EncoderForge: Generating Efficient SQL for Encoders in Machine Learning Inference Pipelines

QINGFENG PAN, East China Normal University[†‡], China
QINGYUAN JING, East China Normal University[†‡], China
CHENYANG ZHANG, East China Normal University[†‡], China
JIAHE ZHI, East China Normal University[†‡], China
CHEN XU[*], East China Normal University[†‡], China
HENG LONG, PingCAP, China

Invoking trained machine learning (ML) pipelines to perform intelligent analysis on data stored in databases has gradually become an essential requirement for many applications. Considering performance and privacy demands, prior studies translate ML pipelines into pure SQL queries for native execution. They typically rely on translating encoders into `CASE` expressions. Unfortunately, databases generally do not deeply optimize `CASE` expressions, as they are not first-class citizens in query optimizers, which may cause inefficient performance of generated queries. To overcome this limitation, we introduce join-based translation approaches, which convert encoders into joins, thus leveraging the high-performance join execution of modern databases. Furthermore, multiple encoders may have various join translation combinations, and join-based translations do not always outperform the case-based ones. This leads to a huge space of candidate translation plans that generate queries having drastically varying execution time. Hence, we propose a plan selector to address this challenge. It employs dynamic programming-based and priority-based selection strategies to identify an efficient translation plan from the huge search space. Moreover, we implement an ML2SQL framework, namely *EncoderForge*, deployable as a plugin across various databases. Experimental results demonstrate that queries generated by EncoderForge achieve up to an order-of-magnitude speedup compared with those produced by existing translation approaches.

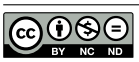
CCS Concepts: • **Information systems** → **Data management systems**; • **Computing methodologies** → *Machine learning*.

Additional Key Words and Phrases: In-database Machine Learning, Category Encoding, SQL Generation

ACM Reference Format:
Qingfeng Pan, Qingyuan Jing, Chenyang Zhang, Jiahe Zhi, Chen Xu, and Heng Long. 2026. EncoderForge: Generating Efficient SQL for Encoders in Machine Learning Inference Pipelines. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 173 (June 2026), 27 pages. https://doi.org/10.1145/3802050

[†]Shanghai Engineering Research Center of Big Data Management
[‡]Engineering Research Center of Blockchain Data Management (East China Normal University), Ministry of Education
[*]Chen Xu is the corresponding author

Authors' Contact Information: Qingfeng Pan, East China Normal University, Shanghai, China, qfpan@stu.ecnu.edu.cn; Qingyuan Jing, East China Normal University, Shanghai, China, qyjing@stu.ecnu.edu.cn; Chenyang Zhang, East China Normal University, Shanghai, China, cyzhangecnu@stu.ecnu.edu.cn; Jiahe Zhi, East China Normal University, Shanghai, China, jhzhi@stu.ecnu.edu.cn; Chen Xu, East China Normal University, Shanghai, China, cxu@dase.ecnu.edu.cn; Heng Long, PingCAP, Shanghai, China, lh@pingcap.com.



ACM 2836-6573/2026/6-ART173
https://doi.org/10.1145/3802050

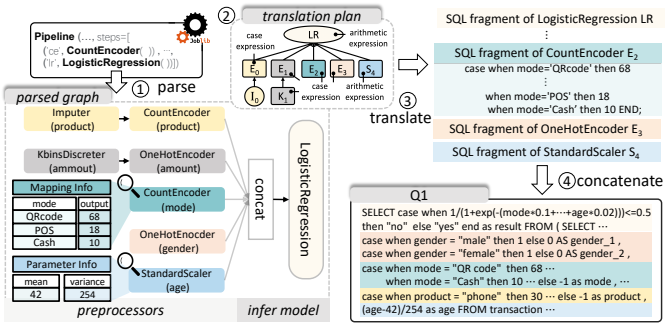


Fig. 1. Translating the Fraud Detection ML Pipeline into Pure SQL

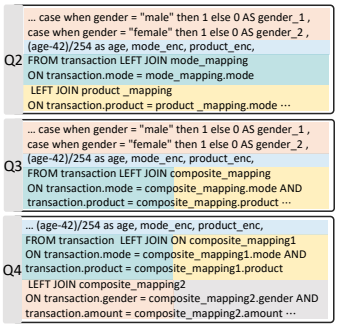


Fig. 2. Equivalent SQL Queries

1 Introduction

Deploying machine learning (ML) inference pipelines in databases to extract deep insights from data is emerging as a promising application direction [25, 26, 36, 38, 45, 60]. In the context of tabular data, traditional ML models like linear regression and tree-based models are widely used and show exceptional effectiveness, outperforming neural network models [12, 18]. Hence, we focus on traditional ML pipelines in this paper. Typically, an ML pipeline consists of multiple preprocessors and an inference model. For example, the fraud detection pipeline in Figure 1 contains preprocessors such as CountEncoder and a LogisticRegression model. Data scientists apply the trained pipeline to transaction data stored in a database to determine whether an order is fraudulent [7].

In general, a database performs such a trained pipeline by invoking ML libraries like scikit-learn in a user-defined function (UDF) [37, 60]. However, this causes the pipeline execution to span across the database and the ML system, introducing overhead such as data conversions and copies. An alternative approach is to implement pipeline semantics within UDF in a customized manner to avoid the overhead. However, this approach suffers from poor portability. Furthermore, databases may provide limited support for parallel execution of UDFs. For example, SQL Server does not allow parallelism in queries that invoke UDFs [50]. In addition, database optimizers typically treat UDFs as black boxes and do not reason about them during optimization [11, 15, 42]. To address these issues, recent works design ML2SQL frameworks to translate ML pipelines into pure SQL queries [16, 34, 36, 54], enabling native execution. In practice, they translate operators in a pipeline into SQL expression fragments and assemble them into a complete pure SQL query. For instance, Q1 in Figure 1 depicts the generated queries of the fraud detection pipeline. Since the generated queries may be executed millions of times for inference on newly arriving data, dominating the overall deployment cost [1, 30, 40], their efficiency is crucial.

Nevertheless, we observe that queries generated by existing ML2SQL frameworks spend significant time executing corresponding `CASE` expressions of encoders in the ML pipeline, resulting in inefficient generated queries. Specifically, the frameworks translate encoders into `CASE` expressions. Taking the encoder on *mode* feature as an example, it encodes payment modes into frequencies and is converted into a `CASE` expression, as illustrated in Q1. The `CASE` expression, essentially an if-then-else structure, introduces string comparison and branching judgments when used to implement encoders. Unfortunately, databases execute such operations inefficiently, even with optimization techniques like just-in-time compilation [14, 31]. We utilize the existing framework to generate SQL queries for multiple public pipelines and execute them in a database. The results show that these queries spend approximately 60%–90% time executing the corresponding `CASE` expressions of encoders, which drives us to revisit their translation approach. In addition, this

observation suggests that databases could further give more attention to the efficient execution of such expressions, particularly in ML-in-DB scenarios. Motivated by the semantics of encoders,we introduce join-based translation approaches that convert encoders into efficient joins. While this enriches the translation approaches, it also necessitates an ML2SQL framework to decide which one to adopt for encoders in a pipeline. This is because a pipeline typically contains multiple encoders, which have various join translation combinations, and join-based translations do not always outperform the case-based ones. This leads to a huge space of candidate translation plans, which generate queries with drastically different execution time.

In this paper, we present *EncoderForge*, an ML2SQL framework that focuses on translating widely used encoders [28] in ML pipelines into efficient SQLs. First, inspired by the characteristics of join operations in databases, we propose two join-based translation approaches, one for single encoders and one for cross-chain encoders, which translate encoders into semantically equivalent joins. Second, to identify a translation plan that produces an efficient query from the candidate translation plans, we devise plan selection strategies, including a dynamic programming-based strategy and a priority-based strategy. In particular, EncoderForge employs one of these two strategies according to the number of encoders in an ML pipeline, so as to strike a balance between plan selection overhead and plan optimality. Our experiments on PostgreSQL show that EncoderForge generates efficient SQL queries with up to 3.2x speedups, in comparison to the ones generated by existing approaches. Moreover, we further evaluate the performance of generated queries on databases such as DuckDB and MonetDB, and the results demonstrate that EncoderForge has good portability.

In the rest of this paper, we describe the background and highlight the motivation of EncoderForge in Section 2. Then, we make the following contributions.

- We propose an *element-as-column join generation* for a single encoder in Section 3, and introduce a novel *cross-chain join generation* for encoders across chains to reduce the number of joins.
- We design a *dynamic programming-based selection* strategy and a *priority-based selection* strategy in Section 4, to find an efficient translation plan effectively from the huge search space.
- We demonstrate the architecture of EncoderForge in Section 5 and it significantly outperforms existing solutions in Section 6.

In addition, we discuss related work in Section 7 and conclude our work in Section 8.

2 Background and Motivation

In this section, we first introduce the background of ML2SQL and encoders in ML pipelines, then present our motivations.

2.1 Background

2.1.1 Machine Learning Pipeline for Inference. An ML pipeline typically comprises multiple preprocessors and an inference model, collectively referred to as ML operators. We focus on ML operators in `scikit-learn` [49] and `category_encoders` [48] libraries, including single-feature preprocessors such as scaling, non-linear transformation, normalization, encoding, discretization, and imputation, as well as linear and small tree-based models. They are widely used in practice [28, 34, 36, 37]. Taking the fraud detection pipeline in Figure 1 as an example, it contains multiple preprocessors, such as StandardScaler and CountEncoder, and a logistic regression inference model. In general, a pipeline provides two core methods, `fit()` for fitting the operator parameters and `predict()` for transforming the newly arriving data and performing inference.

2.1.2 Translating ML operators to SQL. Since parameters are fitted in the training phase, an ML2SQL framework only needs to translate the logic of `predict()` to SQL statements. Specifically, existing frameworks parse a pipeline into a graph containing statistics of ML operators. For instance, the

parsed graph stores the mapping information of the encoder as well as the mean and variance parameters of the scaler. A sequence of preprocessors on the same feature, along with the final model inference, constitutes an operator chain. For the fraud detection pipeline, the graph has five operator chains. Then, the framework traverses operators and specifies the translation approach for each operator to construct a translation plan. Typically, existing frameworks use arithmetic expressions to translate scaling, normalization, and linear models, using `CASE` expressions to translate encoding, discretization, imputation, and tree-based models. Finally, according to the plan in Figure 1, the framework translates ML operators into corresponding SQL fragments, i.e., expressions, and concatenates them to form a complete SQL query Q1. Here, the expressions of preprocessors appear in Q1 as multiple comma-separated items within the `SELECT` clause. Thus far, we obtain a query ready for execution on a target database.

