The VLDB Journal (2025) 34:12
https://doi.org/10.1007/s00778-024-00888-3

REGULAR PAPER

In-database query optimization on SQL with ML predicates

Yunyan Guo[1] · Guoliang Li[1] · Ruilin Hu[1] · Yong Wang[1]

Received: 8 March 2024 / Revised: 21 October 2024 / Accepted: 25 November 2024 / Published online: 23 December 2024

Abstract
Extended SQL with machine learning (ML) predicates, commonly referred to as *SQL+ML*, integrates ML abilities into traditional SQL processing in databases. When processing SQL+ML queries, some methods move data from database (DB) systems to ML systems to support SQL+ML queries. Such methods are not only costly due to maintaining two copies of data, but also pose security risks due to data movement. Fortunately, in-database SQL+ML processing addresses these limitations. However, conventional DB optimizers take ML predicates as UDFs (user-defined functions) and cannot optimize them using query rewriter and cost models. To boost the efficiency of in-database SQL+ML processing, this paper proposes to generate SQL predicates based on ML predicates and add them into SQL+ML queries, which can prune a significant number of irrelevant tuples and thus improve the performance. Optimizing SQL+ML queries presents three challenges: (C1) how to generate valid SQL predicates, (C2) how to select high-quality SQL predicates, and (C3) how to optimize the query using SQL predicates. To address these challenges, we propose `Smart`, which integrates three novel modules into the database optimizer: (1) inference rewrite: generating tight and valid SQL predicates for logical optimization; (2) progressive inference: selecting high-pruning-power but low-overhead SQL predicates to prune irrelevant tuples; (3) cost-optimal inference: optimizing the cost of query plan with selected SQL predicates for physical optimization. We implemented `Smart` in PostgreSQL and evaluated it on four widely-used benchmarks, JOB, TPC-H, SSB, and Flight. Experimental results revealed that `Smart` performed up to three orders of magnitude faster than the state-of-art baselines.

Keywords Query optimization · Model inference · Machine learning · Database system

1 Introduction

Many customers have both SQL analytics and machine learning (ML) requirements on top of their data in databases [20, 47]. Existing methods usually use two independent systems to support this requirement, i.e., using DBMS to support SQL analytics, exporting the data to another ML platform, and using ML platforms to support ML predicates [23, 43]. Obviously this solution is expensive and insecure, because it maintains two copies of the data, transforms the data between the two systems [46], and cannot effectively reuse the hardware environments between the two systems [4].

To alleviate this limitation, the concept of in-database machine learning is introduced [12, 35]. It extends SQL capabilities to support ML operators [5, 37] as *SQL+ML*. SQL+ML encompasses the *ML predicate*, which consists of an inference function by utilizing a trained ML model (e.g., linear regression `lr_model`) and specific columns as features, a comparative operator and a value (e.g., $<$ 14). Figure 1 illustrates an SQL+ML example that the ML predicate reports the tuples where the inference result satisfies the predicate.

The primary objective of in-database ML systems is to optimize SQL+ML queries. However, the processing of ML predicate incurs significant costs due to its computationally intensive operations, which involve *joining* multiple tables, inferring the value for each joined tuple, and returning the tuples that satisfy the ML predicate [19, 51]. A natural idea to accelerate the in-database SQL+ML processing is to *prune irrelevant tuples before performing joins.*

To this end, we propose deriving "valid" SQL predicates, solely based on columns and values, from ML predicates involving ML models. Notably, a "valid" SQL predicate is a necessary condition for an ML predicate, that is, if a tuple satisfies the ML predicate, it must also satisfy the SQL pred-

✉ Guoliang Li
liguoliang@tsinghua.edu.cn

1 Tsinghua University, Beijing, China

```
SELECT  A.id, price
FROM    Apartment as A,  Building as B,
        District as C,    Landlord as D,  Construction as E
WHERE   A.id = B.aid    AND  A.lid = D.id
   AND  B.zone = C.id  AND  B.eid = E.id
   AND  E.name = "XYZ"
   AND  lr_model ( [A.room_num,    B.building_age,
         C.hospital_num,   D.rating_score] ) < 14 AS price
ORDER BY price;
```

Fig. 1 Example of an SQL+ML. In this query, an ML predicate (bold red lines) utilizes a trained ML model to infer and filter tuples

icate[1]. These derived SQL predicates can be optimized by DB query optimizers as well. Collaboratively optimizing ML predicates and other DB operators can early prune some irrelevant tuples using SQL predicates, and consequently enhance the efficiency of SQL+ML processing.

