A Comparative Study of in-Database Inference Approaches

Qiuru Lin [‡1], Sai Wu [‡2], Junbo Zhao [‡3], Jian Dai [#4] , Feifei Li [#5], Gang Chen [‡6]

[‡]*Zhejiang University, China*
[1,2,3,6]{qiurulin, wusai, j.zhao, cg}@zju.edu.cn

[#]*Alibaba Group, China*
[4,5]{yiding.dj, lifeifei}@alibaba-inc.com

***Abstract*—In Alibaba's IoT platform, we face the challenge of processing analytical queries involving both structured and unstructured data. Normally, collaborative queries need deep learning (DL) models and relational algebras to work intertwined to produce sophisticated analytical answers. To be able to support collaborative queries, a variety of approaches have been proposed. In this paper, we present the three most representative ones and study their advantages and limitations. The first one translates the collaborative query into a series of database and DL sub-queries and then maintains the dependence of the intermediate results of two sub-systems and computes the final results on the fly. The second one transforms a DL model to a database built-in User Defined Function(UDF) implemented in C++. The whole collaborative query is then processed by the database system independently. The third one is our novel solution proposed in the paper, DL2SQL, where neural operators underneath DL models are rewritten as SQL queries, and collaborative queries are processed using native SQL syntax. A cost model for our SQL-native neural operators is designed to leverage the database's optimizer to generate an efficient query plan. All three approaches are implemented on the ClickHouse. Finally, we use the real-world workloads on Alibaba's IoT platform as our benchmark and deploy various approaches on both an embedded device and a Cloud server to compare their performance. Results show that DL2SQL outperforms others in most scenarios and is more extensible.**

***Index Terms*—Query Processing, Query Optimization, In-database Inference**

I. Introduction

Deep learning(DL) has shown great successes in many domains, including computer vision, natural language processing, and speech recognition [1]–[6]. Most existing work usually treats date management and deep learning as two lines of research. Deep learning emphasizes the effectiveness of various models, while data management systems contribute to data storage and processing. However, deep learning models are usually packed with database systems to form a solution in real-world applications. Typically, an analytic task in the solution calls for an ETL process to obtain the transformed data or features, a well-tuned DL model for the predictions over the transformed data, and some sophisticated SQL scripts for the final result. Such a non-trivial task incurs significant efforts and high processing overheads.

Take a fabric printing setup on Alibaba's IoT platform as an example. As shown in Fig. 1, multiple sensors are installed on each printer collecting heterogeneous data such as printing video, logging records, meter count, temperature, humidity, etc., and the edge server corresponding to a printer hosts an

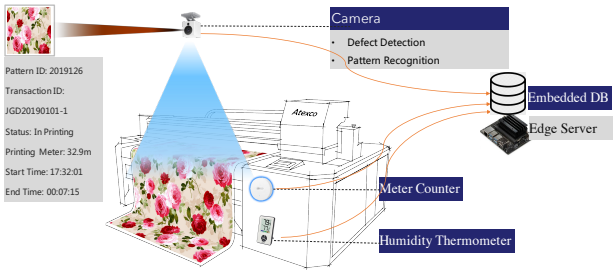


Fig. 1: An Illustration of the Fabric Printing Example

embedded database system collecting those real-time sensor data. For instance, a printing fault detection task needs the sensor readings and criteria to analyze if some faults are caught. In this case, the following query may be issued.

```
SELECT patternID, transID
FROM FABRIC F, Video V
WHERE F.humidity > 80 and F.temperature > 30
        and F.printdate > '2021-01-01'
        and F.printdate < '2021-1-31'
        and F.transID = V.transID
        and V.date > '2021-01-01'
        and V.date < '2021-1-31'
        and nUDF_detect(V.keyframe) = FALSE;
```

The above query seeks the transactions where various defects are detected on textiles. Video data are maintained in a separate table and can be joined with the main table via the transaction ID(transID). All other sensor data and transaction data are aggregated in the main table. $nUDF\_detect$ refers to a User Defined Function(UDF), where a pre-trained neural model is loaded to classify the keyframes of a video for the textile printing into "Defect"(TRUE) or "Not Found"(FALSE). To simplify the discussion, we call the UDF implementing the neural model inference as nUDF and the query combined native query and the nUDF as a collaborative query.

To be able to support collaborative query processing, there are three basic strategies: independent processing, loose integration, and tight integration [7]. Independent processing treats database and Machine Learning(ML) framework as two separate systems, employing the database to manage data and exporting data to the ML framework for model training and inference, such as Microsoft MLS[1], Amazon Sagemaker [8],

[1]https://microsoft.github.io/sql-ml-tutorials/

MindsDB[2]. Loose integration migrates the machine learning code from the ML framework to the database system by re-implementing the corresponding training and inference algorithm in the database using UDF Programming Model, e.g., MadLib [9], Vertica-ML [10], and DB4ML [11]. Tight integration maps the ML tasks into a batch of aggregate query statements and optimizes them using database techniques. On the other hand, the independent processing and loose integration strategies are widely adopted by existing work. The tight integration, to our knowledge, has attracted interest from many researchers, but there is no mature product in the industry. Previous work, such as AC/DC [12] and LMFAO [13], only supports tight integration for traditional machine learning algorithms(e.g., KNN [14] and SVM [15]).

Based on the requirement of our user scenario on Alibaba's IoT platform, we focus on the inference process on low-priced edge devices with limited computation capacity, where the neural model is trained on cloud servers in an offline manner and then deployed on various edge devices for online inference. The performance of online inference is the bottleneck of real-time analysis. Therefore, following the idea of tight integration, we present an in-database inference approach in this paper. Different from the previous work using a tight integration strategy, our approach aims to support deep learning models. Specifically, our in-database inference approach implements the inference pathway of necessary neural operators in SQL, together with the native relational SQL queries forming a system without cross-system I/O cost.

In summary, we make the following contributions:

1) We illustrate and compare three types of in-database inference strategies: independent processing, loose integration, and tight integration. We summarize their advantages and drawbacks.
2) We propose a new tight integration approach, DL2SQL, which transforms the neural model inference into pure SQL statements by implementing popular neural operators as SQL queries.
3) The SQL-implemented neural operators pose new challenges for the database's optimizer. Hence, we propose our customized cost model and apply hint rules for the database's optimizer to choose a proper query plan.
4) We collect data from real applications on our IoT platform and conduct extensive experiments to compare the three strategies. Results show that DL2SQL outperforms others in most cases thanks to its removal of the cross-system overhead.

The remainder of the paper is organized as follows. We formalize our collaborative queries and define four types of queries in Section II. We show the independent processing, loose integration, and our tight integration approaches in Section III. We modify the cost model of the database system in Section IV. We compare all three approaches in Section V. Section VI reviews some related work, and Section VII concludes the paper.

[2]https://mindsdb.com

II. Problem Definition

As discussed, a collaborative query consists of two parts and can be described by $(Q_{db}, Q_{learning})$, where Q_{db} can be processed by a database and $Q_{learning}$ requires the participation of a neural network for inference. Depending on different combinations of Q_{db} and $Q_{learning}$, we classify a collaborative query into four types presented in Table I.

As shown in the first row of Table I, the exemplary Type 1 query aims to retrieve the total printing meters for the "Floral Pattern", where the output of $Q_{learning}$ (i.e., nUDF_classify(V.keyframe) = 'Floral Pattern') is used as a filter for Q_{db} to rule out irrelevant results. In this manner, Q_{db} and $Q_{learning}$ are independent of each other and can be executed in parallel, even if the parallel processing may not be the optimal plan. The second row of Table I gives an exemplary Type 2 query that aims to estimate the defect rates of each pattern. In this example, the aggregate operator in Q_{db} needs the output of nUDF in $Q_{learning}$ to produce the final result. In this case, Q_{db} depends on $Q_{learning}$. Suppose various models are trained for different humidity and temperature combinations. The third row of Table I shows a Type 3 query, where $Q_{learning}$ needs the output of Q_{db} to determine which neural models should be used. In this case, $Q_{learning}$ depends on Q_{db}. Lastly, interdependence may exist between $Q_{learning}$ and Q_{db}. As shown in the fourth row of Table I, the Type 4 query tries to identify whether the printed pattern image is consistent with the pattern recorded in the transaction log. In this case, how to estimate the cost and efficiently process $Q_{learning}$ and Q_{db} becomes a challenging task. As illustrated in the following query, two models(i.e., $detect$ and $classify$) are included in $Q_{learning}$. When the $detect$ model in $Q_{learning}$ predicts that 95% of the original data records are irrelevant, and the $classify$ model in $Q_{learning}$ predicts that more than 60% of the original data records are relevant, it would be more efficient to execute the $detect$ model before the $classify$ model.

```
SELECT patternID, transID
FROM FABRIC F, Video V
WHERE F.transID = V.transID
        and nUDF_detect(V.keyframe) = TRUE
        and nUDF_classify(V.keyframe) = 'Floral Pattern';
```