Table 1. Encoders supported by EncoderForge

Type	Encoders
1-to-1	OrdinalEncoder, CountEncoder, TargetEncoder, WOEEncoder, LeaveOneOutEncoder, MEstimateEncoder, JamesSteinEncoder, CatBoostEncoder, GLMMEncoder, QuantileEncoder
1-to-m	OneHotEncoder, BinaryEncoder, BaseNEncoder, GrayEncoder, HashingEncoder, RankHotEncoder, BackwardDifferenceEncoder, HelmertEncoder, SumEncoder, Embedding-based Encoder, PolynomialEncoder, SummaryEncoder

2.1.3 Blessing and Curse of Encoders. Data scientists use encoders to transform categorical features into a numerical format suitable for inference models, enabling models to leverage the statistical information of these features for higher accuracy. In this paper, we focus on about twenty encoders, as listed in Table 1, including classical encoders such as OneHotEncoder and learned encoders like embedding-based encoder. Although these encoders have different semantics, the essence of an encoder is to transform an input k into an output v according to a mapping information, where v is either a scalar or a vector. Therefore, the above encoders are divided into two types, termed as 1-to-1 and 1-to-m, respectively. Here, the possible values of the input and output are limited, forming two finite sets, denoted as K and V, respectively. Hence, an encoder O is represented as Equation 1. For example, the CountEncoder on the *mode* feature transforms the raw *mode* values, i.e., 'QRcode' and 'POS', into 68 and 18, respectively. In general, existing frameworks translate an encoder into `CASE` expressions, as illustrated in Q1.

$$O(k \in K) \rightarrow (v \in V) \tag{1}$$

Time-consuming Encoders. However, encoders are numerous and account for a high percentage in an ML pipeline, causing SQL queries generated by existing ML2SQL frameworks to spend a substantial majority of time on executing corresponding `CASE` expressions. Concretely, the raw input data of a pipeline generally comprises multiple categorical features, and in certain instances, it may include dozens of such features [36, 37, 39, 62]. Related research [62] investigates twenty-eight publicly available datasets covering classification and regression tasks. It reveals that, on average, categorical features constitute 60.5% of one dataset, with certain datasets composed entirely of categorical features. Data scientists employ encoders on these features for accuracy,

which results in queries generated via frameworks containing extensive CASE expressions to express these encoders. In addition, analysis of 508 trained pipelines indicates that most inference models are lightweight models, i.e., linear models and small tree-based models [37], whose computations are relatively simple. As a result, encoders emerge as the dominant performance bottleneck, and our experiments on public ML pipelines show that about 45%–95% of the execution time is spent evaluating CASE expressions of encoders. Unfortunately, databases generally do not deeply optimize CASE expressions, as they are not first-class citizens in query optimizers. Therefore, it is necessary to accelerate the execution of encoders.

2.2 Motivation

2.2.1 Motivation for Join-based Encoder Translation. As described in Equation 1, the transformation process of the encoder, denoted as E, is essentially about matching outputs based on input values. In relational algebra, the role of a join is to match records from one relation with those from another relation, which is analogous to the transformation logic of an encoder. We manually modify the CASE expressions of E_0, E_1 and E_2 in Q1 to joins, thereby obtaining an equivalent query Q2, as depicted in Figure 2. We run two queries on PostgreSQL for 100 million transaction records, and Q1 takes 158 seconds, while Q2 takes 89 seconds. The results indicate that translating an encoder into a join improves query performance.

Inspired by the denormalization technique [46], it is possible to translate multiple encoders in different chains into one join, thereby reducing the number of join operations. For instance, we manually modify the join of E_0 and E_2 in Q2 to a composite join, i.e., the join condition is a composite predicate. The produced query Q3 shown in Figure 2 is semantically equivalent to Q2, and the two colors in the horizontal direction represent that the composite join expresses the semantics of the two corresponding encoders. The experiment results show that Q3 takes 79 seconds, faster than Q2, indicating that generating a join for multiple encoders is profitable.

Hence, translating encoders into joins has the potential to generate more efficient queries. This motivates us to explore join-based encoder translation approaches, thereby seamlessly leveraging the powerful join execution capabilities of databases.

2.2.2 Motivation for Translation Plan Selection. Clearly, there are three approaches to translating an encoder: translating it into CASE expressions, translating it into a join, or translating it along with other cross-chain encoders into a composite join. However, the queries generated by different plans have different execution time.

Specifically, we modify the join of E_1 and the CASE expression of E_3 in Q3 into a composite join and obtain an equivalent query Q4. We run it under the same settings as illustrated above, and the result shows that Q4 runs slower than Q3. Furthermore, we alter the two composite joins in Q4 into one composite join, resulting in the query having only one join operation. However, this query is even significantly slower than Q1, indicating that different translation plans corresponding to these queries have different costs.

In summary, there are various plans to translate an ML pipeline into different equivalent SQL queries. However, these queries have considerable variation in execution time. This motivates us to explore a cost-based translation plan selection strategy, so as to identify the one that generates efficient query out of all candidate plans.

3 Join-based Encoder Translation

As described in Section 2.2.1, it is profitable to translate encoders into joins. Next, we present the join-based encoder translation, including join generation for a single encoder in Section 3.1 and for cross-chain encoders in Section 3.2. We take the CountEncoder and the OneHotEncoder in the

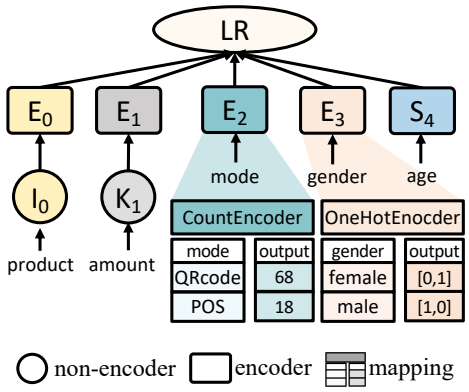


Fig. 3. Mapping of Encoders in the Pipeline

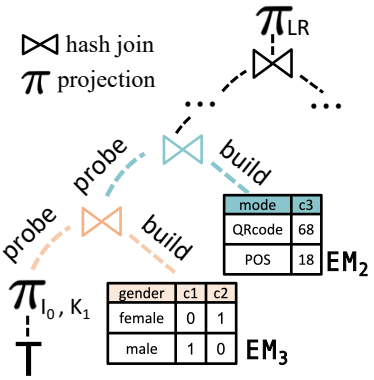


Fig. 4. SQL Plan of Separate Joins

fraud detection pipeline shown in Figure 1 as examples to illustrate how encoders are translated into separate joins and an equivalent composite join.

3.1 Translation for Single Encoder

We first elaborate on the drawbacks of the array-as-column join generation, then present the element-as-column join generation.

3.1.1 Drawback of Array-as-column Join Generation. In fact, Blue Elephants [47] adopts the array-as-column approach to transform encoders in a training pipeline into a join operation. However, applying such an approach to generate a join for encoders in an inference pipeline is inefficient. Taking OneHotEncoder in Figure 3 as an example, the Blue Elephants creates a table where the first column stores the unique values for the *gender* feature, *female*, and *male*. The second column is of type array, storing the corresponding output values, i.e., [0,1] and [1,0]. Then, it performs an equi-join between the training data and the created table to bring the output to the training data, completing the whole transformation process. Subsequently, Blue Elephants conducts data inspection by accessing elements in the array. Since the inspection is typically performed offline, the array access overhead is considered tolerable. Nevertheless, in an inference pipeline, since the downstream model relies on the output of encoders for inference, adopting the above approach results in the generated query frequently accessing arrays during execution. Given that inference pipelines are typically run thousands of times, such overheads are not negligible.

3.1.2 Element-as-column Join Generation. To eliminate the array access overhead, we propose an element-as-column join generation to translate an encoder into an efficient join. In detail, for any type of encoder, i.e., 1-to-1 or 1-to-m, we first create a table in a database for an encoder according to its mapping information, termed as an encoder mapping table, abbreviated as ***EM table***. Note that once the ML pipeline is trained, we create EM tables for all encoders in the database on the fly, which is further explained in Section 5. The created table stores all k in the first column and stores each element of v in an individual column rather than storing v as an array, which is different from the array-as-column generation. The total column number of the EM table is $|v|$+1, where $|v|$ is the number of elements in v. Here, $|v| = 1$ for type 1-to-1 encoders, whereas $|v| > 1$ for type 1-to-m. In addition, we develop a translator that generates a separate join fragment based on the template in Listing 1, to join the EM table with the test table T. Note that the join fragments of the cross-chain encoders are concatenated together and appear as consecutive equi-joins in the final SQL query.

Listing 1. The Separate Join SQL Template

```
v_1, ... , v_|v| FROM T LEFT JOIN EM ON T.k = EM.k;
```

Listing 2. Separate Joins of E_2 and E_3 in the Final Query

```
... c_1, c_2, c_3 FROM T LEFT JOIN EM_2 ON T.mode = EM_2.mode LEFT JOIN EM_3 ON T.gender = EM_3
    .gender ...;
```

Moreover, the translator adds a hint in the generated query to specify the join as a hash join. This is because the created EM table is generally much smaller than T, and the test table is typically not sorted on the join key. Thus, compared to other join algorithms such as nested-loop join and sort-merge join, hash join is more efficient in implementing an encoder, as sort-merge join would incur additional sorting overhead. In practice, mainstream databases like PostgreSQL and SQL Server support enforcing a hash join via a hint [13]. Although certain databases do not support hints, such as DuckDB, they prefer to adopt hash join for equivalent joins [10]. Note that preprocessors like scaling and normalization perform numeric-to-numeric transformations over unbounded input, making them well-suited for translation into arithmetic expressions instead of joins.

Example 3.1. Consider generating two separate joins for CountEncoder E_2 and OneHotEncoder E_3, representing a 1-to-1 and a 1-to-m encoder, respectively. The translator first creates EM_2 and EM_3 in the target database, where $c1$ and $c2$ of EM_3 store the corresponding bits of the output from the OneHotEncoder, as depicted in Figure 4. The translator then generates two separate join fragments and concatenates them into the final SQL query, as shown in Listing 4. Figure 4 depicts the execution plan of the final generated query, which joins transaction table T with two EM tables via hash joins and applies a projection to extract the output values, $c1$, $c2$, and $c3$, thereby completing the transformation.

3.2 Translation for Cross-chain Encoders

Intuitively, it is possible to translate each encoder in the pipeline into a separate join operation thereby leveraging the powerful join execution capabilities of databases. In this section, we first explain the shortcomings of this method and then propose a novel Cartesian product-based composite join generation to address the issues.

3.2.1 Issues of Generating Separate Joins for Cross-chain Encoders. Generating a separate join for each encoder produces a query that contains multiple hash joins, which causes the query to spend substantial time on probing, leading to significant overhead. Specifically, a hash join corresponding to an encoder consists of a build phase and a probe phase. The build phase scans the inner relation, i.e., EM table, and constructs a hash table using a hash function. The probe phase scans the outer relation, i.e., the testing table T, and uses the hash function on each tuple to jump to a location in the hash table to find a matching tuple. For example, consider the SQL execution plan in Figure 4. Database engines typically first build hash tables in memory for EM tables, and then successively perform probing in multiple hash joins over tens of millions of tuples in T. Since T contains orders of magnitude more tuples than an EM table, the probe phase dominates the overall cost.

Denormalization is a technique to reduce the number of joins required during query execution by introducing redundancy into normalized tables. However, if several tables are frequently updated, denormalizing them incurs additional write overhead, because a single update to an original table row requires updating multiple rows in the denormalized table. Hence, while certain practical experiences suggest that denormalization can accelerate complex analytical queries, users must decide whether to adopt it based on their specific needs [9]. In the context of ML pipeline inference,

each EM table is a normalized table whose content is typically determined during training and is not frequently updated afterward. This key characteristic inspires us to apply denormalization to EM tables corresponding to each encoder, thereby reducing the number of joins in generated queries, as elaborated in the next subsection.

3.2.2 Cartesian Product-based Composite Join Generation. To reduce the number of joins derived from encoders across chains, we propose a Cartesian product-based composite join generation inspired by the denormalization technique and characteristics of hash join. The core idea is to reduce the high overhead of probing phases by sacrificing the execution of building phases in a SQL execution plan, thereby accelerating the generated query.

