ALECE: An Attention-based Learned Cardinality Estimator for SPJ Queries on Dynamic Workloads (Extended)

Pengfei Li[1], Wenqing Wei[1], Rong Zhu[1], Bolin Ding[1], Jingren Zhou[1] and Hua Lu[2]
[1]Alibaba Group, China, [2]Roskilde University, Denmark
[1]{lpf367135, weiwenqing.wwq, red.zr, bolin.ding, jingren.zhou}@alibaba-inc.com, [2]luhua@ruc.dk

ABSTRACT

To support efficient query processing, query optimizers in databases have for decades relied on delicate cardinality estimation methods. Inspired by the recent trend of applying machine learning techniques for database tasks, we propose an Attention-based LEarned Cardinality Estimator (**ALECE** for short) for SQL queries. The core idea is to discover the implicit relationships between queries and underlying dynamic data by using attention mechanisms in ALECE's two modules built on top of carefully designed featurizations for data and queries. In particular, the data-encoder module makes organic aggregations among all attributes in the database, whereas the query-analyzer module builds a bridge between the query featurizations and the data aggregations. We experimentally evaluate ALECE on various workloads. The extensive results demonstrate that ALECE clearly outperforms the PostgreSQL's built-in cardinality estimation method and other alternatives in terms of multiple performance metrics. Especially, ALECE achieves nearly optimal performance on dynamic workloads.

1 INTRODUCTION

Cardinality estimation plays a paramount role in the query optimizer of a DBMS [19, 45]. Its purpose is to estimate the number of the result elements of a SQL query before actually executing it, and thus to help the query optimizer to generate good query plans. In the past few decades, the mainstream of cardinality estimation has always been statistical data-driven methods. Such methods condense information about data into lightweight summaries, *e.g.*, histograms, sketches and data distribution approximation, and adopt analytic functions with the summaries as the input to estimate cardinatilies of SQL queries [28, 45, 65]. However, real-world datasets are often very complex and the analytic functions are usually not powerful enough to build correct mappings between the coarse data summaries and the cardinalities of SQL queries. Also, SQL queries often contain join predicates but it is difficult and time-consuming to build particular summaries for each join schema. Computing the joint data distributions is also usually intractable due to high computation and storage overhead.

Recently, traditional methods have been disrupted by learned cardinality estimators that use machine learning models to do the estimations. Some of such estimators [27, 53, 61, 62] learn tighter data distributions from the underlying database and use analytic expressions to estimate the cardinalities. In contrast, query-driven models [33, 63] utilize the feedback of executed queries in a supervised fashion. The latter learn the relation between cardinalities and query distributions, without paying particular attention to the underlying database. However, neither kind of models can fully make use of both data and queries. It is difficult for them to extract individualized useful information for different queries. A few models [15, 33, 39] consider both data and queries. However, they either only use simple and trivial data information and requires sampling operations over relations [33, 56], or do not support processing queries with joins [15] or complex joins [39].

In addition to these drawbacks, existing models have a more critical problem which persuades us to design a more effective cardinality estimator: They do not perform well on **dynamic workloads, that mix queries and data manipulation statements including inserts, deletes and updates.** Such statements tend to make estimations difficult as they influence the data distribution and shift the mapping between true cardinalities and query distributions. When the underlying data changes, the joint data distributions among relations and attributes as well as the mapping between queries and true cardinalities also become different. Thus, single data- or query-driven methods can hardly work on dynamic workloads. Sample-based methods [33, 56], even if they consider both data and queries, will also achieve degraded performance as the distribution of original samples is not compatible with that of the underlying data after executing data manipulation statements. More importantly, existing methods do not answer **how to reasonably link SQL queries and the underlying data** and build an appropriate mapping among the true cardinalities, queries and data—especially when data is not static.

To address the aforementioned problems, we in this paper design an Attention-based LEarned Cardinality Estimator (ALECE) to focus on the estimations for the select-project-join (SPJ) queries. ALECE is both data- and query-driven. When estimating an SPJ query's cardinality, it losslessly featurizes the query into a vector. Meanwhile, it efficiently featurizes the current underlying data in the database into a set of vectors, called ***DB states***, which 'compress' the whole database. Both query and data featurizations are of low space overhead and can be efficiently computed. On top of the DB states and query featurizations, ALECE builds a neural network based model to create reasonable connections between them. The model integrates the information of the DB states and query featurization, and feeds them into a feed-forward regression neural network to make the estimates. Roughly speaking, ALECE first learns to assign different weights to the raw DB states, with each weight showing the correlation between two DB states. This correlation is an useful distribution information to build suitable mapping between the caridinality of an SQL query with the underlying data. Then, the raw DB states are mapped into another set of vectors Z which are the weighted combinations of the raw DB states. Thus, the mapped vectors Z extract useful information and better represent the underlying data. ALECE also learns an another weight for each mapped vector z_i and the query featurization q to measure the influence of z_i on q. Then the weighted combination of Z is a convolution of the DB states and the query featurization. The combination vector is finally used to generate the cardinality estimate c. Fig. 1 depicts ALECE in the context of DBMS's query execution.

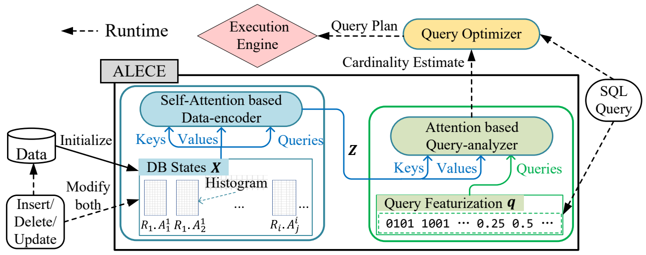


Figure 1: ALECE-based Query Execution.

Our ALECE's design is challenged by two critical issues. First, we need to build the DB states suitable for dealing with data changes in dynamic workloads, and meanwhile make it efficient to access them. To this end, we propose a simple yet effective data featurization approach which is a good approximation of data distributions, sensitive to data changes, and can be computed efficiently. This approach constructs the DB states based on the histograms of each attribute of the relations in the database. Each time a record is inserted, deleted or updated, the DB states set makes reactions by only modifying a few vector elements relevant to the changed table. Also, the basic single attribute distributions and even joint distributions are covered by this set. These factors together make it possible to process dynamic workloads with our design of DB states. Besides, depending on the requirement of the distribution approximation precision, the number of bins of a histogram can be flexibly adapted. Moreover, other useful information relevant to underlying data can be barrier-freely integrated if needed.

Second, we need to extract the implicit relevance between SQL queries and corresponding DB states, and make the information helpful for cardinality estimation. To this end, we adopt the attention mechanism [7, 31, 51] in our model to draw global dependencies between SQL queries and underlying data. The attention mechanism is widely used in a variety of tasks including question answering [26, 48]. Generally, it simulates the process of selection from a set using an attention function that takes as input two main components: a set of queries and a set of key-value pairs. It figures out in an individualized manner which parts of the data play more important roles for different queries, assigns higher weights to the more important and relevant keys for each query, and outputs the combination of the weighted values. Unlike those concepts in a database, a query in attentions is a specific element for which we need to learn a representation, the role of keys is to respond more or less to the query, and the values are used to compose an answer. Nevertheless, the selection process exactly matches our settings where the SQL queries and underlying data are analogy of the queries and key-value pairs in attentions, respectively.

There are two modules in our ALECE where the attentions are used in different ways. On the one hand, the 'data-encoder' module uses a self-attention whose inputs of queries, keys and values all come from the DB states. The self-attention allows the DB states, which correspond to different attributes, to interact with each other. By using the self-attention, the data-encoder module learns the implicit joint distribution information among attributes and computes a smarter representation of the underlying data. On the other hand, in the 'query-analyzer' module, the query set of the attention is exactly a set covering only one featurization vector of a SQL query, while the keys and values come from the output of the data-encoder module. The query-analyzer module outputs a fixed-dimensional 'answering' vector integrating the information from the query and data representations. We then use a simple linear regression model to map the answering vector to a cardinality estimate.

Compared to the state-of-the-art cardinality estimation methods, ALECE is able to make more reasonable use of both queries and underlying data. With the help of the two attentions, it answers the questions that 'which parts of data should a SQL query pays more attention to?' and 'how to find the more important data?' A SQL query usually focuses on some local parts of selected attributes. Also, the join conditions make particular tuples contribute more to the cardinality. Moreover, ALECE is able to adapt to dynamic workloads. In practice, learned models need to be trained with past queries and corresponding DB states, and estimate cardinalities for future queries. The performance of existing query-driven models often dramatically degrades when making predictions on a dynamic database. In contrast, ALECE can make immediate and suitable reactions to data changes by modifying the DB states, and learn an appropriate but implicit mapping between the true cardinality and the query featurization accompanied with the corresponding DB states. Our experimental results show that ALECE is able to make accurate estimates even when the distribution of the underlying data changes. Thus, ALECE is less sensitive to data changes.

To evaluate the effect of ALECE in end-to-end query execution, we extended an existing work [23] to an improved benchmark. It is able to correctly generate sub-queries for queries on workloads, feed the corresponding cardinality estimation results through external methods to PostgreSQL's query optimizer, and make direct and fair comparison among different methods on query execution time.

In our evaluation, ALECE achieves the best cardinality estimation performance on various dynamic workloads. Experimental results show ALECE improves the average end-to-end query time by up to 2.7× faster on the benchmark workload, very close to the optimal results acquired by using true cardinalities. This demonstrates that our ALECE makes more accurate cardinality estimates and helps the query optimizer find better query plans.

We make the following major contributions in this paper:

- We propose necessary principles for a method to featurize the underlying database data and SPJ queries. Accordingly, we design a featurization schema to losslessly featurize a SPJ query and make a reasonable compression of the data. The featurizations can be efficiently updated to support dynamic workloads.
- Based on the featurizations of queries and data, we propose an attention based learned cardinality estimator ALECE, together with detailed analyses.
- ALECE is designed to be a 'whitebox' which not only gives estimates but also clear rationale to integrate the SPJ queries and underlying data together in processing dynamic workloads.
- We experimentally validate ALECE 's advantages over more than half dozen representative alternatives on real datasets based dynamic and static workloads.

The rest of the paper is organized as follows. Section 2 gives the preliminaries. Section 3 presents the featurizations of data and queries. Section 4 elaborates on ALECE, followed by an analysis of it in Section 5. Section 6 reports the experimental studies. Section 7 reviews the related work. Section 8 concludes the paper and points out the future work. In addition, due to space limit, we give more experimental results and analyses in an appendix [1].

2 PRELIMINARIES AND AN OVERVIEW

This section presents our research problem and gives an overview of ALECE. Table 1 lists important notations used in the paper.

Table 1: Notations

R_i	A relation in the database
A^i_j	The jth attribute of the relation R_i
N	The number of relations in the database
T	The number of all attributes in the database
$\boldsymbol{X} = \{\boldsymbol{x}_i\}_{i=1}^T$	The set of data featurizations (a.k.a. DB states)
d_x	The number of histogram bins (dimensionality) for a DB state
$\boldsymbol{q} = \langle \boldsymbol{q}_J, \boldsymbol{q}_F \rangle$	A SQL query and its vectorized featurization
d_q	The dimension of a query featurization vector
n_{enc}	The number of attention layers in the data-encoder module
n_{ana}	The number of attention layers in the query-analyzer module
$\boldsymbol{K}, \boldsymbol{V}, \boldsymbol{Q}$	The input keys/values/queries of an attention function

2.1 Cardinality Estimation Problem

Suppose a database $\mathbf{D}$ has a set of relations $\{R_1, \cdots, R_N\}$. A relation R_i has n_i attributes, *i.e.*, $R_i = (A^i_1, \cdots, A^i_{n_i})$. Each attribute A^i_j can be either *categorical* or *numerical*: the domain of a categorical attribute is a finite set and can be 1-to-1 mapped to an integer set $\{1, \cdots, \mathsf{max}^i_j\}$; the domain of a numerical one is $[\mathsf{min}^i_j, \mathsf{max}^i_j]$.

Problem Formulation. Given a SQL query $\boldsymbol{q}$ and a dynamic database $\mathbf{D}$, we want to estimate the *cardinality of* $\boldsymbol{q}$, denoted as $\mathsf{c}(\boldsymbol{q}, \mathbf{D})$, *i.e.*, the number of resulting tuples when $\boldsymbol{q}$ is executed on $\mathbf{D}$.

In this paper, we focus on select-project-join SQL queries with conjunctive filter predicates; the cardinality $\mathsf{c}(\boldsymbol{q}, \mathbf{D})$ is the number of tuples after joins and filters, as the following counting query:

$$\mathsf{c}(\boldsymbol{q}, \mathbf{D}) : \texttt{SELECT COUNT}(*) \texttt{ FROM } R_{i_1}, \cdots, R_{i_n} \quad (1)$$
$$\texttt{WHERE join predicates } \boldsymbol{J} \texttt{ AND filter predicates } \boldsymbol{F}$$

where $\boldsymbol{q}$ involves n relations $R_{i_1}, \cdots, R_{i_n}$, with a set of join predicates $\boldsymbol{J}$ which is a conjunction of join conditions each in the form of "$R_i.A^i_x = R_j.A^j_y$", and a set of filter predicates $\boldsymbol{F}$. This formulation allows us to support not only PK-FK joins but also more general joins by specifying join predicates on pairs of joinable attributes (which may or may not be primary/foreign keys) in $\boldsymbol{J}$. A filter predicate is an relational expression in the form of "$R_i.A^i_j$ op const" where op $\in \{<, \leq, >, \geq, =\}$ and const is a fixed value. In the query $\boldsymbol{q}$, an attribute can appear in a join or a filter predicate, or both. The support for LIKE predicates is left for future work.

