Enhanced Featurization of Queries with Mixed Combinations of Predicates for ML-based Cardinality Estimation

Magnus Müller
magnus.mueller@uni-mannheim.de
Database Research Group
University of Mannheim
Mannheim, Germany

Lucas Woltmann
lucas.woltmann@tu-dresden.de
Dresden Database Research Group
Technische Universität Dresden
Dresden, Germany

Wolfgang Lehner
wolfgang.lehner@tu-dresden.de
Dresden Database Research Group
Technische Universität Dresden
Dresden, Germany

ABSTRACT

Estimating query result sizes is a critical task in areas like query optimization. For some years now it has been popular to apply machine learning to this problem. However, surprisingly, there has been very little research yet on how to present queries to a machine learning model. Machine learning models do not simply consume SQL strings. Instead, a SQL string is transformed into a numerical representation. This transformation is called *query featurization* and is defined by a query featurization technique (QFT). This paper is concerned with QFTs for queries with many selection predicates. In particular, we consider queries that contain both predicates over different attributes and multiple predicates per attribute. We identify a desired property of query featurization and present three novel QFTs. To the best of our knowledge, we are the first to featurize queries with mixed combinations of predicates, i.e., containing both conjunctions and disjunctions. Our QFTs are model-independent and can serve as the query featurization layer for different machine learning model types. In our evaluation, we combine our QFTs with three different machine learning models. We demonstrate that the estimation accuracy of machine learning models significantly depends on the QFT used. In addition, we compare our best combination of QFT and machine learning model to various existing cardinality estimators.

1 INTRODUCTION

The ever-increasing amount of data and the high demand for low-latency query response times puts pressure on Database Management Systems (DBMS). To meet the requirements, query engines and query optimization remain the focus of database research. In query optimization, *cardinality estimates*, i.e., estimates for the number of rows in a query result, play a crucial role. In particular, cardinality estimates serve as parameters to cost functions whose outputs determine the decision-making in query plan selection. Research findings indicate that query optimizers benefit greatly from improved cardinality estimates [16, 19]. Another field that benefits from accurate cardinality estimates is approximate query processing, in particular count queries. Here, accuracy is traded for response time [1, 4]. Over the decades, the database research community has examined many cardinality estimation techniques. The earliest ones build on independence and uniformity assumptions [25] and are still common in today's DBMS. Later, many synopses-driven approaches were integrated, from sketches [6, 24] to histograms [18, 26]. Sampling techniques for cardinality estimation are versatile, but selective selection predicates cause inaccurate estimates [3, 30].

Background. For some years now, Machine Learning (ML) has been applied to the cardinality estimation problem [8, 12, 32, 33]. In general, ML means *arbitrary function approximation*. The function that underlays the cardinality estimation problem in databases is

$$\textbf{query} \times \textbf{data} \rightarrow \textbf{cardinality} \tag{1}$$

Note that the data component often remains unmentioned even though it is relevant since the query result of `SELECT count(*) FROM R WHERE R.A < 5` differs, depending on the content in `R`. Nonetheless, in this paper, we assume the data to be fixed and approximate Equation 1 by the two-step mapping

$$\textbf{query} \rightarrow \textbf{vector} \rightarrow \textbf{cardinality} \tag{2}$$

The mapping **query** → **vector** is denoted by *query featurization* and benefits from expert knowledge. Query featurization is necessary since machine learning models do not consume SQL strings. Instead, a numerical representation of the query is consumed. Techniques for query featurization are the focus of this paper. The second mapping **vector** → **cardinality** is the machine learning part and benefits from ML knowledge, in particular algorithm choice and parameter tuning.

Despite the fact that all learning-based cardinality estimation techniques need query featurization, most current approaches do not focus on the query featurization part [12, 32]. For instance, Kipf et al. learn the featurized representation of a query from simple per-predicate featurizations. However, learned query featurization leads to obfuscated query representations that ultimately result in sub-optimal query result size estimates. We argue that a smart query featurization technique (QFT) can be identified and implemented - instead of merely learned. Hence, this paper focuses on improved QFTs.

Main contribution. In this paper, we present three novel QFTs. We compare their pros and cons, in particular how well they represent queries and how different types of ML models benefit from good query featurization in terms of estimation accuracy and the number of queries needed for training. In particular, we present the following findings:

- A formal definition of *good* query featurization, to which we refer as *lossless query featurization*. To the best of our knowledge, we are the first to define requirements for query featurization. The definition is presented in Section 3 and motivates the design of our QFTs.
- We present (one established and) two novel QFTs for conjunctive queries, i.e., queries whose predicates are connected by `AND`. Unlike [33], we consider queries with multiple predicates per attribute.
- We present one additional QFT for queries including both conjunctions as well as disjunctions, i.e., predicates connected by `OR`. To the best of our knowledge, we are the first to consider disjunctions in ML-based cardinality estimation.

- We provide an extensive evaluation showing the effect of QFT on the estimation accuracy of several established machine learning models [5, 12, 32]. As part of our evaluation, we also consider the impact of QFTs on estimation accuracy under query drift and different numbers of training queries.

Outline. The structure of the paper is as follows: In the next section, we outline preliminaries in terms of query featurization and machine learning. Then, in Section 3, we present our three new QFTs and discuss their advantages and limitations in detail. In Section 4, we focus on implementation details of our QFTs as part of learning-based cardinality estimators. Next, we evaluate our new QFTs under different ML models with two data sets in Section 5. In Section 6, we discuss how to generalize our approach, for instance to queries with string predicates and aggregations. Section 7 presents related work. Finally, we draw a conclusion and discuss future work.

2 PRELIMINARIES

This section discusses preliminaries. In Section 2.1, existing query featurization techniques for selection predicates and join predicates are covered. Section 2.2 discusses basics of machine learning models (ML models).

2.1 Query Featurization

In this section, we present existing techniques for query featurization techniques – QFTs. Recall that a QFT encodes a query Q into a numerical vector, named *feature vector*. This feature vector then serves as input to a machine learning model – in particular one used for cardinality estimation. In query featurization, the critical part is the encoding of the predicates and joins in input query Q. In this section, we present existing approaches to featurize selection predicates and join predicates.

2.1.1 Predicate Encoding. First, we focus on the encoding of selection predicates. We describe how existing approaches featurize simple predicates, i.e., a predicate that compares an attribute value to a literal using one of the comparison operators in $\{=, >, <, \geq, \leq, \neq\}$. We present the approaches by [7, 12, 32]. Each simple predicate is featurized in the same fashion in all these approaches - the difference lies in how the per-predicate encodings are combined.

To encode a predicate like `A > 5`, the predicate is split and encoded into three parts. (1) Attribute `A` is encoded in a one-hot vector, i.e., only one entry is set, like 001 for a relation with three attributes, (2) the literal 5 is encoded to $\frac{5-\min(A)}{\max(A)-\min(A)}$, which is always a number in $[0, 1]$, and (3) the comparison operator > is encoded to a 3-entries binary vector, where each entry represents one of $\{=, >, <\}$, so at most two entries can be (meaningfully) set. The featurization of `A > 5` could then look like this:

$$\underbrace{001}_{A}\ \underbrace{010}_{>}\ \underbrace{0.27}_{5}$$

To combine multiple selection predicates, there exist two approaches:

In **Singular Predicate Encoding**, as used in [7, 32], for a table with m attributes, the feature vector $\mathcal{F}$ has $4{\cdot}m$ entries. Since specific entries in the feature vector are reserved for each attribute, there is no need to encode the attribute id itself. For $m = 3$ and a query with predicates `A > 5 AND B = 7`, the query featurization looks like:

$$\overbrace{\underbrace{010}_{>}\ \underbrace{0.27}_{5}}^{A}\ \overbrace{\underbrace{100}_{=}\ \underbrace{0.15}_{7}}^{B}\ \overbrace{\underbrace{000}_{\text{no}}\ \underbrace{0.0}_{\text{pred.}}}^{\text{third attribute}}$$

Note that all entries are set to 0 for attributes for which the query contains no predicate. Further note that there can only be up to one predicate per attribute and there is no (good) way to support disjunctions, so all predicates must be connected by `AND`.

In **Predicate Set Convolution**, as used in [12], each selection predicate is featurized and collected in a set P. A convolution step is applied to P in the ML model. The convolution weights are learned during training. The advantage of Predicate Set Convolution is that it supports multiple predicates per attribute. However, disjunctions are not supported. In addition, Predicate Set Convolution as a QFT does not allow for statements in terms of generalization.