Overall, the translation approach constructs a composite EM table for multiple cross-chain encoders and generates a composite join, rather than separate joins for each encoder. This enables the test data to probe the composite EM table using a composite key, thereby reducing the number of probes. We formally describe the composite mapping process in Equation 2, where O_1 to O_n represent encoders in different chains, and $\times$ denotes a Cartesian product.

$$\begin{aligned}&[O_1(k_1 \in K_1) \rightarrow (v_1 \in V_1), ..., O_n(k_n \in K_n) \rightarrow (v_n \in V_n)] \Rightarrow \\ &O((k_1, ..., k_n) \in K_1 \times ... \times K_n) \rightarrow ([v_1, ..., v_n] \in V_1 \times ... \times V_n)\end{aligned} \tag{2}$$

Specifically, to generate a composite join for cross-chain encoders, the translator uses SQL derived from the template in Listing 3 to perform a Cartesian product against multiple separate EM tables, producing a composite EM table. As a result, the composite EM table contains all possible combinations of rows from the original separate EM tables. In addition, the translator generates a single composite join using the composite join template, rather than multiple separate joins. Note that in the final SQL query, composite joins, along with other separate and composite joins, appear as consecutive equi-joins.

Listing 3. The Product and Composite Join SQL Template

```
-- The Cartesian product SQL Template
CREATE TABLE compEM AS
SELECT * FROM EM1 CROSS JOIN ... CROSS JOIN EMn;
-- The Composite Join SQL Template
T LEFT JOIN compEM ON T.k1=compEM.k1 AND ... AND T.kn=compEM.kn
```

Listing 4. Composite Join of E_2 and E_3 in the Final Query

```
... c1, c2, c3 FROM T LEFT JOIN EM2 ON T.mode = EM2.mode AND T.gender = EM2.gender ...;
```

Note that in most scenarios with structured data, the features processed by encoders do not contain unseen unique values in the training data. This is because the possible values of features like courses, country, or product types are limited and predefined [62]. For such scenarios, the composite join generated through the above approach is semantically equivalent to the original pipeline. Because the composite EM table contains all composite inputs derived from the input values of mappings of encoders, any such input is guaranteed to match a corresponding output in the composite EM table. However, when a feature does not ensure the absence of unseen values in the test data, we disallow generating composite joins for the encoder on that feature to guarantee semantic equivalence.

Compared to generating joins separately, generating a composite join for cross-chain encoders may improve the performance of the resulting query. This is because the generated query replaces multiple separate hash probes with a single composite hash probe, thereby reducing the number

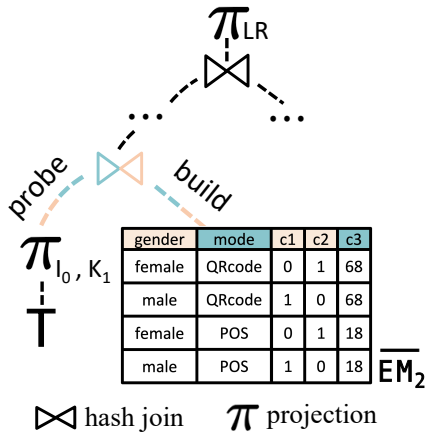


Fig. 5. SQL Plan of Composite Join

of probes and overhead, such as materialization and virtual function calls [31]. Moreover, certain database optimizers, such as PostgreSQL, determine the degree of parallelism based on table size. Given that the composite EM table is larger than the original single EM table and introduces a higher build cost, the optimizer may enable a parallel hash join on the composite EM table, thereby improving query performance. Nevertheless, performing a Cartesian product over n encoders yields a composite EM table whose cardinality increases from $\sum_{i=1}^{n} nuv_i$ to $\prod_{i=1}^{n} nuv_i$, where nuv_i denotes the number of unique values in the i-th encoder. This increases I/O overhead and introduces more hash computations during the build phase, which in turn degrades query performance. Additionally, in PostgreSQL, if the inner table exceeds the size of *work_mem*, the join switches to a grace hash join or a hybrid hash join. This adds an extra hash computation per tuple and introduces disk spill operations, thus causing performance degradation. Therefore, whether to adopt Cartesian product-based composite join generation requires further evaluation.

Example 3.2. For simplicity, we assume the mapping of the encoder on the *mode* feature contains only two unique values. Take generating a composite join for E_2 and E_3 as an example, the translator first creates EM_2 and EM_3, then performs a Cartesian product to construct a larger EM table $\overline{EM_2}$ which has four rows, as depicted in Figure 5. The translator then generates a composite join fragment and concatenates it into the final SQL query, as presented in Listing 2. Clearly, in the SQL plan of separate joins, each tuple in T, such as [..., 'female', 'POS'], performs two probings on EM_3 and EM_2 to match the target values [0, 1] and 18, respectively. In contrast, each tuple in the SQL plan of composite join requires only a single probing to match the combined target value [0, 1, 18] in $\overline{EM_2}$. Given that T contains tens of millions of tuples, the reduction in probing overhead is substantial. In addition, since the size of $\overline{EM_2}$ is comparable to the sum of the sizes of the original EM tables, the composite join introduces negligible additional building overhead. It is feasible to generate a composite join that encompasses more encoders in the same manner, such as E_1, E_2, and E_3, and we do not elaborate further here for the sake of brevity.

Comparison of Three Translation Approaches. Up to this point, we have three approaches for translating an encoder: translating it into a `CASE` expression, translating it into a separate hash join, or translating it along with encoders on other chains into a composite join. However, no single translation approach consistently outperforms the others. Concretely, in database execution engines, hash join and `CASE` expression follow two fundamentally different computational paradigms: the

former relies on key-based mapping to achieve efficient data association and leverage mature join optimization of databases. In contrast, the latter performs row-wise conditional evaluation and benefits from short-circuiting, allowing remaining branches to be skipped once a condition is satisfied, thereby reducing computation cost. Hence, the former two translation approaches may produce SQL that outperforms the other. In addition, choosing between generating separate joins or a composite join involves a trade-off, as we discussed in Section 3.2.2.

4 Translation Plan Selection

The next step is to determine which translation approach to adopt for each encoder. We define the translation plan selection problem in Section 4.1 and describe the solution in Section 4.2 - 4.3.

4.1 Translation Plan Selection Problem

In general, adopting different translation plans to translate an ML pipeline produces multiple equivalent queries, and our goal is to find a translation plan that yields an efficient query. In this section, we first define the translation plan selection problem and then discuss the key challenges involved in addressing it.

Variant of Set Partitioning Problem. The plan selection problem is formally defined as follows. The input includes:

(1) A set of encoders $E = \{e_1, \ldots, e_n\}$ in the input pipeline, where each encoder initially forms an independent group.
(2) Three optional translation designations for a group, including `CASE` expression, separate join and composite join.
(3) Cost model to evaluate the cost of a group $\text{cost}(G_i) \rightarrow \mathbb{R}^+$.

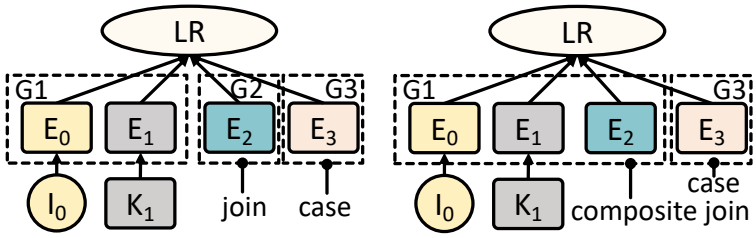
The output is a translation plan $P = (G_1, G_2, ...)$, where G_i is an encoder group, each group is designated a translation approach. The objective is to identify the optimal translation plan P^*:

$$\arg\min_{P} \sum_{G_i \in P} \text{cost}(G_i), \text{ where } \bigcup_i G_i = E,\ G_i \cap G_j = \emptyset \text{ for all } i \neq j$$

The constraint is encoder group designated as join can be partitioned together; A group with $|G_i| = 1$ can only be designated as separate join or `CASE` expression; A group with $|G_i| \geq 2$ can only be designated as composite join.

This is a variant of the set partitioning problem because selecting a plan requires partitioning the encoder set E into disjoint groups G_i, as in classical set partitioning. Here, it is unnecessary to consider the ordering of all groups, since the intermediate result size during execution of generated queries derived from different orders is monotonically increasing. According to the Yannakakis theory [55, 58, 61], the execution time of these queries differs by at most a constant factor. However, the plan selection problem differs in that each group must additionally be designated with one of the three translation approaches, subject to the special constraints that do not exist in the classical set partitioning problem. Here, the generated join or `CASE` expression for encoders in a group G_i incurs different execution cost, and we adopt a cost model to evaluate these costs, which is further described in Section 5.

We can construct a translation plan P through the following two steps. For each encoder in a pipeline, we first designate whether to translate it into a `CASE` expression or a separate join. Then, we partition the encoders designated as separate join translation into disjoint groups. Figure 6 presents two possible translation plans that generate queries with different execution time. Taking the plan in Figure 6(a) as an example, it has three encoder groups. Encoders in G_1 are translated to a composite join, while E_2 and E_3 are translated into a join and a `CASE` expression, respectively. Alternatively, the plan in Figure 6(b) translates E_0, E_1 and E_2 into a composite join. An intuitive



(a) Translation Plan 1 (S4 Omitted) (b) Translation Plan 2 (S4 Omitted)

Fig. 6. Examples of Two Possible Translation Plans

method of finding the efficient plan is to enumerate and compare plans. However, this may lead to an extreme overhead.

Challenge 1: Numerous Designation Options. In the first step, an encoder is designated to apply the case-based translation approach or the separate join translation, resulting in a combinatorial explosion. Consider a pipeline containing n cross-chain encoders. There are a total of 2^n candidate designation options that need to be considered, as different options may have varying costs.

Challenge 2: Explosive Grouping Options. In the second step, it is possible to arbitrarily partition encoders designated as separate join translation to produce all grouping options. Given a designation option including m such encoders, the number of candidate grouping options is $B_m = \sum_{k=1}^{m} S(m,k)$. Here, B is the Bell number and $S(m,k)$ is the Stirling number of the second kind, which counts the number of ways to partition a set of m elements into k nonempty subsets. When m is fifteen, the B_m exceeds one billion.

Comparison with Classical Query Optimization. Database optimizers employ efficient query optimization to find the optimal join orders with the minimal cost from the space of semantically equivalent orders [22, 23, 29, 32, 41, 53]. However, it is fundamentally an order-dependent problem, which is different from the translation plan selection problem. Moreover, optimizers in existing popular databases like Postgres and DuckDB fail to address the translation plan selection problem, because they generally do not support rewriting two joins into an equivalent composite join, nor do they support equivalence transformations between `CASE` expressions and joins. Hence, it is necessary to explore solutions to the problem.

4.2 Dynamic Programming-based Selection

The set partition problem is a classic NP-complete problem, typically solved via dynamic programming [6]. In particular, the translation plan selection problem also exhibits optimal substructure and overlapping subproblems. First, since a plan cost is the sum of costs of all groups, the optimal grouping of all encoders could be obtained by combining optimal groupings of their subsets. Second, different groupings repeatedly involve identical grouping problems on subsets, making memorization effective for avoiding repeated solving. Consequently, we design a dynamic programming-based strategy, detailed in Algorithm 1.