ALECE in the Optimization of SPJ Queries. The estimation results for SQL queries in Format (1) are able to to provide support for the query optimizer on widely-used **SPJ** (select-project-join) queries with **conjunctive** condition in the following format:

$$\boldsymbol{q}_{\mathsf{SPJ}} : \texttt{SELECT AGG}_1, \cdots \texttt{AGG}_m \texttt{ FROM } R_{i_1}, \cdots, R_{i_n} \quad (2)$$
$$\texttt{WHERE join predicates } \boldsymbol{J} \texttt{ AND filter predicates } \boldsymbol{F}$$
$$\texttt{ORDER BY attribute\_set\_1 GROUP BY attribute\_set\_2}$$

where AGG is an aggregate function over one or multiple attributes which can be COUNT, AVG, MIN and MAX, *etc*, or can be simply omitted. The join predicates set $\boldsymbol{J}$ and filter predicate set $\boldsymbol{F}$ carry the same meanings with that in Format (1).

To obtain the execution plan for a given SPJ query $\boldsymbol{q}_{\mathsf{SPJ}}$, the query optimizer of the modern DBMS like PostgreSQL will first decompose $\boldsymbol{q}_{\mathsf{SPJ}}$ into a series of sub-queries in a fixed order [23]. Then, it needs to estimate the cardinalities of these sub-queries. Thus, a cardinality estimator is supposed to focus on the sub-queries of $\boldsymbol{q}_{\mathsf{SPJ}}$ instead of $\boldsymbol{q}_{\mathsf{SPJ}}$ itself. Usually, the formats of the sub-queries are simpler than that of the original query. Considering these factors, we focus on the **cardinality estimation for the sub-queries of $\boldsymbol{q}_{\mathsf{SPJ}}$ which are in Format (1)**. Furthermore, ALECE is also helpful to the optimization of a SQL query $\boldsymbol{q}_{\mathsf{C}}$ more complex than the SPJ ones as long as $\boldsymbol{q}_{\mathsf{C}}$ has sub-queries following Format (1).

Estimation Model on Dynamic Workloads. In reality, the data in a DBMS is seldom static but most often continuously changes. Thus, it is beneficial to design cardinality estimators having the ability to make accurate estimates on a dynamic database. In this work, we define a 'dynamic workload' as a sequence of SQL statements including queries, inserts, deletes and updates. In other words, the associated underlying data changes for the queries in a dynamic workload. To satisfy this difficult requirement, we train a model ALECE which decouples data and queries and is able to make estimates for queries on dynamic workloads. It is worth emphasizing that we allow the database $\mathbf{D}$ to be varying as part of the input. In addition, our ALECE does not need the whole database data in its estimations. Instead, it adopts an efficient method to make a 'compression' of the dynamic database, namely 'DB states', and updates it when data manipulation statements are executed. The details are given in Section 3.1. Although varying underlying data is allowed, we assume a static database schema. The support for dynamic schemata is left for future work.

2.2 Overview of ALECE

Our ALECE's composition and its role in query execution is shown in Fig. 1. ALECE consists of two modules: the data-encoder module adopts the self-attention to figure out the links among all attributes and to learn their join distribution information, whereas the data-query attention in the query-analyzer module focuses on discovering which parts of the data-encoder's output have tighter relations with the SQL queries. We decouple the SQL queries and underlying data in ALECE to make it possible to create a reasonable connection between the true cardinality and the SQL query on a dynamic database. Our ALECE is both data-driven and query-driven. It requires a process of offline training to make online estimates.

Offline Training. Training ALECE needs a dataset of queries and their true cardinalities, as well as the corresponding database information when these queries are executed. Usually, the training dataset can be accessed through collecting the feedback of the historical queries executed on a dynamic database for a period of time. We start from featurizing the initial database and generate a set of fixed-dimensional vectors, namely 'DB states' (Section 3.1). Later, the come-in statements are sequentially processed. For the insert, delete and update statements, we modify the DB states accordingly. When a query comes in, it will also be featurized into a fixed-dimensional vector (Section 3.2). The query featurization and the current DB states will be packed up together as an element of the training data while the true cardinality is used as the label. Next, ALECE is trained with end-to-end gradient descent methods.

Online Estimation. A well-trained ALECE can make online estimation on both static and dynamic workloads. Similarly, we need to featurize the queries and the corresponding database data, feed the featurizations into ALECE and get the estimates. When the testing workload is static, *i.e.,* it contains no data update statements, the DB states are constant. Otherwise, the DB states will continuously change and the latest featurization will always be used.

3 FEATURIZATIONS OF DATA AND QUERIES

The underlying database data and SQL queries are required to be featurized numerically such that our ALECE can deal with. Any featurization method is able to be flexibly adopted by ALECE as long as it satisfies some principles. First, the underlying data needs to be featurized into a set of fixed-dimensional vectors covering enough distribution information. Second, any SPJ query should be losslessly mapped to a fixed-dimensional vector such that ALECE could better understand it. Also, to effectively support the dynamic workloads, the featurization method is supposed to be efficient and of low storage overhead. Following these principles, we propose an method of featurizing the database data and SQL queries numerically. The details are given in Section 3.1 and 3.2, respectively. Moreover, Section 3.3 discusses the properties of our featurization method and how the featurizations help process dynamic workloads. It is noteworthy that our featurization method itself does not have obvious technical novelty. However, it is specifically designed for our ALECE and perfectly fit the requirements of ALECE's inputs.

3.1 Data Featurization

In our settings, the data featurization $\boldsymbol{X}$, also known as the 'DB states', is a compression of the whole database, which can roughly describe the data of each attribute and the relationships among them. Our ALECE requires the data featurization to be a set of vectors of the same dimension. Here we use the set of histograms for each attribute as the DB states, *i.e.,* $\boldsymbol{X} = \{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_T\}$ where $\boldsymbol{x}_i$ is the histogram of the ith attribute and $T = \sum_{i=1}^{N} n_i$ is the number of all attributes in the database. How to order the T attributes will be introduced in Section 3.2. This featurization method is simple but powerful and we can efficiently access and update the histograms.

In particular, the values of categorical attributes are first converted to consecutive integers numbered from 1. Given an attribute A_i, we use $\mathsf{dom}(A_i)$ to denote its domain or the converted integer set if A_i is categorical. It is easy to show that $\mathsf{dom}(A_i) \subseteq \mathsf{D}(A_i) = [l, u)$ where $l = \inf \mathsf{Dom}(A_i)$ and $u = \sup \mathsf{Dom}(A_i) + \epsilon$ with $\epsilon \to 0^+$. Then, given a time stamp t and the database data at t, we create a $\boldsymbol{d_x}$-bin-histogram for each A_i. In particular, let $a = \frac{u-l}{d_x}$ and β_j be the number of A_i's values in $[l + (j-1) \cdot a, l + j \cdot a)$ for $1 \le j \le d_x$, the histogram $\boldsymbol{x}_i$ for attribute A_i could be easily accessed with $\boldsymbol{x}_i = [\beta_1, \cdots, \beta_{d_x}]$. Our DB states at time t is simply a set of T elements each being a d_x-dim histogram vector. In practice, each β_i in all histogram vectors will be scaled to the range [0, 1] through a suitable affine transformation. The value of d_x can be flexibly modified according to the complexity of the data distribution. Apparently, a larger d_x will make the data featurization capture more distribution information among the attributes but result in extra time and storage overhead. Usually, when strong correlations exist among attributes, a larger value tends to be used for d_x.

3.2 Query Featurization

Following and extending the existing work [54, 63], we featurize a SQL query $\boldsymbol{q}$ into a fixed length vector $\boldsymbol{q}$[1]. It is a simple concatenation of two separately generated parts $\boldsymbol{q_J}$ and $\boldsymbol{q_F}$, which featurize the join predicates $\boldsymbol{J}$ and filter predicates $\boldsymbol{F}$, respectively.

Join featurization. For the N relations numbered from 1 to N, we use $m_1 = \lceil \log_2(N+1) \rceil$ bits to featurize the id of each relation. Similarly, $m_2 = \lceil \log_2(n_{max} + 1) \rceil$ bits can featurize the ids of all attributes in any relation, where $n_{max} = \max(\{n_1, \cdots, n_N\})$ is the maximum number of attributes in a relation. Thus, any attribute $R_i.A_j^i$ can be uniquely identified with a binary vector of dimension $m = m_1 + m_2$. The first m_1-dimensional and the last m_2-dimensional sub-vectors identify the relation and the attribute, respectively. Then a join predicate P is featurized by a $2m$-dim binary vector $E_J(P)$ with the first and second half sub-vectors refer to the left and right hand side of P, respectively.

Suppose there are Δ possible join patterns, the join featurization $\boldsymbol{q_J}$ of a SQL query $\boldsymbol{q}$ is a $(2m \cdot \Delta)$-dim binary vector, containing Δ $2m$-dim sub-vectors, indicating which join patterns $\boldsymbol{q}$ covers and featurizing their referred attributes. If the join predicates of $\boldsymbol{q}$ contain the ith join pattern P_i, the ith sub-vector of $\boldsymbol{q_J}$ equals to $E_J(P_i)$. Otherwise, this sub-vector is set to zero. Unlike existing work [63] that simply featurizes whether each join condition appears in the query, our featurization way is more compact and incorporates more information about the joins, which is helpful to ALECE.

In practice, we define the attributes in the left and right hand side of a join predicate are 'equivalent' and find all equivalence classes in $\boldsymbol{J}$. Afterwords, we re-organize the join predicates based on the equivalence classes. Given an equivalence class $C = [A_1, \cdots, A_t]$ containing t attributes sorted by their m-dim featurizations, we re-create $(t-1)$ join predicates with the ith one to be $A_i = A_{i+1}$. By doing this to each equivalence class and packing the corresponding join predicates together, we generate a new join predicate set $\boldsymbol{J'}$. The join featurization is actually performed with $\boldsymbol{J'}$ instead of $\boldsymbol{J}$. In this way, two equivalent join predicate sets in explicitly different forms will be featurized to be the same. For example, in the following formula, join predicate set $\boldsymbol{J_1}$ and $\boldsymbol{J_2}$ both have two same equivalence classes. They will be converted to another set $\boldsymbol{J'}$.

$$
\begin{aligned}
\boldsymbol{J_1} &: \left(A_1^1 = A_1^2 \text{ and } A_1^2 = A_1^3\right) \text{ and } \left(A_3^3 = A_2^2\right) \\
\boldsymbol{J_2} &: \left(A_1^1 = A_1^2 \text{ and } A_1^3 = A_1^2\right) \text{ and } \left(A_2^2 = A_3^3\right) \\
\boldsymbol{J'} &: \underbrace{\left(A_1^1 = A_1^2 \text{ and } A_1^2 = A_1^3\right)}_{\text{Equi-class-1}} \text{ and } \underbrace{\left(A_2^2 = A_3^3\right)}_{\text{Equi-class-2}}
\end{aligned}
$$

Filter featurization. We sort all attributes according to their m-dim featurizations and use A_k to denote the kth one among T attributes. Without loss of generality, we assume $\mathsf{D}(A_k)$ is $[0, 1)$ for each attribute A_k. Thus, the product space $\mathsf{D}(A_1) \times \cdots \mathsf{D}(A_k) \times \cdots \times \mathsf{D}(A_T) = [0, 1)^T$. Apparently, the filter predicates $\boldsymbol{F}$ are equivalent to a hyper-rectangle $[l_1, u_1) \times \cdots [l_T, u_T)$ which is a subset of $[0, 1)^T$. In particular, for any filter condition on the attribute A_k, we convert it into an equivalent one in the form like $\sigma_{lb \le A_k < ub}$. Then, the values of l_k and u_k are set to lb and ub, respectively. Specifically,

$$
\begin{aligned}
&(lb \le A_k) \sim (lb \le A_k < 1), \; (lb < A_k) \sim (lb - \epsilon \le A_k < 1), \\
&(A_k < ub) \sim (0 \le A_k < ub), \; (A_k \le ub) \sim (0 \le A_k < ub + \epsilon) \\
&(A_k = x) \sim (x \le A_k < x + \epsilon), \text{ where } \epsilon \to 0^+.
\end{aligned}
$$

Above, $\sim$ denotes the equivalence operator.

Accordingly, the featurization of filter predicates $\boldsymbol{q_F}$ is a $2T$-dim vector composed of the boundary points of the search hyper-rectangle, *i.e.,* $E_f = [l_1, u_1, l_2, u_2, \cdots, l_T, u_T]$. In practice, each l_i and u_i will be normalized to [0, 1].

[1]Without ambiguity, $\boldsymbol{q}$ denotes both a SQL query and its vectorized featurization.

Concatenation. By concatenating $\boldsymbol{q_J}$ and $\boldsymbol{q_F}$, we get the d_q-dim featurization vector of SQL query $\boldsymbol{q}$. Fig. 2 shows an example.

SQL query $\boldsymbol{q}$: SELECT $*$ FROM $\mathrm{R}_1, \mathrm{R}_2, \mathrm{R}_3, \cdots$ WHERE

$R_1.A_1^1 = R_2.A_1^2$ AND $R_2.A_2^2 = R_3.A_3^3$ AND $\cdots$

AND $0.25 \le R_1.A_1^1 < 0.5$ AND $\cdots$

$$\text{Featurization } \boldsymbol{q}: \overbrace{\underbrace{0101\ 1001}_{R_1.A_1^1 = R_2.A_1^2}\ \underbrace{1010\ 1111}_{R_2.A_2^2 = R_3.A_3^3}\ \cdots}^{\text{join featurization } \boldsymbol{q_J}}\ \overbrace{\underbrace{0.25\ 0.5}_{0.25 \le A_1 < 0.5}\ \underbrace{\cdots}_{l_k \le A_k < u_k}}^{\text{filter featurization } \boldsymbol{q_F}}$$

with $m_1 = m_2 = 2$

Figure 2: An example of query featurization.