2.1.2 Join Encoding. We now focus on query featurization for queries containing joins. In particular, this section discusses how to handle the tables and join predicates contained in some input query Q. Again, there are two general approaches to encode joins:

With **local models**, as used in [7, 32, 33], one model is built per sub-schema, i.e., either per base table or per join result. To estimate the result cardinality of some query, the selection predicates in the query are featurized and forwarded to the corresponding local models. The advantages are that (1) the size of the feature vector remains small and (2) when data in some tables changes, only the models for sub-schemata containing this table have to be retrained. At first sight, a downside is the number of models, since there are $2^n - 1$ sub-schemata, i.e., combinations of n tables. However, in real applications, this number is reduced by relying on System R formulas [25, 31] where models are built exactly for those sub-schemata for which the assumptions from [25], i.e., uniformity and independence assumptions, do not hold.

On the other hand, a **global model**, as used in [12, 14], represents a single estimator capable to estimate result cardinalities for all queries containing arbitrary sub-schemata of n tables. In global models, the feature vector of a featurized query must also represent the accessed tables. Assuming that tables are joined following their key/foreign-key relationships, any QFT can be adapted to global models by appending a binary vector, where each entry corresponds to a specific table, to the feature vector. For instance, for tables 1, 2, 3, and 4, the binary vector 1101 corresponds to a query where tables 1, 2, and 4 are joined (following their key/foreign-key relationships). For comparison, 0100 corresponds to a query on base table 2. In MSCN [12], the approach taken is slightly different. Each table contained in the query is represented by a unique one-hot vector. Then, all one-hot vectors are collected in one set. Similarly, all join predicates are collected in a separate set.

2.2 Machine Learning Models

In this section, we present two types of Machine Learning (ML) models that are commonly used for cardinality estimation. Here, we present the general concepts of these models. In our evaluation in Section 5, we report the qualities of all model types enhanced with the QFTs presented in Section 3.

In general, ML models are functions mapping vectors of *features* X to a *label* or *target* y. Sometimes, the input vectors are referred to as samples. These are not to be confused with samples

generated by random sampling.

$$f: X \to y, X \in \mathbb{R}^d, y \in \mathbb{R} \tag{3}$$

The number of features d is the *dimension* of the feature space. A major advantage of ML approaches is their capability to retrieve the mapping function on their own. This is called *training*, whereas the application of a model, i.e., the application of the learned function, is called a *forward pass*. In general, training requires much more time than the forward pass. In this work, we concentrate on ML approaches using *supervised learning*. Supervised Learning uses labeled example data during training to derive the function f. If the label is a continuous variable, like a cardinality, we call it a *regression problem*. For the training, it is essential that the mapping in the example data to its labels is deterministic and that the same input produces the same label all the time:

$$\forall x_1, x_2 \in X: x_1 = x_2 \Rightarrow f(x_1) = f(x_2) \tag{4}$$

Nothing is worse for an ML model than the same input leading to different labels. This usually leads to a low prediction quality either by an averaged output or by a preference of the model for a specific label for the same input. In such a case, the entropy is maximal and the model cannot derive any information. Therefore, query featurization must be collision-free to avoid these problems. If two different queries are featurized to the same feature vector, the model sees the same input for both queries but different result cardinalities. This contradicts the requirement of determinism and leads to inferior predictions for the problem.

2.2.1 Neural Networks. Neural networks have been applied to various problems in DBMS [12, 13, 32]. Neural networks are able to solve supervised ML problems by modeling the dependencies within the problem as a black box. With increasing hardware performance over the last years, their training has become feasible on a large scale. Therefore, it seems to be evident that the database community relies on neural networks to solve the complex problem of cardinality estimation. Here, we focus on Multi-layer Perceptron or Feed-forward Networks (NN) and Multi Set Convolutional Networks (MSCN) because previous work has shown that these kinds of neural networks work best for queries with joins on complex schemata [12, 32]. In our evaluation in Section 5, we use the original network architecture from [12] and [32]. To show both the importance and versatility of query featurization, we extend [12, 32] to use our new QFTs.

2.2.2 Gradient Boosting. It has been observed that neural networks can be too complex and their training time takes too long. Hence, [5] proposes to use smaller and faster models, based on Gradient Boosting (GB). GB is a tree-based approach where weak learners are combined based on the residuals of a preceding learner. The GB estimator sums over P weak predictors F_p, each with weight λ_p, and adds a constant c.

$$\hat{f}(x) = \sum_{p=1}^{P} \lambda_p F_p(x) + c \tag{5}$$

In our case, each weak predictor F_p is a simple decision tree. The general structure of GB models makes them very fast in training and forward passes. In our evaluation in Section 5, we show that GB trains and converges faster. Therefore, fewer training samples are required. This leads to improved estimation accuracy and faster setup times per model. Like with the neural network approaches, we combine GB with different QFTs and compare the performance of the QFT × ML model combinations.

All neural networks and GB models are input-agnostic, i.e., for a fixed input vector length, they can work with any numeric vector presented to them. This is very useful in the context of this work since it allows us to vary the QFT without having to modify the models' architecture.

Finally, note that we also tested simpler models, like *linear regression* and *support vector regression*. However, we do not include these ML models in the further discussion and evaluation since their estimates are worse by a significant factor.

3 QUERY FEATURIZATION

Query featurization is the process of encoding a query Q in a numerical vector, named *feature vector*. This feature vector then serves as input to a machine learning model.

This section presents three novel query featurization techniques, where we focus on the encoding of selection predicates. The scope is restricted to simple predicates, where an attribute value is compared to a literal using one of the comparison operators in $\{=, >, <, \geq, \leq, \neq\}$. We consider conjunctions as well as certain disjunctions of predicates. We also consider arbitrarily many predicates per attribute. Our QFTs can be applied to either a single table or, using the techniques presented in Section 2.1, to queries containing joins. Extensions to groupings and certain string predicates are discussed in Section 6.

The better a feature vector represents the predicates in a query, the better the query featurization technique. As part of our assessment of QFTs, we use the terms *information loss* and *lossless* from the field of data compression and apply the following definition.

Definition 3.1 (Lossless Query featurization). The feature vector $\mathcal{F}$ is a lossless query featurization of query Q iff there exists a function from $\mathcal{F}$ to a query $\widetilde{Q}$ such that Q and $\widetilde{Q}$ have the same query result.

In other words, a query featurization is lossless if the feature vector is as expressive as the query. This implies that a featurization is, in the sense of queries with equal results, invertible. Note that the definition is similar to the requirements of a good autoencoder. Note that the lossless property is not either satisfied or violated by some QFT in general. Instead, a QFT satisfies the lossless property for a certain class of queries. When a QFT does not satisfy the lossless property for a class of queries, then information loss occurs, since an ML algorithm, to which feature vectors serve as input, cannot distinguish between different input queries with the same feature vector representation.

The following shows that the previously discussed Singular Predicate Encoding does not satisfy Definition 3.1 for a query Q with $k > 1$ predicates on some attribute A. Recall that Singular Predicate Encoding can represent only one of the k predicates in Q's feature vector $\mathcal{F}$. The information about the other $k-1$ predicates is lost. Thus, $\mathcal{F}$ may represent a selective query with many predicates on attribute A or a less selective query with few predicates on A. In terms of accuracy, this means an ML model struggles to derive an accurate cardinality estimate from the feature vector $\mathcal{F}$. As we will see, Range Predicate Encoding, presented in Section 3.1, is prone to the same problem for queries with many predicates per attribute.

3.1 Range Predicate Encoding

This section presents a straightforward but useful extension of Singular Predicate Encoding as discussed in Section 2.1. More evolved techniques are discussed in the next two sections. Range

Predicate Encoding is a QFT that allows to encode, per attribute, either (1) a range predicate, where both open or closed ranges are supported, or (2) an equality predicate. In existing work, range predicates were discussed in [33], but since their model was optimized for point queries, their cardinality estimate for queries containing range predicates is the sum of cardinality estimates for multiple point queries, which is computationally feasible only for small, discrete ranges. For large ranges, [33] employs a sampling technique.

Our predicate encoding technique builds on the observation that, in databases, all types of point and range predicates can be encoded to closed ranges. For instance, `A = 5` becomes $[5, 5]$ and `A` $\leq$ `5` becomes $[\min(A), 5]$. While the difference between range predicates including or excluding endpoints is often marginal, this can still be addressed: For integer attributes it is easy to see that `A < 5` corresponds to $[\min(A), 4]$ and for decimal attributes we can use a small step size, e.g. $[\min(A), 4.9]$. Since this is beneficial for some ML models, all ranges are normalized to $[0, 1]$ using the min and max values of each attribute.

The benefit of Range Predicate Encoding is that all queries with up to one equality, open range, or closed range predicate per attribute are featurized losslessly. Queries with multiple predicates per attribute are not supported. Neither are disjunctions, so all predicates must be connected by `AND`.

3.2 Universal Conjunction Encoding

The QFTs discussed thus far could be *read of* directly from the query but share one common caveat: Only a limited number of predicates can be encoded in the feature vector without information loss. The problem is inherent to the previous featurization techniques: SQL queries are of arbitrary length but feature vectors have a fixed length!