The strategy identifies the translation plan with the lowest cost for an ML pipeline that contains n cross-chain encoders. Specifically, the selection strategy represents a subset of encoders in the parsed graph as a binary mask, i.e., a bitmask, and incrementally increases the bitmask to traverse all subset states (Line 1). First, for each bitmask, i.e., an encoder subset, if it contains only one encoder, the strategy constructs two translation plans: one translating the encoder to a separate join, and the other translating it to a `CASE` expression. Otherwise, the strategy constructs a translation

Algorithm 1 The Process of BitMask DP-based selection

Input: A parsed ML pipeline graph
Output: An efficient translation plan, P^*

```
1:  for bitMask from 1 to 2^n − 1 do
2:      planSet ← enumPlan(bitMask)
3:      subPlan ← costEvaluate(planSet)
4:      memorize(subPlan, bitMask)
5:  return getMinCostPlan(planSet)
6:  function enumPlan(bitMask)
7:      subMask ← bitMask
8:      subPlans ← boundaryPlanGen(subMask)
9:      subPlanSet.add(subPlans)
10:     while (subMask − 1) ≥ floor(bitMask/2) do
11:         subMask ← (subMask − 1) & bitMask
12:         complementMask = bitMask & ~ subMask
13:         subPlan ← combine(subMask, complementMask)
14:         subPlanSet.add(subPlan)
15:     return subPlanSet
```

plan that translates encoders with a corresponding bit being 1 into a composite join (Line 8). The strategy then enumerates all non-empty subsets and combines its optimal translation subplan with the optimal subplan of its complement to build a plan for the current subset (Line 10 - 14). Next, the strategy uses a cost model to estimate the cost of each subplan and memorize the one that has the minimum cost (Line 3 - 4). The strategy finally returns a translation plan for the input pipeline when the bitmask equals 2^n.

Example 4.1. Figure 7 illustrates the dynamic programming process for selecting translation plans for the four encoders in the fraud detection pipeline. The strategy first constructs the optimal subplan for the encoder subset with the bitmask '0001', translating E_3 to a CASE expression. It then increments the bitmask by one and constructs a plan for the next encoder subset. When the bitmask reaches '0011', the subset contains two encoders, E_2 and E_3. The strategy constructs a subplan for the current encoder subset based on the optimal solutions of subsets '0001' and '0010'. It then compares its cost with that of the composite join and selects the one with the lower cost as the optimal subplan. Similarly, when the bitmask increases to '0111', the strategy enumerates all possible subplans for E_1, E_2, and E_3. The strategy iterates the above process until the bitmask reaches '1111', and ultimately selects the translation plan $\{E_0^j\}\{E_1, E_2\}\{E_3^c\}$, i.e., generating a separate join for E_0 and a composite join for E_1 and E_2, and a CASE expression for E_3.

Issues of the Dynamic Programming Selection. Compared to an enumeration strategy, the dynamic programming strategy improves the efficiency of plan selection by reusing solutions, while theoretically ensuring the identification of the lowest-cost translation plan. However, it still requires enumerating all non-empty subsets, which results in an algorithm complexity of $O(3^n)$. For pipelines with a moderate number of encoders, such as around ten, the strategy effectively identifies a plan. Yet, the selection overhead becomes prohibitive as the number of encoders increases. However, in practice, an ML pipeline may contain more than thirty encoders. Therefore, the dynamic programming strategy fails to select a translation plan for complex pipelines effectively.

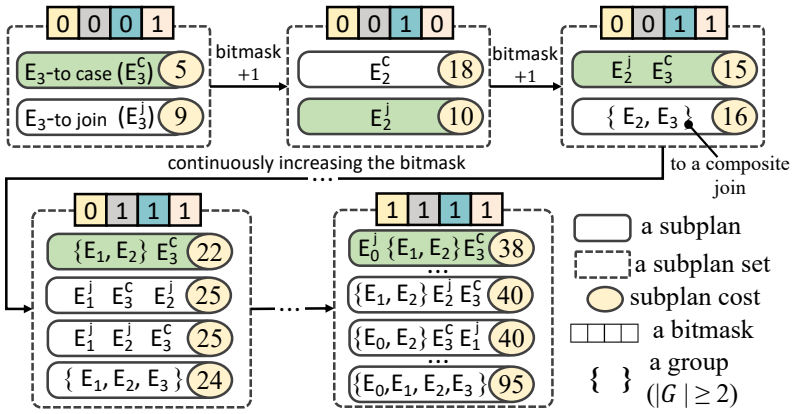


Fig. 7. Dynamic Programming-based Selection

4.3 Priority-based Plan Selection

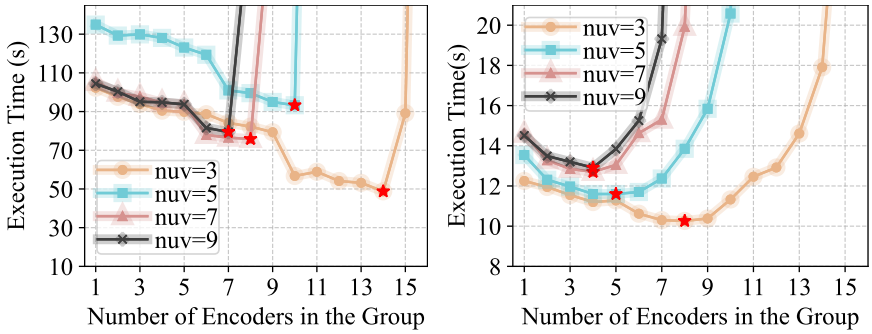
To reduce the complexity, a straightforward strategy is to disable the translation approach for cross-chain encoders. We first discuss its limitations, then propose a priority-aware greedy selection.

4.3.1 Limitation of Disabling Composite Join Generation. Most studies [33, 52, 53], including databases like DuckDB and PostgreSQL, suggest that performing a cross-join between two tables generally produces large intermediate results, which significantly degrade query performance. This indicates that translating cross-chain encoders into a composite join typically degrades the performance of the generated query, compared to translating them into separate joins. Thus, an alternative selection strategy is to prohibit cross-chain encoders from being grouped together, i.e., disabling Cartesian product-based composite join generation, and individually choose a translation approach for each encoder from the other two approaches. Clearly, this strategy bypasses the huge search space discussed in the challenges, reducing the complexity to $O(n)$.

However, EM tables are typically small, so a cross-join over multiple such tables yields a moderate intermediate result. In this context, completely disabling composite join generation prevents discovering better translation plans. Specifically, as elaborated on Section 3.2.2, translating two encoders into a composite join reduces the expensive overhead on the probe phase while increasing that on the build phase. However, for encoders with a small EM table, termed as *small encoders*, the size of their composite EM table, i.e., the intermediate result, is relatively small compared to the input tables. As a result, the increased overhead on the build phase is insignificant and worthwhile. Since ML pipelines typically include multiple small encoders, constructing a translation plan based on the composite-join translation approach is essential for generating efficient queries. For example, the bank pipeline comprises eleven encoders, yet the largest EM table contains fewer than two hundred rows. Clearly, the strategy that disables composite-join translation fails to find a better plan for such an ML pipeline.

4.3.2 Priority-aware Greedy Selection. We design a priority-aware greedy selection driven by the following rationales.

Rationale 1: *Picking case expressions or separate joins locally for each encoder generally yields more efficient queries than applying a single approach to translate all encoders.* Specifically, a `CASE` expression is essentially a sequence of branching instructions over a column, with its cost determined by the number of rows and the complexity of the expression. Similarly, the cost of a hash join



(a) PostgreSQL (b) DuckDB

Fig. 8. Impact of Encoder Group Size

depends on the sizes of its input tables. Therefore, the total cost of the query corresponding to the entire pipeline is equal to the sum of the costs of all joins or CASE expressions. Hence, selecting a translation approach locally for each encoder ensures that the performance of the generated query is at least on par with translating all encoders into CASE expressions or joins, and may even generate more efficient queries.

Rationale 2: *Compared to translating all encoders into separate joins, iteratively prioritizing the grouping of smaller encoders/groups together is more likely to yield efficient queries. However, we have to avoid grouping excessive encoders into one group.* Here, a small group refers to a group where the composite EM table corresponding to the encoders within it is small. Formally, we denote the cost of a hash probe by $\mathbb{C}_{pb}$, and the cost of constructing an entry of a hash table during the build phase by $\mathbb{C}_{bd}$. Assuming a plan applies separate join translation for all encoders in a pipeline, the cost of the plan is represented as $\mathbb{C}_{Sepquery}$. In addition, supposing the optimal plan further partitions encoders into k groups based on the above plan, each containing l encoders, its cost is approximately equal to $\mathbb{C}_{Optquery}$. Therefore, the difference in cost between the two translation plans is presented as follows.

$$\begin{aligned}\Delta = \mathbb{C}_{Sepquery} - \mathbb{C}_{Optquery} &= (\mathbb{C}_{pb} \cdot |T| \cdot n + \mathbb{C}_{bd} \cdot \textstyle\sum_{i=1}^{n} nuv_i) \\ &- (\mathbb{C}_{pb} \cdot |T| \cdot k + \mathbb{C}_{bd} \cdot \textstyle\sum_{i=1}^{k} \prod_{j=1}^{l_i} nuv_{ij}) \\ &= \mathbb{C}_{pb} \cdot |T| \cdot (n-k) + \mathbb{C}_{bd} \cdot \textstyle\sum_{i=1}^{n} nuv_i - \underline{\mathbb{C}_{bd} \cdot \textstyle\sum_{i=1}^{k} \prod_{j=1}^{l_i} nuv_{ij}}\end{aligned} \tag{3}$$

We aim to identify a grouping option that maximizes the difference. Despite various grouping options, the non-underlined part in Equation 3 is equal across all these options. Thus, maximizing the difference is equivalent to minimizing the underlined part. Clearly, to achieve the goal, we should iteratively merge the two smallest encoders/groups into one.

However, if the resulting composite EM tables become excessively large, it dramatically degrades query performance instead. As shown in $\mathbb{C}_{Optquery}$, the term on the left of the plus sign represents the probe cost on the test table, while the term on the right represents the cost of building hash tables for all EM tables. As the number of groups decreases, i.e., more encoders are translated into composite joins, the probe cost decreases linearly, while the build cost increases exponentially. To verify this, we conduct experiments on two databases. In detail, we construct synthetic pipelines, each containing twenty cross-chain encoders and a testing table with ten million rows. All encoders in a pipeline have the same *nuv*. For each pipeline, we increase the number of encoders included in

Algorithm 2 The Process of Priority-aware Greedy Selection

Input: A parsed ML pipeline graph, *graph*
Output: An efficient translation plan, P^*

```
1: // Locally Translation Designating Stage
2: designate(graph.encoders)
3: // Priority-based Grouping Stage
4: encoders ← pickJoin(graph.encoders)
5: sorting(encoders) // sort encoders/groups by nuv
6: while true do
7:     group ← merging(encoders[0], encoders[1])
8:     if costEvaluate(encoders)< previous_cost then
9:         encoders.replaceInsert([0, 1], group)
10:    else plan ← planConstruct(graph, encoders), break
11: return plan
```

a group. As shown in Figure 8, the execution time of generated queries first decreases and then rises sharply. Particularly, on PostgreSQL, after the inflection point, the composite EM table exceeded *work_mem*, causing the join to degrade to a grace hash join.

Based on the above rationales, we propose a priority-aware greedy strategy in Algorithm 2, outlined in two stages. In short, it starts with a translation approach designating, followed by iterative sorting and merging until the termination condition is reached.

❶ **Locally Translation Designating Stage.** This stage reduces the search space while ensuring the performance of the generated query is comparable to or even exceeds that of using a single approach to translate all encoders, as explained in Rationale 1. Specifically, for each encoder, this step uses a cost model to evaluate the cost of executing it via a `CASE` expression or a separate join, then chooses the translation approach with the lowest cost (Line 2).

