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 [3] in both modules instead of performing a single function. Fig. 1 depicts its details.

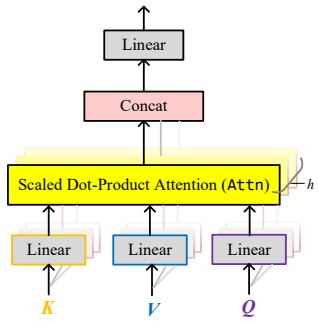


Figure 1: Multi-head Attention, adapted from [3].

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 [3].

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

M3
R4.D1

The *CardEst* benchmark [2] 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, 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 know what the sub-queries are and feed the corresponding cardinality estimates accessed through external estimators, 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 [2] 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 schema. Our improved benchmark [1] 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-queries generation and cardinality estimates replacement. We leave multiple knobs to let users decide when to generate sub-queries and which method's estimation results used to generate the query plans for the queries on a workload $\boldsymbol{W}$.

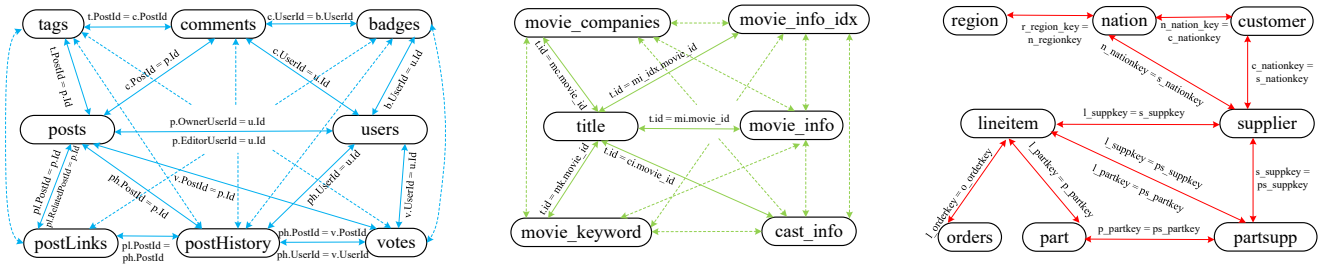
1) Sub-queries generation. This function is related with only one knob: *print_sub_queries*. Given a query, the optimizer needs the cardinality estimations of the sub-queries of two types: the *single* sub-queries only involve single tables whereas the *join* sub-queries cover join conditions. By setting 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_sub_queries.txt`' and '`join_sub_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 [1] for more details.



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

Figure 2: 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. 2 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

M9 M11 R5.D1 R5.D5

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 1 reports the relevant comparative results.

Table 1: 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	-	-	-
	Uni-Samp	15,397	**1.33**	6.64	$>10^{10}$	$>10^{10}$	4.44	**0.02**	413
	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	-	-	-
	Uni-Samp	19,146	**1.32**	3.31	6.00	$>10^{10}$	31.83	**0.48**	187
	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	-	-	-
	Uni-Samp	19,319	1.16	3.09	$>10^{10}$	$>10^{10}$	30.95	**0.14**	26.24
	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 1, the end-to-end performance of cardinality estimators on the static workloads is different from the counterparts on the dynamic workloads. Overall, the performance of the methods on static workloads is better than that on the dynamic workloads. Nevertheless, ALECE still performs best among all methods. ALECE results in up to 1.66, 2.00, 2.58, 1.97 and at least 3 times faster query execution than PG, Uni-Samp, 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 or more than 11.64×, 10^9×, 166×, 77× and 494× smaller Q-error compared to PG, Uni-Samp, 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 that ALECE achieves, the slightly extra overhead is acceptable.

ALECE vs. MLP on static workloads. Referring to Table 1, 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 layers 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 2 reports the comparison results.