Universal Conjunction Encoding builds on the observation that both the number of attributes in the data set, denoted by m, and the data domain of each attribute remains constant. Hence, while the same attribute may occur in multiple predicates, we use the fact that no query references more than m distinct attributes. The data-driven idea that follows is to (1) partition the data domain of each attribute, (2) give each partition one entry in the feature vector, and (3) assign a value to each entry that indicates whether the partition it represents satisfies the predicates in query Q. This technique allows us to encode queries with arbitrarily many simple predicates connected via `AND`. To implement this idea, we discretize the domain of each attribute A into $n_A = \min(n,\ \max(A) - \min(A) + 1)$ partitions, where n denotes some maximum number of partitions per attribute, e.g. 64. The feature vector entry corresponding to $v \in A$ has zero-based index $\left\lfloor \frac{val - \min(A)}{\max(A) - \min(A) + 1} \cdot n_A \right\rfloor$. Hence, the partition each entry represents consists of consecutive values. Each feature vector entry indicates via a *categorical value* whether the corresponding partition qualifies the predicates in Q. We use 0 to indicate that no value qualifies, $\frac{1}{2}$ to indicate that some values qualify, and 1 to indicate that all values qualify. The concatenation of the per-attribute featurization yields the total feature vector.

Note that the optimal choice of the maximum number of partitions n depends on the frequency distribution of the values in the m attributes. In general, we observe that each partition covers $1/n$ of the domain of some attribute A. Hence, with n=32, each partition covers roughly 3% of an attribute's domain. In the context of query optimization, this granularity usually suffices. Indeed, our evaluation (Sec. 5.4) supports n=32 as a reasonable heuristic. For attributes with high skew, a larger n may be necessary. Observe that it is easy to extend our approach to choose an attribute-specific n. One could also apply sophisticated partitioning techniques from the field of histograms, like v-optimal [23] and q-optimal [18] partitioning.

Example: Suppose a table with numeric attributes `A`, `B`, and `C`, where $\min(A) = -9$, $\max(A) = 50$, $\min(B) = 0$, $\max(B) = 115$ and `C` contains only values in $\{1, 2\}$. Let n=12 be the maximum per-attribute feature vector length. Then, a query with predicates `A < 7 AND B >= 30 AND B <= 100 AND B <> 66` induces the following feature vector (ignore gray entries for now):

$$\underbrace{1\,1\,\tfrac{1}{2}\,0\,0\,0\,0\,0\,0\,0\,0.32}_{A<7}\ \underbrace{0\,0\,0\,\tfrac{1}{2}\,1\,1\,\tfrac{1}{2}\,1\,1\,1\,\tfrac{1}{2}\,0\,0.48}_{30\leq B\leq 100\ \wedge\ B\neq 66}\ \underbrace{1\,1\,1.0}_{\text{no pred.}}$$

($B\neq 66$ marks the entry for $B = 66$.)

With respect to `A < 7`, note that 7 maps to the fourth entry in the vector of A since, according to the above zero-based index formula, $\lfloor (7 - (-9))/(50 - (-9) + 1) \cdot 12 \rfloor = 3$. This fourth entry is set to $\frac{1}{2}$. All entries to the left are 1 to indicate that values smaller than 7 qualify. Accordingly, all entries to the right, but within the bounds of A's vector, are 0. Since there is no predicate on attribute `C`, and its data domain consists of only two values, `C`'s featurization is the all-one vector 11. To reproduce the rest of the example, handle the predicates on B accordingly.

Algorithm 1 describes the steps taken to featurize a query Q. Ignore the gray lines for now, they are discussed later. In line 1 we define a map that associates each attribute A, where A is an attribute in the relation under consideration, to a vector with n_A entries (see above). Initially, each entry in the mapped vector is set to 1. Then, for each predicate p in query Q, p is decomposed into the attribute A it refers to, its comparison operator op, and the literal val to which A is compared. A and val are then used to compute the index idx of the vector entry corresponding to val. If the vector entry $\mathcal{M}_A[idx]$ is 1, it is set to $\frac{1}{2}$. In particular, $\mathcal{M}_A[idx]$ can only be decreased, e.g., once it is zero, it remains zero. Then in lines 6 to 16, depending on the comparison operator op, vector entries that correspond to attribute values which do not qualify p are set to 0. Hence, each predicate p sets specific entries to 0 (or $\frac{1}{2}$). This captures the property that further predicates in a conjunction can make a query only more selective. The final feature vector, returned in line 21, is the concatenation of all per-attribute vectors.

Note that, strictly speaking, there are no queries that can be featurized losslessly with Universal Conjunction Encoding, unless there are more feature vector entries than distinct values. However, the following lemma captures a convergence property that holds.

Lemma 3.2. *Let Q be some query with an arbitrary conjunction of simple predicates. Denote by $\mathcal{F}_n$ the feature vector as produced by Algorithm 1 for query Q, where n denotes the maximum number of feature vector entries per attribute.*

The sequence $(\mathcal{F}_n)_{n\in\mathbb{N}}$ converges to a lossless query featurization (cf. Definition 3.1).

In the lemma, *converge* means that, as the number of buckets n is increased beyond a certain level, the feature vector does not change anymore. Below this level, each increase in n reduces the information loss. The lemma derives directly from the way each feature vector entry corresponds to a specific partition of an attribute. Hence, for large feature vectors where each entry corresponds only to one distinct value of an attribute, Universal

Algorithm 1 Query featurization using Universal Conjunction Encoding

FeaturizeArbitraryConjunction(Q)

Let $\mathcal{M}$ map attributes to all-one per-attribute vectors
for $p \in$ Predicates(Q)
  $A,\ op,\ val$ = Decompose(p)
  $idx = \frac{val - \min(A)}{\max(A) - \min(A) + 1} \cdot n_A$
  if $\mathcal{M}_A[idx] == 1$: $\mathcal{M}_A[idx] = \frac{1}{2}$
  if $op \in$ {"=", "is"}
    $\mathcal{M}_A[!idx] = 0$
  elseif $op \in$ {">", ">="}
    $\mathcal{M}_A[0 : idx] = 0$
    $min_A = \max(min_A, val)$ // initially $\min(A)$
  elseif $op \in$ {"<", "<="}
    $\mathcal{M}_A[idx{+}1 : \text{Len}(\mathcal{M}_A)] = 0$
    $max_A = \min(max_A, val)$ // initially $\max(A)$
  elseif $op \in$ {"! =", "<>"}
    // do nothing
    $nots_A \cup= val$ // initially empty set
for $A \in$ Attributes($\mathcal{M}$)
  c_A = CountBetween($nots_A, max_A, min_A$)
  $r_A = \max(max_A - min_A - c_A,\ 0)$
  Append($\mathcal{M}_A,\ r_A\ /\ (\max(A) - \min(A) + 1)$)
return Concat_all($\mathcal{M}$) // feature vector $\mathcal{F}$

Conjunction Encoding is a lossless query featurization for all queries with arbitrary conjunctions of simple predicates. For smaller feature vectors, this featurization loses only information up to the size of the attribute partition that a feature vector entry represents.

So far, we did not discuss the gray lines in Algorithm 1. Here, per-attribute selectivity estimates are appended to the feature vector. The per-attribute selectivity estimate of some attribute A is the ratio of A's domain that qualifies by the predicates on A. In the gray lines of Algorithm 1, this estimate is obtained as the size of A's domain that qualifies r_A divided by the total size of A's domain $\max(A) - \min(A) + 1$. This corresponds to an estimate under uniformity assumption like in [25]. The model benefits when the partitions in the feature vector are coarse-grained or when trained on a few queries.

Note that for attributes with sufficiently small domain sizes, one feature vector entry corresponds to one distinct value of some attribute. In our implementation of Algorithm 1, we recognize this case and set entries only to 0 or 1 (but not $\frac{1}{2}$). For brevity, we omitted this aspect in Algorithm 1.

3.3 Limited Disjunction Encoding

To the best of our knowledge, Limited Disjunction Encoding is the first QFT that is designed to take both conjunctions and disjunctions, i.e. predicates connected by AND as well as OR, into account. Limited Disjunction Encoding is essentially a generalization of Universal Conjunction Encoding. Note that others [33] have mentioned disjunctions, but only referred to the inclusion-exclusion principle.

Before we present Limited Disjunction Encoding, we address its limitations - the name already suggests that we cannot handle arbitrary disjunctions. We restrict ourselves to the following class of queries:

Definition 3.3 (Mixed query). A **mixed query** is a conjunction of the compound predicates for an arbitrary subset of attributes. A **compound predicate** P_A for some attribute A is an arbitrary combination (AND/OR) of arbitrarily many simple predicates on A.