In this paper, we study how to optimize an SQL+ML query through employing valid SQL predicates within DB optimizers.

Challenge 1: How to generate valid SQL predicates that achieve the equivalence guarantee? The generated SQL predicate should be valid and tight, ensuring that it does not include false-negative tuples and minimizes the occurrence of false-positive tuples.

Contribution 1: Inference rewrite for SQL predicate generation. By combining the model's weight and structure with conditions in ML predicate, we derive valid SQL predicates. Inference rewrite incorporates these SQL predicates into the query tree as a logical optimization step in DB optimizer.

Challenge 2: How to choose high-quality SQL predicates? The number of valid SQL predicates is large when ML predicate involves many columns across numerous tables. However, multiple SQL predicates result in execution overhead and pruning redundancy. Therefore, it becomes crucial to judiciously select a high-quality subset of valid SQL predicates.

Contribution 2: Progressive inference for SQL predicate selection. We compute the pruning redundancy and overhead of different SQL predicates, and select a subset with the maximum pruning performance. Aiming to minimize the overhead, the selected SQL predicates infer tuples progressively.

Challenge 3: Where to place the SQL predicates in the plan tree? Each SQL predicate can be positioned at different nodes in the plan tree to affect the subsequent operators' costs, while the relative positions of SQL predicates influence each other. Therefore, it is crucial to identify the cost-optimal positions for them.

Contribution 3: Cost-optimal inference for SQL predicates placement. We propose computing the cost of placing multiple SQL predicates at different positions, and then design a dynamic programming algorithm that finds the cost-optimal positions.

We summarize the main contributions of this paper.

(1) To the best of our knowledge, this is the first work focused on optimizing ML predicates within the DB optimizer.

(2) We propose a logical optimization technique, inference rewrite, that generates SQL predicates while preserving equivalence.

(3) We introduce the progressive inference to select high-quality SQL predicates, aiming to improve query performance.

(4) We devise a cost-optimal inference technique to optimize the physical positions for SQL predicates.

(5) We have implemented `Smart` into PostgreSQL. Experimental results on most commonly used ML models, including linear regression, logistic regression, and decision tree models, showed that `Smart` outperformed state-of-the-art baselines by 2.28-1000x on four benchmarks with real-world datasets.

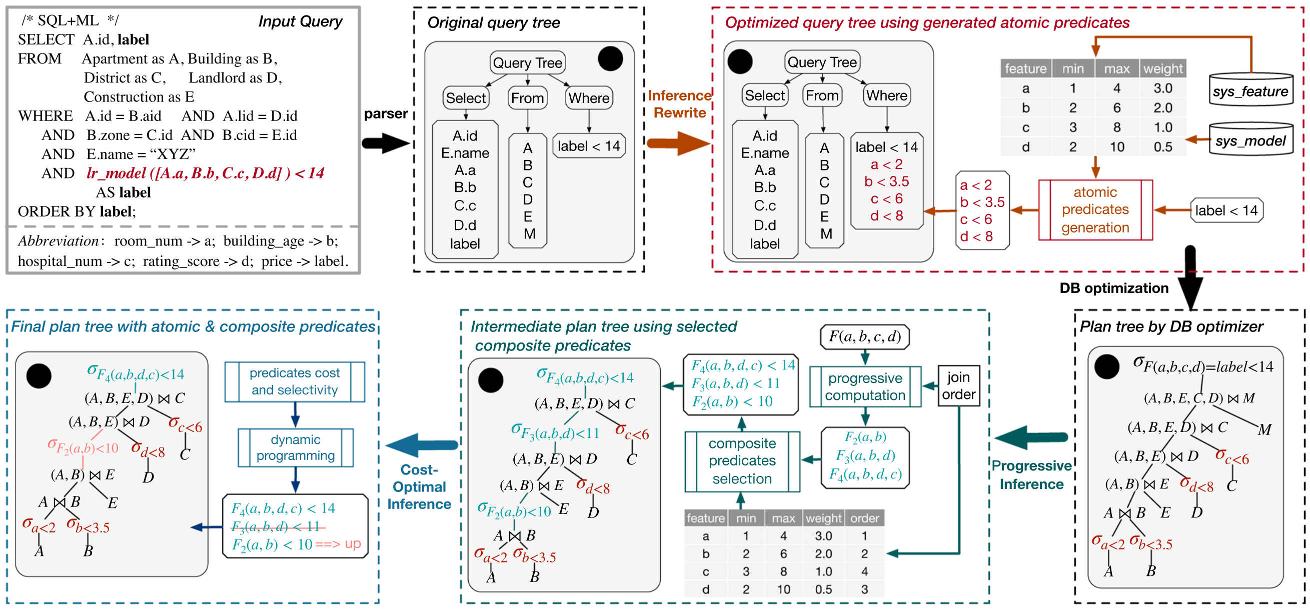
Notably, the "State of Data Science and Machine Learning" report [22] indicates that the most commonly used ML algorithms are "linear or logistic regression" (80.3%) and followed closely by "decision tree" (74.1%), which are popular on relational datasets. Our techniques can also be adapted for linear/non-linear SVM and random forests with slight modification, and can be seamlessly integrated in most modern DB query optimizers.