3.3 Discussions

3.3.1 Featurization properties. As we claimed, our way of featurizing database data and SQL queries has the following properties:
1) Efficiency. Building the data featurizations of a static database requires looking over each relation once only. An insert/delete/update statement only influences one relation and modifying the relevant histograms takes $O(m)$ time where m is the number of the records involved. The time complexity of featurizing a SQL query is $O(|\boldsymbol{J}| + |\boldsymbol{F}|)$, which is small and can often be ignored.
2) Low space overhead. The DB states only contain $T \cdot d_x$ float numbers. In practice, we usually set d_x smaller than 100. The dimensionality of a query featurization is $2m \cdot \Delta + 2T$. Usually, the value of m is small. In our experiments, m is smaller than 10. Our way of re-generating join predicates based on equivalence classes will usually reduce Δ, the number of possible joins. Also, the joins are basically performed on attributes with primary keys. According to our observations, Δ is usually less than N^2 and featurizing a query on a 8-relation database requires less than 1000 float numbers.
3) Stability. The dimensions of data and query featurizations are fixed, no matter how the database or queries change. This ensures the featurizations can be easily processed with learned models.

3.3.2 Why our featurizations work on dynamic workloads? It is noteworthy that no matter how the underlying data changes, the featurization of a given query will not change. In contrast, the DB states will vary with the change of the data in the database. It is a compression of the whole database and able to catch the distribution characteristics of each attribute. Our model takes both featurizations as input and is able to 'convolute' the query featurizations with the DB states. Thus, when the data changes, it can react properly and give different predictions for the same query with different DB states. The experiments in Section 6 show that our model outperforms other state-of-the-art methods on both static and dynamic workloads.

4 DESIGN OF ALECE

Given the DB states set $\boldsymbol{X}$ and a SQL query featurization $\boldsymbol{q}$, our ALECE can reasonably discover the implicit relations between them that are required for cardinality estimation. The mystery behind lies in the attention mechanisms [51] twice used in our ALECE. In this section, we will review the motivation of ALECE's design, introduce the details of ALECE including its two modules processing $\boldsymbol{X}$ and $\boldsymbol{q}$, respectively, the training process of ALECE, and make a deep discussion on the properties of ALECE.

4.1 Motivations and ALECE Overview

An attention function maps a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors [51]. Here, the concepts of key-value pairs and queries can be analogous to retrieval systems. Take a user's search behavior on an e-commerce platform like Amazon as an analogy. When the search engine receives a query (the text in the search bar), it maps the query against a set of keys (item names, tags and descriptions, *etc.*) associated with values (candidate items) in the database and outputs the best matched items. The output is a weighted sum of the values, where each weight is computed by a compatibility function that measures the relevance between the query and keys.

The idea behind the attention mechanism is to encode the input key-value pairs set, and utilize the most relevant parts of the keys, associated with values, with the query in a flexible manner. Through a weighted combination of all encoded input vectors, this mechanism 'answers' the query with the most relevant vectors getting the highest weights. This idea perfectly fits in our research problem as a SQL query's featurization $\boldsymbol{q}$ is a natural query vector. Besides, the DB states $\boldsymbol{X}$ can be seen as the item information in the above example and be used as the keys and values in the attention functions. Thus, we extend this idea and design ALECE, which takes full advantage of the attention mechanism to accurately estimate cardinalities of SQL queries on dynamic workloads. Fig. 3 illustrates the structure of ALECE.

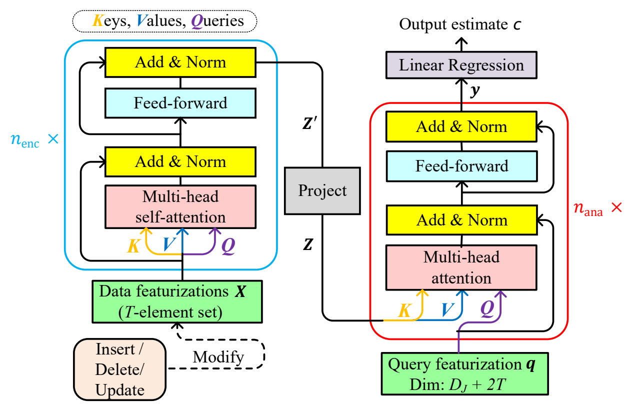


Figure 3: ALECE structure.

Our attention-based model ALECE is designed like a soft lookup table. It fetches information from a better representative of the set of DB states $\boldsymbol{X}$ using features from the transformation of $\boldsymbol{q}$ as indices. It is composed of two modules. The left **data-encoder** module maps $\boldsymbol{X} = \{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_T\}$ to another set of representations $Z = \{z_1, \cdots, z_T\}$. It learns the implicit joint distribution information among all attributes by computing the relevance between any pairs of DB states through a stack of self-attention layers. This distribution information is embodied in the set Z. Given Z, the right **query-analyzer** module adopts another stack of attention layers to measure the relevance between the query featurization $\boldsymbol{q}$ and Z, and generates an 'answer' vector $\boldsymbol{y}$. Finally, vector $\boldsymbol{y}$ is fed into a linear regression layer and $\boldsymbol{q}$'s cardinality given DB states $\boldsymbol{X}$ is estimated and returned.

4.2 Attentions in ALECE

Attention background. Before showing the data-encoder and

Table 2: The overall picture of ALECE's two modules, given DB states set X and SQL query featurization $\boldsymbol{q}$

Data-encoder (Self-Attention)			Query-analyzer		
Input source	***What to learn?***	***Output***	***Input source***	***What to learn?***	***Output***
Keys: $\boldsymbol{X}$ Values: $\boldsymbol{X}$ Queries: $\boldsymbol{X}$	Relevance among the DB states $\boldsymbol{X}$, and thus the joint distribution information of multiple attributes.	Another vector set $\boldsymbol{Z}$ covering joint distribution information of attributes	Keys: $\boldsymbol{Z}$ Values: $\boldsymbol{Z}$ Queries: $\boldsymbol{q}$	Relevance between $\boldsymbol{q}$ with the elements in $\boldsymbol{Z}$, showing which parts of data are more important.	Final 'answering' vector $\boldsymbol{y}$ which will be directly turned into the cardinality estimate c.

query-analyzer modules in ALECE, it is worth briefly introducing the attention mechanism. In neural networks, attention is a technique that is meant to mimic cognitive attention. Its motivation behind is that the network should devote more focus to the important parts of the data instead of treating all data equally. It uses an attention function to discover which parts of the data should be emphasized. The function maps a query and a set of key-value pairs to an output, which is a weighted sum of the values. Usually, the function computes the similarity (relevance) between each pair of query and key with some metric, and uses it to produce the weight assigned to the corresponding value.

Our ALECE uses the 'Scaled Dot-Product Attention' [51], namely Attn, as the attention functions in the attention layers from both the data-encoder and query-analyzer modules. The input keys and values are packed together into matrices $\boldsymbol{K}$ and $\boldsymbol{V}$, of dimension $S \times d_k$ and $S \times d_v$, respectively, where S denotes the size of the original key-value pair set.[2] The d_k-dim query vector is converted to a $1 \times d_k$ matrix $\boldsymbol{Q}$. The function Attn computes the dot products of the given query $\boldsymbol{Q}$ with all keys $\boldsymbol{K}$, divides each by $\sqrt{d_k}$, and applies a softmax function to obtain the weights on the values $\boldsymbol{V}$:

$$\mathsf{Attn}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \mathrm{softmax}\Big(\frac{\boldsymbol{Q} \cdot \boldsymbol{K}^T}{\sqrt{d_k}}\Big)\boldsymbol{V}$$

The dot product above is used as the similarity metric. To prevent the dot products from growing too large such that the gradients of the softmax function 'vanishing', we divide the dot product by the factor $\sqrt{d_k}$. In practice, the function Attn is computed with batches of queries simultaneously to improve the efficiency. Before feeding the input to Attn, we adopt a multi-head projection mechanism [51] to first project the queries, keys and values multiple times with h different linear projections. This operation is able to enhance the representation ability of ALECE. More details are given in Section A in the appendix [1].

Attentions in ALECE's two modules. The multi-head attention layer appears twice in our ALECE. The first is a self-attention layer in the data-encoder module. It is fed with the same query, key and value matrix $\boldsymbol{X}$ of dimension $T \times d_x$, and outputs a matrix $\boldsymbol{Z'} \in \mathbb{R}^{T \times d_x}$ that will be later projected to another matrix $\boldsymbol{Z} \in \mathbb{R}^{T \times d_q}$. Then, $\boldsymbol{Z}$ is used as the key and value matrix of the other attention layer in the query-analyzer module, where the input query set contains only one SQL query featurization vector. It is noteworthy that the "queries" set in the self-attention layers does not come from the SQL query. In addition, we do not need to order the vectors in the DB states set. Thus, unlike transformer-like models to process sequences [8, 14], our ALECE does not need the positional encodings [51] or positional embeddings [17] of the attention layers' inputs.

Table 2 states the input sources and output of the two modules' attention layers, and what each module is able to learn. The rationale behind them is given in the following two subsections.

[2] In the rest of the paper, we regard a set of vectors as a matrix. In other words, we do not distinguish a set of key/value/query vectors with a key/value/query matrix.

4.3 Data-encoder

Suppose a random variable V's value is taken randomly with an equal possibility from the possible values of an attribute, the corresponding histogram can be regarded as the rough distribution function of V. Thus, the DB states set consists of a series of 'marginal' distributions describing single attributes. However, SQL queries tend to cover join predicates which requires to know the join distribution information among multiple attributes. Usually, the distributions of different attribute values are not independent. It is difficult and impractical to directly access the joint distribution from the marginal distributions. To address this important issue, we design the data-encoder module, taking use of attentions, to establish a bridge between the marginal distributions and joint distribution.

The data-encoder takes as input the DB states set $\boldsymbol{X}$ and feeds it into a $\boldsymbol{n}_{\mathbf{enc}}$-stacked layers. Each layer is identical and has two sub-layers. The first is the multi-head self-attention sub-layer N_{sa} which takes the same inputs of keys, values and queries—three matrices equal to $\boldsymbol{X}$ or the output of the last layer, and outputs another T-element set $\tilde{\boldsymbol{Z}}_i$. On top of N_{sa}, the feed-forward sub-layer FF uses stacked fully connected networks and nonlinear activation functions, *e.g.,* ReLU, to map $\tilde{\boldsymbol{Z}}_i$ into the data representation set $\boldsymbol{Z}'_i$. Besides, to address the degradation problem and ease the model training, we employ a residual connection [25] around each sub-layer, followed by layer normalization [6], following the settings in [51]. Thus, the output of each sub-layer is LayerNorm($\boldsymbol{V}$ + sub-layer($\boldsymbol{V}$)). The sub-layer is either N_{sa} or FF. The analytic expression of the output representations $\boldsymbol{Z'}$ is described as follows.

$$\boldsymbol{Z'} = \boldsymbol{Z}'_{n_{\text{enc}}}, \text{ where } \boldsymbol{Z}'_0 = \boldsymbol{X}, \text{ and } \forall 1 \le i \le n_{\text{enc}},$$

$$\tilde{\boldsymbol{Z}}_i = \mathsf{LayerNorm}\big(\boldsymbol{Z}'_{i-1} + \mathsf{N}_{\mathsf{sa}}(\overbrace{\boldsymbol{Z}'_{i-1}}^{\text{keys}}, \overbrace{\boldsymbol{Z}'_{i-1}}^{\text{values}}, \overbrace{\boldsymbol{Z}'_{i-1}}^{\text{queries}})\big),$$
$$\boldsymbol{Z}'_i = \mathsf{LayerNorm}(\tilde{\boldsymbol{Z}}_i + \mathsf{FF}(\tilde{\boldsymbol{Z}}_i)), \text{with } \mathsf{FF}(\boldsymbol{V}) = \mathrm{ReLU}(\boldsymbol{W}\boldsymbol{V} + \boldsymbol{b})$$

Finally, the output matrix $\boldsymbol{Z'}$ will be linearly projected to the representation matrix $\boldsymbol{Z} \in \mathbb{R}^{T \times d_q}$. This projection is to align the dimensionality of $\boldsymbol{Z}$ with that of the query featurization $\boldsymbol{q}$.

The keys, values and queries in the attention layers are the same set. They are either the DB states set $\boldsymbol{X}$ or the output of the previous stacked layer. This setting makes each element in the output set of a layer attend to all outputs in the previous layer and thus attend to all DB states. More importantly, the self-attention layer quantitatively 'calculates' the relevance between a pair of elements from any two histograms. It is noteworthy that each element of a histogram describes the local distribution of an attribute. Thus, the data-encoder module is able to more organically discover the implicit connections between any pair of DB states, and usually could exhibit behavior related to the joint distributions of multiple attributes in the output set. Compared to other neural network architectures like multilayer perceptron (MLP) [24], self-attention could yield more interpretability and have higher representative abilities. Through self-attention layers, we create links among all

DB states, or equivalently, all attributes in the database. After fine-tuning the parameters of the self-attention and feed-forward layers, the relationship information helpful to the cardinality estimation task is implicitly covered and encoded into the output Z of the data-encoder module. This information will be processed and utilized in the query-analyzer module.