For example, the following mixed query asks for orders in the TPC-H dataset from either 1994 or 1996, with July 4th excluded in both years, that are either in progress (P) or finished (F) and with a price range from 1000 to 2000:

```
SELECT count(*) FROM Orders WHERE
(o_orderdate >= '1994-01' AND o_orderdate <= '1994-12'
  AND o_orderdate <> '1994-07-04'
    OR
 o_orderdate >= '1996-01' AND o_orderdate <= '1996-12'
  AND o_orderdate <> '1996-07-04') AND
(o_orderstatus = 'P' OR o_orderstatus = 'F') AND
(o_totalprice > 1000 AND o_totalprice < 2000);
```

Note that the query contains three compound predicates, each enclosed by parentheses. Our example, as well as the original TPC-H and TPC-DS queries, illustrate disjunctions with both categorical data and ordinal data. Note that mixed queries do not have to follow a CNF or DNF form.

As for this class of queries, disjunctions occur only locally, i.e., per attribute, it is sufficient for our approach to address them at this level. The key idea of Limited Disjunction Encoding is to regard each conjunction in each compound predicate as a query that can be featurized using Universal Conjunction Encoding. Then, for each compound predicate, the per-conjunction featurizations can be merged by taking the entry-wise max over all per-conjunction featurizations. This merging technique captures the property that additional disjunctions make queries only less selective. As in the previous section, the final feature vector is the concatenation of all per-attribute vectors.

Example: To illustrate the idea, we featurize the WHERE clause `(A > -2 AND A <= 30 AND A != 7 OR A >= 42) AND B >= 39.5`. Suppose the attributes A, B, C have the min and max values from the example in the previous section. The compound predicate on A consists of two conjunctions. For each conjunction, a featurized representation is generated using Universal Conjunction Encoding. In particular, `A > -2 AND A <= 30 AND A != 7` is featurized to

$$\underbrace{0\tfrac{1}{2}1\tfrac{1}{2}111\tfrac{1}{2}0000}_{-2<A\leq 30 \wedge A\neq 7}$$

and `A >= 42` is featurized to

$$\underbrace{0000000000\tfrac{1}{2}1}_{A\geq 42}.$$

The above per-conjunction vectors must then be merged by taking the entry-wise max, i.e.,

$$\underbrace{0\tfrac{1}{2}1\tfrac{1}{2}111\tfrac{1}{2}00\tfrac{1}{2}1}_{-2<A\leq 30 \wedge A\neq 7 \vee A\geq 42}$$

The compound predicate on B only consists of `B >= 39.5`, which is regarded as one conjunction. The featurized representation is

$$\underbrace{0000\tfrac{1}{2}11111111}_{B\geq 39.5}.$$

As in the example from the previous section, since attribute C is not mentioned in the query, its featurization is the all-one vector 11. The concatenation of the per-attribute vectors gives the final

Algorithm 2 Query featurization using Limited Disjunction Encoding

```
FeaturizeLimitedDisjunction(Q)
1  Let M map attributes to all-one per-attribute vectors
2  for cp ∈ CompoundPredicates(Q)
3      Let V be an all-zero feat. vec. for Attr(cp)
4      for d ∈ Split(cp, "OR")
5          f = FeaturizeArbitraryConjunction(d)
6          V = entrywise_max(V, f)
7      M[Attr(cp)] = V
8  return Concat_all(M)
```

feature vector:

$$\underbrace{0\tfrac{1}{2}1\tfrac{1}{2}111\tfrac{1}{2}00\tfrac{1}{2}1}_{-2<A\leq 30\wedge A\neq 7\vee A\geq 42}\overbrace{0000\tfrac{1}{2}11111111}^{B\geq 39.5}\underbrace{11}_{\text{no pred.}}.$$

For the sake of brevity, we do not include per-attribute selectivity estimates, which were illustrated in gray in the previous section.

Algorithm 2 outlines the implementation of Limited Disjunction Encoding. As before, the input is query Q, and we start with a map from attributes to vectors. Then, for each compound predicate cp, a vector with all entries set to zero is created. The subroutine $\textsc{Attr}(cp)$ returns the attribute of cp, and V contains $n_{\textsc{Attr}(cp)}$ many entries. Recall that cp is a disjunction of multiple conjunctions. Starting in line 4, each conjunction is regarded as a query that serves as input to Algorithm 1 which returns a feature vector f. Then, to capture the property that further disjunctions make queries only less selective, V is merged by taking the entry-wise maximum for each entry in V and its corresponding entry in f. For simplicity, assume that the subroutine entrywise_max knows which entries in f refer to V. The final per-attribute V is stored in $\mathcal{M}$. As before, the concatenation of all per-attribute vectors gives the final feature vector for input query Q.

Since merging feature vectors as in line 6 of Algorithm 2 directly resembles the semantics of `OR`, and the feature vectors to be merged converge to a lossless feature vector by Lemma 3.2, it follows that Limited Disjunction Encoding converges to a lossless query featurization of mixed queries.

4 IMPLEMENTATION

This section outlines how we adopt different ML models to use our QFTs. In our implementation, the code of the QFTs is independent of the ML model. Hence, each ML model may use any QFT. As discussed in Section 2.2, we focus on NN, MSCN, and GB models.

4.1 Local Models

The local model approach can be implemented with both NN and GB because local models operate on sub-schemata and therefore can be any arbitrary ML model [32]. We decide to use the original NN from [32] and a modified version where each neural network is replaced by a GB model. Of course, we adjust the necessary hyperparameters.

Extending NN and GB to use different QFTs is rather straightforward since both models are input-agnostic. In particular, the model's architecture remains as is and only the feature vectors presented to a model change. To achieve this, the code pipeline is adjusted so that queries flow through the configured QFT routine.

4.2 Global Models

As a common representative for global models in cardinality estimation, we extended the MSCN [10] implementation to use our new QFTs. Since, in MSCN, query featurization is intertwined with the model, we briefly outline our adjustments. MSCN uses a three-part featurization dividing the information from the query into three vector sets. The three vector sets are: (1) the joins, (2) the tables and their samples, and (3) the predicates. Our QFTs address the featurization of the predicates. The other two vector sets remain untouched by our implementation. To maintain the set logic of MSCN, we featurize all predicates referencing the same attribute into one feature vector. Each per-attribute feature vector is then labeled by the attribute id and added to a vector set. This vector set serves as the predicate vector set for MSCN, exactly as in the original implementation [10].

5 EVALUATION

This section presents our experimental evaluation. We evaluate all QFT × ML model combinations. In the evaluation, we use two different data sets under different query workloads. We evaluate estimation accuracy under different scenarios, take memory consumption into account, and compare our best QFT × ML model combination to other established cardinality estimators. In particular, we give empirical answers to the following questions:

- Which QFT leads to the best estimation accuracy?
- Does the number of attributes or selection predicates explain dispersion in estimation accuracy?
- Does our approach improve query run times in Postgres?
- Does query drift impact estimation accuracy?
- Does the QFT choice impact training convergence?
- What is the time & memory cost of QFTs and models?
- What QFT × ML model combination do we recommend?
- How does our favored QFT × ML model combination compare to established cardinality estimation techniques?

Data sets & query workloads. We use two real-world data sets together with corresponding query workloads. The first data set, forest cover type (forest) [17] is popular both in the machine learning and cardinality estimation community and contains more than 580k entries with 55 attributes. For forest, we generate a query workload with *conjunctive queries*, i.e., predicates connected by `AND`. We draw k, $1{\leq}k{\leq}55$ distinct attributes uniformly at random and randomly generate a closed range predicate for each. Additionally, we generate $l, 0{\leq}l{\leq}5$ not-equal predicates, for each of the k chosen attributes, that exclude values from the aforementioned range, where l is drawn uniformly at random. For instance, one of the queries from our evaluation is:

```
SELECT count(*) FROM forest
WHERE A7 >= 160 AND A7 <= 225 AND
A8 >= 45 AND A8 <= 237 AND A8 <> 220 AND A8 <> 186
```

In addition, we generate a second query workload with mixed queries, in the sense of Definition 3.3. The generation is the same as for conjunctive queries, except that we repeat the generation for the per-attribute predicates between $m, 1{\leq}m{\leq}3$ times and concatenate them via `OR`. For an example see the query below Definition 3.3. For both conjunctive and mixed queries, we generated 100k training queries and another 25k test queries.

As a second data set, the Internet Movie Database (IMDb) [16] is used. IMDb contains data on more than 2.5 million movies with around 4 million actors from more than 135 years. For testing, we use JOB-light, a collection of 70 hand-written SQL queries containing joins from [12]. The JOB-light queries contain between 2

and 5 joins. The selection predicates are only conjunctions of 1 to 5 predicates on 1 to 4 different attributes. The queries contain at most one range per attribute. For training, 231k generated training queries are used. In all query workloads, since training and test sets are disjoint, test set leakage is avoided.