2 System framework

In this section, we first define SQL+ML and then present a framework for optimizing SQL+ML queries.

SQL+ML queries. We extend SQL to support ML predicates, where each ML predicate can be modelled as "`MLmodel(Columns) OP value`", where `MLmodel` is an ML model, e.g., linear regression, `Columns` is a list of columns, `OP` is a comparison operator (e.g., $<, >, =$), `value` is a user-specific value. The ML model will compute a value for each tuple on the specified columns, and each tuple satisfies the predicate if the corresponding model inference value meets the condition. For example, in Fig. 1, "`lr_model([A.room_num, B.building_age, C.hospital_num, D.rating_score]) < 14`" is an ML predicate, which reports tuples with inferred prices are smaller than 14. An SQL where the `WHERE` clause contains ML predicates is called an SQL+ML query. Notice that the model in the ML predicate can be created by "`CREATE MODEL lr_model USING`

[1] We will discuss how to generate such SQL predicates in Sects. 3.2 and 3.3.


```
/* SQL+ML */                                   Input Query
SELECT  A.id, label
FROM     Apartment as A, Building as B,
         District as C,   Landlord as D,
         Construction as E
WHERE   A.id = B.aid    AND  A.lid = D.id
   AND   B.zone = C.id  AND  B.cid = E.id
   AND   E.name = "XYZ"
   AND   lr_model ([A.a, B.b, C.c, D.d] ) < 14
            AS label
ORDER BY label;
```

Abbreviation: room_num -> a; building_age -> b; hospital_num -> c; rating_score -> d; price -> label.

feature	min	max	weight
a	1	4	3.0
b	2	6	2.0
c	3	8	1.0
d	2	10	0.5

feature	min	max	weight	order
a	1	4	3.0	1
b	2	6	2.0	2
c	3	8	1.0	4
d	2	10	0.5	3

Fig. 2 A running example of SQL+ML query optimization in Smart

linear_regression FEATURES Columns FROM Tables;" and trained offline, and then the model will be maintained in databases (e.g., in the view of database systems) and can be used in SQL+ML queries. In addition, if the columns in the ML predicates are from multiple tables, the model training queries and inference queries should explicitly specify the join conditions, i.e., tables A, B, C, and D appear in the FROM clause and the join conditions are in the WHERE clause in Fig. 1.

Key idea. If an SQL+ML query contains an ML predicate, traditional databases typically first compute the intermediate table for the SQL+ML query by omitting the ML predicate, then infer values of these tuples based on the model in the ML predicate, and finally prune the tuples that do not satisfy the ML predicate. However, this approach definitely incurs high computational costs, especially when the intermediate table comprises a large number of tuples after multiple joins. Therefore, it's crucial to prune irrelevant tuples either before or during the join operators, especially when the intermediate table is substantial but only a small proportion of tuples satisfy the ML predicate. To achieve this goal, we design **SQL**+**ML** optimizer **AR**chi**T**ecture (Smart), which generates, selects, and places valid SQL predicates (e.g., column OP value) to optimize ML predicates within the DB optimizer. Given that these SQL predicates can prune a significant number of irrelevant tuples, Smart can notably improve the performance of SQL+ML in DB systems.

We classify SQL predicates into *atomic predicates* with single table (e.g., A.room_num < 4), and *composite predicates* with multiple tables (e.g., w_1 * A.room_num + w_2 * B.building_age < 10, where w_1 and w_2 are two weights of lr_model). Next we discuss how to generate SQL predicates, whether to use SQL predicates, and how to place them within DB optimizer.