❷ **Priority-based Grouping Stage.** Stage ❶ designates certain encoders to apply the separate-join translation approach, while stage ❷ performs grouping on these encoders. In detail, it picks out these encoders and sorts them by *nuv* in ascending order (Line 4 - 5). Here, the *nuv* is computed exactly from the number of unique values in encoders, without relying on a cost model. Then, based on Rationale 2, the stage merges the two smallest encoders into a new group, rather than blind exploration of all possible grouping options, thereby further reducing search space (Line 7). Next, this stage compares the cost of the subplan before and after merging to prevent the selected translation plan from producing excessively large composite EM tables, thereby avoiding OOM issues and disk spills. If the merging reduces the cost, this stage replaces the original encoders/groups with the resulting group and inserts it after encoders/groups whose *nuv* is smaller than its own (Line 9). In particular, the *nuv* of a resulting group is the product of *nuv* values of encoders it contains. Otherwise, it terminates the iteration and returns the translation plan (Line 10), since any further merging would degrade the performance of the generated query.

Example 4.2. As shown in Figure 9, the strategy first selects a translation approach for each encoder, adopting the `CASE` expression approach for E_3. Subsequently, it picks E_0, E_1, E_2, and sorts them based on *nuv*. The strategy merges the first two encoders, E_1 and E_2, into a group and compares the cost before and after merging. Since merging reduces the cost from 20 to 17, the strategy replaces E_1 and E_2 with the resulting group and inserts it into the encoders set according to *nuv* order. Next, the strategy attempts to merge the group with E_0 to form a larger group. However,

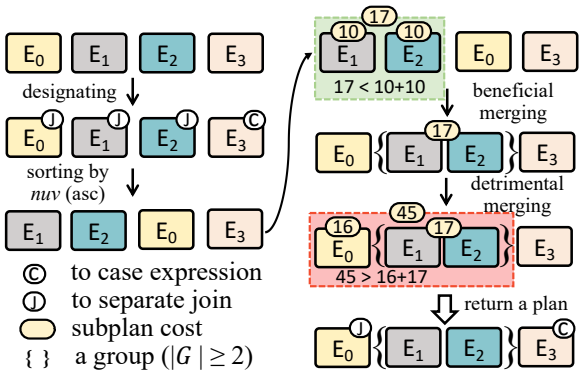


Fig. 9. Priority-aware Greedy Selection

because the merging increases the cost from 33 to 45, the strategy does not adopt this merge and returns the final translation plan.

Discussion. The primary overhead of the priority strategy stems from the iterative merging and the insertion of the resulting group, with a time complexity of $O(n^2)$. While this allows it to select a translation plan for complex pipelines in an acceptable time, it sacrifices plan optimality compared to the dynamic programming-based strategy. For example, for the encoders designated in Step ❶ to be translated using `CASE` expressions, generating composite joins for them may lead to more efficient queries. To balance selection overhead and plan quality, EncoderForge employs two strategies based on the number of cross-chain encoders in the ML pipeline. By default, it applies the DP strategy when the number of encoders is fewer than six; otherwise, it adopts the priority strategy. Note that this threshold is user-configurable. In scenarios such as trading prediction and sensor status monitoring, where generated queries are executed repeatedly on newly arriving data of similar scale, users may increase this parameter to obtain more efficient queries.

5 EncoderForge Architecture

EncoderForge is an ML2SQL framework that translates trained ML pipelines into pure SQL, enabling native in-database execution. In particular, it focuses on converting encoders into efficient SQLs.

EncoderForge provides an interface similar to scikit-learn for building and training ML pipelines. Once the training of a pipeline is completed, it automatically creates EM tables for each encoder in the target database. In our experiments, the sizes of all EM tables, including composite EM tables, range from 8 KB to 11 MB, and creating EM tables in PostgreSQL takes 16 seconds on average (0.1 seconds – 113 seconds), accounting for 11% of the total training time on average (1%–39%). Besides, EncoderForge consists of a *compiler*, an *optimizer*, and a *translator*, as depicted in Figure 10. The compiler extracts operator parameter information from the ML pipeline and parses it into a graph. The translation plan selector of the optimizer employs the dynamic programming-based and priority-based strategies to select an efficient translation plan adaptively. Here, the strategies rely on a learning-based cost model to estimate the generated SQL fragments, like joins. Following prior research [17, 24, 44], we construct the model in two steps. Concretely, the collector in profiler first gathers execution statistics of join operations from the target database via profiling queries. The creator then builds an ML model that predicts the cost of join operations by fitting the statistics. Note that these two steps are performed offline; that is, they only need to be executed once when EncoderForge is initially deployed on the target database. However, if available, it is possible to

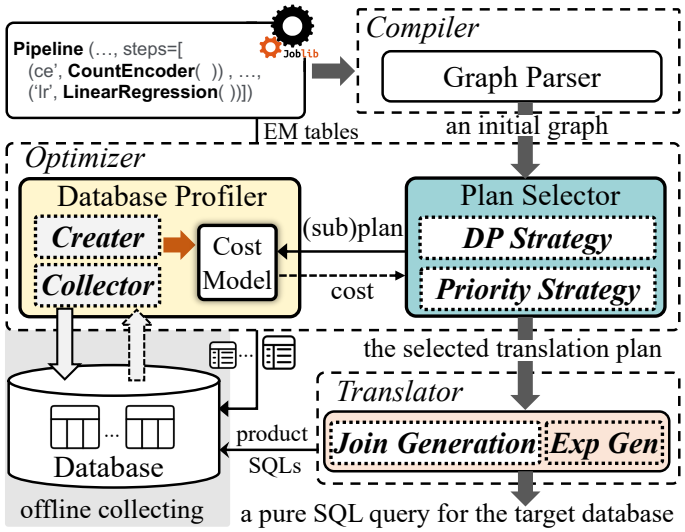


Fig. 10. Architecture of EncoderForge

employ the built-in cost model of the target database to evaluate the cost of plans. The translator generates join or CASE expression fragments for encoders according to the grouping information in the selected translation plan, and uses the generated Cartesian product SQLs to produce composite EM tables. For other ML operators, such as normalization, inference models, the translator converts them into arithmetic or CASE expressions. Finally, it stitches these SQL fragments together to produce an efficient pure SQL query that expresses the input ML pipeline.

6 Experimental Evaluation

6.1 Experimental Setting

We conduct experiments on a server equipped with two Intel(R) Xeon(R) Gold 6126 CPUs @ 2.60GHz, with 256GB of RAM and an 8TB AVAGO HDD. The software stack includes Ubuntu 18.04, scikit-learn 1.3.2, Python 3.11, PostgreSQL 17.2, MonetDB v11.48.0, and DuckDB 1.1.3. By default, we set the parallelism of database engines to four to execute the pipelines.

Workloads. We use eight publicly available datasets that are widely used in related works and data science tasks, and construct eight pipelines for the dataset and provide their detailed description in Table 2. Specifically, P3, P6, and P8 are derived from the public benchmark, FTBench [39], while the remaining pipelines are designed based on the top-performing solutions from Kaggle competitions. To match the testing dataset sizes in production, we scale the size of each dataset to 100M rows by replicating it several folds following the experiments in prior studies [36, 37].

6.2 Effectiveness of Join-based Translation

In this section, we study the effectiveness of two join generation approaches on PostgreSQL. The baseline follows the approach in existing ML2SQL frameworks such as Raven [37], which translates encoders within pipelines into CASE expression (denoted as *case-based*). Besides, we use the approach of Blue Elephants [47] to convert encoders into join (denoted as *blue elephants*).

6.2.1 Effectiveness of Translation for Single Encoder. To evaluate the effectiveness of translation for single encoder, we translate each encoder in pipelines into a separate join via the approach presented

Table 2. Overview of Experimental Machine Learning Pipelines

ID	Dataset	# of Features (num./cat.)	Pipeline Description	Origin
P1	UkAir	8 (5/3)	SS(5), OHE(1), CE(2), 3-53 nuv	OpenML
P2	Job	9 (1/8)	KBins+OHE(1), OHE(8), 3-20 nuv	Kaggle
P3	Adult	14 (6/8)	KBins+OHE(6), OHE(8), 2-40 nuv	FTBench
P4	Bank	14 (3/11)	SS(3), LE(11), 2-172 nuv	Kaggle
P5	Cat	22 (5/17)	MS(5), OHE(11), LE(6), 2-11932 nuv	Kaggle
P6	Criteo	33 (13/20)	OHE(7), BE(9), LE(4), 3-108192 nuv	FTBench
P7	Mushroom	21 (0/21)	OHE(22), 2-12 nuv	Kaggle
P8	KDD	162 (95/67)	KBins(46), SS(49), OHE(53), LE(14), 2-19938 nuv	FTBench

Meaning of Abbreviations: SS-StandardScaler, OHE-OneHotEncoder, CE-CountEncoder, KBins-KBinsDiscretizer, LE-LabelEncoder, MS-MinMaxScaler, BE-BinaryEncoder

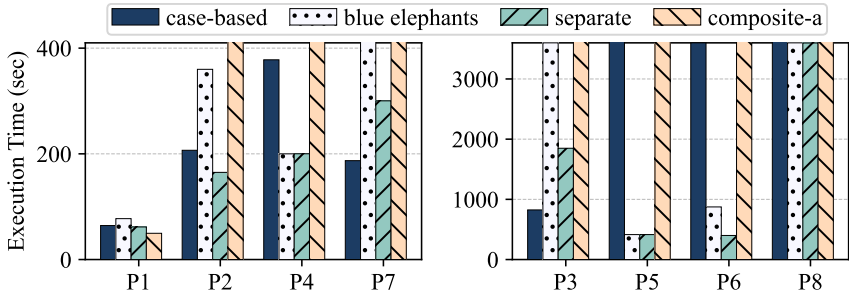


Fig. 11. Execution Time of Generated SQL Queries

in Section 3.1, denoted as *separate*. Figure 11 shows the execution time of queries generated using the case-based, blue elephant, and the separate approaches, on PostgreSQL.

Overall, compared to the case-based approach, the separate approach generates faster queries but reduces the performance of generated queries on certain pipelines. Specifically, for P2, P4-P5, and P8, queries generated by the separate approach are 7.20x faster on average, ranging from a speedup of 1.25x to 20.20x. In particular, P6 includes multiple encoders with a huge number of distinct values. Executing these encoders via case expressions introduced extensive branching judgments and string comparisons, resulting in query execution times exceeding one day. In contrast, the hash join efficiently accomplishes the encoding process through hash matching, taking less than seven minutes. However, for P7, the execution time of the generated query via the separate approach increases by 60.5%. This is because the encoders in this pipeline have only a few unique values, so executing them with `CASE` expressions is more efficient. Note that, for P3, the generated query is about one thousand seconds slower than the query generated by the case-based approach. This is because PostgreSQL executes the former with a parallelism of one, whereas the latter is executed with a parallelism of four. Moreover, the Blue Elephants adopts an array-as-column layout to create EM tables, which introduces extra array access overhead. Hence, the queries it generates are generally slower than those produced by our separate approach.