Error metric. We use the q-error metric [19], defined as $\max(\frac{x}{e}, \frac{e}{x})$, to measure the deviation between a cardinality x and its estimate e. The q-error is a relative and symmetric metric. Essentially all relevant work on ML-based cardinality estimation employs the q-error [5, 7, 8, 12, 32, 33]. In our evaluation, we consider only queries with non-empty results, and all estimates are ≥ 1. Hence, we can ignore that the q-error is undefined for zero inputs. Note that the relative error $\frac{|e-x|}{x}$ is an insufficient metric for its systematic preference to estimators that underestimate [28].

Experimental setup. All QFTs are implemented in Python. For the NN, we use the Keras/TensorFlow implementation provided by the authors of [32]. The GB models are built with lightGBM. MSCN and its modifications are built upon the code published alongside the paper [10]. We ran all experiments on an AMD A10-7870K Radeon R7 machine with 32 GB memory and an NVIDIA Tesla K20c. The neural networks are trained only with the hyperparameters from their papers due to long training times. The GB models, on the other hand, are trained with full hyperparameter tuning implying the presented results are based on the best configurations.

Abbreviations. Throughout this section, the following abbreviations serve as labels in plots:

simple	Singular Predicate Encoding
range	Range Predicate Encoding
conjunctive	Universal Conjunction Encoding
complex	Limited Disjunction Encoding
GB	Gradient Boosting
NN	Feed-Forward Network
MSCN	Multi Set Convolutional Network

Unless stated otherwise, Universal Conjunction Encoding and Limited Disjunction Encoding use 64 per-attribute entries each.

5.1 Quality under all QFT × Model Combinations

In this section, we analyze the estimation accuracy of all QFT × ML model combinations discussed in this paper. Recall that the new QFTs presented in this paper are Range Predicate Encoding (range), Universal Conjunction Encoding (conjunctive), and Limited Disjunction Encoding (complex). Singular Predicate Encoding (simple) serves as a benchmark comparison. The QFTs are combined with the ML models feed-forward neural network (NN), gradient boosting (GB), and Multi-Set Convolutional Network (MSCN).

Figure 1 shows the estimation errors observed for the forest data set. As usual, the bottom and top of a box are the 25% and 75% quantiles, and the middle band is the median. The lower and upper whiskers show the 1% and 99% percentiles, respectively. All boxplots refer to the conjunctive query workload, except for Limited Disjunction Encoding. Limited Disjunction Encoding refers to the mixed query workload, for which, to the best of our knowledge, no good comparison exists. To indicate this difference, we separate the plots of Limited Disjunction Encoding via a vertical line.

We note three things in Figure 1: (1) Given the Singular Predicate Encoding and Range Predicate Encoding QFT, the local model choice, GB or NN, does not make a significant difference. (2) Given the Universal Conjunction Encoding and Limited Disjunction Encoding QFTs, GB and MSCN outperform NN. And (3), given the GB or MSCN model, Universal Conjunction Encoding and Limited Disjunction Encoding clearly outperform the other two QFTs.

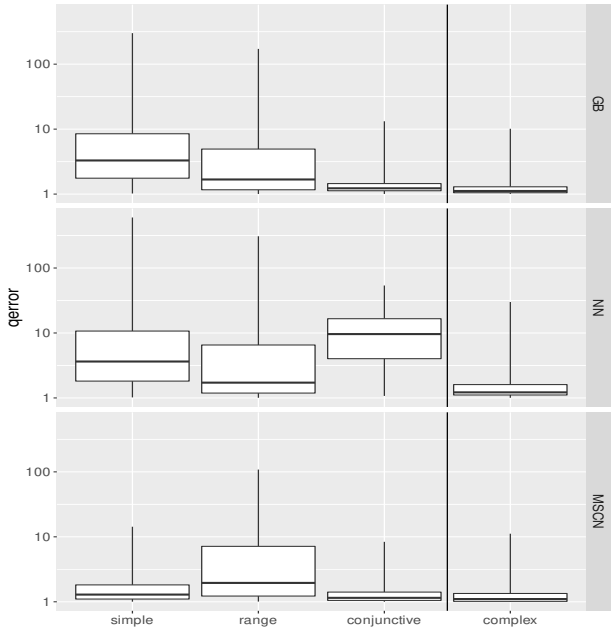

Figure 1: Error distribution by QFT × ML model combination. Dataset is forest. Limited Disjunction Encoding shows errors for mixed query workload. All other QFTs are for conjunctive query workload.

In Figure 2, we explore the estimation accuracy for different numbers of attributes mentioned in the queries. We only show GB since NN underperforms GB over all number of attributes and MSCN performs worse than GB on joins queries, as will be shown later. Observe how the accuracy worsens for all QFTs in the number of attributes. Further observe that Universal Conjunction Encoding outperforms Singular Predicate Encoding and Range Predicate Encoding. Note that while disjunctions are generally regarded as challenging in cardinality estimation, Limited Disjunction Encoding performs about as well as Universal Conjunction Encoding.

Figure 3 shows the estimation accuracy by number of predicates in the queries, again only for GB. Note that queries with two predicates refer to queries with a single range predicate, one predicate for the lower bound and the other one for the upper bound. Further, note that queries with three predicates refer to queries with one closed range predicate from which one value is excluded via a not-equal predicate. For its limited predicate encoding abilities, the observation that only Singular Predicate Encoding struggles with two predicates meets the expectation. Accordingly, as we go from two to three predicates, we observe a spike in the 99% error quantile (upper whisker) of Range Predicate Encoding. Observe that the accuracy further worsens as the number of predicates increases. Universal Conjunction Encoding and Limited Disjunction Encoding perform more consistently over varying numbers of predicates.

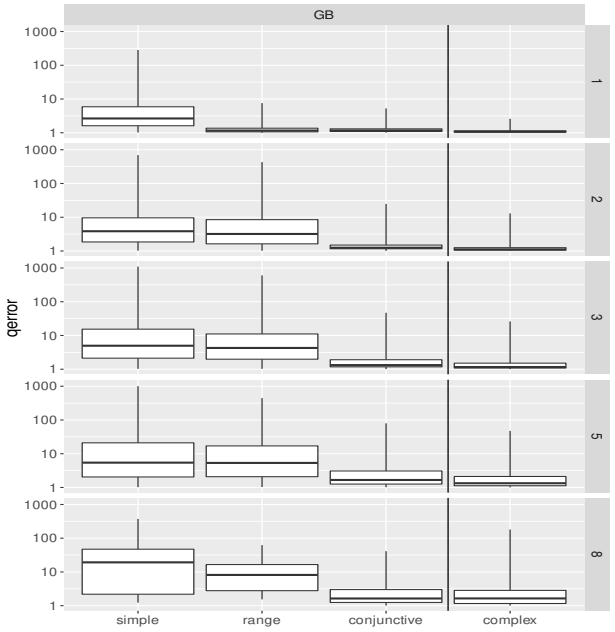

Figure 2: Estimation errors per QFT in the number of attributes mentioned in the queries.

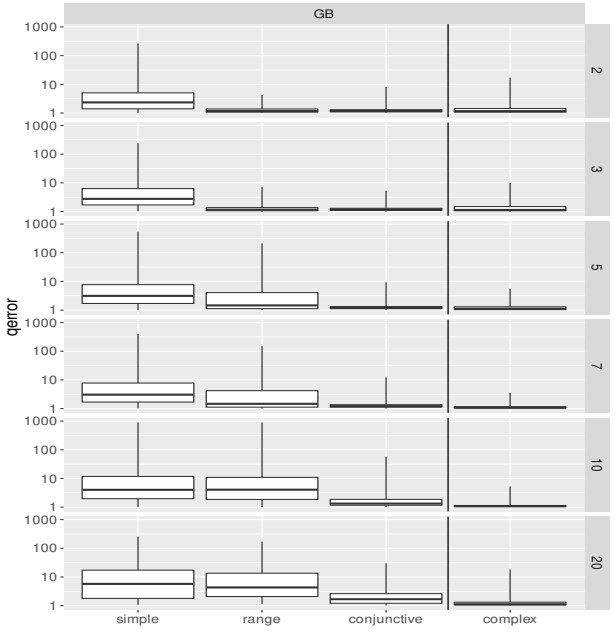

Figure 3: Estimation errors per QFT in the number of predicates in the queries.