III. Collaborative Query Processing

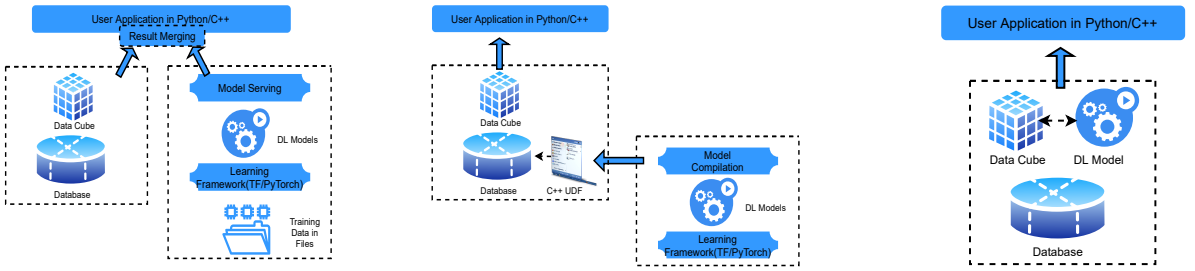
To support collaborative query processing, we study three possible approaches in this section. The first two approaches are adopted by existing work [7], while the tight integration one based on native SQL neural operators is our novel approach.

A. Independent Processing

Given a collaborative query, as illustrated in Fig. 2a, the application layer is responsible for parsing the query, generating a series of Q_{db} and $Q_{learning}$ sub-queries, distributing sub-queries onto corresponding the database system and the DL system, collecting results from both the database and the

TABLE I: Types of nUDF Queries

Query Type	Correlations	Example Query	Difficulty
Type 1	Q_{db}/$Q_{learning}$ independent	SELECT sum(meter) FROM FABRIC F, Video V WHERE F.printdate> '2021-01-01' and F.printdate<'2021-1-31' and F.transID=V.transID and V.date> '2021-01-01' and V.date<'2021-1-31' and nUDF_classify(V.keyframe)='Floral Pattern'	Easy
Type 2	Q_{db} depends on $Q_{learning}$	SELECT patternID, count(nUDF_defect(V.keyframe)=TRUE)/sum(meter) FROM FABRIC F, Video V WHERE F.printdate>'2021-01-01' and F.printdate<'2021-1-31' and F.transID=V.transID and V.date>'2021-01-01' and V.date<'2021-1-31' GROUP BY patternID	Medium
Type 3	$Q_{learning}$ depends on Q_{db}	SELECT patternID, transID FROM FABRIC F, Video V WHERE F.humidity>80 and F.temperature>30 and F.printdate>'2021-01-01' and F.printdate<'2021-1-31' and F.transID= V.transID and V.date>'2021-01-01' and V.date<'2021-1-31' and nUDF_detect(V.keyframe)=FALSE	Medium
Type 4	$Q_{learning}$ and Q_{db} depend on each other	SELECT patternID FROM FABRIC F, Video V WHERE F.printdate>'2021-01-01' and F.printdate<'2021-1-31' and F.transID=V.transID and V.date> '2021-01-01' and V.date<'2021-1-31' and F.patternID != nUDF_recog(V.keyframe)	Hard



(a) Independent Processing (b) Loose Integration (c) Tight Integration: DL2SQL

Fig. 2: Three Types of Collaborative Query Strategies

DL system, and computing the final result. The dependency between Q_{db} and $Q_{learning}$ is actually maintained by the user application. The Q_{db} part is handled by the database, where a data cube can be established to speed up the processing of aggregation queries. The $Q_{learning}$ part is processed by a DL system, where DL models are typically trained in an offline manner and used by the model serving component offered by the DL system in an online manner.

For complex Type 4 queries shown in Table I, the application layer needs to do a lot of coordination work, including forwarding prediction results from the DL system to the database system and transferring the querying results from the database system to the DL system. It incurs significant I/O and data transformation overheads between two systems because data transferring between two systems is a must. In addition, serialization and de-serialization of intermediate results between relational data schema and tensor data schema are often required. The independent processing strategy is easy to implement for a fixed set of collaborative queries. However, it is difficult for it to support arbitrary collaborative queries. Different collaborative queries require different hand-crafted query processing codes in the application layer since different collaborative queries usually correspond to different data transformations and DL models. It leads to tremendous development efforts.

B. Loose Integration

To reduce the cross-system I/O overheads, as shown in Fig. 2b, the loose integration approach implements the model inference process as a built-in UDF of the database system. In fact, using the UDF to extend database capability has been well studied. Many systems, such as MindsDB and MLDB[3], adopt the loose integration strategy, realizing an inference UDF by calling the corresponding APIs provided by the DL system. In this case, UDF works as a bridge between the database system and the DL system.

In this paper, we further replace the inference API calls with the compiled C++ codes to improve the efficiency of such kinds of UDFs. As shown in Fig. 2b, the model compilation component is responsible for compiling a DL model to binary files that can be directly used by a database kernel. The compilation process contains three steps. Take the PyTorch as an example. First, the model trained in PyTorch is converted to Torch script via the tracing tool(provided in torch.jit.trace). Second, a ScriptModule object is generated from the script and serialized as a binary file. Third, the binary file is loaded into the database kernel and directly used by the database system. The database kernel has to be recompiled to enable the corresponding UDF (i.e., compiled binary files). After the recompilation, the database system does not need to interact with the DL system anymore.

The loose integration strategy provides a unified database interface for collaborative query processing and avoids heavy data transformation and I/O overheads. Nevertheless, a new model requires a recompilation of the database kernel and the generated binary file. It cannot optimize the execution process of the corresponding query plan since the binary file is treated as a black box, and its execution cost cannot be effectively estimated.

[3]https://github.com/mldbai/mldb

TABLE II: Supported Neural Operators and Structures

Neural Blocks	Variants	SQL Support
Pooling	Average Pooling	Supported
	Max Pooling	Supported
Activation	ReLU	Supported
	Sigmoid	Supported
Normalization	Batch Normalization	Supported
	Instance Normalization	Supported
Full Connection	N.A.	Supported
Convolution	N.A.	Supported
Deconvolution	N.A.	Supported
Residual Block	N.A.	Supported
Indentity Block	N.A.	Supported
Dense Block	N.A.	Supported
Attention Block	Basic Attention	Supported
	Self Attention	Unsupported
RNN	LSTM	Unsupported
	GRU	Unsupported
Graph Convolution	N.A.	Supported by Graph DB

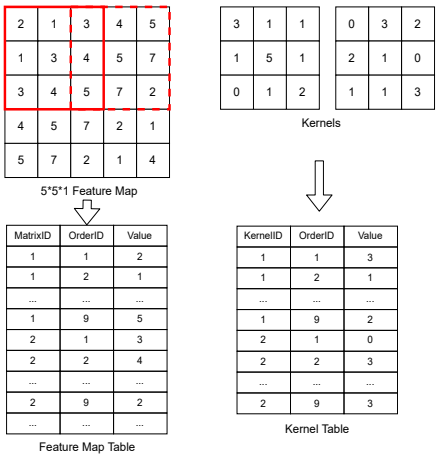


Fig. 3: Table Schema for Convolution

C. Tight Integration: DL2SQL

To avoid the coordination of the application layer, prevent the communication between the database and the DL system, and fully leverage the internal optimization techniques offered by a DBMS, we propose a new solution, DL2SQL, in the process of researching the tight integration strategy. DL2SQL turns a deep learning model into relational tables, where each record represents a parameter in the model and converts the deep learning operators into the operations over the relational tables. Fig. 2c illustrates the idea of DL2SQL, where neural model resides in a database system, the inference is also conducted inside a database system, and no DL system is involved during the processing of collaborative query processing.

Different from Madlib [9] focusing on transitional machine learning models, DL2SQL supports deep learning models including Convolution Neural Network(CNN), such as LeNet [1], VGG [4], ResNet [6], etc. Table II shows the supported neural operators and blocks in our current implementation.

In the following, we discuss the implementation details of SQL-based neural operators and our optimization techniques. First, we present how to implement CNN in a database system in Section III-C1. Then, we introduce how to implement other neural operators in the database in Section III-C2.

1) Implementing CNN as SQLs: CNN is viewed as a basic yet representative neural network layer that is widely used for image classification. In the convolutional layer, a kernel matrix scans over the input data to produce a feature map for the next layer. Normally, values on the feature maps are computed by taking the sum of the result of an element-wise multiplication of the kernel and the input matrix. The dimensions of the kernel can be adjusted to produce different feature maps with different dimensions. In the following, we first present how the kernel matrix and feature map are stored in a database. Then we demonstrate how the actual convolution is conducted, leveraging a native SQL query.

