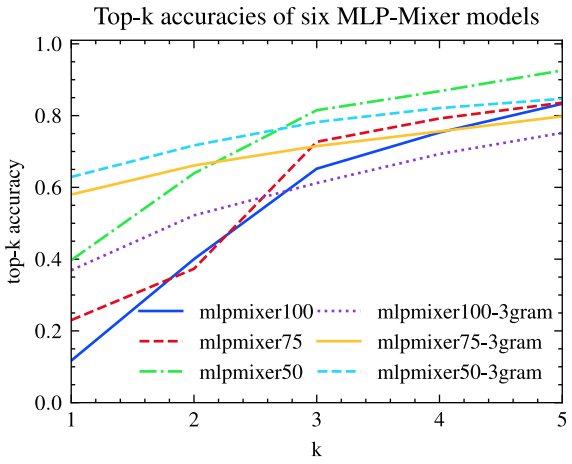
(a) Word-based Transformer models



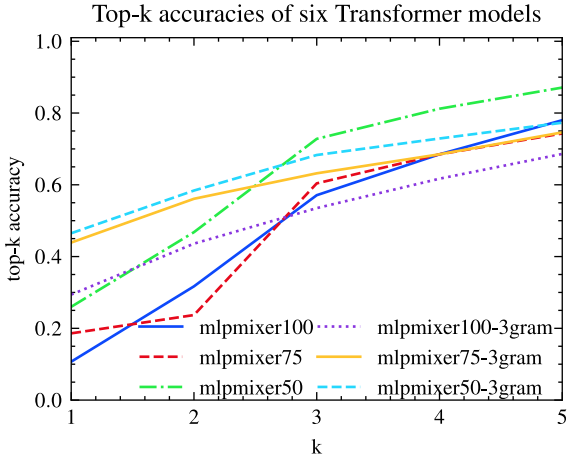






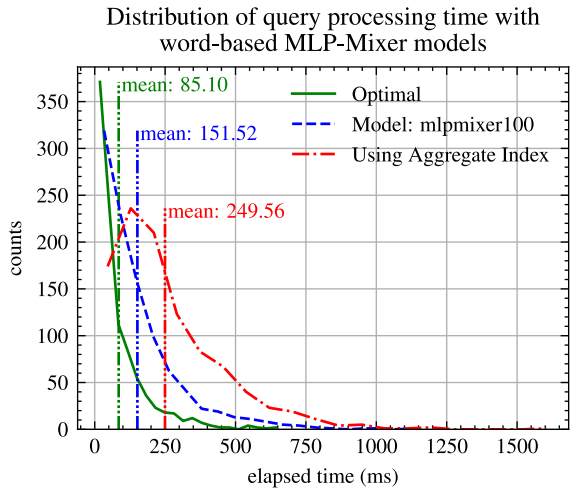


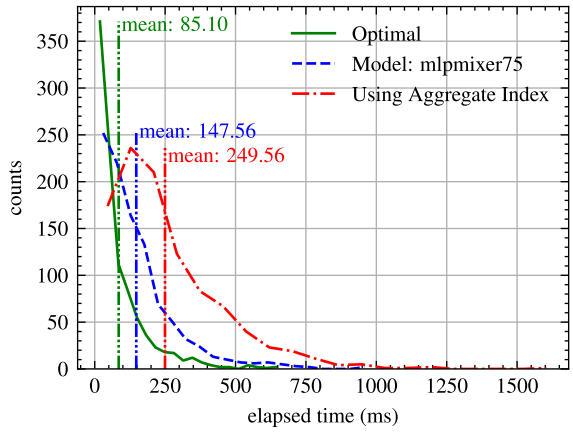
(a) Under the workload A.

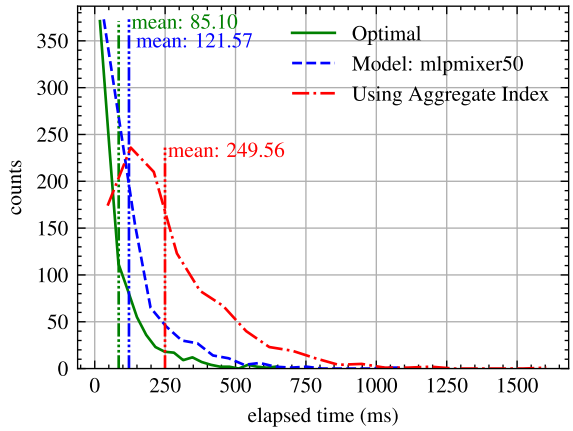


(b) Under the workload B.