For queries containing joins, we report the JOB-light results in Table 1, which shows for each QFT × ML model combination the errors of the mean, median, 99% quantile, and maximum. Since JOB-light consists of only 70 queries, the 99% quantile contains all errors except the worst outlier. For the feed-forward neural network model, Universal Conjunction Encoding significantly outperforms the other two QFTs. Overall, the estimates of GB + range are best. This comes as no surprise since JOB-light queries contain at most one point- or range predicate per attribute. Universal Conjunction Encoding was configured to have

Table 1: 70 hand-written JOB-light join queries

model + QFT	mean	median	99%	max
NN + simple	144.47	10.67	2507.34	3331.07
NN + range	110.23	7.60	2050.50	3573.30
NN + conj	19.97	5.74	129.45	134.37
GB + simple	4.03	1.88	34.06	56.39
GB + range	**3.92**	1.65	**29.77**	**45.51**
GB + conj	8.88	**1.52**	106.10	114.55

Table 2: JOB-light join queries: Local vs. Global Models

model + QFT	mean	median	99%	max
MSCN w/o mods (global)	138.94	11.23	4209	5460
MSCN + conj (global)	119.83	5.26	1465	1811
NN + conj (local)	19.97	5.74	129	134

Table 3: Effect of per-attribute selectivity estimates

model	mean	median	99%	max
GB+conj w/ attrSel	2.65	1.12	20.19	4709.14
GB+conj w/o attrSel	2.93	1.23	25.78	3876.95
GB+comp w/ attrSel	2.95	1.11	18.31	6051.11
GB+comp w/o attrSel	2.92	1.06	16.00	8823.52
NN+conj w/ attrSel	3.65	1.36	19.80	23912.81
NN+conj w/o attrSel	4.00	1.28	16.93	38377.30
NN+comp w/ attrSel	5.08	1.21	37.54	16482.75
NN+comp w/o attrSel	39.74	3.20	268.39	246047.41

8 per-attribute entries for the NN and 32 per-attribute entries for GB. Limited Disjunction Encoding is not shown since JOB-light does not contain disjunctions and, hence, the feature vectors of Limited Disjunction Encoding and Universal Conjunction Encoding are equal.

Recall from Section 2.1.2, that for queries containing joins, there is a difference between local and global models. While Table 1 shows only local models, we address global models separately in Table 2 and focus on MSCN. *MSCN w/o mods* refers to the original MSCN from [12]. MSCN + conj is a modified version that uses our Universal Conjunction Encoding QFT. Observe that the errors in MSCN, both on average and over all quantiles, are significantly reduced with our QFT. In addition, note the gap between the accuracy of global and local models. Since the local model NN and the global model MSCN are both neural network approaches, we again show NN + conj in the last line of Table 2 to illustrate their difference. Observe that NN + conj has significantly lower errors than both MSCN w/o mods and MSCN + conj. Since the global model MSCN struggles with joins, we recommend to use local models.

Recall that, in Universal Conjunction Encoding and Limited Disjunction Encoding, we append per-attribute selectivity estimates to the feature vectors. In Table 3 we investigate the effect of these estimates. For multiple QFT × ML model combinations, we show results obtained with per-attribute selectivity estimates (w/ attrSel) and without per-attribute selectivity estimates (w/o attrSel). Note that in most cases the difference is marginal. However, the benefit is that in all except one case, the worst case error (max) is reduced.

Table 4: End-to-end run times for the JOB-light in PostgreSQL

Postgres	Our approach	True cardinalities
144.95s	142.45s	142.20s

5.2 Comparison with Other Estimators

We identified GB + Universal Conjunction Encoding as the best estimator for conjunctive queries. For mixed queries (conjunctions and disjunctions), GB + Limited Disjunction Encoding is our best estimator. In this section, we compare our best query QFT × ML model combinations to the following estimators:

- *Postgres* is the cardinality estimator from PostgresSQL version 13.2., essentially independence assumption.
- *Sampling* is a 0.1% Bernoulli sample of the data. The sample is drawn independently per query.
- *Multi-set Convolutional Network (MSCN)* without modifications, where we did not use the optional sampling to solely judge the prediction accuracy of the ML model itself.

Section 7 describes the above competing estimators in detail.

Figure 4 shows our best and the competing estimators for both the mixed and conjunctive query workload over the forest data set. In the plot, the query workload is partitioned by the number of attributes in each query. We discuss the *Conjunctive Queries* first. In the plot, note that all estimators lose accuracy with an increasing number of attributes. The accuracy of the Postgres estimator is worse compared to both ML approaches. For sampling, we observe a familiar phenomenon: It works in most cases but has large tail errors. MSCN performs well, but the error varies a lot for different runs as presented in [33]. Note though the difference between MSCN and our approach (*GB + conj*). In some cases, as specified by the number of attributes, we significantly outperform MSCN.

Looking at the *Mixed Queries*, we again note that all estimates worsen in the number of attributes mentioned in the queries. We observe that the Postgres estimator performs worst. For sampling, we again note bad tail errors. Our approach (*GB + complex*) has a slightly larger median and 75% error but significantly lower 99% errors than sampling. Since the standard implementation of MSCN does not support disjunctions, its performance cannot be demonstrated for this query workload.

5.3 End-to-End Performance

We integrated Universal Conjunction Encoding with a global neural network into the PostgresSQL database system. The goal is to measure the improvement in query run times over PostgresSQL with its standard cardinality estimator. We report the numbers for the JOB-light benchmark in Table 4. Observe that the run time improves by only 1.7%. At first sight, this comes as a surprise since, for PostgresSQL's standard cardinality estimator, [12] observed a mean q-error of 174 on the JOB-light benchmark and also Figure 4 indicates weak cardinality estimates. Further investigation reveals that our run times are close to the run times with the true cardinalities, as can be observed by the run time difference of only 0.25s. We believe the only marginal run time improvement is due to the defensive behavior and limited search space of the PostgresSQL optimizer. Note that, in commercial database systems, significantly faster run times for improved cardinality estimates have been observed [16].

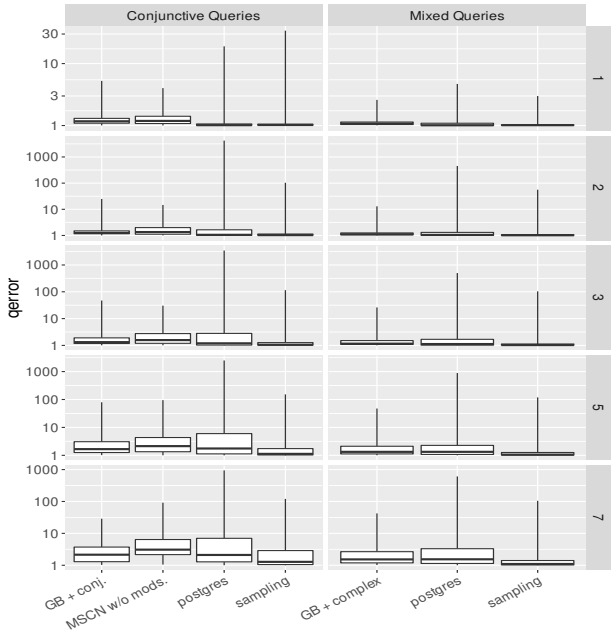

Figure 4: Our best QFT × ML model combinations (GB + conj./complex) are compared to established estimators. We consider both query workloads (conjunctive and mixed) for the forest data set. The figure shows the q-error distributions for different numbers of attributes in the queries.

Table 5: Accuracy for different feature vector lengths

no. entries	Bytes feat. vec.*	mean	median	99%	max
8	72	16.98	1.63	149.51	169.90
16	136	11.49	**1.52**	111.61	123.06
32	264	**8.88**	**1.52**	**106.10**	**114.55**
64	520	20.13	1.90	278.45	313.93
256	2136	86.68	1.69	1347.91	1539.26

*Affects only input layer. Rest of model remains the same.

5.4 Feature Vector Length

This section compares the performance of Universal Conjunction Encoding with different feature vector lengths. In Section 3.2, we discussed that the number of per-attribute entries in the feature vector is a parameter that can be varied. Table 5 shows the accuracy for the JOB-light query workload for {8, 16, 32, 64, 256} per-attribute entries. Note that one additional entry is used for the per-attribute selectivity estimate (printed in gray in the illustration in Section 3.2). Table 5 shows the memory a feature vector temporarily occupies for a featurized query. This is equal to the input layer size of a model. The rest of model stays the same. The ML model used is GB. We observe that, for the JOB-light workload, the best choice is 32 per-attribute entries for Universal Conjunction Encoding. For smaller number of entries, information loss dominates, i.e., the feature vector is too far away from a lossless query featurization, thus causing larger errors. For more than 32 entries, the ML model struggles to learn the patterns encoded in the feature vector, which happens when the feature vector is too long, given the number of training queries.

5.5 Concept Drift

This section is concerned with how well eight QFT × ML model combinations generalize to unseen input/output pairs. In particular, *concept drift* describes the change of input and/or output characteristics. We make the following observation.

Key observation. *At any point in time, the data stored and/or queries executed in a DBMS may change abruptly and drastically.*

This observation does not apply to all machine learning tasks. For instance, cats in image recognition usually do not change their shape abruptly and drastically (evolution takes time).