Storage. To model such a convolutional layer, we store the feature map in FeatureMap table and the kernel matrix in Kernel table, respectively, as shown in Fig. 3. Suppose we have two 3×3 kernels with striding 2 and no padding. We generate 4 matrices for the convolutional layer and maintain them together in a feature map table formatted as {MatrixID, OrderID, Value} for an input 5×5×1 feature map. The OrderID is employed to serialize the matrix values for each kernel computation. Similarly, the kernel matrices are also vectorized and stored in a kernel table [4]. As shown in Fig. 3, the first row in the feature map table is $\{1, 1, 2\}$ corresponding to the first element in the 5×5×1 matrix, and the first row in the kernel table is $\{1, 1, 3\}$ corresponding to the first element whose value is 3 in the kernel matrix. Note that some elements in the feature map may be stored redundantly, owing to the mechanism of CNN. For instance, $\{2, 1, 3\}$ and $\{1, 3, 3\}$ correspond to the same element in the feature map.

Apart from the feature map and kernel matrix, the other hyper-parameters (e.g., kernel size, stride, and padding size) are stored in a meta-data table. In addition, the input and the generated intermediate feature maps are stored in the database in accordance with the feature map table schema.

Algorithm 1 illustrates how to turn an original feature map F into a feature map table. As shown in lines 7-13, for each channel in F, we calculate the currently involved elements in F, assign them with the self-increasing OrderID, and put the triples into the feature map table. Then, as shown in lines 14-18, we update the coordinate values x and y and continue to process the rest of the original feature map.

Computation. After storing the convolutional layer in the database, we can perform the convolutional operation, relying on the feature map table and the kernel table.

The computation is straightforward. The convolution operation is essentially realized by an inner-join between the feature map table and the kernel table. As illustrated in **Q1** below, the feature map table and kernel table are joined under the condition of equal OrderID. Then the result is computed via the sum of the result of an element-wise multiplication.

[4]If the feature maps contain multiple channels, we maintain a feature table for each channel.

Algorithm 1: Generation of Feature Map Table

Input: Input F, Kernel K
Output: SQLs for Creating the Feature Map Table

```
1  FeatureMap = ∅, k = K.height, s = K.striding,
     Order_Header = 0
2  for i = 1 to F.channel do
3      MatrixID = 1
4      for y = 1 to F.height do
5          for x = 1 to F.width do
6              OrderID ← Order_Header
7              for Coordinate_y = y to y + k do
8                  for Coordinate_x = x to x + k do
9                      Values ← F.Value(Coordinate_x,
                         Coordinate_y)
10                     Insert {MatrixID, OrderID,
                         Values} into FeatureMap
11                     OrderID ++
12                 end
13             end
14             x ← x + s, MatrixID ++
15         end
16         y ← y + s
17     end
18     Order_Header ← Order_Header + k^2
19 end
20 return FeatureMap
```

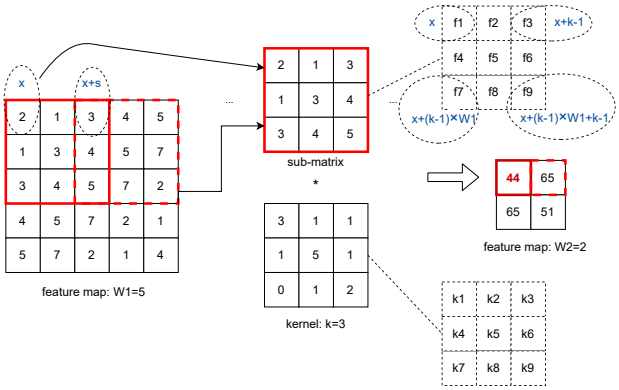


Fig. 4: Array Indices for Convolution

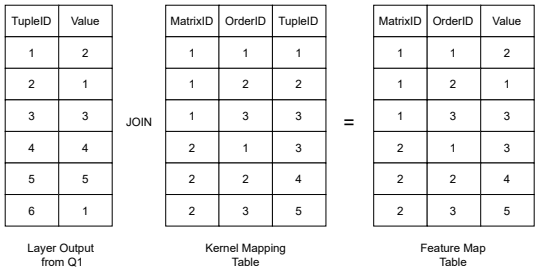


Fig. 5: Mapping of Feature Map Table

Q1:
```
CREATE TEMP TABLE Layer_Output(
SELECT MatrixID as TupleID,
       SUM(A.Value * B.Value) as Value
FROM FeatureMap A INNER JOIN Kernel B
ON A.OrderID = B.OrderID
GROUP BY KernelID, MatrixID);
```

Figure 4 demonstrates how a feature map is produced by **Q1**. As shown, the inner-join is performed for each sub-matrix and the kernel. For example, we can apply **Q1** for the sub-matrix $\begin{Bmatrix} 2 & 1 & 3 \\ 1 & 3 & 4 \\ 3 & 4 & 5 \end{Bmatrix}$ and the kernel $\begin{Bmatrix} 3 & 1 & 1 \\ 1 & 5 & 1 \\ 0 & 1 & 2 \end{Bmatrix}$. After applying the sum of element-wise multiplication, we obtain 44.

Since a neural network usually consists of multiple layers, we need to transform the output of convolution into the input feature map table format for subsequent use. Clearly, the output of **Q1** is in a different schema from the input feature map table illustrated in Fig. 3. The output of **Q1** is a tuple list, using the previous MatrixID as the TupleID for each record. To convert the tuple list to a matrix (i.e., feature map), we introduce a **kernel mapping table** that records the relationship among MatrixID, OrderID and TupleID, and is able to map TupleID to MatrixID via a join operation, as shown in Fig. 5. In fact, when the kernel, the input feature map, padding, and stride are all fixed, the relationships among MatrixID, OrderID, and TupleID are fixed as well. As shown in Fig. 4, let k denotes the width (the height as well) of the kernel, s denotes the stride, p denotes the padding. If $p = 0$, in the feature map, we have ⟨TupleID=1, Value=2⟩, ⟨TupleID=2, Value=1⟩, ⟨TupleID=3, Value=3⟩. For the first sub-matrix, we have ⟨MatrixID=1, OrderID=1, Value=2⟩, ⟨MatrixID=1, OrderID=2, Value=1⟩, ⟨MatrixID=1, OrderID=3, Value=3⟩, etc. Suppose $s = 2$. For the subsequent sub-matrix, we have ⟨MatrixID=2, OrderID=1, Value=3⟩, ⟨MatrixID=2, OrderID=2, Value=4⟩, ⟨MatrixID=2, OrderID=3, Value=5⟩, etc. As such, we can obtain the **kernel mapping table** T, where ⟨MatrixID=1, OrderID=1, TupleID=1⟩, ⟨MatrixID=1, OrderID=2, TupleID=2⟩, ⟨MatrixID=1, OrderID=3, TupleID=3⟩, ⟨MatrixID=2, OrderID=1, TupleID=3⟩, etc.

Abstractly, we are able to re-index the tuple list T_{i+1} starting with index x and the width of kernel k, and T_{i+1} is generated from the ith convolution layer as follows. $T_{i+1} = \{x,...,x + W_i^2 - 1\}$, where W_i is the width of the input feature map of the ith convolution layer. The sub-sequence $\{x, ..., x + k - 1, x + W_i, ..., x + W_i + k - 1, ..., ..., x + (k-1) \times W_i ..., x + (k-1) \times W_i + k - 1\}$ belongs to the first sub-matrix and is assigned to MatrixID=1. Likewise, we are able to construct the second sub-matrix, the third sub-matrix and etc.

Algorithm 2 shows the specific steps of generating the **kernel mapping table**. As shown from line 6 to line 10, the MatrixID, OrderID, and TupleID values of each record in the kernel mapping table are generated from a nested loop that produces k^2 elements. Since the kernel mapping table only

Algorithm 2: Generation of Mapping Table

Input: FeatureMap F, Kernel K
Output: SQLs for Creating the Kernel Mapping Table

MappingTable = $\emptyset$, $MatrixID$ = 0, x_Header = 0, k = $K.height$, s = $K.striding$
W_1 = $F.width$, W_2 = $\frac{W_1-k}{s}$+1
while $MatrixID < W_2^2$ **do**
  count = 0, $x \leftarrow x\_Header$
  while count $< W_2$ **do**
    for i = 0 to k **do**
      for j = 0 to k **do**
        $OrderID \leftarrow j + i * k$
        $TupleID \leftarrow x + j + W_1 * i$
        Insert $\{MatrixID, OrderID, TupleID\}$ into MappingTable
      end
    end
    $MatrixID$++, count++
    $x \leftarrow x$ + s
  end
  $x\_Header \leftarrow x\_Header$ + W_1 * s
end
return MappingTable

depends on k, W_i, and s [5]. Therefore, we generate the involved mapping tables in an offline way. After a mapping table has been generated, the below query is submitted to create a new feature map table:

Q2:
```
CREATE View FeatureMap(
SELECT MatrixID, OrderID, Value
FROM Layer_Output A, Kernel_Mapping B
WHERE A.TupleID = B.TupleID);
```

After **Q2** is performed, we can apply **Q1** again to conduct the CNN computation for the next layer.