Atomic Predicates. (1) Generation: Each ML predicate contains multiple tables, and we generate a corresponding atomic predicate for each table. *(2) Selection:* The cost of executing atomic predicates is almost negligible when compared to the cost of table scan and join operators. Each atomic predicate highly likely prunes many tuples in the corresponding basic table, so Smart directly preserves the generated atomic predicates. *(3) Placement:* The atomic predicates are common in SQL and optimized by DB. Thus, Smart simply appends the atomic predicates to the scan node in the query tree.

Overall, the generation, selection, and placement of atomic predicates constitute the *inference rewrite* module as shown in Figs. 2 and 3. Since Smart aims to leverage existing optimization algorithms to optimize atomic predicates, this module is deployed at the beginning of logical optimization stage.

Composite predicates. (1) Generation: Composite predicates are generated by combining multiple tables and their corresponding model weights. They provide tighter necessary conditions than atomic predicates, thus can further prune irrelevant tuples. The generation function, which extends from the atomic predicate generation function, will be discussed in Sect. 3.3. *(2) Selection:* Unfortunately, the total number of candidate composite predicates that can be generated reaches up to $2^D - D - 1$, where D represents the number of columns in the ML predicate, resulting in significant time and space costs when generating all of them. Hence,

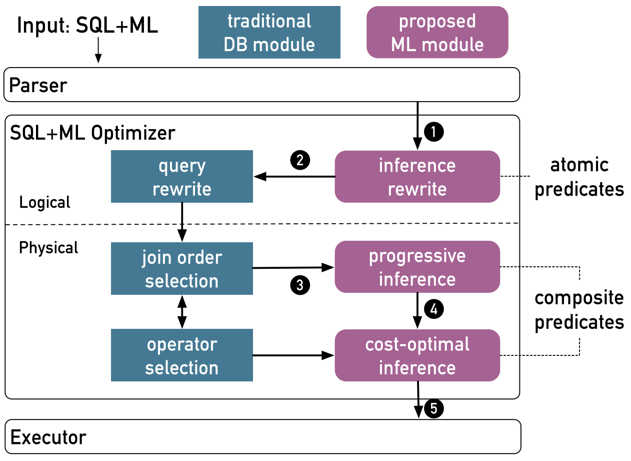


Fig. 3 The framework of `Smart`

it becomes crucial to select effective composite predicates that possess high pruning ability. We propose a tree-based method to evaluate the pruning redundancy between different composite predicates and select the high-quality composite predicates to maximum pruning performance in Sect. 4, which corresponds to the *progressive inference* module in Figs. 2 and 3. *(3) Placement:* Different places lead to varying total costs. Firstly, the computational cost is affected by the positions of other composite predicates due to progressive inference. Secondly, the pruning benefits change based on positions. Considering both factors, `Smart` utilizes a dynamic programming algorithm to optimize the plan tree's cost through iteratively determining the optimal placement for each selected composite predicate from bottom to top, after localizing cascading effects. This is the *cost-optimal inference* module in Figs. 2 and 3.

System workflow. The framework is shown in Fig. 3, and Fig. 2 illustrates a running example. A user-given SQL+ML is parsed into a query tree ❶ as the input of `Smart`. In the logical optimization stage, the inference rewrite module rewrites ML predicate using atomic predicates, leading to a modified query tree ❷. Then it undergoes traditional modules, producing a physical plan tree ❸. In the physical optimization phase, the progressive inference module replaces ML predicate with composite predicates as ❹, and the cost-optimal inference module refines them to obtain the cost-optimal plan tree ❺.

Optimizing SQL+ML directly within DB optimizer is notably more effective. Since ML predicates and DB operators are intricately intertwined, they should not be optimized separately. DB optimizers optimize ML predicates using statistical data and cost estimation models. `Smart` treats SQL+ML as a cohesive entity, merging both the newly introduced modules and the existing ones to enhance the performance collaboratively.

Different from existing frameworks. `Smart` integrates ML predicate optimization directly within the DBMS query optimizer, unlike Raven [23, 38], which externally compiles ML models into SQL and relies on the database's native capabilities. Raven's key contribution lies in partitioning tasks between SQL for the database and Python for TensorFlow while optimizing the intermediate representation (IR). In contrast, our method focuses on pruning data rather than simplifying models and introduces new SQL predicates to reduce data processed during query execution, especially in complex queries where all tables are necessary. Our approach and Raven's can be seen as complementary, with Raven's output serving as input for our optimizer.