5.5.1 Query Drift. This section presents an experiment where we simulate query drift. In this experiment, we use query workloads with different characteristics for training and testing. *Low-dimensional* queries, mentioning at most two distinct attributes, are used for training. For testing, *high-dimensional* queries, mentioning at least three distinct attributes, are used. Here, the changed feature vectors are query drift for the ML model. In particular, the more attributes mentioned in the predicates of a query, the fewer entries are set to 1 in the query's feature vectors. In addition, the output characteristics, i.e., the query result sizes, change, making it hard for a model to generalize. The low-dimensional queries, used for training, have a mean query result size of 174 566 and 307 093 for the conjunctive and mixed workload, respectively. The high-dimensional queries, used for testing, however, have mean query result sizes of only 79 805 and 131 376. Hence, to maintain a good estimation accuracy during testing, the model must, on average, produce estimates less than half as large as during training. Figure 5 shows the corresponding plots. The rows with 3, 5, and 8 attributes correspond to the test queries. The rows with 1 or 2 attributes correspond to the training queries, which we normally do not show, but are meant to illustrate the impact of the query drift. For GB, all featurizations generalize well. This can be observed by comparing Figure 5 to Figure 2, which shows the same aspects but without query drift. Taking a closer look, note that the 99% quantile error for the 8-attributes case is larger in Figure 5 than it was in Figure 2. The right column of Figure 5 illustrates the case for the NN. For NN, a clear difference between low-dimensional training queries (≤ 2 attributes) and high-dimensional test queries (> 2 attributes) can be noted. The notable difference indicates that the NN overfits during training, but less for Limited Disjunction Encoding and Universal Conjunction Encoding.

5.5.2 Data Drift. Over time, the data stored in a database changes and all cardinality estimators become outdated. In ML, this is known as *data drift*. This section discusses how to react to data drift. We start by reporting measured run times for learning an estimator for 125k mixed queries for the forest data set. Note though that only 100k of the queries are used for training. We spend 3.5 days generating the queries and 1.5 minutes for featurization. The training depends on the model: 6 seconds for GB, 21 minutes for NN, and 41 minutes for MSCN. Based on these numbers, we conclude that how to obtain queries and result cardinalities is critical. In Snowflake [27], queries and their result cardinalities can be derived from query logs or profiles. Featurization and training are rather cheap, in particular for estimators like GB. Since reinforcement learning is slow when looking at cumulated run times [22], we simply recommend to reconstruct models after data drift occurred. For deciding when to reconstruct, we recommend to follow Larson et al. [15], who propose to base the decision on query feedback.

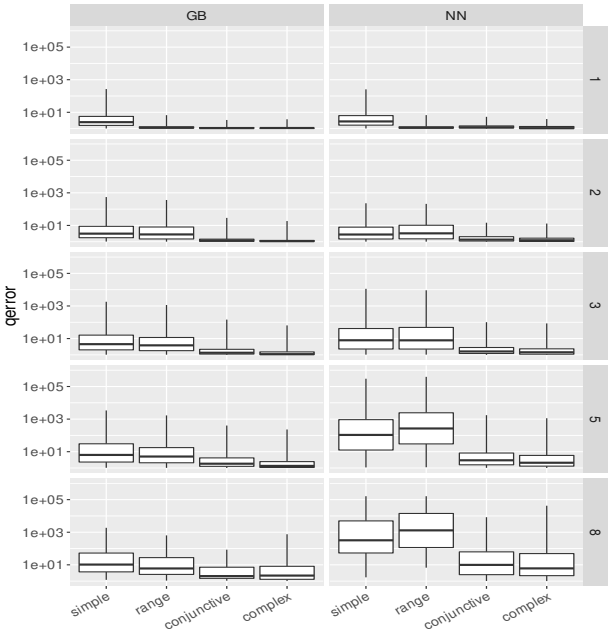

Figure 5: Training on queries with up to two attributes in query. Testing on queries with at least three attributes. Not all featurization/model combinations compensate this *query drift* well.

Table 6: Average estimation error for different model/size/featurization combinations. Forest data set.

training queries	GB conj	GB comp	GB range	GB simple
10k	5.96	4.71	58.23	76.93
20k	4.31	4.11	56.07	63.98
30k	3.83	3.79	45.82	58.32
40k	3.43	3.83	43.74	54.23
50k	3.24	3.72	32.48	51.20
100k	2.93	2.96	32.50	47.29
training queries	**NN conj**	**NN comp**	**NN range**	**NN simple**
10k	28.44	17.91	283.20	386.20
20k	19.70	12.18	232.70	325.50
30k	13.15	10.44	98.17	267.80
40k	19.56	5.88	70.69	313.70
50k	8.32	4.45	57.37	149.02
100k	5.71	5.08	74.2	130.3

5.6 Training Convergence

We report on how the average error changes in the number of training queries in Table 6. First, we observe that for all QFT × ML model combinations, the average error decreases, as the number of training queries increases. Next, note that the errors for NN are significantly larger than for GB. Finally, observe that, given the number of training queries, Universal Conjunction Encoding and Limited Disjunction Encoding have significantly smaller average errors for both GB and NN than the other QFTs.

5.7 Time & Memory Consumption

In Table 7, we report on the time consumption of each QFT. The table shows the average time to featurize the forest workload

Table 7: Time consumption of QFTs for forest dataset

QFT	Simple	Range	Conj.	Comp.
μs per query	21.6	29.7	43.2	72.9

queries. We note two things: (1) All QFTs run fast. All QFTs take significantly less than 100μs to featurize a query. (2) The time to featurize a query grows in the complexity of the QFT, with Singular Predicate Encoding being the fastest and Limited Disjunction Encoding being the slowest.

We also report on the memory consumption of each estimator considered in this evaluation. Since Postgres relies on independence and uniformity assumptions, its estimator's memory footprint is negligible. For sampling, the size of a 0.1% sample depends on the data. For the forest dataset (142 MB in the Postgres DB), the sample consumes around 142 kB. For MSCN, unfortunately, we cannot report the exact memory consumption, but only a lower bound of 320 kB based on the number of trainable parameters. Gradient boosting (GB) is the smallest estimator with only 4.8 kB memory consumption. Thus, we confirm results from [5], who observed that ML models can be small. The NN is the largest estimator with a footprint of more than 1 MB.

6 DISCUSSION

This section discusses extensions and limitations of our approach to make it applicable for general-purpose cardinality estimation.

`GROUP BY` clauses. To the best of our knowledge, thus far, only Kipf et al. [11] covered `GROUP BY` clauses in ML-based cardinality estimation. However, `GROUP BY` clauses can significantly impact query result sizes. We outline how to featurize `GROUP BY` clauses such that combination with any QFT is easy. Suppose a binary vector with as many entries as attributes in the table under consideration, and we have defined some arbitrary mapping from attributes to vector entries. Then, this vector exactly describes the `GROUP BY` clause by setting the entry of each of the grouping attributes to 1. For instance, suppose we have the 5 attributes $A1$ to $A5$, then 01010 exactly corresponds to the clause `GROUP BY A2, A4`.

String predicates. The state-of-the-art approach to support strings is to use a dictionary encoding [33]. This approach works for equality predicates. However, range predicates could only be supported for sorted dictionaries, and `C like a%` predicates are not supported at all. The problem is that they cannot be encoded in the feature vector. However, Universal Conjunction Encoding and Limited Disjunction Encoding naturally support the encoding of such predicates. Consider, for example, a column where all entries are strings of lower case letters, i.e., `[a-z]*`. With 26 entries in the per-attribute vector, each entry corresponds to the most significant letter of a word, e.g. words starting with `d` are represented by the fourth entry. For enhanced accuracy, more entries can be used.

Inclusion-Exclusion Principle. Yang et al. [33] noted that the inclusion-exclusion principle (IEP) helps them with queries that contain disjunctions of predicates, which they cannot featurize. Here, we go into detail to show that the IEP is not a practical, i.e., efficient, solution. The reason is that IEP replaces one cardinality estimation problem with many. In particular, for a query with n predicates connected by OR, the IEP states that an estimate for the result cardinality is obtained by solving $2^n - 1$ cardinality estimation problems whose results must then be accumulated. For the exact formula, we refer to [2]. Here, we present an illustrative example: Consider a query with the `WHERE` clause: p_1 `OR` p_2 `OR` p_3, where each p_i is a predicate like `A > 5`. Let $|p_1 \vee p_2 \vee p_3|$ denote the cardinality of the query result. Then, according to the IEP, $|p_1 \vee p_2 \vee p_3|$ can be computed as:
$|p_1| + |p_2| + |p_3| - |p_1 \wedge p_2| - |p_1 \wedge p_3| - |p_2 \wedge p_3| + |p_1 \wedge p_2 \wedge p_3|$.
Note how the cardinality of a single query with n=3 predicates connected by OR is computed via $2^3 - 1 = 7$ query result cardinalities, each of which must be estimated, and might introduce errors. Hence, we believe the practical approach is to have QFTs that allow to featurize disjunctions of predicates.

7 RELATED WORK

This section discusses related work. First, we discuss established, non-ML cardinality estimation. Then, ML-based cardinality estimation techniques are discussed.