2) Implementing Other Neural Operators: In a typical CNN-based model, CNN is often used together with Batch Normalization(BN), Rectified Linear Unit(ReLU), and Pooling operators.

BN is an approach that normalizes the layers' inputs by re-centering and re-scaling, making neural networks faster and more stable. BN can be computed as follows.

$$\text{BN} = \frac{input - \mu}{\sqrt{\sigma^2 + \varepsilon}} \tag{1}$$

where $input$ is the input data, μ is the mean of $input$, σ^2 is the variance of $input$, and ε is a constant to prevent the denominator from being zero. ReLU is a common activation function used in neural networks to overcome the gradient disappearance, which can be computed as follows.

$$\text{ReLU} = max(0, input) \tag{2}$$

[5]Here, $p = 0$.

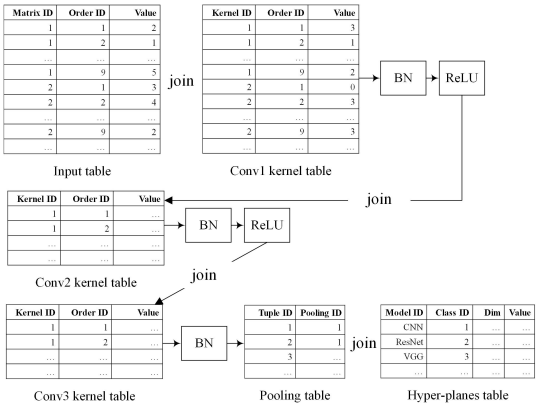


Fig. 6: Summary of a Complete CNN

Obviously, the BN and ReLU layer can be directly implemented as the SQL math algebra, so we omit the implementation details.

The pooling operator needs to split the input feature map into multiple sub-matrices for computation like the convolutional layer. The only difference with the convolutional layer is that the pooling layer performs some aggregation operations, such as the calculation of the maximum value in each sub-matrix instead of convolution. Therefore, we propose a process similar to **Q1** to realize the pooling operator. Assuming we adopt the max-pooling aggregation, **Q3** shows how the pooling is performed in a database system.

Q3:
```
CREATE TEMP TABLE Pooling_Output(
SELECT MatrixID as TupleID,
       MAX(A.Value) as Value
FROM FeatureMap A
GROUP BY MatrixID);
```

Some operators, such as full connection and basic attention, can be derived from the implementations of CNN. For example, the full connection can be considered a specific CNN operator with kernel size 1 and no striding. The basic attention operator is a variant of full connection. So we discard the details of those operators.

After implementing the basic neural operators, we are able to put them together and build more sophisticated neural networks. Fig. 6 demonstrates how a 3-layer CNN model with BN and ReLU is established upon the basic neural operators implemented in a database system. It is normally used as the basic building block for the neural models of other structures.

Next, we focus on how a residual block in ResNet is implemented. A typical residual block consists of three convolution blocks and a shortcut block. We simulate the shortcut block using the following SQL:

Q4:
```
CREATE VIEW feature_cbshortcut_conv(
SELECT MatrixID, OrderID, Value
FROM ( SELECT MatrixID as TupleID,
       SUM(A.Value * B.Value) as Value
```

```
        FROM FeatureMap A
        INNER JOIN Kernel B
        ON A.OrderID = B.OrderID
        GROUP BY KernelID, MatrixID)
        as Layer_Output, Kernel_Mapping
WHERE Layer_Output.TupleID
        = Kernel_Mapping.TupleID);

CREATE TEMP TABLE feature_cbshortcut_conv_bn(
SELECT MatrixID, OrderID, ((Value - (SELECT
        AVG(Value) FROM feature_cbshortcut_conv)) /
        ((SELECT stddevSamp(Value) FROM
        feature_cbshortcut_conv) + 0.00005)) as Value
FROM feature_cbshortcut_conv);
```

Q4 first conducts a CNN operator for the input and then applies a batch normalization.

We neglect the details of the three convolution blocks, which are almost identical to the shortcut block. Let feature_cb3_conv_bn denote the result produced by the last convolution block. Below we show how the residual link is formed, along with a ReLU activation:

Q5:
```
CREATE TEMP TABLE cb_output(
SELECT A.MatrixID, A.OrderID,
        A.Value + B.Value as Value
FROM feature_cbshortcut_conv_bn A,
        feature_cb3_conv_bn B
WHERE A.MatrixID = B.MatrixID);

UPDATE cb_output SET Value = 0 where Value < 0;
```

D. Comparison of Three Approaches

In this subsection, we present our comparison of the three strategies, which is summarized in Table III.

The **independent processing** strategy is easy to implement because it takes the database and the DL system as two black boxes, leaving the application responsible for parsing the collaborative queries and dealing with the internal dependency. Adopting this strategy, we can easily support complex neural models and distributed processing. Meanwhile, we can seamlessly introduce new hardware, such as GPUs or APUs, into the system for speedup. However, this strategy lacks reusability and requires rewriting the application layer for processing a new type of collaborative query.

The **loose integration** requires the developer to compile their trained models into serializable files and then write customized UDF to load models from those files. The built-in UDF can support parallel processing or GPU acceleration if the underlying database system provides such APIs. Nevertheless, we need to rewrite and recompile our UDF for each new model.

The **DL2SQL** strategy re-implements various neural operators in a database system, using native SQL clauses. The implemented neural operators can be easily assembled to realize various neural networks. Moreover, since we only rely on translated SQL neural operators to process a collaborative query, the database system is able to optimize the query processing in a natural manner when the execution cost can be effectively estimated.

All approaches need to maintain feature maps and intermediate results and hence incur additional storage overhead. We show the storage cost of all approaches in Table IV. We use ResNet5 to ResNet40 as our example models, and more details can be found in the experiment section. DL2SQL represents its neural model in relational tables, incurring a slightly higher cost than the other two approaches, which maintain models in file systems using compression. The additional cost, however, is acceptable for modern hardware, even edge devices.

IV. Collaborative Query Optimization

As mentioned above, the **DL2SQL** strategy can leverage the DBMS optimizer to generate its query plan. However, we discover that the DBMS optimizer cannot precisely estimate the size of intermediate results generated by the neural operators. It normally over-estimates the number of join results which will be exaggerated exponentially after several iterations of neural operator computations. On the other hand, we can accurately model the cost using the kernel sizes and input features based on the implementations of neural operators. In this section, we present our customized optimization approach to generate hints for the database optimizer.

A. Cost Model of SQL Implementation

To properly estimate the processing overhead of neural operators, we need to estimate the cardinality of each feature map table. Let F_{in} and F_{out} represent the input and output tensors of the convolutional layer, respectively, where $F_{in} \in \mathbb{R}^{H_{in} \times W_{in} \times N_{in}}, F_{out} \in \mathbb{R}^{H_{out} \times W_{out} \times N_{out}}$. H_{in} and W_{in} are the height and width of F_{in}, respectively, and N_{in} is the channel size of F_{in}. H_{out}, W_{out} and N_{out} are defined similarly for the output feature map.

Let k_h and k_w denote the height and width of the CNN kernels [6]. Let s and p denote the striding and padding value (we assume that we adopt the same striding and padding value for height and width). The relationship between the input tensors and output tensors can be computed as:

$$H_{out} = \frac{H_{in} + 2p - k_h}{s} + 1, W_{out} = \frac{W_{in} + 2p - k_w}{s} + 1 \quad (3)$$

In our SQL implementation, the sizes of kernel tables in the current layer and next layer are $k_{in} = k_h * k_w * N_{in}$ and $k_{out} = k_h * k_w * N_{out}$. The cardinality of the input feature can be estimated as $T_{in} = H_{out} * W_{out} * k_{in}$.

For a fixed kernel, the join selectivity between the feature map table and the kernel table is estimated as:

$$S_J = \frac{1}{k_{in}} \quad (4)$$

Then, the cardinality of the output feature map table is:

$$T_{out} = T_{in} * S_J * k_{out} \quad (5)$$

[6] In most cases, k_h equals to k_w.