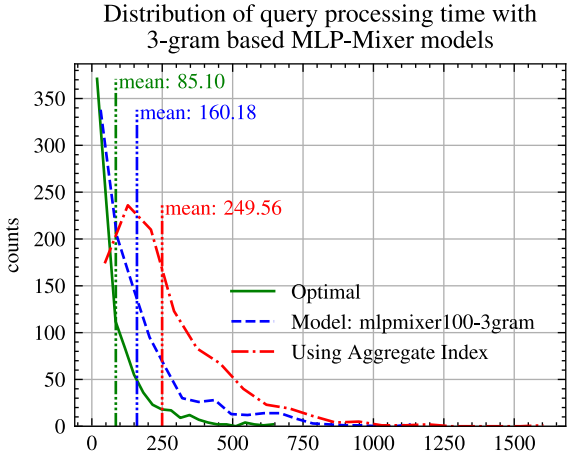
Fig. 19: Top-k accuracies of six MLP-Mixer models under the workload A and B.

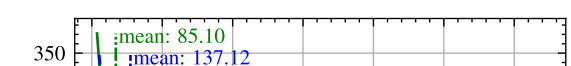


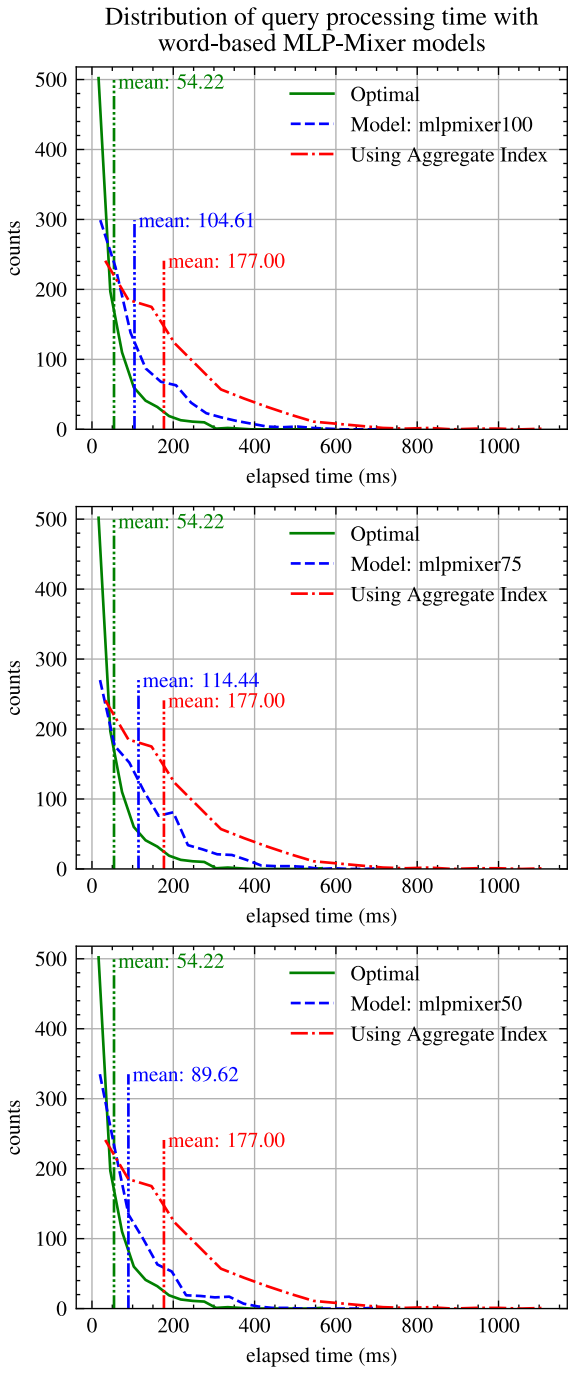




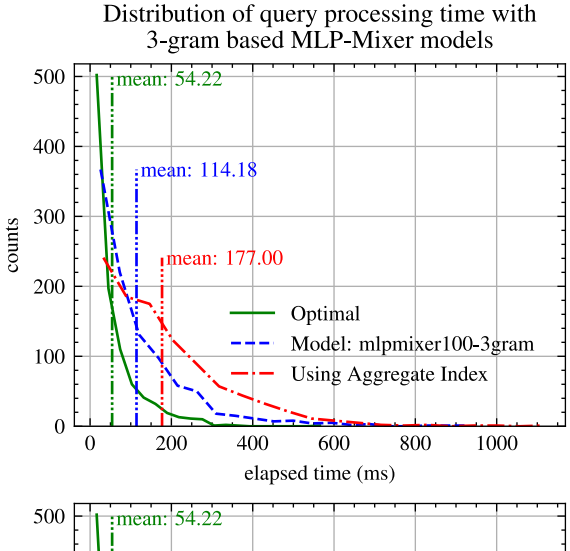
(a) Word-based MLP-Mixer models

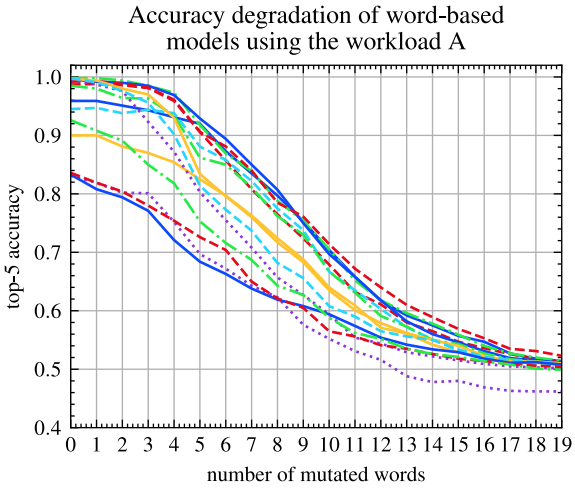




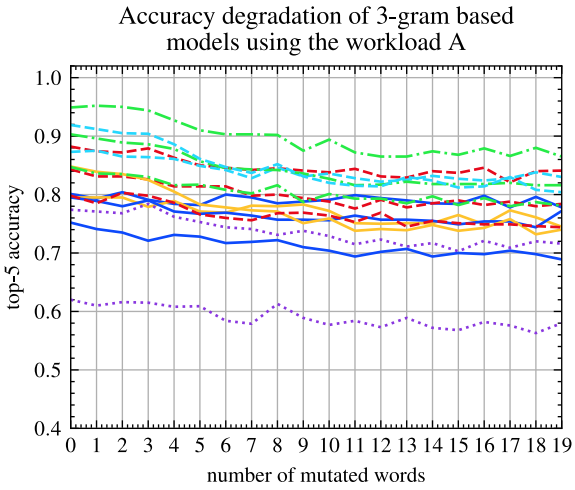