The first cardinality estimation technique is by Selinger et al. [25] and is commonly referred to as independence assumption. For instance, *Postgres* implements this estimator. In fact, the per-selection-predicate estimates rely on uniformity assumptions, i.e., it is assumed that each value in the domain has the same frequency. The overall estimate is calculated as the product of the per-selection-predicate estimates. Many DBMS use frequency histograms to avoid the uniformity assumption [9, 21].

Another established method for cardinality estimation is sampling. In *Bernoulli sampling*, one draws a random sample R' from table R, hence each tuple from R is drawn independently and with the same probability. Let $|R'(Q)|$ denote the number of tuples in R' that satisfy the predicates in some query Q and suppose that R' is a p percent sample of R, then the final cardinality estimate is $\frac{|R'(Q)|}{p}$. For join size estimation, *Correlated sampling* [29] or *Two-Level Sampling* [3] give better estimates than Bernoulli sampling.

Many *sketches* for specific estimation problems were invented, e.g.,. HyperLogLog [6] for estimating the number of distinct values in an attribute. Several join size sketches are described in [24]. An approach to incorporate selection predicates is presented in [20].

For the rest of this section, we focus on well-known ML-based cardinality estimation techniques and their connection to QFTs.

Kipf et al. featurize queries into different sets and learn their cardinalities with a specific Multi Set Convolutional Network (MSCN) architecture [12]. This approach supports both base table and join size estimates with multiple predicates. However, its QFT is lacking domain knowledge and explainability, since it learns an implicit black box featurization through its structure during training.

Woltmann et al. [32] extend [12] to handle arbitrary subsets of a database schema by training several local models. Whereas this approach reduces the disadvantages of Kipf et al., this work only examines one type of QFT for one predicate per attribute.

Naru [33] uses autoregressive models to learn the conditional joint probability of point queries. This introduces overhead for range queries since their estimate is the sum over multiple point queries. Additionally, the order of attributes needs to be fixed, which makes generalization difficult. Note that both [12, 33] combine their approach with sampling to achieve their best results in terms of estimation accuracy. *Naru* uses a very simple QFT that also only allows one predicate per attribute. The authors do not present details about its impact on the models' quality. A similar approach by Hasan et al. [7], who acknowledge the

impact of query featurization on the models' quality but do not further research in this direction.

Dutt et al. [5] present similar approaches to [12, 32], but focus on the estimator models' complexity, like memory footprint and training time. This work supports gradient boosting as a lightweight model architecture but does not detail the impact of QFTs.

DeepDB [8] is a workload-independent approach. It uses Sum-Product-Networks (SPN) to model distributions on the base table level and to combine their outputs for multi-predicate queries over joins without the requirement of labeled example queries. However, it relies to some extent on sampling for finding matching join attributes for the construction of SPN. Similar to Kipf et al., DeepDB implicitly learns a black box featurization for queries.

8 CONCLUSION

We presented three new query featurization techniques (QFTs) and their impact on learning-based cardinality estimation. To the best of our knowledge, we are the first to consider queries containing both conjunctions and disjunctions of predicates for learning-based cardinality estimation. Previous QFTs either do not support disjunctions or only by rewriting to conjunctions. One key aspect of our work is to formally define desired properties of QFTs, i.e., lossless query featurization. Our QFTs are derived from this definition. The experimental evaluation indicates that using our QFTs leads to more accurate cardinality estimates. We find that our QFTs are robust since we examine the query workloads from different view points, like the number of predicates, attributes mentioned, and also query drift. In addition, our QFT × ML model combinations compete well against established cardinality estimators. We demonstrated that our QFTs are model-independent by using them as a plug-in featurization layer for existing ML models. Hence, other researchers may choose to use our QFTs for their work on ML models.

For further research, one can test the influence of our QFTs with even more established ML model architectures to improve their estimation accuracy. In addition, we aim to support featurization for even broader classes of queries, including queries with nested sub-queries, arbitrary string predicates, or group-by clauses.

REFERENCES

[1] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 29–42.

[2] Andreas Björklund, Thore Husfeldt, and Mikko Koivisto. 2009. Set partitioning via inclusion-exclusion. *SIAM J. Comput.* 39, 2 (2009), 546–563.

[3] Yu Chen and Ke Yi. 2017. Two-level sampling for join size estimation. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 759–774.

[4] Graham Cormode, Minos Garofalakis, Peter J Haas, and Chris Jermaine. 2012. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases* 4, 1–3 (2012), 1–294.

[5] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. *Proceedings of the VLDB Endowment* 12, 9 (2019), 1044–1057.

[6] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*. Discrete Mathematics and Theoretical Computer Science, 137–156.

[7] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep learning models for selectivity estimation of multi-attribute queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1035–1050.

[8] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: learn from data, not from queries! *Proceedings of the VLDB Endowment* 13, 7 (2020), 992–1005.

[9] IBM. 2021. *IBM Db2: Histograms in workload management*. https://www.ibm.com/support/producthub/db2/docs/content/SSEPGG_11.5.0/com.ibm.db2.luw.admin.wlm.doc/doc/c0052789.html Accessed: 2021-09-21.

[10] Andreas Kipf. 2019. *Learned Cardinalities in PyTorch*. https://github.com/andreaskipf/learnedcardinalities Accessed: 2022-05-02.

[11] Andreas Kipf, Michael Freitag, Dimitri Vorona, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2019. Estimating filtered group-by queries is hard: Deep learning to the rescue. In *1st International Workshop on Applied AI for Database Systems and Applications*.

[12] Andreas Kipf, Thomas Kipf, Radke, Leis, Boncz, and Kemper. 2019. Learned cardinalities: Estimating correlated joins with deep learning. *CIDR* (2019).

[13] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. 489–504.

[14] Seetha Lakshmi and Shaoyu Zhou. 1998. Selectivity estimation in extensible databases-a neural network approach. In *VLDB*, Vol. 98. 24–27.

[15] Per-Ake Larson, Wolfgang Lehner, Jingren Zhou, and Peter Zabback. 2007. Cardinality estimation using sample views with quality assurance. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 175–186.

[16] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[17] M. Lichman. 2013. UCI Machine Learning Repository. https://archive.ics.uci.edu/ml/datasets/covertype

[18] Guido Moerkotte, David DeHaan, Norman May, Anisoara Nica, and Alexander Böhm. 2014. Exploiting ordered dictionaries to efficiently construct histograms with q-error guarantees in SAP HANA. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 361–372.

[19] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proceedings of the VLDB Endowment* 2, 1 (2009), 982–993.

[20] Magnus Müller, Daniel Flachs, and Guido Moerkotte. 2021. Memory-efficient key/Foreign-key join size estimation via multiplicity and intersection size. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 984–995.

[21] Oracle. 2021. *Database SQL Tuning Guide: Histograms*. https://docs.oracle.com/database/121/TGSQL/tgsql_histo.htm#TGSQL366 Accessed: 2021-09-21.

[22] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2019. An empirical analysis of deep learning for cardinality estimation. *arXiv preprint arXiv:1905.06425* (2019).

[23] Viswanath Poosala, Peter Haas, Ioannidis, and Shekita. 1996. Improved histograms for selectivity estimation of range predicates. *ACM SIGMOD* (1996).

[24] Florin Rusu and Alin Dobra. 2008. Sketches for size of join estimation. *ACM Transactions on Database Systems (TODS)* 33, 3 (2008), 1–46.

[25] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System *(SIGMOD '79)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/582095.582099

[26] Michael Shekelyan, Anton Dignös, and Johann Gamper. 2017. Digithist: a histogram-based data summary with tight error bounds. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1514–1525.

[27] Snowflake. 2017. *Understanding Your Snowflake Utilization, Part 1: Warehouse Profiling*. https://www.snowflake.com/blog/understanding-snowflake-utilization-warehouse-profiling/ Accessed: 2021-11-11.

[28] Chris Tofallis. 2015. A better measure of relative prediction accuracy for model selection and model estimation. *Journal of the Operational Research Society* 66, 8 (2015), 1352–1362.

[29] David Vengerov, Andre Cavalheiro Menck, Mohamed Zait, and Sunil P Chakkappen. 2015. Join size estimation subject to filter conditions. *VLDB* (2015).

[30] TaiNing Wang and Chee-Yong Chan. 2020. Improved correlated sampling for join size estimation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 325–336.

[31] Lucas Woltmann, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2020. Best of Both Worlds: Combining Traditional and Machine Learning Models for Cardinality Estimation *(aiDM '20)*. Association for Computing Machinery, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3401071.3401658

[32] Lucas Woltmann, Claudio Hartmann, Maik Thiele, Dirk Habich, and Wolfgang Lehner. 2019. Cardinality estimation with local deep learning models. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019*, Rajesh Bordawekar and Oded Shmueli (Eds.). ACM, 5:1–5:8. https://doi.org/10.1145/3329859.3329875

[33] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep unsupervised cardinality estimation. *VLDB* (2019).