TABLE III: Comparison of Three Approaches

Approaches	Implementation Complexity	Flexibility and Reusability	Opportunities for Optimizations	Scalability	I/O Cost	Support for GPUs
Independent Processing	Easy	Need to rewrite the codes for a new query	Consider databases and DL systems as black boxes and unable to optimize	High	High	Easy
Loose Integration	Medium	Need to rewrite and recompile the UDFs for a new query	UDF cannot be optimized by the database's optimizer	Medium	Medium	Depends on the database
Tight Integration(DL2SQL)	Hard	Translate the query into SQL neural operators	Create new cost model and apply the database's optimizer	Medium	Low	Depends on the database

TABLE IV: Storage Overheads with Different Model Depths

Model Depth	DL2SQL(KB)	DB-UDF(KB)	DB-PyTorch(KB)
5	4096	3838	3253
10	21504	17325	14803
15	38910	32003	26354
20	56316	45586	37905
25	73722	60543	49455
30	91128	73899	61006
35	108534	88577	72556
40	125354	102942	84107

To speed up the join processing, we build indices on columns $MatrixID$, $OrderID$, and $KernelID$. The processing of join is performed by scanning the feature map table and probing the kernel tables. Kernel tables are probed multiple times during the processing. Since each probing produces a value in the output table, we can use T_{out} to denote the times that the kernel tables are probed. Accordingly, the cost of the join operation is approximated as:

$$C_{join} = T_{in} + T_{out} * k_{in} \tag{6}$$

The next step of a CNN operator is the mapping process shown in Fig. 5. The cost is approximately identical to scanning the output table since the kernel mapping table is typically small, which is fully maintained in the L2 cache. As such, the overall CNN cost is approximately:

$$C_{out} = T_{in} + T_{out} * k_{in} + T_{out} \tag{7}$$

For the next CNN operator, we can use N_{out} to replace N_{in} and continue the estimation. The cardinality of the input for the next CNN operator is approximately:

$$T_{in}{}' = k_{out} * \left[\frac{(\frac{T_{out}}{k_{out}})^{\frac{1}{2}} + 2p - k_h}{s} + 1 \right]^2 \tag{8}$$

Other neural operators, such as BN, ReLU, Pooling, only need to scan the feature map table once. Thus, the cost can be calculated as a linear function to the feature map. The residual block, on the other hand, consists of several convolution blocks and is estimated based on the cost of convolution.

B. Hints for Collaborative Query

As shown in Table I, the two parts, Q_{db} and $Q_{learning}$, have a dependence on each other in some collaborative queries. So we define some rules as the hints for the database's optimizer to choose a proper query plan.

To achieve this, we need to evaluate the selectivity of nUDF in a collaborative query. We first learn the probability

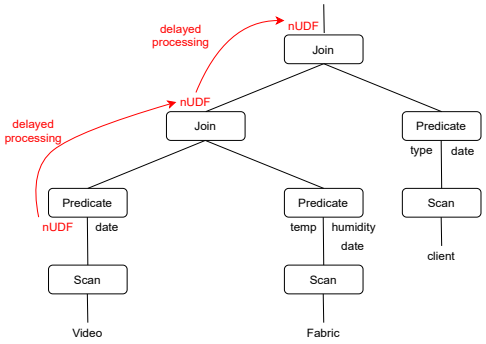


Fig. 7: Delayed Processing of nUDF

distribution Pr for each class c_i in an arbitrary nUDF. Given an input and all pre-defined classes $\{c_0, c_1, ..., c_n\}$, we have:

$$\sum_{i=0}^{n} Pr(c_i) = 1, Pr(c_i) \leq 1 \tag{9}$$

During the offline training process of the neural model for nUDF, we build a histogram $H(c_i)$ for each target class c_i. $H(c_i)$ counts the number of training samples that are predicted to be c_i. So the function Pr can be empirically estimated as:

$$Pr(c_i) = \frac{H(c_i)}{\sum_{i=0}^{n} H(c_i)} \tag{10}$$

Finally, for the nUDF testing the input sample against class c_i, we use $Pr(c_i)$ as its selectivity.

Given a collaborative query, we have the following rules. First, if the nUDF appears as a prediction, we have two strategies: 1) evaluating the nUDF during the table scan; 2) delaying the evaluation as much as possible. The first strategy incurs a full cost for the nUDF, but it avoids unnecessary tuples participating in the following processing. On the other hand, the second strategy reduces the cost of processing nUDF, since many tuples are pruned in previous operations. The optimizer generates the final query plan by comparing different cost estimations. Fig. 7 shows the idea of the second strategy. We rely on our customized cost model of the database system to compare the two strategies and generate a hint to select a better one.

Second, if the nUDF appears in the *select* clause, we will evaluate the nUDF as the last operator. In this way, the database optimizer can generate a plan by ignoring the nUDF in the beginning.

Third, we adopt the symmetric hash join algorithm if the nUDF appears as the join condition. Suppose we have a join

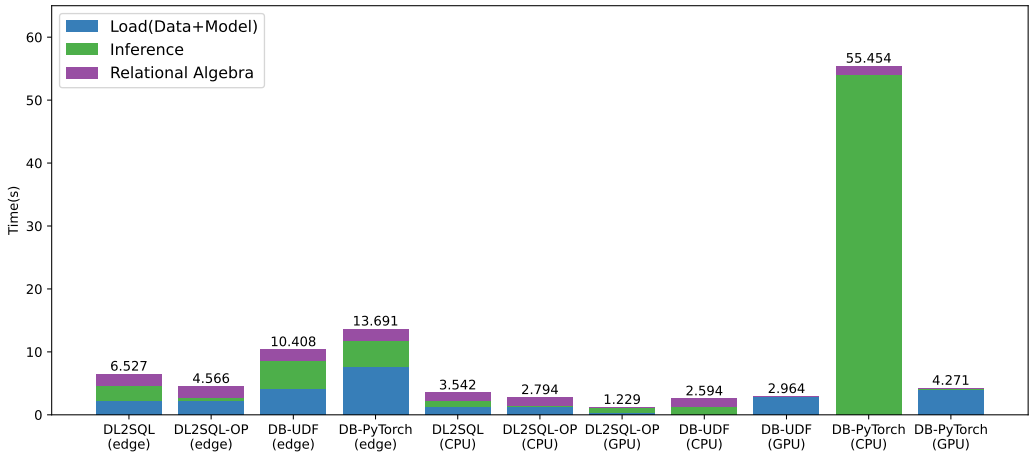


Fig. 8: Average Performance of Multiple Queries on Different Devices with Selectivity=0.01%

defined as $T_0.nUDF(x) = T_1.y$. Namely, the two tables are joined on the nUDF result of column x and column y. We maintain hash tables for $nUDF(x)$ (i.e., the prediction results of $nUDF$) and values of y, respectively. For each incoming value of T_0/T_1, we probe all values at the corresponding hash bucket of T_1/T_0 to perform the join. Ideally, the whole hash tables are maintained in memory. We apply the LRU strategy to select the victim when the buffer is full. Because the $nUDF$ is performed in a batch manner (a batch of feature maps are fed to the model together), we apply a bucket-based LRU strategy. Once we detect a new value from $T_0.nUDF(x)$, we will load the corresponding bucket of T_1 completely into the memory, avoiding the consecutive cache misses.

V. Evaluation

We implement all three strategies on an in-memory version ClickHouse and deploy the modified database on an embedded edge device powered by the ARM V8 CPU with 32GB memory. For the DL framework, we adopt PyTorch and its C++ Distributions LibTorch for serving models. For comparison, we also deploy them on our in-house server (with Quadro P6000 GPU and Xeon CPU).

We evaluate the performance of the three implementations using the dataset collected from Alibaba's IoT platform for the textile printing scenario. Our testing database consists of five tables: video, fabric, client, order, and device. The video table maintains the surveillance data generated by the IoT cameras for the textile printers. Videos are split into small clips using an equal-size time window. The fabric table records the basic information of the fabric pattern, e.g., the pattern image and its ID. The client table and order table denote the customers and their printing orders. The device table maintains the sensor data generated for each printer, such as temperature and humidity. There are 100 million tuples in total (the sizes of tables follow a ratio of "100:10:1:10:1"). To reduce the storage overhead, we resize the resolution of all videos to $224\times224\times3$ and have more than 100GB of videos.

We train a model repository consisting of 20 neural networks for various tasks, such as textile defect detection, clothes classification, textile type classification, and textile pattern recognition. We adopt the ResNet34 [6] as the backbone model for each task and tune the parameters for different tasks. Furthermore, to reduce the computation overhead, we apply the distillation technique [16]–[18] to learn a student CNN composed of three Conv+BN+ReLu layers (the prediction accuracy is 87% compared to the 93% of the ResNet34). In the experimental results, we report and analyze the performance of our student model first and then examine the results of ResNet with different depths.

We create a query template for each type displayed in Table I that picks a random DL task corresponding to a model in the model repository. All neural models are trained offline, and the neural model corresponding to a collaborative query is integrated into the system on the fly. We generate 100 queries for each type with a preset selectivity on the SQL predicates and mix them as our query benchmark to report the average processing cost per query.