To further explore the performance differences of various implementations, we pick twenty-six representative encoders, including OneHotEncoder, LabelEncoder and BinaryEncoder, with the number of unique values (*nuv*) ranging from 2 to 11926. Figure 12 shows the execution time of queries using `CASE` expressions and three common types of joins. We sort the encoders by the *nuv*

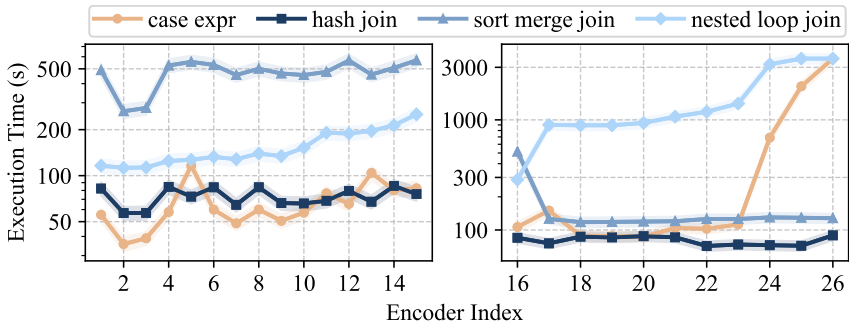


Fig. 12. Performance of Different Implementations

in ascending order, and the x-axis denotes their indices. The results show that nested loop join and sort-merge join are always slower than hash join because they introduce additional overhead such as sorting, as mentioned in Section 3.1. When the value of *nuv* is small, neither CASE expression nor hash join consistently outperforms the other. However, as *nuv* increases, hash join gradually outperforms the CASE expression, and the performance gap continues to widen.

In summary, translating encoders into separate joins is beneficial, especially when the *nuv* of the encoder is large. However, translating via CASE expressions may also yield more efficient queries.

6.2.2 Effectiveness of Translation for Cross-chain Encoders. To evaluate the effectiveness of translation for cross-chain encoders, we aggressively translate all encoders in a pipeline into a composite join, denoted as *composite-a*, as depicted in Figure 11.

Overall, compared to case-based and separate approaches, it generates faster queries on P1; however, it causes a substantial performance drop on other pipelines. In detail, for P1, by reducing probe overhead, the query generated using the composite-a approach is 1.24x faster than that of the separate approach. For other pipelines, the generated queries are slower. For certain complex pipelines like P5, generating the composite EM table takes more than an hour. Therefore, although generating a composite join for cross-chain encoders has potential, it should not be used recklessly.

In conclusion, compared with the case-based translation approach adopted in existing studies, the two join-based approaches generate faster or slower queries. Hence, relying on a single approach to translate all encoders for a pipeline may miss more efficient queries. This highlights the necessity of exploring a translation plan selection strategy, which is further studied in the next section.

6.3 Efficiency of Translation Plan Selection

In this section, we study the efficiency of different translation plan selection strategies, as well as the performance of queries generated by the plans they select. In particular, we evaluate dynamic programming-based (DP) and priority-based selections against two baselines. The enumeration strategy exhausts all translation plans, evaluates their cost, and chooses the one with the lowest cost, denoted as *enum*. Another strategy, introduced in Subsection 4.3.1 disables translation for cross-chain encoders, referred to as *disable*.

6.3.1 Effect of Different Translation Plan Selection Strategies. To evaluate the effect of different selection strategies, we measure their compilation time, i.e., the time for identifying an efficient translation plan for an ML pipeline. In particular, we restrict the search time to one hour; that is, the selection process is terminated if the strategy fails to return a plan within an hour.

As depicted in Figure 13, the enumeration strategy returns a plan for P1 in a short time, since it only has fifteen candidate plans. However, as the number of encoders increases, the number of

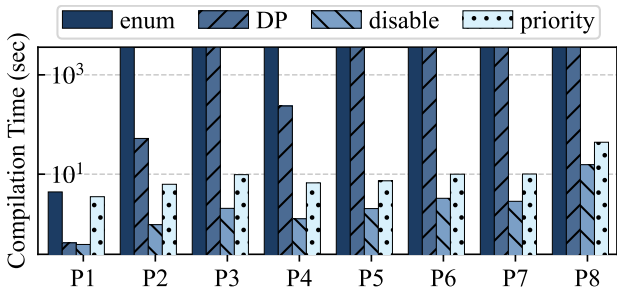


Fig. 13. Compilation Time on Plan Selection

candidate plans grows rapidly, which prevents the enum from returning an efficient plan within an hour. By contrast, the DP strategy avoids redundant computation by reusing solutions to subproblems, thereby improving selection efficiency. For P1, it takes less than one second. For P4 which has about four million candidate plans, the DP selects an efficient plan in four minutes. Nevertheless, as we discussed in Section 4.2, the DP strategy requires enumerating all non-empty subsets with a complexity of $O(3^n)$, causing the plan selection for the remaining pipelines to exceed one hour. We also evaluate the accuracy of the cost model. On Postgres, it yields an MAE of 17.96, an R^2 of 0.81, and an average median Q-Error of 1.31 across workloads, which is comparable to the accuracy reported in prior studies.

In contrast, the disable strategy substantially prunes the search space by disabling the translation for cross-chain encoders, thus returning a translation plan for a given pipeline quickly. However, it misses more efficient plans, which is discussed in the next subsection. As for the priority strategy, it exploits the rationales introduced in Subsection 4.3.2 to select efficient plans within a short time, in a greedy manner. In particular, for complex pipelines such as P8, which contains sixty-seven encoders, the priority strategy takes about forty seconds to select a plan. Note that, for plain SQL queries, spending about one minute on selecting a plan is unacceptable. However, this is worthwhile and acceptable for a trained ML pipeline, because a pipeline is typically executed thousands to millions of times for inference [1, 30, 40], allowing the optimization overhead to be amortized across multiple executions. Hence, the priority strategy efficiently selects a translation plan, which addresses the issue of high selection overhead in DP strategy fundamentally.

6.3.2 Performance of Generated SQL Queries. A trained ML pipeline may be executed thousands of times, so the efficiency of the generated query is crucial. Figure 14 shows the execution time of queries generated by translation plans selected via different strategies.

Specifically, because the DP strategy explores all candidate plans, it finds the optimal translation plan and thus generates efficient queries for P1, P2 and P4. Compared with the separate strategy, the queries generated by the DP strategy achieve an average speedup of 1.54x. For P1, the selected plan of the DP translates all encoders into one composite join. For P2 and P4, the selected plans of the DP translate encoders in the pipelines into multiple composite joins.

As for the disable strategy, since it does not fully explore the search space, the queries generated from the translation plans it selects for P1, P2 and P4 are slower than those from the DP strategy. However, for certain pipelines where the DP strategy fails to return a plan within one hour, the disable strategy produces more efficient queries than the case-based and the separate strategies. For P5, compared with the faster query produced by the case-based and the separate strategies, the query generated by the disable strategy is 1.24x faster. In particular, for P3, the disable strategy generates a query that is 108% slower than the query generated by the case-based approach. The

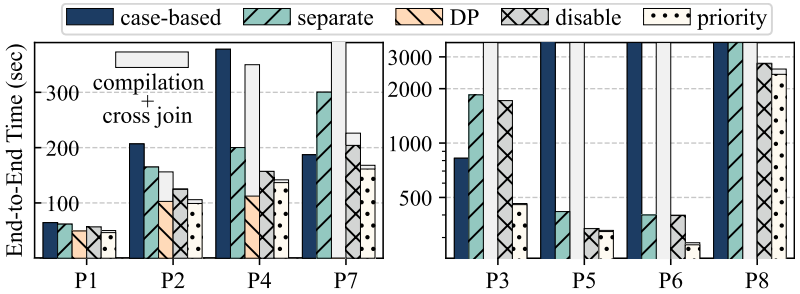


Fig. 14. End-to-end Execution Time

reason is that PostgreSQL runs the latter with a degree of parallelism (DOP) of four, whereas the former is executed with a degree of one, even though the maximum DOP is set to four.

Unlike the disable strategy, the priority strategy considers the translation for cross-chain encoders, thereby generating more efficient queries. Concretely, for P1 and P2, the performance of the generated queries is comparable to that generated by the DP strategy. For P3 and P6-P8, compared to the fastest query generated by other strategies, the generated queries of the priority strategy achieve a speedup of 1.39x on average, ranging from 1.14x to 1.81x. For P6 and P8, the selected plans translate encoders using a combination of `CASE` expressions, separate joins, and composite joins. For the remaining pipelines, the selected plans translate encoders using a combination of `CASE` expressions and composite joins. Taking P2 as an example, the plan selected by the disable strategy translates encoders in the pipeline into four `CASE` expressions and five separate joins. In contrast, for encoders converted as separate joins, the plan returned by the priority strategy translates them into two composite joins, reducing the overhead of probing. Hence, compared to other strategies, the DP strategy and priority strategy effectively select more efficient translation plans, thereby improving the execution performance of ML pipelines.

We further evaluate the end-to-end time, including the execution time of the generated queries, represented by the colored bars, and the compilation time together with the time spent performing cross-joins, represented by the gray bars. Note that the time spent on performing cross-joins does not exceed two seconds. As shown in Figure 14, the priority-based strategy outperforms other strategies, as it generates faster queries in a short time. Although the compilation time on P8 of the priority strategy is almost one minute, it only accounts for 2.5% of the total execution time, which is thus negligible. In addition, since the DP requires more time to select a translation plan, it typically results in longer overall execution time. For example, its total time on P4 is 146.8% slower than that of the priority strategy. Nevertheless, the query generated by the DP strategy is 1.21x faster than that generated by the priority strategy.

In summary, the DP strategy selects plans that generate more efficient queries. However, it fails to cope with the huge search space issue in complex pipelines, while the priority strategy efficiently returns the plans, addressing the issue fundamentally.

6.4 Discussion

6.4.1 Scalability. We vary the dataset size from 100 to 1 million rows to evaluate the scalability of EncoderForge (denoted as EF), measuring the execution time and throughput of generated queries without limiting parallelism. In general, across eight workloads, EF achieves speedup ranging from 1x to 20x on execution time. For simple pipelines, such as P1, EF chooses to translate all encoders into `CASE` expressions, producing queries identical to those generated by the case-based

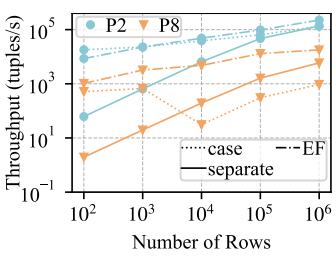


Fig. 15. Scalability of EF

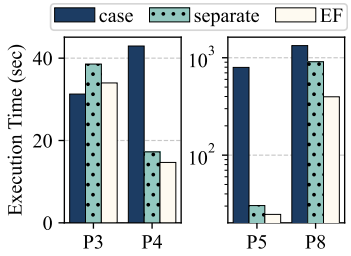


Fig. 16. EF on DuckDB

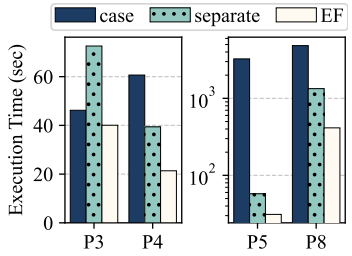


Fig. 17. EF on MonetDB

approach. For the remaining pipelines, EF is comparable to or outperforms the latter. For instance, on P2 with 100 rows, EF is slightly slower, which we attribute to a cost model misprediction. However, EF generates faster queries as the dataset size increases. For P5, P6, and P8, EF always significantly outperforms the case-based approach. As for throughput, EF generally outperforms other approaches. Note that throughput improves as the number of rows increases, as the overhead of operations such as building hash tables for EM tables can be amortized. We present results for representative pipelines P2 and P8 in Figure 15.

6.4.2 Portability. Since EF may be deployed on different databases, we evaluate its portability. To differentiate from PostgreSQL, we select two other types of databases: the embedded database DuckDB and the in-memory database MonetDB. The results show that EF generates efficient queries on various databases and could adaptively produce different queries suitable for each database.