While both PP [34] and `Smart` aim to filter data before inference, they address different challenges. PP focuses on unstructured data where inference dominates costs, whereas our method targets joined relational data where join operations are the bottleneck. PP adds probabilistic predicates but does not address the cost of joins in relational data, limiting its effectiveness. Our method generates SQL predicates specifically to optimize joins, reducing intermediate dataset sizes and ensuring query equivalence without requiring additional model training. Despite the differences, our approach can complement PP by optimizing join operations for its probabilistic predicates in relevant scenarios.

Orion [27] proposed that in each iteration of the ML model training process, the total gradient calculation on the total tuples can be accelerated over factorized joins [8, 33]. Different from Orion, `Smart` filters irrelevant tuples through ML predicate in query before joins.

3 SQL predicate generation

In this section, we take linear regression, logistic regression, decision tree as examples to discuss how to generate SQL predicates.

3.1 Preliminary

Linear regression inference. A linear regression model f has a model weight array $\mathbf{w} = \{w_0, \ldots, w_D\}$. Given a tuple $\mathbf{x} = \{\mathbf{x_1}, \ldots, \mathbf{x_D}\}$, the inferred numeric value of model f is defined as $f(\mathbf{x}, \mathbf{w}) \in \mathbf{R}$:

$$f(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{d=1}^{D} w_d \cdot x_d \tag{1}$$

We abbreviate this linear regression model as `lr_f`, then users can write an SQL+ML query with a range ML predicate using it.

Range ML predicate. Given a numeric value $l_{up} \in \mathbf{R}$ as the upper bound of the inferred value with the inference function $f(\mathbf{x}, \mathbf{w})$, the range ML predicate can be written in an

SQL+ML query as "`lr_f([Col`$_1$`, ..., Col`$_D$`])` $< l_{up}$" where `Col`$_d$ is the column name of d-th feature, which selects a list of tuples

$$(\mathbf{x}, f(\mathbf{x}, \mathbf{w})), \mathbf{x} \in \{\mathbf{X}\}, \text{ s.t. } f(\mathbf{x}, \mathbf{w}) < l_{up}$$

Logistic regression inference. A logistic regression model g has a model weight array $\mathbf{w} = \{w_0, \dots, w_D\}$, and $\mathcal{S}(x) = \frac{1}{1+e^{-x}}$ is the *sigmoid* function. Given a tuple $\mathbf{x} = \{\mathbf{x_1}, \dots, \mathbf{x_D}\}$, the inferred category label of model g on tuple $\mathbf{x}$ is defined as $g(\mathbf{x}, \mathbf{w}) \in \{\texttt{FALSE}, \texttt{TRUE}\}$:

$$g(\mathbf{x}, \mathbf{w}) = \begin{cases} \texttt{FALSE}, \text{ s.t. } \mathcal{S}(w_0 + \sum_{d=1}^{D} w_d \cdot x_d) < 0.5 \\ \texttt{TRUE}, \text{ s.t. } \mathcal{S}(w_0 + \sum_{d=1}^{D} w_d \cdot x_d) \geq 0.5 \end{cases} \tag{2}$$

Similarly, we abbreviate this logistic regression model as `lr_g`, then users can use equality ML predicates to write SQL+ML.

Equality ML predicate. Given a category label `TRUE` as the required inferred label with the inference function $g(\mathbf{x_n}, \mathbf{w})$, the equality ML predicate can be written in an SQL+ML query as "`lr_g([Col`$_1$`, ..., Col`$_D$`]) = TRUE`" where `Col`$_d$ is the column name of d-th feature, which selects a list of tuples

$$(\mathbf{x}, \texttt{TRUE}), \mathbf{x} \in \{\mathbf{X}\}, \text{ s.t. } g(\mathbf{x}, \mathbf{w}) = \texttt{TRUE}$$

We take a binary classification as an example, and our method can be easily extended to support multi-class classification.

Decision tree inference. A decision tree model h is typically structured as a tree. Each internal node tests a feature condition (one column predicate) for the tuple $\mathbf{x}$, and the traversal turns left if the condition is satisfied or turns right otherwise. Each leaf node represents the inferred value. The inferred value of model h for tuple $\mathbf{x}$ is obtained by traversing the tree from the root to a leaf: $h(\mathbf{x}) \in \{v_1, v_2, \dots, v_k\}$. The inferred value is one of the unique decision labels $\{v_1, v_2, \dots, v_k\}$, which can be reached through various paths in the tree.

Similarly, the equality ML predicate can be written as "`dt_h([Col`$_1$`, ..., Col`$_D$`]) = v`", where $v \in \{v_1, v_2, \dots, v_k\}$.