A. Overall Result

Fig. 8 shows the overall performance of different strategies using the student models. By default, the accumulative selectivity of Q_{db} predicates is 0.01%. We perform our experiments with the following different configurations:

- DL2SQL and DL2SQL-OP denote the DL2SQL approach without and with query optimization, respectively.
- DB-UDF implements the UDF approach.
- DB-PyTorch represents the independent approach where the database and PyTorch work like black boxes.

We evaluate the approaches on two hardware settings. The first one is our edge device equipped with an ARM CPU and no GPU (the first four bars in Fig. 8). The second one is a typical GPU server(Quadro P6000 GPU and Xeon CPU) from the Alibaba Cloud. We test both CPU and GPU modes for the four approaches on the Cloud server. We break down the cost into three parts: loading cost, inference cost, and relational algebra cost. The loading cost includes the cost of loading neural models and data into the system. It also contains the

TABLE V: Performance Comparison with Different Selectivity on Edge Server

	DL2SQL-OP			DB-UDF			DB-PyTorch		
Selectivity(%)	***Inference(s)***	***Loading(s)***	***All(s)***	***Inference(s)***	***Loading(s)***	***All(s)***	***Inference(s)***	***Loading(s)***	***All(s)***
0.01	**0.441**	**2.256**	**2.697**	4.558	4.617	9.175	4.199	7.589	11.788
0.1	**0.263**	**1.129**	**2.783**	4.63	4.631	9.261	4.2	7.589	11.789
0.2	**0.618**	**2.175**	**2.793**	4.54	4.531	9.071	4.199	7.591	11.79
0.4	**0.857**	**2.259**	**3.116**	4.516	4.41	8.926	4.21	7.592	11.802
0.6	**1.308**	**2.261**	**3.569**	4.341	4.277	8.618	4.23	7.594	11.824
0.8	**2.254**	**2.231**	**4.485**	4.437	4.23	8.667	4.24	7.597	11.837
1	4.651	**2.174**	**6.825**	4.568	4.292	8.86	**4.24**	7.599	11.839

TABLE VI: Performance Comparison with Different Model Depths on Edge Server

Model Configuration		**DL2SQL-OP**			**DB-UDF**			**DB-PyTorch**		
Depth	***Parameters***	***Inference(s)***	***Loading(s)***	***All(s)***	***Inference(s)***	***Loading(s)***	***All(s)***	***Inference(s)***	***Loading(s)***	***All(s)***
5	828418	**0.138**	**1.198**	**1.336**	2.282	2.243	4.525	2.478	1.957	4.435
10	3781890	**0.165**	2.341	**2.506**	2.291	**2.274**	4.565	1.982	2.524	4.506
15	6734850	**0.199**	3.237	**3.436**	2.29	**2.263**	4.553	1.987	2.558	4.545
20	9687810	**0.227**	4.555	4.782	2.309	**2.282**	4.591	1.975	2.572	**4.547**
25	12640770	**0.258**	4.508	4.766	2.321	**2.308**	4.629	2.001	2.596	**4.597**
30	15593730	**0.289**	4.306	4.595	2.321	**2.327**	4.648	1.959	2.542	**4.501**
35	18546690	**0.319**	4.593	4.912	2.332	**2.341**	4.673	1.961	2.543	**4.504**
40	20909570	**0.348**	6.194	6.542	2.343	**2.358**	4.701	1.969	2.546	**4.515**

I/O cost between the database system and the DL system. The inference cost is the prediction time incurred by the DL model. The relational algebra cost denotes the overhead of processing relational operators in the database system.

Fig. 8 shows the overall performance of all the approaches with different configurations. On the edge device, the DL2SQL-OP performs the best. Compared with the other approaches, both the loading and inference costs are the lowest. It avoids unnecessary inference compared with the DL2SQL by employing the proposed optimization. On the GPU server, all approaches work better than their counterparts on the CPU, except the DB-UDF approach. Although the usage of GPU effectively reduces the prediction overhead, the loading cost increases significantly, owing to the I/O between CPU memory and GPU memory.

In addition, we vary the query selectivity values and neural model depths and evaluate the performance of different approaches. The results are reported in Table V and VI, respectively. We first change the overall selectivity of the relational predicates from 0.01% to 1%. The DL2SQL-OP consistently performs better than the others, but we also observe that the gap between them is narrowed as more predictions are triggered. Interestingly, the DB-UDF and DB-PyTorch are less affected by the selectivity, indicating the selectivity has an insignificant correlation with the inference overhead for DB-UDF and DB-PyTorch.

In Table VI, we directly use ResNet5 to ResNet40 to evaluate the effect of different neural model depths. We omit the relational algebra cost because the cost of processing relational operators is two or three orders of magnitude smaller than the inference and loading costs for a deeper neural model. We set the selectivity as 0.1% and calculate the average cost. Clearly, the results show that DL2SQL-OP works better

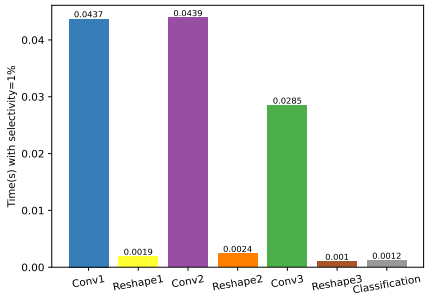


Fig. 9: Costs of CNN Blocks in DL2SQL

than DB-UDF and DB-PyTorch in terms of inference. For a shallow neural network, the running time of DL2SQL-OP is smaller than DB-UDF and DB-PyTorch. Nevertheless, for a deeper neural model, DB-PyTorch outperforms the other two. After performing a breakdown analysis, we find that the DL2SQL still achieves a better inference performance, but its loading cost (load the neural model from relational tables) increases significantly, suggesting that the existing relational table format may slow down the tensor manipulations.

B. Detailed Analysis

Fig. 9 reports the running time of each CNN block in the model(distilled from ResNet34 to 3 Conv+BN+ReLu blocks) used in Fig. 8. In Fig. 9, $Conv1$ denotes the total cost of all neural operators used in the first CNN block. $Reshape1$ represents the cost of the mapping process shown in Fig. 5. $Classification$ is the cost of softmax prediction. According to the experimental results, the main bottleneck is the convolution operators, suggesting that a greater parameter size leads to more running time.

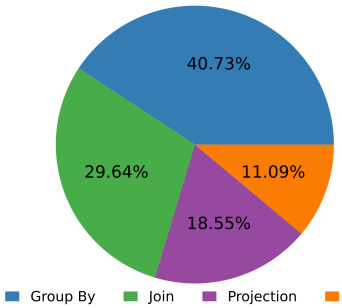


Fig. 10: Percentage of Processing Costs in DL2SQL

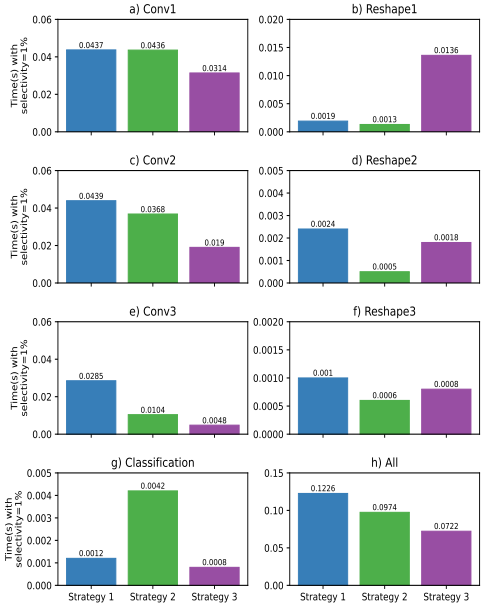


Fig. 11: Performance of Different Pre-Join Strategies

In addition, we profile the execution of generated SQL and investigate the running time distribution of different SQL clauses. Fig. 10 shows the costs of different relational operations in the generated queries. The results show that the relatively expensive operations are Join and GroupBy. Furthermore, we find that many joins are processed inefficiently. In order to join the feature map table and kernel table in Fig. 3, the feature map table is scanned and searched for corresponding tuples in the kernel table using OrderID.

One way to avoid inefficient Join and GroupBy operators is to pre-join those tables together during the data generation process. Fig. 11 shows the performance of CNN blocks using three different pre-join strategies. The default strategy is to adopt no pre-join as before. The second strategy is to avoid the join in the mapping process (i.e., **Q2** presented in Section III) and the GroupBy operation of pooling (i.e., **Q3** presented in Section III). The third strategy avoids the join process between the feature map table and kernel table. As illustrated in Fig. 11, avoiding unnecessary joins can effectively improve the performance of CNN blocks. This indicates that there are many optimization opportunities that can be explored.

C. Effect of Cost Model

Fig. 12 shows estimations of the default database cost model, our customized cost model, and the actual running cost for Type 1 queries. Note that cost models return estimated I/O+CPU costs, and we normalize costs into the running time based on the ratio $r = \frac{seq\_time}{seq\_scan\_cost}$.