Specifically, on DuckDB, for P4, P5, and P8, EF outperforms both alternatives. Queries generated by EF are 1.56x faster on average, ranging from a speedup of 1.17x to 2.28x, as shown in Figure 16. Taking P5 as an example, the translation plan of EF generates three composite joins in total, reducing multiple probing steps, thereby producing a more efficient query. For other pipelines, the query generated by EF achieves performance comparable to the best queries produced by the other two strategies. For example, for P3, the query generated by the case-based strategy is faster, taking 31.3 seconds, while the query generated by EF takes 33.9 seconds.

On MonetDB, as shown in Figure 17, queries generated by EF for P3-P5 and P8 are on average 2.02x faster than the fastest queries generated by the other two strategies, ranging from 1.15x to 3.23x. Here, we note an interesting observation. EF tends to group more encoders together on MonetDB, i.e., translating more encoders into one composite join. For instance, for P4, the translation plan produced by EF on DuckDB partitions eleven encoders into seven groups, whereas on MonetDB, the produced plan divides the encoders into only three groups. We attribute this to the materialization execution model and in-memory architecture of MonetDB [19]. The former enables a probe reduction to avoid extensive memory access, while the latter eliminates the need to scan composite EM tables from disk during the build phase. Hence, translating more encoders into a composite join offers greater performance benefits.

6.4.3 Comparison with Other Solutions. On Postgres and DuckDB, we compare EF with UDF-based solutions and external execution using scikit-learn, without limiting the parallelism.

EF *vs.* Python UDF. We invoke the scikit-learn library within the Python UDF of Postgres and DuckDB to execute trained pipelines, which is a common practice [37, 51, 60]. However, the execution time of all pipelines exceeds half an hour on Postgres and fifteen minutes on DuckDB, which is not presented in Figure 18 and Figure 19. In addition, although IMBridge [60] improves the execution of such Python UDFs in DuckDB, the execution time still exceeds 10 minutes. In contrast,

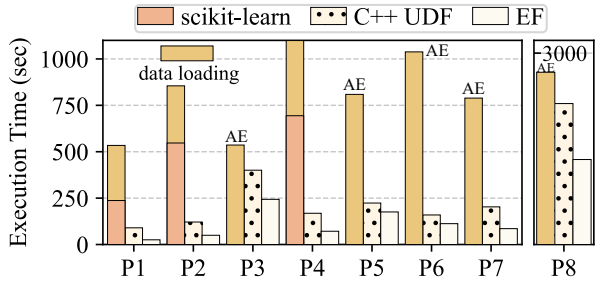


Fig. 18. Other Solutions on Postgres

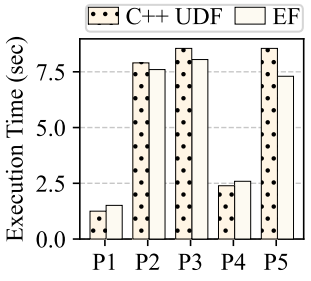


Fig. 19. UDF on DuckDB

EF translates ML computations into pure SQL, thereby eliminating the cross-engine overheads and completing execution in 1.6 seconds to 3 minutes on DuckDB.

EF *vs.* C++ UDF. We compare EF with a more efficient alternative approach. Specifically, on Postgres, we implement the computation logic of the encoders in the pipeline within a single C++ UDF, using `unordered_map` for the mappings. To avoid redundant loading, all `unordered_map` instances are defined as global variables. We then execute queries that invoke the created functions. Here, we enable expression compilation and mark all functions as `PARALLEL SAFE`. Overall, EF achieves an average speedup of 2.09x, ranging from 1.27x to 3.54x, as shown in Figure 18. We attribute the performance differences to two main factors. First, Postgres launches seven workers for queries generated by EF, i.e., with a parallelism of eight. In contrast, although the C++ UDF approach avoids the overhead of virtual function calls [31], Postgres launches two workers for query of P8 and zero workers for other queries that invoke UDF. We attribute this to the difficulty of the optimizer in accurately estimating the cost and benefits of UDF parallelism [15, 42]. We add encoders with large number of unique values to pipelines like P6, where the largest mapping table reaches 193 MB. However, the queries invoking the UDF remain non-parallel. Second, EF translates multiple encoders into composite joins, whereas databases typically treat UDFs as black boxes and fail to optimize the statements inside [43].

On DuckDB, we implement vectorized UDFs in a similar manner. Figure 19 presents the results of P1-P5, due to page limitations. Here, queries invoking UDFs achieve a parallelism degree of 48, identical to that of pure SQL queries, indicating that DuckDB fully enables parallelism for UDFs. Overall, the UDF approach benefits from aggressive low-level optimizations of the compiler, whereas EF generates composite joins and exploits the execution engine of DuckDB. Clearly, the two approaches are comparable.

In summary, EF provides good portability, enabling encodings to seamlessly benefit from database parallelism capability and optimizations; however, its performance upper bound is constrained by the underlying database engine. By contrast, C++ UDF approach allows customized implementations and eliminates virtual function call overhead, particularly in Postgres. Moreover, developers with expertise in the target database could further modify the engine to fully support UDF parallelism. However, UDFs differ across databases at the API and execution layers, resulting in higher development complexity and lower portability for C++ UDF approach.

EF *vs.* scikit-learn. We employ ConnectorX [56] to accelerate data loading from Postgres. As shown in Figure 18, EF significantly improves the execution efficiency of pipelines. For P1, P2, and P4, EF improves execution performance by 8.50x on average. Even when excluding data loading time, the generated queries are faster than scikit-learn. For other pipelines, scikit-learn throws `ArrayMemoryError` (AE) exceptions during inference.

7 Related Work

Translation of ML Operations. Translating ML operations into SQL is a promising research direction. A number of works, such as SmartLite [25], sql-einsum [2], translate tensor computations in deep learning models to relational operations. Similarly, certain works map linear algebra to relational algebra. Luo et al. [27] extend the DB engine to efficiently support operations such as matrix multiplication, while TRA [59] leverages the idea of relational algebra to optimize tensor computations in a novel way. However, neither focuses on the encoder translation problem, because encoder computations fall outside the scope of linear algebra [37].

In contrast, ML2SQL frameworks like Raven [37] and MASQ [34] focus on translating traditional ML operations into efficient SQL and design logical optimization. However, including the latest studies [26, 36], they only express encoders as `CASE` expressions. Although Blue Elephants [47] employs join to express OneHotEncoder, it overlooks the execution characteristics of database engines. To address these limitations, EncoderForge introduces two join-based translation approaches that leverage the powerful computation capability of join. Furthermore, unlike existing frameworks, EncoderForge does not apply a uniform translation approach to all encoders. Instead, it selects a translation approach for each encoder in an ML pipeline to maximize the performance of the generated queries, which may include a mix of `CASE` expression, separate and composite joins.

Unified Intermediate Representation (IR). A series of studies maps ML computations and relational algebra into an IR. Weld [35] and Lara [21] aim to capture global semantics via an IR and propose advanced co-optimization techniques. Yet they do not consider how to translate encoders in an inference ML pipeline into efficient SQL when the execution backend is a database. SubOperator [20] uses MLIR to achieve co-optimization of ML and relational algebra, but likewise does not explore how to translate encoders into efficient SQL. However, the queries produced by EncoderForge when translating an ML pipeline can serve as input to SubOperator, since it is essentially a compilation-based relational engine.

Selection of Candidate Plans. Selecting an efficient plan from all candidate plans is a widely studied problem. ML systems like SystemDS and ReMac adopt dynamic programming (DP) for selecting the optimal matrix multiplication chains or optimal redundancy elimination [3–5, 8]. Database optimizers employ efficient methods based on DP to find the optimal join orders. As the number of joins increases, databases like Postgres and DuckDB use heuristic methods to handle the exponential growth of the search space in DP. These works inspire our adaptive translation plan selection. Additionally, prior studies train ML models to predict the cost of candidate plans [17, 24, 44, 57], and we develop the cost model by drawing on techniques from them.

8 Conclusion

In this paper, we introduce EncoderForge, an ML2SQL framework that translates encoders in ML pipelines into efficient SQLs, thereby improving the performance of generated queries. We first present two translation approaches to generate joins for encoders. Then, we propose the dynamic programming and the priority selection strategies to identify a translation plan that generates an efficient query. Experiment results show that queries generated by EncoderForge are faster than those generated by existing approaches. In addition, EncoderForge is easily ported to different databases, enabling databases to efficiently execute ML pipelines natively.

Acknowledgments

We thank the anonymous SIGMOD reviewers for their constructive and insightful feedback and suggestions. This work was supported by the National Natural Science Foundation of China (No. 62272168), and the Natural Science Foundation of Shanghai (No. 23ZR1419900).

References

[1] Amazon. 2025. The total cost of ownership (TCO) of Amazon SageMaker. https://pages.awscloud.com/rs/112-TZM-766/images/Amazon_SageMaker_TCO_uf.pdf.

[2] Mark Blacher, Julien Klaus, Christoph Staudt, Sören Laue, Viktor Leis, and Joachim Giesen. 2023. Efficient and Portable Einstein Summation in SQL. *Proc. ACM Manag. Data (PACMMOD)* 1, 2 (2023).

[3] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, and Sebastian Benjamin Wrede. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *Proceedings of the 10th Conference on Innovative Data Systems Research (CIDR)*.

[4] Matthias Boehm, Matteo Interlandi, and Chris Jermaine. 2023. Optimizing Tensor Computations: From Applications to Compilation and Runtime Techniques. In *Companion of the 2023 International Conference on Management of Data (SIGMOD)*. 53–59.

[5] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, and Niketan Pansare. 2018. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *Proc. VLDB Endow. (PVLDB)* 11, 12 (2018), 1755–1768.

[6] Xuyi Cai, Ying Wang, and Lei Zhang. 2023. Optimus: An Operator Fusion Framework for Deep Neural Networks. *ACM Transactions on Embedded Computing Systems (TECS)* 22, 1 (2023), 1:1–1:26.

[7] Shaosheng Cao, Xinxing Yang, Cen Chen, Jun Zhou, Xiaolong Li, and Yuan Qi. 2019. TitAnt: Online Real-time Transaction Fraud Detection in Ant Financial. *Proc. VLDB Endow. (PVLDB)* 12, 12 (2019), 2082–2093.

[8] Zihao Chen, Baokun Han, Chen Xu, Weining Qian, and Aoying Zhou. 2022. Redundancy Elimination in Distributed Matrix Computation. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*. 573–586.

[9] ClickHouse. 2025. Denormalizing Data. https://clickhouse.com/docs/data-modeling/denormalization.

[10] DuckDB. 2025. plan_comparison_join. https://github.com/duckdb/duckdb/blob/main/src/execution/physical_plan/plan_comparison_join.cpp.

[11] Yannis Foufoulas and Alkis Simitsis. 2023. Efficient Execution of User-Defined Functions in SQL Queries. *Proc. VLDB Endow.* 16, 12 (2023), 3874–3877.

[12] Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. 2022. Why do tree-based models still outperform deep learning on typical tabular data?. In *Proceedings of the 2022 Annual Conference on Neural Information Processing Systems (NeurIPS)*.

[13] The PostgreSQL Global Development Group. 2025. pg_hint_plan. https://www.postgresql.org/about/news/pg_hint_plan-v160-released-2712/.