4.4 Query-analyzer

The query-analyzer module attempts to discover and measure the relevance, through attention layers, between the SQL query and each element of Z, the output of the data-encoder module covering joint distribution information among attributes.

This module is also a stacked structure composed of $\boldsymbol{n_{\text{ana}}}$ identical layers. Similar to the data-encoder module, each layer here is composed of a multi-head attention sub-layer and a feed-forward fully connected sub-layer. Also, residual connections are employed around each sub-layer, followed by layer normalization. Unlike the data-encoder module, the input sets of key-value pairs and queries to the 'data-query' attention sub-layer here are not from the same place. We use Z, the output of the data-encoder module, as both the input key and value matrices of the attention sub-layer, while the query set here comes from either the query featurization or the output of the previous layer. It is noteworthy that the input query set and the output set of each attention sub-layer have only one d_q-dim element.

The data-query attention sub-layer establishes a bridge between the queries and the data. It individualize each SQL query featurization and presents different 'answers' by enhancing the influences of some parts of the input key-value pair set Z, while diminishing other parts. Learning which part of the data is more important depends on the relations between queries and keys, and this is measured with the attention functions. Suppose a SQL query contains join predicate $R_1.A_1^1 = R_2.A_2^2$ and filter predicate $R_1.A_2^1 > 1$, and the attributes $R_1.A_1^1$, $R_2.A_2^2$ and $R_1.A_2^1$ are numbered i, j and k, respectively. The data-query attention sub-layer will pay more attention to the part of the vectors in the set Z that are relevant to $\boldsymbol{x}_i$, $\boldsymbol{x}_j$ and $\boldsymbol{x}_k$. The effect of particular attention can be realized through suitable parameters of different layers in both modules.

After accessing $\boldsymbol{y}$, the output of the final query-analyzer layer, we use a simple linear regression layer LR to calculate a scalar value as the cardinality estimate c. The process of accessing c with the input of Z and $\boldsymbol{q}$ is described as follows.

$$\mathrm{c} = \mathsf{LR}(\boldsymbol{y}), \text{ where } \boldsymbol{y} = \boldsymbol{y}_{n_{\text{ana}}}, \boldsymbol{y}_0 = \boldsymbol{q}, \text{ and } \forall 1 \le i \le n_{\text{ana}}, \quad (3)$$
$$\boldsymbol{y}_i' = \mathsf{LayerNorm}\big(\boldsymbol{y}_{i-1} + \mathsf{N}_{\text{dq}}(\overbrace{Z}^{\text{keys}}, \overbrace{Z}^{\text{values}}, \overbrace{\boldsymbol{y}_{i-1}}^{\text{queries}})\big),$$
$$\boldsymbol{y}_i = \mathsf{LayerNorm}(\boldsymbol{y}_i' + \mathsf{FF}(\boldsymbol{y}_i'))$$

4.5 Training of ALECE

Fine-tuning the parameters of ALECE requires a training dataset of which each element is a 3-tuple $(\boldsymbol{q}_i, \boldsymbol{X}_i, \mathrm{c}_i')$, where $\boldsymbol{q}_i$ is the query featurization of a SQL query $\boldsymbol{q}_i$ and $\boldsymbol{X}_i$ is the associated DB states set of the dynamic database. By executing $\boldsymbol{q}_i$ on the database of which the DB states set is $\boldsymbol{X}_i$, we will get the true cardinality c_i', which will be used as the label. In practice, we will take the logarithm of c_i' to make the range of the labels not too large. Collecting the training dataset is not difficult. Usually, we only need to collect the feedback of executed queries on a dynamic database. Then, we will get the training dataset with three lists $\mathcal{X} = [X_i]$, $Q = [\boldsymbol{q}_i]$ and $\mathbf{c'} = [\mathrm{c}_i']$. They will be splitted into batches to train our ALECE.

We use the mean-weighted-squared-error function MWSE taking input of the batch card predictions $\mathbf{c_b}$ and the true cards $\mathbf{c_b'}$ as well as their weights $\boldsymbol{w_b}$ with batch size B as the loss function, *i.e.,* $\mathsf{MWSE}(\mathbf{c_b}, \mathbf{c_b'}) = \frac{1}{B}\sum_{i=1}^{B} w_i(\mathrm{c}_i - \mathrm{c}_i')^2$. The parameters of the linear regression layer, the attention and feed-forward sub-layers in both modules are trained by gradient descent with batches in an end-to-end fashion. Here, the value w_i is proportional to that of $\log \mathrm{c}_i$. In particular, $w_i = \frac{\log \mathrm{c}_i}{\sum_j \log \mathrm{c}_j}$. We use the weight w_i in the loss function because it is usually beneficial to emphasize the queries with larger true cardinalities as their execution times tend to be longer. The procedure of training ALECE using the these lists $\mathcal{X}$, Q and $\mathbf{c'}$ is detailed in Algorithm 1.

Algorithm 1: TRAIN-ALECE

Input : Three data lists: $\mathcal{X} = [X_i]$, $Q = [\boldsymbol{q}_i]$, $\mathbf{c'} = [\mathrm{c}_i']$
The maximum number of training epochs M_e
Output : Well-trained ALECE
1 Compute the weight vector $\boldsymbol{w} = [w_i]$ according to $\mathbf{c'}$.
2 **for** $i \leftarrow 0$ **to** M_e **do**
3   `Train-Epoch`$(\mathcal{X}, Q, \mathbf{c'}, \boldsymbol{w})$
4   **if** ALECE performs better on validation set **then**
5   Assign current parameters to ALECE.
6 **return** ALECE whose parameters are fine-tuned.
7 **Function** `Train-Epoch`$(\mathcal{X}, Q, \mathbf{c'}, \boldsymbol{w})$**:**
8   Shuffle and split $\mathcal{X}, Q, \boldsymbol{c}, \boldsymbol{w}'$, generate n_B equal-size batches $[(\mathcal{X}_b, Q_b, \mathbf{c'}_b), \boldsymbol{w}_b]$, $b = 1, \cdots, n_B$.
9   **for** $b \leftarrow 0$ **to** n_B **do**
10   Compute the vector $\mathbf{c}_b$ with Eq (3).
11   Compute the MWSE loss given $\mathbf{c}_b$, $\mathbf{c'}_b$ and $\boldsymbol{w}_b$, and apply gradient decent to tune model parameters.

In practice, a training dataset is split into two parts: the first part is the three data lists used to fit the parameters in the learning process (lines 7-10); the second part is used as the validation set to choose best parameters for ALECE and avoid overfitting (lines 4-5). Also, we will apply the early stopping strategy [42], to stop the training process when the error on the validation set grows.

5 ANALYSIS OF ALECE

In this section, we discuss why ALECE is suitable for dynamic workloads and further analyze its properties.

5.1 ALECE on Dynamic Workloads

It is critical for databases to process dynamic workloads, *i.e.,* mixture of queries and data manipulation operations. This requires a DBMS' query optimizer to be able to accurately predict the cardinalities of the queries performed on a continuously changing database. To reduce the related overhead, the DBMS regularly maintains its cardinality estimators instead of modifying it immediately after each update. Between two model-maintaining time points, the cardinality estimator model is usually fixed but still expected to make accurate estimates.

The traditional Histogram-based cardinality estimators [13, 22, 41, 45] currently used in systems like PostgreSQL [21] make simple and often unreasonable assumptions, like the mutual independence of attributes. Their estimates are inaccurate, especially when the data is dynamic. This renders it difficult for the optimizer to choose good query plans. Existing learned cardinality estimation methods either do not support dynamic workloads [63] or need to completely re-build the model at model-maintaining time points [27, 61, 65]. This process is often too time-consuming to be feasible.

Compared to existing methods [27, 61, 65], the training of ALECE is efficient. Also, it is easily maintained: it can be incrementally updated at model-maintaining time points. Moreover, its architecture design is reasonable and suitable for processing dynamic workloads. It is noteworthy that the SQL queries and underlying data are decoupled in ALECE such that ALECE can learn something whenever the DB states $\boldsymbol{X}$ or the query featurization $\boldsymbol{q}$ in the training dataset changes. Thus, ALECE is able to avoid overfitting certain datasets. Instead, it applies the attention mechanisms on its two modules to learn the properties of the data distribution which 'generates' the underlying data, and how these properties influence the cardinality of a SQL query. It is able to reasonably approximate $\mathrm{c}(\boldsymbol{q}, \mathrm{DB})$, the true but implicit mapping among the queries, dynamic data and cardinalities. When either the query featurization or DB states, or both of them, get changed, ALECE can still output accurate results. Our experimental results indicate that ALECE achieves good performance even if there is distribution discrepancy between the training and test data.

A straightforward baseline without attentions. It is mentioned in Section 3.3 that the DB states will be dynamically modified when data is inserted to/deleted from the database. This renders it possible to use a stable well-trained model to predict the cardinalities of queries on a dynamic workload. As the dimensions of DB states and query featurizations are fixed, we can also adopt a straightforward method without attentions to process them. For example, for each pair of data featurizations $\boldsymbol{X}(t_i)$ and query featurization $\boldsymbol{q}_i$, we flatten the matrix $\boldsymbol{X}(t_i)$ into a vector and concatenate the vector with $\boldsymbol{q}_i$ to generate another vector $\boldsymbol{v}_i$. Subsequently, $\boldsymbol{v}_i$ and the associated layer are fed into a common supervised neural network like multilayer perceptron (MLP) [24]. However, a straightforward neural network like MLP is usually not powerful enough to discover the implicit relations between SQL queries and the underlying data. Its performance heavily relies on the similarity between the distributions of training and test datasets. When the workload is static, *i.e.,* the database data never changes, straightforward methods perform well. However, the cardinality estimator model needs to use the current training data it observes to make predictions for 'future' data. The distribution of future DB states may be highly different from that of the current data. In contrast, the application of the attention mechanisms in our ALECE make it possible to make accurate estimates even when the distributions of the underlying data change. Experimental results in Section 6 show the great advantages of ALECE over MLP on processing dynamic workloads.

5.2 Overhead of ALECE and Its Extension

Several other good properties make ALECE more practical and attractive as a cardinality estimation method in modern DBMS's query optimizers. First, the storage and training overhead of ALECE is affordable. On the one hand, the sizes of ALECE on three datasets are both smaller than 23 MB. On the other hand, training ALECE from scratch requires less than 12 minutes. Also, the estimation latency of ALECE is less than 11 ms. These overheads make ALECE able to process real world workloads.

Second, different from FLAT [65] and DeepDB [27], ALECE directly estimates the cardinality instead of the selectivity of a SQL query. To get the cardinality, the selectivity estimation methods usually needs to estimate the size of the corresponding join table with sampling-based techniques [64]. However, when multiple relations are involved, the statistical variance often becomes large and results in highly inaccurate estimates.

Last but more importantly, ALECE establishes a general framework that is not limited to cardinality estimation tasks. By replacing COUNT(∗) with with other aggregation functions, the special aggregate analytic queries of Format (2) can be further transformed to more general ones. As a matter of fact, ALECE can be easily extended to approximate the results of more general aggregate analytic queries. When processing queries with other kinds of aggregate functions, we only need to make slight modifications to the data and query featurizations, including featurizing the extra aggregate functions and optional GROUP BY clauses in the query featurizations, and incorporating more data description information specific to the aggregate queries. We left the support of the general aggregate analytic queries for future work.

6 EXPERIMENTAL STUDIES

This section reports the experiments that compare ALECE with selected alternatives. All mothods are implemented in Python 3.9 and evaluated on a Linux server with a 96-core Xeon(R) Platinum 8163 CPU and 376GB memory. The implementation of ALECE is open [2]. Due to space limit, we present additional experimental results and analyses in Section C in an appendix [1].

6.1 Experimental Settings

Datasets. We use two real datasets to evaluate all models.

- **STATS** contains 8 relations (*users, posts, postLinks, postHistory, comments, votes, badges, tags*) with 43 attributes [3]. There are 1,029,842 records in total. An existing open workload with 146 queries are marked as 'testing queries'. They are associated with 2,603 sub-queries. We created another 1000 different queries with sub-queries, which are used as the training and validation data.
- **Job-light** is a subset of the IMDB dataset [4]. It contains 6 relations (*cast_info, movie_info, movie_companies, movie_keyword, movie_info_idx, title*) with 14 attributes. There are 62,118,470 records in total. The testing query set contains 208 queries, associated with 3,248 sub-queries. Similarly, another 2,000 queries as well as their sub-queries are created for training the models.
- **TPCH** (1 GB) [5] is a widely-used benchmark dataset of a suite of business oriented relations (*customer, lineitem, nation, orders, part, partsupp, region and supplier*). We remove the 'comment' attribute from each relation and use the remaining 46 attributes. There are 8,661,245 records in total. We randomly create 123 testing queries and 1,554 testing sub-queries for evaluation. Another 2,000 queries and their sub-queries are used for training.

The join information among relations in these datasets are shown in Fig. 5 in the extended version [1]. All joins in the queries on the Job-light dataset are PK-FK joins, while the queries on the other datasets involve more complex many-to-many joins.

Dynamic workloads. For each dataset, we create three different dynamic workloads, each of which is a random mix of inserts, deletes, updates and query statements. These workloads are differentiated according to the ratios among the numbers of inserts, deletes and updates, and the distributions of the inserted records:

- **Insert-heavy:** #inserts : #deletes : #updates = 2 : 1 : 1.
- **Update-heavy:** #inserts : #deletes : #updates = 1 : 1 : 2.
- **Dist-shift** is a special Insert-heavy workload but having skewed distribution of the inserted records. In particular, the inserted records are selected intentionally such that their first attributes' values are the first 30% smallest among all data[3].