We vary the sizes of CNN kernel and input feature map and illustrate the results in Fig. 12a and Fig. 12b, respectively. The CNN computation incurs more overhead for a larger kernel size and feature map size. Our customized outperforms the default DBMS cost model (note that we adopt the log-scale for y-axis).

We further examine the estimation for each neural operator and show the results in Fig. 13. It also indicates that our customized cost model can return a more precise estimation.

Finally, we verify the effectiveness of hints for collaborative queries in Fig. 14. By varying the selectivity, we observe that hint rules can significantly improve the performance by pruning unnecessary computation.

VI. Related Work

The fusion of two research tracks, database, and artificial intelligence, is recently accelerating due to the blooming of deep learning techniques. Wang et al. [19] summarizes the challenges and opportunities.

AI for DB receives many research interests from the database community. A series of works have been proposed for database tuning [20]–[23], database index design [24]–[29], and database query optimization [30]–[35].

Ottertune [20] groups database knobs based on their impacts on the performance, and employs a learning approach to predict proper values for important database knobs. CDBTune [21] and QTune [22] extend the idea by using a reinforcement learning model to pick a proper configuration for the database system regarding a pre-defined query workload. Some new DBMSs, such as SkinnerDB [36] and NeoDB [30], have incorporated learning-based tuning and optimization techniques into their core module, showing a significant improvement over conventional tuning approaches.

Kraska et al. [24] considers the database indexing process as a learning process for the cumulative distribution function (CDF). Neural models are applied to simulate the CDF and to improve the performance. In a recently proposed benchmark [37], the learned indexes outperform conventional indexes, such as B-tree and B^w-tree [38], for a large margin.

Another branch of research focuses on query optimizations. For example, learning-based approaches are adopted to support cardinality estimation [34], plan generation [35], data partitioning [39], and transaction processing [40].

DB for AI is a new emerging topic, where state-of-the-art database techniques are applied to support learning-based processing. Most existing works expand the database's capability using UDF extensions. For instance, MadLib [9] is an open-source library that provides in-database analysis capabilities. It uses UDAF to implement different ML algorithms and provides a set of SQL-based algorithms for machine learning, data

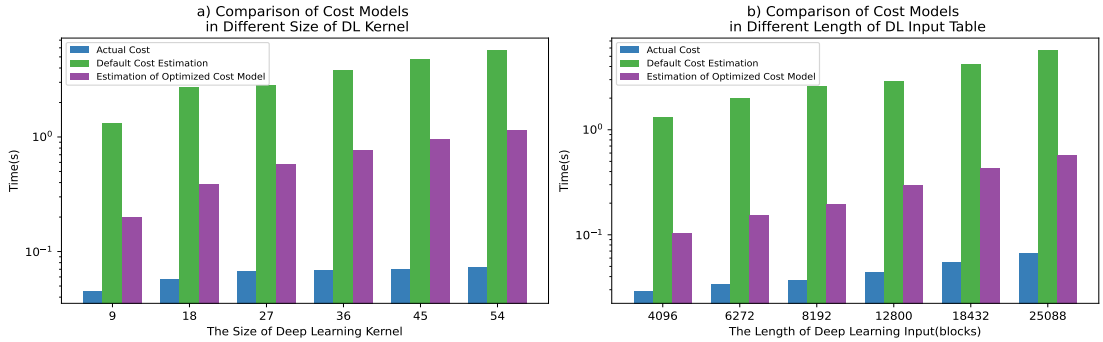


Fig. 12: Comparison of Different Cost Models in Collaborative Query

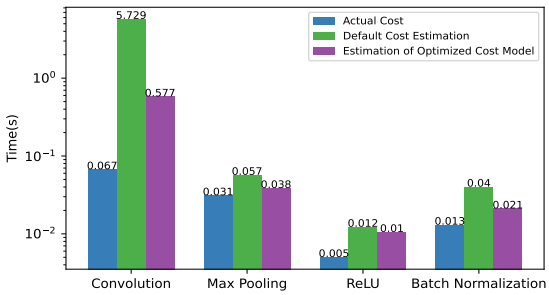


Fig. 13: Comparison of Different Neural Operators

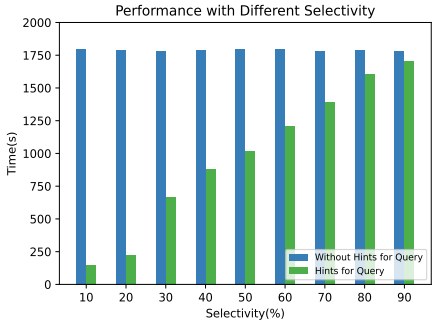


Fig. 14: Performance of Hints for Collaborative Query

mining, and statistics. Vertica-ML [10] uses a P2P architecture that enables data parallelism and model parallelism. Each ML algorithm consists of a Metafunction and several UDFs. The Metafunction passes parameters and implements the algorithm control logic to call the specific UDTF. UDTF uses the SDK provided by Vertica to implement the details.

Different from the UDF approach, MLog [41] proposes a high-level declarative language that integrates machine learning models into DBMS. Learning-based tasks written as MLog will be translated into TensorFlow jobs for processing and hence, also face the problem of system barrier.

Some other researchers apply database techniques to facilitate the machine learning process. LMFAO [13] represents the data-intensive computation required by ML models as batches of group-by aggregates over the relational joins in the database. Karanasos et al. [42] integrates an open-sourced machine learning engine, ONNX, into the SQL server and propose a co-optimization approach. Oracle AutoML [43] presents a pipeline used in Oracle to support the process of AutoML. Jankov et al. [44] examines how to exploit database engines to train backpropagation models efficiently. ModelDB [45] proposes using the DBMS to manage gradually evolved machine learning models.

VII. Conclusion

In this paper, we investigate a new type of query, the collaborative query, and study three possible processing strategies. The first strategy considers both the database and the Deep Learning(DL) system as black boxes and processes the collaborative query by invoking the APIs of the corresponding system. It incurs high I/O costs. The second strategy implements the DL models as built-in User Defined Function(UDF) of the database system, where the collaborative query can be processed without relying on a DL system. It incurs high compilation costs. The third strategy, our novel DL2SQL, implements popular neural operators using SQL syntax. It solves the above problems. The processing of the collaborative query is translated into a series of SQL queries that can be naturally optimized. To evaluate the performance of the three strategies, we use a textile printing dataset from Alibaba's IoT platform and a collection of real-world tasks. The experimental results show that the DL2SQL significantly outperforms the others for small-to-medium size neural models. Moreover, if a pre-join strategy is introduced, the efficiency of DL2SQL can be further improved, suggesting that a lot of optimization work can be done on top of DL2SQL.

Acknowledgment

This work is supported by the Fundamental Research Funds for the Zhejiang Provincial Natural Science Foundation (Grant No. LZ21F020007), the National Natural Science Foundation of China (Grant No. 61872315) and Alibaba Group through Alibaba Innovative Research (AIR) Program.

References

[1] Y. LeCun et al., "Lenet-5, convolutional neural networks," URL: http://yann. lecun. com/exdb/lenet, vol. 20, no. 5, p. 14, 2015.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," Advances in neural information processing systems, vol. 25, pp. 1097–1105, 2012.

[3] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[4] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

[5] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015. IEEE Computer Society, 2015, pp. 1–9. [Online]. Available: https://doi.org/10.1109/CVPR.2015.7298594

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016. IEEE Computer Society, 2016, pp. 770–778. [Online]. Available: https://doi.org/10.1109/CVPR.2016.90

[7] M. A. Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich, "Learning models over relational data using sparse tensors and functional dependencies," ACM Trans. Database Syst., vol. 45, no. 2, pp. 7:1–7:66, 2020. [Online]. Available: https://doi.org/10.1145/3375661

[8] A. V. Joshi, "Amazon's machine learning toolkit: Sagemaker," in Machine Learning and Artificial Intelligence. Springer, 2020, pp. 233–243.

[9] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar, "The madlib analytics library or MAD skills, the SQL," Proc. VLDB Endow., vol. 5, no. 12, pp. 1700–1711, 2012. [Online]. Available: http://vldb.org/pvldb/vol5/p1700_joehellerstein_vldb2012.pdf

[10] A. Fard, A. Le, G. Larionov, W. Dhillon, and C. Bear, "Vertica-ml: Distributed machine learning in vertica database," in Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 755–768. [Online]. Available: https://doi.org/10.1145/3318464.3386137

[11] M. Jasny, T. Ziegler, T. Kraska, U. Röhm, and C. Binnig, "DB4ML - an in-memory database kernel with machine learning support," in Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 159–173. [Online]. Available: https://doi.org/10.1145/3318464.3380575

[12] M. A. Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich, "AC/DC: in-database learning thunderstruck," in Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD 2018, Houston, TX, USA, June 15, 2018, S. Schelter, S. Seufert, and A. Kumar, Eds. ACM, 2018, pp. 8:1–8:10. [Online]. Available: https://doi.org/10.1145/3209889.3209896

[13] M. Schleich and D. Olteanu, "LMFAO: an engine for batches of group-by aggregates," Proc. VLDB Endow., vol. 13, no. 12, pp. 2945–2948, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p2945-schleich.pdf

[14] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," Journal of the royal statistical society. series c (applied statistics), vol. 28, no. 1, pp. 100–108, 1979.