[14] Philipp M. Grulich, Aljoscha P. Lepping, Dwi P. A. Nugroho, Varun Pandey, Bonaventura Del Monte, Steffen Zeuch, and Volker Markl. 2024. Query Compilation Without Regrets. *Proc. ACM Manag. Data (PACMMOD)* 2, 3 (2024), 165.

[15] Surabhi Gupta and Karthik Ramachandra. 2021. Procedural Extensions of SQL: Understanding their usage in the wild. *Proc. VLDB Endow. (PVLDB)* 14, 8 (2021), 1378–1391.

[16] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. 2021. Putting Pandas in a Box. In *Proceedings of the 11th Conference on Innovative Data Systems Research (CIDR)*.

[17] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction. *Proc. VLDB Endow. (PVLDB)* 15, 11 (2022), 2361–2374.

[18] Zezhou Huang, Rathijit Sen, Jiaxiang Liu, and Eugene Wu. 2023. JoinBoost: Grow Trees Over Normalized Data Using Only SQL. *Proc. VLDB Endow. (PVLDB)* 16, 11 (2023), 3071–3084.

[19] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin* 35, 1 (2012), 40–45.

[20] Michael Jungmair and Jana Giceva. 2023. Declarative Sub-Operators for Universal Data Processing. *Proc. VLDB Endow. (PVLDB)* 16, 11 (2023), 3461–3474.

[21] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. An Intermediate Representation for Optimizing Machine Learning Pipelines. *Proc. VLDB Endow. (PVLDB)* 12, 11 (2019), 1553–1567.

[22] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2021. A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration. *Data Science and Engineering (DSE)* 6, 1 (2021), 86–101.

[23] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow. (PVLDB)* 9, 3 (2015), 204–215.

[24] Jiexing Li, Arnd Christian König, Vivek R. Narasayya, and Surajit Chaudhuri. 2012. Robust Estimation of Resource Consumption for SQL Queries using Statistical Techniques. *Proc. VLDB Endow. (PVLDB)* 5, 11 (2012), 1555–1566.

[25] Qiuru Lin, Sai Wu, Junbo Zhao, Jian Dai, Meng Shi, Gang Chen, and Feifei Li. 2023. SmartLite: A DBMS-Based Serving System for DNN Inference in Resource-Constrained Environments. *Proc. VLDB Endow. (PVLDB)* 17, 3 (2023), 278–291.

[26] Kuan Lu, Zhihui Yang, Sai Wu, Ruichen Xia, Dongxiang Zhang, and Gang Chen. 2025. Adda: Towards Efficient in-Database Feature Generation via LLM-based Agents. *Proc. ACM Manag. Data (PACMMOD)* 3, 3 (2025), 125:1–125:27.
[27] Shangyu Luo, Zekai J. Gao, Michael N. Gubanov, Luis Leopoldo Perez, and Christopher M. Jermaine. 2017. Scalable Linear Algebra on a Relational Database System. In *Proceedings of the 33rd IEEE International Conference on Data Engineering (ICDE)*. 523–534.
[28] Federico Matteucci, Vadim Arzamasov, and Klemens Böhm. 2023. A benchmark of categorical encoders for binary classification. In *Proceedings of the 2023 Annual Conference on Neural Information Processing Systems (NeurIPS)*.
[29] Guido Moerkotte and Thomas Neumann. 2008. Dynamic programming strikes back. In *Proceedings of the 2008 International Conference on Management of Data (SIGMOD)*. 539–552.
[30] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. 2020. A Tensor Compiler for Unified Machine Learning Prediction Serving. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 899–917.
[31] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.
[32] Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. 677–692.
[33] Kiyoshi Ono and Guy M. Lohman. 1990. Measuring the Complexity of Join Enumeration in Query Optimization. In *Proceedings of the 16th Conference on Very Large Data Bases (VLDB)*, Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek (Eds.). 314–325.
[34] Matteo Paganelli, Paolo Sottovia, Kwanghyun Park, Matteo Interlandi, and Francesco Guerra. 2023. Pushing ML Predictions Into DBMSs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 35, 10 (2023), 10295–10308.
[35] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, and Matei Zaharia. 2017. Weld: A Common Runtime for High Performance Data Analytics. (2017).
[36] Qingfeng Pan, Jiahe Zhi, Chenyang Zhang, Chen Xu, Zhao Zhang, Anita Shao, Guanglei Bao, Qiu Cui, Xiaowei Chen, and Aoying Zhou. 2025. Machine Learning Inference Pipeline Execution Using Pure SQL Based on Operator Fusion. In *Proceedings of the 41st IEEE International Conference on Data Engineering (ICDE)*. 3397–3410.
[37] Kwanghyun Park, Karla Saur, Dalitso Banda, Rathijit Sen, Matteo Interlandi, and Konstantinos Karanasos. 2022. End-to-end Optimization of Machine Learning Prediction Queries. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*. 587–601.
[38] Yuchen Peng, Zhongle Xie, Ke Chen, Gang Chen, and Lidan Shou. 2025. Towards Automatic and Efficient Prediction Query Processing in Analytical Database. In *Proceedings of the 41st IEEE International Conference on Data Engineering (ICDE)*. 2253–2266.
[39] Arnab Phani, Lukas Erlbacher, and Matthias Boehm. 2022. UPLIFT: Parallelization Strategies for Feature Transformations in Machine Learning Workloads. *Proc. VLDB Endow. (PVLDB)* 15, 11 (2022), 2929–2938.
[40] Ashwin Prasad, Sampath Rajendra, Kaushik Rajan, R. Govindarajan, and Uday Bondhugula. 2024. SilvanForge: A Schedule-Guided Retargetable Compiler for Decision Tree Inference. In *Proceedings of the 30th Symposium on Operating Systems Principles (SOSP)*, Emmett Witchel, Christopher J. Rossbach, Andrea C. Arpaci-Dusseau, and Kimberly Keeton (Eds.). 488–504.
[41] Bernhard Radke and Thomas Neumann. 2019. LinDP++: Generalizing Linearized DP to Crossproducts and Non-Inner Joins. In *Proceedings of the 2019 Conference on Database Systems for Business, Technology and Web (BTW)*, Vol. P-289. 57–76.
[42] Karthik Ramachandra and Kwanghyun Park. 2019. BlackMagic: Automatic Inlining of Scalar UDFs into SQL Queries with Froid. *Proc. VLDB Endow. (PVLDB)* 12, 12 (2019), 1810–1813.
[43] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César A. Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of Imperative Programs in a Relational Database. *Proc. VLDB Endow. (PVLDB)* 11, 4 (2017), 432–444.
[44] Maximilian Rieger and Thomas Neumann. 2025. T3: Accurate and Fast Performance Prediction for Relational Database Systems With Compiled Decision Trees. *Proc. ACM Manag. Data (PACMMOD)* 3, 3 (2025), 1–27.
[45] Ricardo Salazar-Díaz, Boris Glavic, and Tilmann Rabl. 2024. InferDB: In-Database Machine Learning Inference Using Indexes. *Proc. VLDB Endow.* 17, 8 (2024), 1830–1842.
[46] G. Lawrence Sanders and Seungkyoon Shin. 2001. Denormalization Effects on Performance of RDBMS. In *Proceedings of the 34th Hawaii International Conference on System Sciences (HICSS)*. IEEE Computer Society.
[47] Maximilian E. Schüle, Luca Scalerandi, Alfons Kemper, and Thomas Neumann. 2023. Blue Elephants Inspecting Pandas: Inspection and Execution of Machine Learning Pipelines in SQL. In *Proceedings of the 26th International Conference on Extending Database Technology (EDBT)*. 40–52.
[48] scikit learn. 2025. Category Encoders. https://contrib.scikit-learn.org/category_encoders/.
[49] scikit learn. 2025. Preprocessing data. https://scikit-learn.org/stable/modules/preprocessing.html.

[50] SQL Server. 2026. Performance of scalar UDFs. https://learn.microsoft.com/en-us/sql/relational-databases/user-defined-functions/scalar-udf-inlining?view=sql-server-ver17#performance-of-scalar-udfs.
[51] Tarique Siddiqui, Arnd Christian König, Jiashen Cao, Cong Yan, and Shuvendu K. Lahiri. 2025. QURE: AI-Assisted and Automatically Verified UDF Inlining. *Proc. ACM Manag. Data (PACMMOD)* 3, 1 (2025).
[52] Yutian Sun, Tim Meehan, Rebecca Schlussel, Wenlei Xie, Masha Basmanova, Orri Erling, Andrii Rosa, Shixuan Fan, Rongrong Zhong, Arun Thirupathi, Nikhil Collooru, Ke Wang, Sameer Agarwal, Arjun Gupta, Dionysios Logothetis, Kostas Xirogiannopoulos, Amit Dutta, Varun Gajjala, Rohit Jain, Ajay Palakuzhy, Prithvi Pandian, Sergey Pershin, Abhisek Saikia, Pranjal Shankhdhar, Neerad Somanchi, Swapnil Tailor, Jialiang Tan, Sreeni Viswanadha, Zac Wen, Biswapesh Chattopadhyay, Bin Fan, Deepak Majeti, and Aditi Pandit. 2023. Presto: A Decade of SQL Analytics at Meta. *Proc. ACM Manag. Data (PACMMOD)* 1, 2 (2023), 189:1–189:25.
[53] Jeffrey Tao, Natalie Maus, Haydn Thomas Jones, Yimeng Zeng, Jacob R. Gardner, and Ryan Marcus. 2025. Learned Offline Query Planning via Bayesian Optimization. *Proc. ACM Manag. Data (PACMMOD)* 3, 3 (2025), 179:1–179:29.
[54] tidypredict. 2025. Run Predictions Inside the Database. https://tidypredict.tidymodels.org/.
[55] Qichen Wang, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. 2025. Yannakakis+: Practical Acyclic Query Evaluation with Theoretical Guarantees. *Proc. ACM Manag. Data (PACMMOD)* 3, 3 (2025), 235:1–235:28.
[56] Xiaoying Wang, Weiyuan Wu, Jinze Wu, Yizhou Chen, Nick Zrymiak, Changbo Qu, Lampros Flokas, George Chow, Jiannan Wang, Tianzheng Wang, Eugene Wu, and Qingqing Zhou. 2022. ConnectorX: Accelerating Data Loading From Databases to Dataframes. *Proc. VLDB Endow. (PVLDB)* 15, 11 (2022), 2994–3003.
[57] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Stage: Query execution time prediction in amazon redshift. In *Companion of the 2024 International Conference on Management of Data (SIGMOD)*. 280–294.
[58] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Proceedings of the 7th Conference on Very Large Data Bases (VLDB)*. 82–94.
[59] Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois, and Chris Jermaine. 2021. Tensor Relational Algebra for Distributed Machine Learning System Design. *Proc. VLDB Endow. (PVLDB)* 14, 8 (2021), 1338–1350.
[60] Chenyang Zhang, Junxiong Peng, Chen Xu, Quanqing Xu, and Chuanhui Yang. 2025. Mitigating the Impedance Mismatch between Prediction Query Execution and Database Engine. *Proc. ACM Manag. Data (PACMMOD)* 3, 3 (2025), 189:1–189:28.
[61] Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. 2025. Debunking the Myth of Join Ordering: Toward Robust SQL Analytics. *Proc. ACM Manag. Data (PACMMOD)* 3, 3 (2025), 1–28.
[62] Wenbin Zhu, Runwen Qiu, and Ying Fu. 2024. Comparative Study on the Performance of Categorical Variable Encoders in Classification and Regression Tasks. *CoRR* abs/2401.09682 (2024).

Received October 2025; revised January 2026; accepted February 2026