To generate a dynamic workload, we randomly select about $\frac{2}{3}$ of the records as the initial datasets to bulk load all the relations. Subsequently, the insert and update statements in a dynamic workload will insert the remaining $\frac{1}{3}$ of the records to the database, while the delete and the update statements will remove or change some records. For simplicity and clarity, we assume that each delete or update statement only influences one record. These statements are equally splitted into two parts: the former 'training' part and the latter 'evaluation' part. We make three copies of the training queries and their sub-queries and randomly spread them into the training part. Further, we count the total number of records in the database, denoted with Γ_0, after executing all the statements in the training part. Note that the true cardinalities of the queries in the training part are available to the query-driven methods and used as the labels in the training dataset.

The way of mixing testing queries with the data manipulation statements in the evaluation part are slightly different. For each testing query, we pack it up with its sub-queries together. Then, those packs are shuffled and randomly mixed with data update statements in the evaluation part. Also, we'd like to know how cardinality estimators perform when a certain amount of data in the database changes. Thus, we assume that when each testing query is executed, the associate **changing rate** ρ of the underlying database data is larger than a pre-defined threshold. Suppose the numbers of inserts, deletes and updates by time stamp t are $\mathtt{I_t}$, $\mathtt{D_t}$ and $\mathtt{U_t}$, respectively. The changing rate $\rho(t)$ is defined as $\rho(t) = \frac{\mathtt{I_t}+\mathtt{D_t}+\mathtt{U_t}}{\Gamma_0}$. When there is no ambiguity, t is usually omitted.

Both training and testing queries are randomly generated. In particular, to generate a query, we run a series of Bernoulli tests with p=0.5 to determine if a join condition like $R_1.A_1$=$R_2.A_2$ appears. The attributes appear in the filter predicates and their operators ($\le$, $\ge$ *etc*) are determined likewise. The right hand side values of the filter predicates are randomly sampled from the initial dataset.

Competitors. We include the following representative methods:

- **PG** is the simplest 1D histogram based cardinality estimation method used in PostgreSQL [21].
- **DeepDB** [27] is based on a Sum-Product-Network (SPN) [40]. It learns a pure data-driven model to capture the data's joint probability distribution. Following the same settings in [27], we set the RDC independence threshold to 0.3 and split each SPN node with at least 1% of the input data.
- **FLAT** [65] improves SPN based on factorize-sum-split-product network (FSPN) [60], a new graphical model, to adaptively model the joint probability density function (PDF) of attributes. It is also data-driven.
- **NeuroCard** [61] is a data-driven method, extending Naru [62] into the multi-table case, while Naru is a Deep Auto-Regression (DAR) [20] based cardinality estimation algorithm for a single table. The sampling size of the NeuroCard model is set to 8,000, following the settings in the paper.
- **FactorJoin** [57] combines classical join-histogram methods with learned single table cardinality estimates into a factor graph.
- **MLP** [44] is the baseline neural network based method introduced in Section 5.1.
- **MSCN** [33] is a multi-set convolutional network which adopts the information from both queries and data.
- **NNGP** [63] adopts the Neural Network Gaussian Process (NNGP) model [35] to learn from queries as well as their true cardinalities in a supervised manner.

In addition, we also include the comparison with the results generated by true cardinalities. This **Optimal** method measures the best performance a method can achieve.

These competitors are chosen because they have better overall performance over other statistical and learned cardinality estimators. The comparisons are reported in benchmark and evaluation papers [23, 49] and other existing works like [58, 61, 65]. Table 3 summarizes the properties of all methods. The better performance is indicated in bold.

Table 3: Properties of different methods

Method	Data-driven	Query-driven	Building time	Space overhead	Suitable for dynamic workload
PG [21]	✓	×	**small**	**low**	×
DeepDB [27]	✓	×	medium	medium	×
FLAT [65]	✓	×	large	high	×
NeuroCard [61]	✓	×	medium	medium	×
FactorJoin [57]	✓	×	medium	**small**	×
MLP [44]	×	✓	**small**	**low**	×
MSCN [33]	✓	✓	**small**	**small**	×
NNGP [63]	×	✓	**small**	medium	×
ALECE (ours)	✓	✓	**small**	medium	✓

Evaluation Metrics. We use three metrics to evaluate all methods:

- **E2E time** is the average query execution time by feeding the query optimizer the estimated cardinalities of sub-queries. Those estimation results are acquired using cardinality estimation methods. It is the most important evaluation metric as it directly connects to the query optimizer and objectively shows if a cardinality estimation method could help improve the query performance of a DBMS. This type of evaluation requires an improved benchmark over the existing work [23]. This benchmark can integrate the estimation results by an external method to the PostgreSQL. We show its details in Section B in the appendix [1].
- **P-error** [23] measures the gap between the optimal query plan and the generated plan based on the estimated cardinalities. In particular, given a query $\boldsymbol{q}$, a query plan P and the set $\mathbf{c}$ of estimated/true cardinalities of $\boldsymbol{q}$'s all sub-queries, the DBMS will output an estimated cost $C(P, \mathbf{c})$ with a cost function C. By feeding the query optimizer $\mathbf{c}^T$, the set of true cardinalities of $\boldsymbol{q}$'s all sub-queries, we can get the optimal query plan P^T for $\boldsymbol{q}$. Similarly, a cardinality estimation method A outputs for $\boldsymbol{q}$ a set $\mathbf{c}^E$ which results in another query plan P^E. Then the P-error of A and $\boldsymbol{q}$ is defined as follows:
$$\text{P-error}(\mathrm{A}, \boldsymbol{q}) = \frac{C(P^E, \mathbf{c}^T)}{C(P^T, \mathbf{c}^T)}$$
where the denominator $C(P^T, \mathbf{c}^T)$ is the optimal execution cost,

[3] Most primary keys are the first attributes.

Table 4: Overall performance of cardinality estimation models on dynamic workloads

Data	Model	Insert-heavy													Update-heavy								
		E2E Time(S)	Q-error				P-error				Size (MB)	Building Time(Min)	Latency (ms)	E2E Time(S)	Q-error				P-error			
			50%	90%	95%	99%	50%	90%	95%	99%					50%	90%	95%	99%	50%	90%	95%	99%
STATS	PG	7, 790	190	$1.4\cdot10^5$	$1.1\cdot10^6$	$1.8\cdot10^7$	2.60	25.44	41.23	243	-	-	-	4, 337	524	43, 096	$3.7\cdot10^5$	$1.1\cdot10^7$	1.78	19.52	29.01	178
	NeuroCard	>30, 000	17.37	1, 388	7, 402	$3.0\cdot10^5$	2.38	39.18	824	84, 415	90.76	23.69	32.42	22, 193	18.49	1, 252	6, 784	$3.1\cdot10^5$	2.10	31.75	347	11, 730
	FLAT	>30, 000	12.77	1, 979	12, 897	$8.6\cdot10^5$	2.22	70.08	2, 202	8, 738	210.33	53.77	53.77	24, 319	16.00	2, 376	17, 811	$9.4\cdot10^5$	2.20	29.76	724	$1.4\cdot10^5$
	FactorJoin	>30, 000	22.62	2, 593	31, 936	$1.6\cdot10^6$	2.35	66.12	876	8, 384	1.64	0.42	1.33	>30, 000	23.76	2, 939	36, 198	$2.4\cdot10^6$	2.75	115	912	4, 409
	MLP	>30, 000	$2.4\cdot10^6$	$>10^{10}$	$>10^{10}$	$>10^{10}$	7.35	491	2, 294	6, 887	8.63	3.11	3.17	>30, 000	$2.3\cdot10^6$	$>10^{10}$	$>10^{10}$	$>10^{10}$	6.66	502	3, 025	11, 584
	MSCN	27, 758	20.09	2, 870	17, 037	$2.6\cdot10^5$	2.46	87.23	736	8, 738	**1.61**	11.87	**1.02**	25, 766	18.89	2, 382	16, 947	$1.8\cdot10^6$	2.56	64.64	246	46, 721
	NNGP	12, 883	9.88	827	4, 652	$2.6\cdot10^5$	1.41	5.31	27.33	3, 588	32.44	**0.89**	32.66	11, 985	8.15	567	3, 639	$2.5\cdot10^5$	1.52	5.62	13.81	2, 753
	ALECE	**2, 901**	**1.29**	**5.07**	**11.54**	**62.52**	**1.07**	**1.34**	**1.59**	**2.32**	22.31	9.25	8.25	**2, 402**	**1.42**	**5.06**	**10.40**	**44.69**	**1.08**	**1.45**	**2.13**	**2.86**
	Optimal	2, 871	1	1	1	1	1	1	1	1	-	-	-	2, 349	1	1	1	1	1	1	1	1
Job-light	PG	6, 128	1.59	7.14	20.85	223	1.10	1.32	1.48	2.89	-	-	-	4, 747	1.67	6.20	14.09	70.13	1.10	1.39	1.64	3.06
	DeepDB	26, 636	11.31	663	6, 328	$5.4\cdot10^5$	1.66	13.71	26.01	49.57	11.52	11.93	11.92	23, 190	10.64	628	4, 883	$8.8\cdot10^5$	1.50	11.81	27.71	135
	NeuroCard	27, 744	14.73	978	6, 766	$4.7\cdot10^5$	1.81	12.40	37.25	144	42.93	9.11	15.90	16, 553	13.49	1, 000	6, 489	$1.5\cdot10^6$	1.60	15.10	25.94	50.68
	FLAT	23, 876	11.22	626	5, 786	$4.4\cdot10^5$	1.66	13.98	27.26	89.67	11.10	17.83	17.83	22, 531	10.13	490	4, 221	$1.1\cdot10^6$	1.42	12.87	34.54	195
	FactorJoin	16, 665	25.14	3, 456	27, 938	$2.5\cdot10^6$	1.52	5.45	10.23	47.09	4.66	26.22	1.33	13, 071	27.31	6, 134	36, 155	$2.2\cdot10^6$	1.42	4.83	9.43	22.75
	MLP	5, 610	4.30	23.14	59.37	2, 411	1.13	1.70	2.10	3.12	6.19	2.68	2.28	5, 351	5.59	62.02	138	1, 441	1.17	2.02	2.88	4.09
	MSCN	26, 665	24.42	1, 809	9, 217	$1.4\cdot10^5$	2.07	20.98	28.58	106	**1.56**	33.37	**0.77**	29, 118	26.77	1, 448	7, 980	$1.5\cdot10^5$	2.59	19.87	61.18	257
	NNGP	15, 430	$4.5\cdot10^8$	$>10^{10}$	$>10^{10}$	$>10^{10}$	3.57	8.39	11.97	24.47	32.44	**0.83**	29.86	4, 959	4.99	59.86	190	50, 036	1.20	2.05	2.99	4.96
	ALECE	**5, 168**	**1.14**	**2.08**	**3.85**	**14.82**	**1.07**	**1.24**	**1.32**	**1.77**	22.25	7.73	7.28	**3, 839**	**1.09**	**1.93**	**3.63**	**16.28**	**1.07**	**1.23**	**1.29**	**1.57**
	Optimal	5, 063	1	1	1	1	1	1	1	1	-	-	-	3, 792	1	1	1	1	1	1	1	1
TPCH	PG	**3, 230**	1.35	4.88	7.90	16.16	1.44	**1.57**	**1.59**	**1.62**	-	-	-	2, 385	1.33	5.01	6.87	14.64	**1.41**	**1.56**	**1.59**	**1.62**
	NeuroCard	10, 616	43.43	7, 071	27, 735	$3.5\cdot10^5$	3.75	10.91	22.48	106	704.98	34.38	34.77	6, 562	45.73	5, 896	20, 931	$1.2\cdot10^5$	3.16	11.04	16.49	245
	FLAT	11, 428	46.04	12, 236	70, 383	$8.4\cdot10^7$	3.56	10.91	25.71	1, 182	168.65	10.06	41.42	7, 864	42.32	7, 328	26, 470	$3.7\cdot10^5$	3.56	15.82	21.70	425
	MLP	7, 261	51, 475	$1.4\cdot10^7$	$3.5\cdot10^7$	$4.7\cdot10^9$	4.71	14.11	22.74	36.09	8.63	3.57	3.62	8, 342	79, 317	$3.7\cdot10^7$	$9.1\cdot10^7$	$>10^{10}$	5.12	17.63	32.74	63.39
	MSCN	7, 718	36.36	6, 709	26, 744	$4.4\cdot10^5$	3.58	15.11	25.71	387	**1.58**	23.43	**0.79**	8, 353	41.66	5, 457	21, 433	$1.7\cdot10^5$	3.80	14.86	25.69	472
	NNGP	18, 618	$1.3\cdot10^9$	$>10^{10}$	$>10^{10}$	$>10^{10}$	5.56	47.40	59.18	99.55	32.44	**0.92**	41.76	3, 128	25.88	432	986	4, 968	1.54	7.98	10.90	20.66
	ALECE	3, 233	**1.15**	**2.29**	**3.37**	**8.86**	**1.43**	**1.57**	1.60	**1.62**	22.34	11.62	10.35	**2, 380**	**1.25**	**2.11**	**2.95**	**10.87**	**1.41**	**1.56**	**1.59**	**1.62**
	Optimal	3, 227	1	1	1	1	1	1	1	1	-	-	-	2, 378	1	1	1	1	1	1	1	1

and the numerator $C(P^E, \mathbf{c}^T)$ is the cost by feeding the true cardinalities of sub-queries to the query plan generated by method A. In other words, $C(P^E, \mathbf{c}^T)$ is the actual execution cost of $\boldsymbol{q}$ if method A is adopted.