[15] V. D. S. A., "Advanced support vector machines and kernel methods," Neurocomputing, vol. 55, no. 1-2, pp. 5–20, 2003. [Online]. Available: https://doi.org/10.1016/S0925-2312(03)00373-4

[16] G. E. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," CoRR, vol. abs/1503.02531, 2015. [Online]. Available: http://arxiv.org/abs/1503.02531

[17] B. Heo, J. Kim, S. Yun, H. Park, N. Kwak, and J. Y. Choi, "A comprehensive overhaul of feature distillation," in 2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019. IEEE, 2019, pp. 1921–1930. [Online]. Available: https://doi.org/10.1109/ICCV.2019.00201

[18] B. Heo, M. Lee, S. Yun, and J. Y. Choi, "Knowledge transfer via distillation of activation boundaries formed by hidden neurons," in The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019. AAAI Press, 2019, pp. 3779–3787. [Online]. Available: https://doi.org/10.1609/aaai.v33i01.33013779

[19] W. Wang, M. Zhang, G. Chen, H. V. Jagadish, B. C. Ooi, and K. Tan, "Database meets deep learning: Challenges and opportunities," SIGMOD Rec., vol. 45, no. 2, pp. 17–22, 2016. [Online]. Available: https://doi.org/10.1145/3003665.3003669

[20] B. Zhang, D. V. Aken, J. Wang, T. Dai, S. Jiang, J. Lao, S. Sheng, A. Pavlo, and G. J. Gordon, "A demonstration of the ottertune automatic database management system tuning service," Proc. VLDB Endow., vol. 11, no. 12, pp. 1910–1913, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol11/p1910-zhang.pdf

[21] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li, "An end-to-end automatic cloud database tuning system using deep reinforcement learning," in Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019, P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds. ACM, 2019, pp. 415–432. [Online]. Available: https://doi.org/10.1145/3299869.3300085

[22] G. Li, X. Zhou, S. Li, and B. Gao, "Qtune: A query-aware database tuning system with deep reinforcement learning," Proc. VLDB Endow., vol. 12, no. 12, pp. 2118–2130, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p2118-li.pdf

[23] D. V. Aken, D. Yang, S. Brillard, A. Fiorino, B. Zhang, C. Billian, and A. Pavlo, "An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems," Proc. VLDB Endow., vol. 14, no. 7, pp. 1241–1253, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p1241-aken.pdf

[24] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018, G. Das, C. M. Jermaine, and P. A. Bernstein, Eds. ACM, 2018, pp. 489–504. [Online]. Available: https://doi.org/10.1145/3183713.3196909

[25] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. B. Lomet, and T. Kraska, "ALEX: an updatable adaptive learned index," in Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 969–984. [Online]. Available: https://doi.org/10.1145/3318464.3389711

[26] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska, "Tsunami: A learned multi-dimensional index for correlated data and skewed workloads," Proc. VLDB Endow., vol. 14, no. 2, pp. 74–86, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol14/p74-ding.pdf

[27] R. Marcus, E. Zhang, and T. Kraska, "Cdfshop: Exploring and optimizing learned index structures," in Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 2789–2792. [Online]. Available: https://doi.org/10.1145/3318464.3384706

[28] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "Radixspline: a single-pass learned index," in Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020, R. Bordawekar, O. Shmueli, N. Tatbul, and T. K. Ho, Eds. ACM, 2020, pp. 5:1–5:5. [Online]. Available: https://doi.org/10.1145/3401071.3401659

[29] X. Tang, S. Wu, G. Chen, J. Gao, W. Cao, and Z. Pang, "A learning to tune framework for LSH," in 37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021. IEEE, 2021, pp. 2201–2206. [Online]. Available: https://doi.org/10.1109/ICDE51399.2021.00224

[30] R. C. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: A learned query optimizer," Proc. VLDB Endow., vol. 12, no. 11, pp. 1705–1718, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p1705-marcus.pdf

[31] S. Hasan, S. Thirumuruganathan, J. Augustine, N. Koudas, and G. Das, "Deep learning models for selectivity estimation of multi-attribute queries," in Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 1035–1050. [Online]. Available: https://doi.org/10.1145/3318464.3389741

[32] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making learned query optimization practical," in SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021, G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds. ACM, 2021, pp. 1275–1288. [Online]. Available: https://doi.org/10.1145/3448016.3452838

[33] L. Ma, W. Zhang, J. Jiao, W. Wang, M. Butrovich, W. S. Lim, P. Menon, and A. Pavlo, "MB2: decomposed behavior modeling for self-driving database management systems," in SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021, G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds. ACM, 2021, pp. 1248–1261. [Online]. Available: https://doi.org/10.1145/3448016.3457276

[34] Z. Yang, A. Kamsetty, S. Luan, E. Liang, Y. Duan, P. Chen, and I. Stoica, "Neurocard: One cardinality estimator for all tables," Proc. VLDB Endow., vol. 14, no. 1, pp. 61–73, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol14/p61-yang.pdf

[35] X. Yu, G. Li, C. Chai, and N. Tang, "Reinforcement learning with tree-lstm for join order selection," in 36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020. IEEE, 2020, pp. 1297–1308. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00116

[36] I. Trummer, J. Wang, D. Maram, S. Moseley, S. Jo, and J. Antonakakis, "Skinnerdb: Regret-bounded query evaluation via reinforcement learning," in Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019, P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds. ACM, 2019, pp. 1153–1170. [Online]. Available: https://doi.org/10.1145/3299869.3300088

[37] L. Bindschaedler, A. Kipf, T. Kraska, R. Marcus, and U. F. Minhas, "Towards a benchmark for learned systems," in 37th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2021, Chania, Greece, April 19-22, 2021. IEEE, 2021, pp. 127–133. [Online]. Available: https://doi.org/10.1109/ICDEW53142.2021.00029

[38] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "The bw-tree: A b-tree for new hardware platforms," in 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013, C. S. Jensen, C. M. Jermaine, and X. Zhou, Eds. IEEE Computer Society, 2013, pp. 302–313. [Online]. Available: https://doi.org/10.1109/ICDE.2013.6544834

[39] B. Hilprecht, C. Binnig, and U. Röhm, "Learning a partitioning advisor for cloud databases," in Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 143–157. [Online]. Available: https://doi.org/10.1145/3318464.3389704

[40] Y. Sheng, A. Tomasic, T. Zhang, and A. Pavlo, "Scheduling OLTP transactions via learned abort prediction," in Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019, R. Bordawekar and O. Shmueli, Eds. ACM, 2019, pp. 1:1–1:8. [Online]. Available: https://doi.org/10.1145/3329859.3329871

[41] X. Li, B. Cui, Y. Chen, W. Wu, and C. Zhang, "Mlog: Towards declarative in-database machine learning," Proc. VLDB Endow., vol. 10, no. 12, pp. 1933–1936, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol10/p1933-zhang.pdf

[42] K. Karanasos, M. Interlandi, F. Psallidas, R. Sen, K. Park, I. Popivanov, D. Xin, S. Nakandala, S. Krishnan, M. Weimer, Y. Yu, R. Ramakrishnan, and C. Curino, "Extending relational query processing with ML inference," in 10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings. www.cidrdb.org, 2020. [Online]. Available: http://cidrdb.org/cidr2020/papers/p24-karanasos-cidr20.pdf

[43] A. Yakovlev, H. F. Moghadam, A. Moharrer, J. Cai, N. Chavoshi, V. Varadarajan, S. R. Agrawal, T. Karnagel, S. Idicula, S. Jinturkar, and N. Agarwal, "Oracle automl: A fast and predictive automl pipeline," Proc. VLDB Endow., vol. 13, no. 12, pp. 3166–3180, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p3166-yakovlev.pdf

[44] D. Jankov, S. Luo, B. Yuan, Z. Cai, J. Zou, C. Jermaine, and Z. J. Gao, "Declarative recursive computation on an RDBMS," Proc. VLDB Endow., vol. 12, no. 7, pp. 822–835, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p822-jankov.pdf

[45] M. Vartak, "MODELDB: A system for machine learning model management," in 8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings. www.cidrdb.org, 2017. [Online]. Available: http://cidrdb.org/cidr2017/gongshow/abstracts/cidr2017_112.pdf