M10 R5.D2

As shown in Table 2, the storage overhead of ALECE is clearly influenced by n_{enc} and n_{ana}. When the values of n_{enc} and n_{ana} get larger, ALECE will also result in larger memory costs. However, the Q-error performance of ALECE is not necessarily positively correlated to the numbers of attention layers in its two

Table 2: The effects of the hyperparameters n_{enc} and n_{ana}

n_{enc}	n_{ana}	Q-error 50%	90%	95%	99%	Size (MB)	Building Time(Min)	Latency (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

modules. When the values of n_{enc} and n_{ana} are set to 2, a smaller number, ALECE achieves the worst estimation accuracy. 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 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 versions with different join predicate featurization ways and observing their performance. The comparison results are reported in Table 3.

Table 3: The effects of join predicate featurization ways

Featurization way	Q-error 50%	90%	95%	99%	Size (MB)	Building Time(Min)	Latency (ms)
Simple [4]	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. 3 show, ALECE is able to achieve the best overall performance. All methods have better Q-error distribution 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.

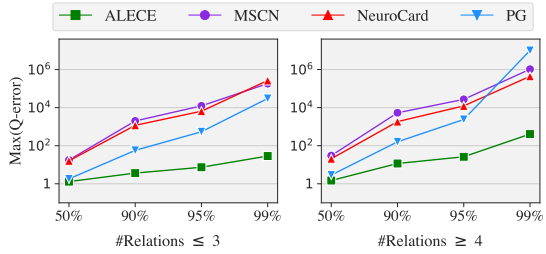


Figure 3: Q-error distributions with number of joins

C.5 Effect of DB State Types

We have conducted additional experiments to investigate the effects of the DB state types. We build three ALECE-variants with different types of the DB states: R5.W3

- **Histogram** is of our used DB state type in the paper.
- **Sample** is built by uniformly sampling from the underlying data. For each relation R, we maintain a sample set S_R of fixed size M. S_R may gets updated when a data manipulation statement referred to the relation R comes in. It is guaranteed that each tuple in R is equal-possibly sampled. In our experiments, the value of M is set to 512. In other words, the dimension of each DB state vector is 512.
- **NE-Hist** is the set of histograms over unequally sized partitions over the value ranges of all attributes. Specifically, we first extract all values of an attribute A from the initial dataset. Then the range of A is partitioned into d_x parts, with each part covering the same number of values. Each element of NE-Hist data featurzations is a vector generated by aligning the value frequency f_i for each i, where f_i refers to the number of values falling in the ith partition. It is noteworthy that the values of all f_i are the same initially. With the data manipulation statements in the workload are executed, the value of f_i will keep changing.

Table 4 shows the Q-error distributions, storage overhead and latency of three ALECE-variants with different types of the DB states on the Insert-heavy workload of the STATS datasets. The results on the other datasets and other workloads are similar and thus omitted.

Table 4: The effects of DB state types

DB state type	Q-error 50%	90%	95%	99%	Size (MB)	Latency (ms)
Histogram	**1.29**	**5.07**	**11.54**	62.52	**22.31**	8.25
Sample	1.52	5.95	12.86	**60.87**	23.23	8.67
NE-hist	1.41	6.01	13.93	77.74	**22.31**	**8.23**

Referring to Table 4, all of the three DB state types are able to help ALECE achieve accurate estimates. It is noteworthy that all of these features can be regarded as the marginal distribution approximation of single attributes. The data-encoder module of ALECE establishes a bridge between the marginal distributions and the joint distribution. It can quantitatively 'calculate' the relevance between a pair of elements from any two DB states. Thus, it can effectively discover the implicit connections between any pair of attributes, which is helpful to the cardinality estimation task.

Nevertheless, the cost of updating histograms when data manipulation statements come in is smaller than that of updating other types of DB states. Thus, our ALECE adopts the simple histograms as the DB states in the paper. In the future, we will investigate more types of DB states.

REFERENCES

[1] online. https://github.com/pfl-cs/ALECE.

[2] 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.

[3] 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.

[4] 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.