- **Q-error** [38] measures the distance between the estimated cardinality P and the true cardinality T of a query. In particular, Q-error$(P, T) = \max(\frac{T}{P}, \frac{P}{T})$.
- **Space overhead** is the memory size used by a method.
- **Building time** indicates the offline training time of query-driven methods or construction time of data-driven methods.
- **Estimation latency** describes the average estimation time per query used by a cardinality estimation method.

As the optimizer only requires the cardinality estimates of the sub-queries, our evaluations also involve the testing sub-queries only.

Parameter Settings. We use a 40-bin-histogram for each attribute, *i.e.*, $d_x = 40$. The values of n_{enc} and n_{ana}, *i.e.*, the number of stacked attention layers in the data-encoder and query-analyzer, respectively, are both set to 4. To train our ALECE, we use an Adam optimizer [32] with a learning rate of 0.01 and a batch size of 128.

6.2 Performance on Dynamic Workloads

For each dynamic workload, we build all methods using the data in the training part to make estimates for the testing sub-queries in the evaluation part. The featurizations of the training queries and sub-queries as well as the corresponding DB states and true cardinalities in the training part form the training data. We use the training data to train the query-driven models including ALECE, MLP, MSCN and NNGP. The data-driven models, namely DeepDB, NeuroCard, FLAT and FactorJoin, are built with the database data after all the statements in the training part of the workload are executed. This setting is compatible with real world scenarios as the cardinality estimation models are updated at regular intervals. In our experiments, the associated changing rate ρ of each testing query is larger than 20%. In other words, when testing queries are executed, at least 20% of the underlying data are changed, compared to the database when cardinality estimators are built.

The estimation results by different methods will be fed into our improved benchmark to compare the end-to-end query times (E2E times) of all methods on dynamic workloads. To investigate the performance gap between these methods and the optimal, we also feed the true cardinalities to the benchmark to get the optimal execution time. We do not compare with DeepDB on the STATS and TPCH datasets since it supports PK-FK joins only. The open implementations of Factorjoin are hard-coded for the STATS and Job-light datasets and does not support the TPCH dataset. Thus, Factorjoin is not tested on this dataset. Besides the E2E time, we also record the Q-error and P-error distributions, building time, storage overhead and latency of all methods on different workloads. These results are shown in Table 4. The storage overhead, building time and latency results on the **Update-heavy** workloads are similar to the counterparts on the Insert-heavy workloads and thus omitted.

End-to-end evaluations. Referring to Table 4, ALECE has clear advantages on the E2E time over the other methods. On the one hand, compared to PostgreSQL's built-in cardinality estimator, ALECE makes query execution up to 2.7× faster. Its E2E time is only slightly larger than that of PG on TPCH's Insert-heavy workload. However, there is almost no gap between the performance of PG and Optimal on this workload. Compared to Optimal, ALECE only results in at most 2.2% extra E2E time. Considering the superiority of ALECE on these workloads, we confidently claim that ALECE performs much better than the built-in estimator in PostgreSQL and is able to greatly improve the query execution performance on dynamic workloads. On the other hand, except for ALECE, only MLP outperforms PG on Job-light's Update-heavy workload. The E2E time of DeepDB, NeuroCard, FLAT, FactorJoin and NNGP is larger than that of PG in all cases. Compared to Optimal, these methods results in more than double of E2E time on most workloads. These results indicate that these existing methods cannot make satisfactory estimates for DBMS on the dynamic workloads.

The P-error comparisons demonstrate that ALECE results in effective query executions from another angle. The P-errors of ALECE on most queries are close to 1. In other words, with ALECE's outputs, most queries can be executed almost as fast as if they are optimized with true cardinalities. At the 95% quantile, ALECE outperforms PG, DeepDB, NeuroCard, FLAT, FactorJoin, MLP, MSCN and NNGP, in terms of P-error, by up to 26×, 21×, 518×, 1,385×, 551×, 1,443×, 463× and 37× times, respectively. All these show that ALECE achieves large superiority over the competitors on helping PostgreSQL process queries more efficiently.

It is noteworthy that ALECE has clearer advantages on the STATS dataset with more complex join patterns. This reflects that our ALECE is able to grasp the implicit relations between complex join patterns and the underlying data significantly better than the alternatives. Besides, ALECE outperforms MLP in terms of E2E time and P-error. The main structure difference of ALECE and MLP is reflected on the adoption of attentions in ALECE's modules. This verifies the positive effects of the attention mechanisms in extracting useful information from underlying data and SQL queries.

Estimation Accuracy. Table 4 also reports on the Q-error distributions of all methods. In general, ALECE clearly outperforms PG on all three datasets. This verifies that the independence assumption in PG is not reasonable for some scenarios. ALECE still performs best among all methods in all cases. At most quantiles, ALECE results in the smallest maximum Q-error. The median Q-error of ALECE in all cases are all close to 1, the optimal value. At the 95% quantile, the Q-error of ALECE in all cases is smaller than 10. In contrast, none of the other methods can reach this level of performance. At this quantile, DeepDB, NeuroCard, FLAT, FactorJoin, MLP, MSCN and NNGP result in up to 84×, 1,643×, 8,220×, >10^4×, 3,480×, >10^4×, 7,927×, >10^4× times larger Q-error than that of ALECE. At other quantiles, these methods are still incomparable with ALECE. These results show that ALECE is more accurate and able to discover the implicit relationships among attributes and those between queries and attributes. Another interesting thing is that smaller Q-error does not necessarily result in smaller E2E time, which is shown by the comparisons among NNGP, DeepDB and NeuroCard, *etc*, on the Job-light dataset. An inaccurate estimate for a single sub-query tends to generate bad query plans and large execution time. It is necessary to ensure accurate estimates for all sub-queries.

Model Construction Efficiency. Referring to Table 4, the training cost of ALECE is small. It requires less than 10, 8 and 13 minutes to fine-tune its parameters for the STATS, Job-light and TPCH datasets, respectively. In contrast, DeepDB, NeuroCard, FLAT and MSCN consume more construction time. For example, building FLAT for the STATS dataset requires nearly an hour, unacceptable in many real-world scenarios. Although Factorjoin requires less construction time on the STATS dataset, its E2E time, P-error and Q-error performance are much worse. Compared to ALECE, MLP and NNGP have simpler structures and thus they need less time to train. However, their representation abilities are not so powerful as that of ALECE. The overwhelming advantage of ALECE on E2E time illustrates the necessity of a more complex structure and slightly more training time.

In terms of latency, FactorJoin, MLP and MSCN are the best. They need less than 10 ms on making an estimate for a query. ALECE's latency is slightly larger but smaller than others. Considering that executing a query on all datasets takes more than 10 seconds on average, estimation latency is not crucial in the overall picture.

Storage Overhead. The storage overhead of the query-driven methods is smaller than that of the data-driven ones in general. The query-driven methods only need to maintain a fixed number of parameters whose sizes are usually much smaller than the 'data summaries' held by the data-driven methods, *e.g.,* the SPN or FSPN in DeepDB and FLAT. FactorJoin and MSCN incur the smallest memory costs. ALECE consumes more memory than MLP and NNGP due to its more complex structure. Compared to NeuroCard and FLAT, ALECE saves up to 75.4% and 89.4% memory cost on the STATS dataset, respectively. ALECE results in a little more memory cost than FLAT and DeepDB on the Job-light dataset. Considering the fact that modern computers usually have large memories and ALECE performs better in terms of E2E time, P-error and Q-error, the extra storage overhead pays off highly.

6.3 Effect of Distribution Shifting

To investigate if ALECE still works well when the distribution of the underlying data greatly changes, we carry out experiments to compare the different methods on the Dist-shift workload. It is noteworthy that the evaluation part of the Dist-shift workload covers highly skewed insert statements. Table 5 reports the E2E time, Q-error and P-error distributions, and latency comparisons among ALECE, PG, NeuroCard, MSCN, NNGP and Optimal. Those competitors are chosen because they have better overall performance on the Insert-heavy and Update-heavy workloads. Due to space limit, the storage overhead and building time comparisons are omitted. These results are similar to the counterparts on the other two workloads.

Table 5: Performance of methods on Dist-shift workloads

Data	Model	E2E Time(S)	Q-error				P-error			
			50%	90%	95%	99%	50%	90%	95%	99%
STATS	PG	8, 432	189	$1.4{\cdot}10^5$	$1.1{\cdot}10^6$	$1.9{\cdot}10^7$	2.60	25.50	42.65	300
	NeuroCard	27, 252	14.30	996	5, 051	$3.9{\cdot}10^5$	2.08	25.69	108	3, 318
	MSCN	26, 697	20.08	2, 802	15, 576	$4.9{\cdot}10^5$	2.75	46.51	465	83, 452
	NNGP	10, 537	9.47	785	4, 370	$2.4{\cdot}10^5$	1.45	7.19	25.19	3, 318
	ALECE	**6, 876**	**1.40**	**5.46**	**11.75**	**118.36**	**1.07**	**1.38**	**1.71**	**11.18**
	Optimal	6, 770	1	1	1	1	1	1	1	1
Job-light	PG	5, 608	1.58	6.12	14.22	76.30	1.10	1.32	1.48	2.89
	NeuroCard	17, 720	14.15	822	4, 799	$3.5{\cdot}10^5$	1.81	12.40	37.25	144
	MSCN	25, 157	28.54	1, 592	5, 261	99, 178	2.06	15.01	27.77	84.35
	NNGP	11, 698	$4.6{\cdot}10^8$	$>10^{10}$	$>10^{10}$	$>10^{10}$	3.50	7.90	11.24	23.38
	ALECE	**4, 763**	**1.16**	**2.23**	**4.34**	**16.22**	**1.07**	**1.24**	**1.32**	**1.77**
	Optimal	4, 708	1	1	1	1	1	1	1	1
TPCH	PG	3, 377	1.23	3.94	5.85	10.28	1.10	1.39	1.64	3.06
	NeuroCard	11, 362	44.65	9, 151	34, 541	$2.8{\cdot}10^5$	1.60	15.10	25.94	50.68
	MSCN	7, 062	41.62	6, 571	28, 837	$2.4{\cdot}10^5$	3.64	9.17	22.00	202
	NNGP	18, 244	$1.4{\cdot}10^9$	$>10^{10}$	$>10^{10}$	$>10^{10}$	5.54	46.30	59.12	99.12
	ALECE	**3, 230**	**1.18**	**2.66**	**4.46**	**11.23**	**1.07**	**1.23**	1.29	**1.57**
	Optimal	3, 226	1	1	1	1	1	1	1	1

Referring to Table 5, the overall Q-error and P-error performance of all methods on the Dist-shift workloads are worse than the counterparts on the Insert-heavy workloads. This is reasonable because compared to the data upon which these methods are built, the distribution of the underlying data is greatly shifted when testing queries are executed. Nevertheless, ALECE still achieves the best E2E time on all datasets. It needs up to 18.5% less time than PG to execute all testing queries. Its E2E time on all three workloads is close to the optimal results and much smaller than that of other competitors. In terms of Q-error and P-error, ALECE also performs best. At the 95% quantile, the most Q-error and P-error of ALECE are 11.75 and 1.71, respectively. In contrast, other competitors' Q-error and

P-error are at least 429 and 8.52 times larger, respectively. All these demonstrate that ALECE is less sensitive to the distribution shifting and able to make accurate estimates even when the distributions of the underlying data changes significantly.

6.4 Effect of d_x

Parameter d_x is used to control the amount of distribution information covered by data featurizations. A larger d_x usually implies more informative featurizations but results in more storage and latency overhead. To investigate the effect of d_x, we build four ALECE s with different d_x values and observe their performance on the Insert-heavy workload of the STATS dataset. The results on the other datasets and other workloads are similar and thus omitted. Table 6 shows the results of ALECE with different values of d_x. As d_x's value increases, ALECE is able to get more accurate estimations. Meanwhile, the building time does not changes much. The storage overhead and latency increase slightly. Also, the value of d_x does not influence the cost of updating DB states when an data update statement is executed. Thus, we suggest to adopt a larger d_x to build the DB states within affordable memory and latency cost.

Table 6: The effects of the hyperparameter d_x

d_x	Q-error				Size (MB)	Building Time(Min)	Latency (ms)
	50%	90%	95%	99%			
10	1.26	5.52	13.57	78.16	**22.15**	9.47	**8.08**
20	1.33	5.49	12.65	65.52	22.21	**8.83**	8.11
40	1.29	5.07	11.54	**62.52**	22.31	9.25	8.25
80	**1.24**	**5.00**	**10.76**	64.25	22.56	9.66	8.78

7 RELATED WORK

Data-driven Cardinality Estimators. Data-driven methods aim to describe the underlying data with statistical or machine learning models. The simple yet efficient 1-D Histogram [45] is used in many well-known DBMS like PostgreSQL. It assumes all attributes are mutually independent and maintains a 1-D (cumulative) histogram for each attribute. To address the problem of unreasonable independence assumption, M-D Histogram based methods [13, 22, 41, 52] build multi-dimensional histograms to model attribute dependency. Although such methods improve the accuracy, the decomposition of the joint attributes is still lossy. Also, they hardly work for queries with complex joins. Sampling-based methods [10, 11, 33, 36, 43] address join queries but they risk high variance and sampling failure when the data distribution or query is complex. Bayesian network (BN) based methods [12, 18, 50] use a directed acyclic graph to model the dependence among attributes, assuming that each attribute is conditionally independent of the remaining attributes given its parents' distributions. BayesCard [58] revitalizes BN using probabilistic programming to improve its inference and model construction speed. Recently, machine learning techniques are adopted in data-driven methods. Deep autoregressive models are adopted in Naru [62] and NeuroCard [61] to decompose the joint distribution of attributes to a product of conditional distributions. DeepDB [27] is built upon Sum-Product Network (SPN) [40] which approximates the joint distribution using several local and simple PDFs. FLAT [65] improves SPN by adopting a factorize-split-sum-product network (FSPN) [60] to adaptively decompose the joint distribution according to the attribute dependence level.