(a) Word-based MLP-Mixer models



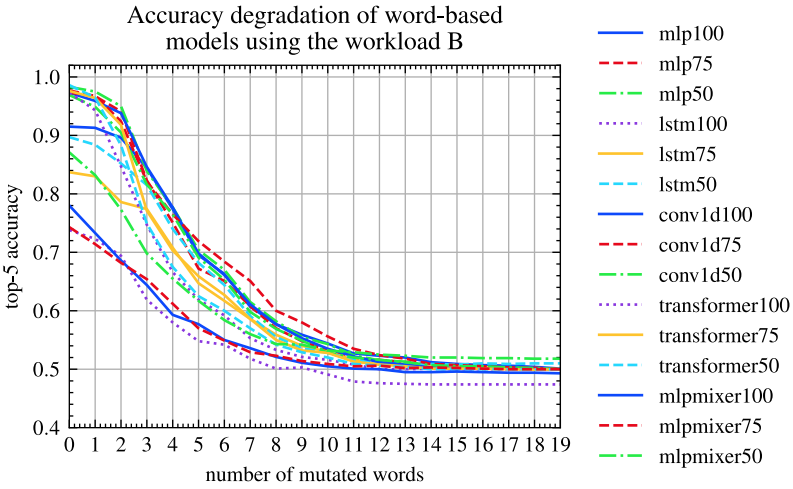


(a) Word-based models

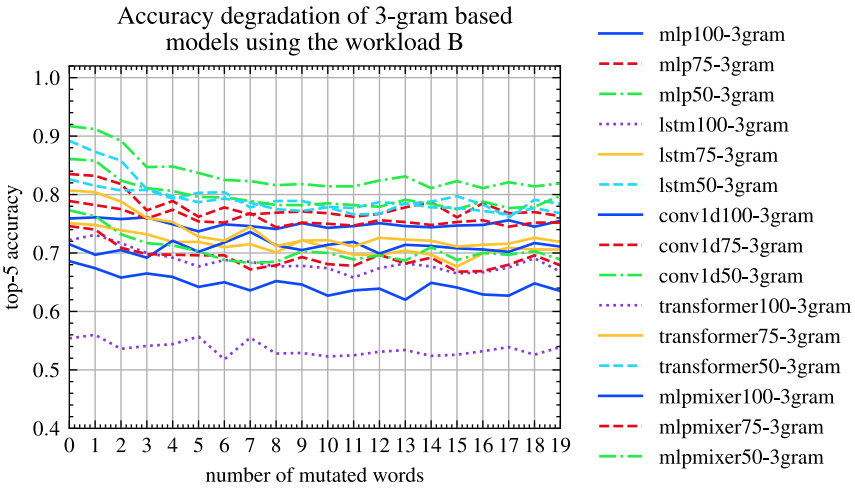


(b) 3-gram based models

Fig. 22: Top-5 accuracy degradation of all models under the workload A.



(a) Word-based models



(b) 3-gram based models

Fig. 23: Top-5 accuracy degradation of all models under the workload B3.