Query-driven Cardinality Estimators. Such estimators focus on modeling the relationships between queries and their true cardinalities. The feedbacks of past queries are utilized to correct and self-tune histograms [9, 16, 29, 46] and update statistical summaries [47, 55]. LW-XGB and LW-NN [15] formulate the cardinality estimation as a regression problem and apply gradient boosted trees and neural networks for regression, respectively. UAE-Q [59] applies the deep auto-regression models and differentiable progressive sampling via the Gumbel-Softmax trick to learn hidden information from queries. The KDE-based join estimators [30] combine kernel density estimation (KDE) with a query-driven tuning mechanism to estimate multivariate probability distributions of a relation and cardinalities of joins. Fauce [37] and NNGP [63] assume a query's cardinality follows a Gaussian distribution and adopt Deep Ensemble [34] and neural network Gaussian process [35] to predict the distribution's mean and variance.

A few works also consider both data and SQL queries. Wu *et. al.* [56] propose a unified deep autoregressive model utilizing both data as unsupervised information and query workload as supervised information. Kipf *et. al.* [33] concatenate basic relation information and query features together and use a multi-set convolutional neural network to process them and make estimates. Negi *et. al.* [39] propose techniques to build sample tables on the join keys and use neural networks to extract information from queries. However, these methods require sampling over static data which results in high sampling overhead. The discrepancy between the distributions of the samples and changed data makes these methods difficult to deal with dynamic workloads. Also, compared to ALECE, these methods do not explain the links among data, queries and true cardinalities. Besides, Negi's work [39] supports PK-FK joins only. Dutt *et. al.* [15] use neural networks and tree-based ensembles to extract information from data and queries to solve the problem of selectivity estimation on a single relation. Thus, their approach does not support joins. In addition, Han *et. al.* [23] propose an end-to-end evaluation benchmark for cardinality estimators. Our used benchmark is an improved version of it. Sun *et. al.* [49] make a comprehensive comparison of the existing cardinality estimators.

8 CONCLUSION AND FUTURE WORK

In this work, we design ALECE, a versatile learned cardinality estimation model, that makes accurate and high-quality estimates for SQL queries. Based on two delicate methods to featurize the underlying database data and the SQL queries, respectively, ALECE adopts the attention mechanisms in its two modules to understand the implicit relations between data and queries. The self-attention layer in the data-encoder module figures out the link among all database attributes. The query-analyzer takes the input of the query featurization and the output of the data-encoder, and puts attention on the more important parts of the data. Extensive experimental results show that ALECE clearly outperforms the state-of-the-art alternatives in terms of multiple evaluation metrics.

For future work, it is interesting to extend ALECE to more general aggregate analytic queries by replacing `COUNT(*)` with other aggregate functions. Also, it is relevant to explore if better data and query featurization methods exist. Moreover, it makes sense to use other types of attention functions in ALECE.

REFERENCES

[1] online. https://github.com/pfl-cs/ALECE/blob/main/ALECE-Extended.pdf.
[2] online. https://github.com/pfl-cs/ALECE.
[3] online. https://relational.fit.cvut.cz/dataset/Stat.
[4] online. http://homepages.cwi.nl/~boncz/job/imdb.tgz.
[5] online. https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp.
[6] Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer Normalization. *arXiv preprint* abs/1607.06450 (2016). http://arxiv.org/abs/1607.06450
[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *ICLR*.
[8] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *NeurIPS*.
[9] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 2001. STHoles: A Multidimensional Workload-Aware Histogram. In *SIGMOD*. 211–222.
[10] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. 1999. On Random Sampling over Joins. In *SIGMOD*. 263–274.
[11] Yu Chen and Ke Yi. 2017. Two-Level Sampling for Join Size Estimation. In *SIGMOD*. 759–774.
[12] C. K. Chow and C. N. Liu. 1968. Approximating discrete probability distributions with dependence trees. *IEEE Trans. Inf. Theory* 14, 3 (1968), 462–467.
[13] Amol Deshpande, Minos N. Garofalakis, and Rajeev Rastogi. 2001. Independence is Good: Dependency-Based Histogram Synopses for High-Dimensional Data. In *SIGMOD*. 199–210.
[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*. 4171–4186.
[15] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek R. Narasayya, and Surajit Chaudhuri. 2019. Selectivity Estimation for Range Predicates using Lightweight Models. *Proc. VLDB Endow.* 12, 9 (2019), 1044–1057.
[16] Dennis Fuchs, Zhen He, and Byung Suk Lee. 2007. Compressed histograms with arbitrary bucket layouts for selectivity estimation. *Inf. Sci.* 177, 3 (2007), 680–702.
[17] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. 2017. Convolutional Sequence to Sequence Learning. In *ICML*. 1243–1252.
[18] Lise Getoor, Benjamin Taskar, and Daphne Koller. 2001. Selectivity Estimation using Probabilistic Models. In *SIGMOD*. 461–472.
[19] Goetz Graefe and William J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*. IEEE Computer Society, 209–218.
[20] Karol Gregor, Ivo Danihelka, Andriy Mnih, Charles Blundell, and Daan Wierstra. 2014. Deep AutoRegressive Networks. In *ICML*, Vol. 32. 1242–1250.
[21] PostgreSQL Global Development Group. 1996. PostgreSQL. https://www.postgresql.org. (1996). Accessed: 2022-10-28.
[22] Dimitrios Gunopulos, George Kollios, Vassilis J. Tsotras, and Carlotta Domeniconi. 2000. Approximating Multi-Dimensional Aggregate Range Queries over Real Attributes. In *SIGMOD*. 463–474.
[23] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *Proc. VLDB Endow.* 15, 4 (2021), 752–765.
[24] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition.* Springer.
[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. 770–778.
[26] Karl Moritz Hermann, Tomás Kociský, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. 2015. Teaching Machines to Read and Comprehend. In *NeurIPS*. 1693–1701.
[27] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proc. VLDB Endow.* 13, 7 (2020), 992–1005.
[28] Raghav Kaushik, Jeffrey F. Naughton, Raghu Ramakrishnan, and Venkatesan T. Chakaravarthy. 2005. Synopses for query optimization: A space-complexity perspective. *ACM Trans. Database Syst.* 30, 4 (2005), 1102–1127.
[29] Andranik Khachatryan, Emmanuel Müller, Christian Stier, and Klemens Böhm. 2015. Improving Accuracy and Robustness of Self-Tuning Histograms by Subspace Clustering. *IEEE Trans. Knowl. Data Eng.* 27, 9 (2015), 2377–2389.
[30] Martin Kiefer, Max Heimel, Sebastian Breß, and Volker Markl. 2017. Estimating Join Selectivities using Bandwidth-Optimized Kernel Density Models. *Proc. VLDB Endow.* 10, 13 (2017), 2085–2096.
[31] Yoon Kim, Carl Denton, Luong Hoang, and Alexander M. Rush. 2017. Structured Attention Networks. In *ICLR*.
[32] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR*.
[33] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*.
[34] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. 2017. Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles. In *NIPS*. 6402–6413.
[35] Jaehoon Lee, Yasaman Bahri, Roman Novak, Samuel S. Schoenholz, Jeffrey Pennington, and Jascha Sohl-Dickstein. 2018. Deep Neural Networks as Gaussian Processes. In *ICLR*.
[36] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation via Random Walks. In *SIGMOD*. 615–629.
[37] Jie Liu, Wenqian Dong, Dong Li, and Qingqing Zhou. 2021. Fauce: Fast and Accurate Deep Ensembles with Uncertainty for Cardinality Estimation. *Proc. VLDB Endow.* 14, 11 (2021), 1950–1963.
[38] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *Proc. VLDB Endow.* 2, 1 (2009), 982–993.
[39] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust Query Driven Cardinality Estimation under Changing Workloads. *Proc. VLDB Endow.* 16, 6 (2023), 1520–1533.
[40] Hoifung Poon and Pedro M. Domingos. 2011. Sum-Product Networks: A New Deep Architecture. In *UAI*. 337–346.
[41] Viswanath Poosala and Yannis E. Ioannidis. 1997. Selectivity Estimation Without the Attribute Value Independence Assumption. In *VLDB*. 486–495.
[42] Lutz Prechelt. 2012. Early Stopping - But When? In *Neural Networks: Tricks of the Trade - Second Edition*. Lecture Notes in Computer Science, Vol. 7700. Springer, 53–67.
[43] Yuan Qiu, Yilei Wang, Ke Yi, Feifei Li, Bin Wu, and Chaoqun Zhan. 2021. Weighted Distinct Sampling: Cardinality Estimation for SPJ Queries. In *SIGMOD*. 1465–1477.
[44] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. 1986. *Learning Internal Representations by Error Propagation.* 318–362.
[45] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *SIGMOD*. 23–34.
[46] Utkarsh Srivastava, Peter J. Haas, Volker Markl, Marcel Kutsch, and Tam Minh Tran. 2006. ISOMER: Consistent Histogram Construction Using Query Feedback. In *ICDE*. 39.
[47] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *VLDB*. 19–28.
[48] Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. 2015. End-To-End Memory Networks. In *NeurIPS*. 2440–2448.
[49] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation: A Design Space Exploration and A Comparative Evaluation. *Proc. VLDB Endow.* 15, 1 (2021), 85–97.
[50] Kostas Tzoumas, Amol Deshpande, and Christian S. Jensen. 2011. Lightweight Graphical Models for Selectivity Estimation Without Independence Assumptions. *Proc. VLDB Endow.* 4, 11 (2011), 852–863.
[51] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NeurIPS*. 5998–6008.
[52] Hai Wang and Kenneth C. Sevcik. 2003. A multi-dimensional histogram for selectivity estimation and fast approximate query answering. In *CASCON*. 328–342.
[53] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. 2021. FACE: A Normalizing Flow based Cardinality Estimator. *Proc. VLDB Endow.* 15, 1 (2021), 72–84.
[54] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready For Learned Cardinality Estimation? *Proc. VLDB Endow.* 14, 9 (2021), 1640–1654.
[55] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a Learning Optimizer for Shared Clouds. *Proc. VLDB Endow.* 12, 3 (2018), 210–222.
[56] Peizhi Wu and Gao Cong. 2021. A Unified Deep Model of Learning from both Data and Queries for Cardinality Estimation. In *SIGMOD*. 2009–2022.
[57] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2023. FactorJoin: A New Cardinality Estimation Framework for Join Queries. In *SIGMOD*.
[58] Ziniu Wu and Amir Shaikhha. 2020. BayesCard: A Unified Bayesian Framework for Cardinality Estimation. *CoRR* abs/2012.14743 (2020). https://arxiv.org/abs/2012.14743
[59] Ziniu Wu, Pei Yu, Peilun Yang, Rong Zhu, Yuxing Han, Yaliang Li, Defu Lian, Kai Zeng, and Jingren Zhou. 2022. A Unified Transferable Model for ML-Enhanced DBMS. In *CIDR*.

[60] Ziniu Wu, Rong Zhu, Andreas Pfadler, Yuxing Han, Jiangneng Li, Zhengping Qian, Kai Zeng, and Jingren Zhou. 2020. FSPN: A New Class of Probabilistic Graphical Model. *CoRR* abs/2011.09020 (2020). https://arxiv.org/abs/2011.09020

[61] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *Proc. VLDB Endow.* 14, 1 (2020), 61–73.

[62] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *Proc. VLDB Endow.* 13, 3 (2019), 279–292.

[63] Kangfei Zhao, Jeffrey Xu Yu, Zongyan He, Rui Li, and Hao Zhang. 2022. Lightweight and Accurate Cardinality Estimation by Neural Network Gaussian Process. In *SIGMOD*. 973–987.

[64] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *SIGMOD*. 1525–1539.

[65] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. *Proc. VLDB Endow.* 14, 9 (2021), 1489–1502.

Appendix

Due to space limit, we present some minute details and relatively less important experimental results and analyses in this appendix.

A MULTI-HEAD ATTENTION

To enhance the representation ability of our ALECE, we adopt a multi-head attention mechanism [51] in both modules instead of performing a single function. Fig. 4 depicts its details.

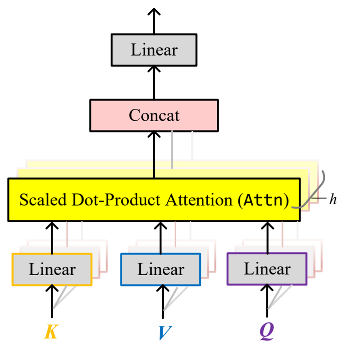


Figure 4: Multi-head Attention, adapted from [51].

A multi-head attention projects the queries, keys and values multiple times with h different linear projections, and then executes the attention function Attn with the input of the projected queries, keys and values in parallel. Compared to a single attention head, it could jointly attend to information from different projected spaces and thus is more powerful. Below is its analytical expressions:

$$\text{MutliHead}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{concat}(\text{head}_1, \cdots, \text{head}_h)\boldsymbol{W}^M,$$

$$\text{where head}_i = \text{Attn}(\boldsymbol{Q}\boldsymbol{W}_i^Q, \boldsymbol{K}\boldsymbol{W}_i^K, \boldsymbol{V}\boldsymbol{W}_i^V)$$

Above, $\boldsymbol{W}_i^Q \in \mathbb{R}^{d_k \times d_m}$, $\boldsymbol{W}_i^Q \in \mathbb{R}^{d_k \times d_m}$ and $\boldsymbol{W}_i^Q \in \mathbb{R}^{d_v \times d_m}$ project each query, key and value vector into $\mathbb{R}^{d_m}$ space respectively; $\boldsymbol{W}^M \in \mathbb{R}^{d_m \times d_v}$ projects the output of weighted values back to $\mathbb{R}^{d_v}$. In our experiments, the value of h is set to 8, following the settings in [51].

It is noteworthy that the choice of the attention function Attn is not unique and there may be other type of Attn-integrated attention mechanism besides the multi-head attention. The attention layers can be flexibly designed depending on situations.

B AN IMPROVED BENCHMARK

The *CardEst* benchmark [23] provides a way to integrate external cardinality estimators into the built-in query optimizer of PostgreSQL. In particular, given a SQL query to be executed, the PostgreSQL's query optimizer needs to get the estimated cardinalities of a series of sub-queries in a fixed order. By enabling a knob, the benchmark provides the function of reading the cardinality estimates from a given file instead of using the estimates by PostgreSQL's built-in estimator. Thus, if we can know what the sub-queries are and feed the corresponding cardinality estimates accessed through external estimation methods, we can directly compare the quality of their generated query plans by watching the end-to-end query execution time.

The idea above is simple and appealing. However, the original benchmark [23] cannot produce correct sub-queries for queries on the dynamic workloads and with complex join predicates. In other words, it works on very limited cases only. These drawbacks motivate us to improve the benchmark to support dynamic workloads and more general join schemata. Our improved benchmark [2] implements correct sub-queries generations function and inherits the cardinality estimates reading function. We have validated it with numerous queries on the STATS, Job-light and TPCH datasets.

Usage guidance. Our benchmark mainly provides two functions: sub-qureies generation and cardinalty estimates replacement. We leave multiple knobs to let users decide when to generate sub-queries and which method's estimation results used to generate and query plans for the queries on a workload $\boldsymbol{W}$.

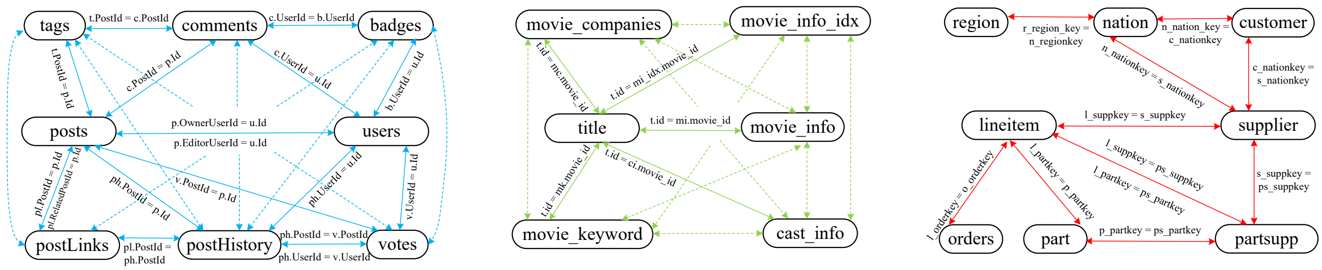
1) Sub-queries generation. This function is related with only one knob: *print_sub_queries*. Given a query, the optimizer needs the cardinality estimations of the sub-queries of two types: the *single* sub-qureies only involve single tables whereas the *join* sub-queries cover join conditions. By setting the value of the knob to True and executing the `Explain` statement for each query on $\boldsymbol{W}$, two files will be generated in the data directory of PostgreSQL: 'single_tbls.txt' and 'join_queries.txt'. They record the above two types of sub-queries, respectively. Each line of either file is a sub-query of a query $\boldsymbol{q}$ on $\boldsymbol{W}$. Also, the line will include the appearance ranking of $\boldsymbol{q}$ among all queries. This information help users know the ties between queries and sub-queries.

2) Cardinality estimates replacement. After estimating the cardinalities of the sub-queries in both files using an external method, *e.g.* ALECE, we can 'inject' them into PostgreSQL to replace the built-in results. First, we need to save the estimation results into two files with each file corresponds to one type of sub-queries, and copy them into the data directory of PostgreSQL. Then, two knobs, namely *read_single_cards* and *read_join_cards*, are supposed to be turned on. Meanwhile, we should set the configuration parameters *single_cards_fname* an *join_cards_fname* to be the names of the above two files, respectively. After these settings, we can run the workload in the usual way. It is noteworthy that the query optimizer will use the cardinality estimates by the external method to generate the query plans. Suppose the files for the single and join sub-queries are named 'single_cards.txt' and 'join_cards.txt', respectively. The setting statements for the knobs and configuration parameters are shown as follows.

```
SET read_single_cards=true;
SET read_join_cards=true;
SET single_cards_fname='single_cards.txt';
SET join_cards_fname='join_cards.txt';
```

Usually, PostgreSQL's built-in estimator is able to give sufficiently accurate cardinality estimates for the single sub-queries. Thus, we only need to estimate the cardinalities of the join sub-queries, turn on the knob *read_join_cards* and set the parameter *join_cards_fname* in most cases.

Please also refer to our github repository [2] for more details.



(a) STATS (b) Job-light (c) TPCH

Figure 5: Joins among relations in three datasets.

C ADDITIONAL EXPERIMENTAL RESULTS

This section presents the additional experimental results that cannot be put in the main body of the paper due to the space limit.

C.1 Join Information among Relations

Fig. 5 shows the join information among relations in the STATS, Job-light and TPCH datasets. For clearity, only part of the joins are exhibited.

C.2 Performance on Static Workloads

We also conduct experiments to compare all methods on the static workloads that consist of query statements only. In particular, we use the whole STATS, Job-light and TPCH datasets to build the data-driven methods. On top of the whole dataset, the training queries and sub-queries as well as their true cardinalities are used as the training and validation data for the query-driven methods. Accordingly, all data-driven and query-driven methods estimate the cardinalities for the testing sub-queries. These estimates are in turn used to generate the E2E evaluation results, Q-error, *etc.* Table 7 reports on the relevant comparative results.

Table 7: Performance of methods on static workloads

Data	Model	E2E Time(S)	Q-error 50%	90%	95%	99%	Size (MB)	Building Time(Min)	Latency (ms)
STATS	PG	12, 777	1.80	21.84	106	7, 950	-	-	-
	NeuroCard	19, 847	2.91	192	1, 511	$1.5 \cdot 10^5$	121.88	27.77	40.16
	MLP	7, 823	1.58	4.62	9.22	76.97	8.52	3.10	3.10
	MSCN	15, 162	3.85	39.56	99.81	1, 273	**1.61**	12.41	**0.79**
	NNGP	20, 181	8.10	694	3, 294	$2.3 \cdot 10^5$	9.56	**0.68**	21.28
	ALECE	**7, 704**	**1.47**	**4.86**	**9.11**	**56.98**	22.31	6.44	8.64
	Optimal	7, 622	1	1	1	1	-	-	-
Job-light	PG	19, 820	1.70	9.41	16.25	50.19	-	-	-
	NeuroCard	18, 153	1.37	4.16	6.35	15.22	49.96	9.83	16.76
	MLP	18, 413	1.40	3.78	7.00	76.30	6.19	1.83	2.25
	MSCN	24, 829	10.94	200	396	2, 741	**1.56**	31.68	**1.04**
	NNGP	>30, 000	6.66	121	414	$1.5 \cdot 10^5$	32.44	**0.88**	30.35
	ALECE	**18, 015**	**1.44**	**3.23**	**5.75**	**47.28**	22.25	5.88	7.32
	Optimal	17, 939	1	1	1	1	-	-	-
TPCH	PG	12, 217	**1.23**	3.95	6.22	11.33	-	-	-
	NeuroCard	12, 020	1.09	2.97	395	450	850.53	44.39	35.98
	MLP	8, 957	1.44	2.50	3.79	10.91	8.63	2.68	3.67
	MSCN	11, 893	4.29	36.76	83.76	321	**1.58**	24.17	**0.80**
	NNGP	13, 183	31.89	1, 052	2, 104	11, 811	32.44	**0.91**	39.66
	ALECE	**8, 717**	1.24	**2.36**	**4.26**	**10.73**	22.34	7.15	10.31
	Optimal	8, 706	1	1	1	1	-	-	-

As shown in Table 7, the end-to-end performance of cardinality estimation methods on the static workloads is different from the counterpart on the dynamic workloads. Overall, the performance of the methods on static workloads is better than that on dynamic workloads. Nevertheless, ALECE still performs best among all methods. ALECE results in up to 1.66, 2.58, 1.97 and at least 3 times faster query execution than PG, NeuroCard, MSCN and NNGP, respectively. The E2E time of ALECE is only at most 1.1% larger than that of Optimal. Next, at most quantiles, ALECE's Q-error is smaller than that of the others on all datasets. At the 95% quantile, ALECE achieves up to 11.64×, 166×, 77× and 494× smaller Q-error compared to PG, NeuroCard, MSCN and NNGP, respectively. ALECE is not the best in terms of training time, latency and memory cost. However, ALECE incurs comparable results with the competitors. Considering the much better E2E time and estimation accuracy ALECE achieves, the slightly extra overhead is acceptable.

ALECE vs. MLP on static workloads. Referring to Table 7, MLP achieves E2E time and Q-error comparable with the counterparts of ALECE, and even less storage overhead, latency and training time. The reason behind is that ALECE is almost equivalent to MLP when processing queries on static workloads, where the DB states for the queries are constant. In this case, only one element exists in the input sets of keys, values and queries of the self-attention layer in the data-encoder module. This is the same to the three sets of the attention layers in the query-analyzer module. Consequently, both attention layers make almost no effects, and the whole ALECE network degenerates to an MLP. Therefore, if there is no underlying data change, we can simply train an MLP to estimate cardinalities.

C.3 Hyperparameter Studies

To study the effects of more hyperparameters, we build different ALECE versions and observe their performance. Similarly, we only show the comparison results on the Insert-heavy workload of the STATS dataset. The results on the other datasets and other workloads are similar and thus omitted.

Effects of n_{enc} and n_{ana}. n_{enc} and n_{ana} are the numbers of stacked multi-head attention layers in ALECE's data-encoder and query-analyzer modules, respectively. To investigate if they affect ALECE's performance, we train a series of ALECE with different n_{enc} and n_{ana} values. Table 8 reports the comparison results.

When the values of n_{enc} and n_{ana} are set to 2, a smaller number, ALECE achieves the worst estimation accuracy. However, the Q-error performance of ALECE is not necessarily positively correlated to the numbers of attention layers in its two modules. When both n_{enc} and n_{ana} equal 4 or 6, ALECE has the overall best Q-error performance. However, larger n_{enc} or n_{ana} will result in larger

Table 8: The effects of the hyperparameters n_{enc} and n_{ana}

n_{enc}	n_{ana}	Q-error				Size	Building	Latency
		50%	90%	95%	99%	(MB)	Time(Min)	(ms)
2	2	1.37	6.33	13.69	90.04	**10.36**	**7.12**	**6.13**
2	4	1.32	5.95	11.77	67.02	16.33	8.43	7.47
4	2	1.35	5.88	11.27	78.26	16.33	7.69	7.62
4	4	**1.29**	5.07	11.54	**62.52**	22.31	9.25	8.25
4	6	1.32	5.57	11.01	73.36	28.29	10.17	8.80
6	4	1.45	5.59	11.60	69.10	28.29	9.05	8.92
6	6	**1.29**	**4.74**	**10.45**	75.22	34.26	9.92	9.19-

storage and latency overhead. Thus, we set the values of both parameters to 4.

Effects of join condition featurization ways. In Section 3.2, we mention that compared to simply featurize whether each join predicate appears in the SQL query, our way of additionally featurizing the relations involved in joins are more compact, informative and helpful to the estimations. To verify this claim, we carry out an ablation study by building two ALECE s with different join predicate featurization ways and observing their performance. The comparison results are reported in Table 9.

Table 9: The effects of join predicate featurization ways

Featurization way	Q-error				Size	Building	Latency
	50%	90%	95%	99%	(MB)	Time(Min)	(ms)
Simple [63]	1.36	5.78	**11.48**	73.85	**22.28**	9.42	**8.20**
Ours	**1.29**	**5.07**	11.54	**62.52**	22.31	**9.25**	8.25

Apparently, our method of featurizing join predicates result in better Q-error performance, at the very slightly extra expense of memory cost and latency.

C.4 Effect of Number of Relations in Join

We also investigate the effect of the number of relations involved in a query on ALECE and the alternatives. According to the number of relations involved, the testing sub-queries on STATS's Insert-heavy workload are divided into two categories. The sub-queries in both categories covers ≤ 3 and ≥ 4 join predicates, respectively. Then, we observe each method's Q-error distributions on different categories. As the results in Fig. 6 show, ALECE is able to achieve the best overall performance. All methods have better Q-error estimation performance on queries involving less relations. However, the gap between ALECE's achieved Q-error on queries with less and more relations is far smaller than that of the competitors. This implies that ALECE is effective at understanding the implicit relationships between true cardinalities and complex join patterns.

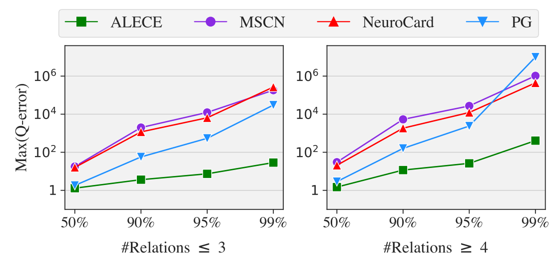


Figure 6: Q-error distributions with number of joins