SQL predicate generation. Given an SQL+ML query Q, SQL predicate generation aims to generate a set of valid SQL predicates, such that adding them into Q will keep equivalent with original query.

3.2 Atomic predicate generation

An ML predicate is a condition that combines columns from multiple tables, whereas an atomic predicate, which involves only a column from a single table, represents a necessary but not sufficient (less restrictive) condition. We first discuss linear models and then move on to discuss tree models. We consider each table only maintains one column, then multiple columns are easy to expend.

Linear models. To derive the atomic necessary conditions, it is necessary to exclude other columns in other tables with additional conditions. Thus, we propose incorporating the min/max ranges of all columns from DBMS catalogs, $\forall i \in \{1, \dots, D\}, x_{i,min} \leq \mathbf{x_i} \leq x_{i,max}$, to create a combined inequality. DBMS catalogs generally store up-to-date statistic data for column values. Theorem 1 serves as an example of the derivation.

Theorem 1 (Atomic Necessary Condition for Linear Models) *When an SQL+ML query includes a range ML predicate as "`lr_f([Col`$_1$`, ..., Col`$_D$`])` $< l_{up}$" with the inference function in Equation (1), and let*

$$M_i = l_{up} - w_0 - \sum_{d \in S_1}^{d \neq i} w_d \cdot x_{d,min} - \sum_{d \in S_2}^{d \neq i} w_d \cdot x_{d,max}$$

then the ith necessary condition of the ith column value is derived as:

$$\mathbf{x_i} \begin{cases} <(M_i/w_i)\,, \text{ if } w_i > 0 \\ >(M_i/w_i)\,, \text{ if } w_i < 0 \end{cases}$$

where $x_{d,max}$ and $x_{d,min}$ are the maximum and minimum values of column `Col`$_d$. Also, $S_1 \cup S_2 = \{1, \dots, D\}$, $S_1 \cap S_2 = \emptyset$, and $\forall d \in S_1$ satisfies $w_d > 0$, $\forall d \in S_2$ satisfies $w_d < 0$.[2]

For an equality ML predicate `lr_g([Col`$_1$`,...,Col`$_D$`]) = TRUE`, it holds that $\mathcal{S}(w_0 + \sum_{d=1}^{D} w_d \cdot \mathbf{x_d}) \geq 0.5$ according to Equation (2), indicating that

$$w_0 + \sum_{d=1}^{D} w_d \cdot \mathbf{x_d} \geq 0.$$

Therefore, it is similar to the range ML predicates setting l_{low} to 0.

The model weights can be either positive (S_1) or negative (S_2), leading to different derivations in Theorem 1. However, for the sake of simplicity in further discussions, we consider each negative weight w_d as logically positive, represented by the transformation:

$$[w_d, \mathbf{x_d}, x_{d,min}, x_{d,max}] \leftarrow (-1) \cdot [w_d, \mathbf{x_d}, x_{d,max}, x_{d,min}]$$

[2] There is no need to consider $w_d = 0$ for inference since that means the corresponding column is useless.

By doing so, we only need to consider situations where all weights are positive. Subsequently, we present the atomic necessary conditions for the ML predicates.

Decision tree. For each path in the decision tree, every branch represents a condition. Within the conditions of a path, a column may appear multiple or zero times. By intersecting multiple occurrences and using $x_{min} < x < x_{max}$ for the omitted one, we transform them into D distinct necessary conditions.

Theorem 2 (Atomic Necessary Condition for Decision Tree) *When an SQL+ML query includes an ML predicate as "`dt_h([Col`$_1$`, ..., Col`$_D$`]) =`v" and $v \in \{v_1, \ldots, v_k\}$ from model h, there exists a set of paths $\mathcal{P}(v) = \{P_i, \ldots, P_J\}$ that lead to this label, where J is the size of $\mathcal{P}(v)$. For any specific path $P_j \in \mathcal{P}(v)$, it consists of a set of conditions, which are denoted as $P_j = \{c_{j,1}, \ldots, c_{j,D}\}$. For the ith column $\mathbf{x}_i$, the atomic necessary condition for lable v is derived as a logical disjunction:*

$$\mathbf{x}_i : c_{1,i} \vee c_{2,i} \vee \ldots \vee c_{J,i}$$

Atomic necessary conditions. A single SQL+ML query can include multiple ML predicates. For example, "`lr_f(Cols) BETWEEN` l_{low} `AND` l_{up}" includes two ML predicates sharing the same model and columns, or "`dt_h1(Cols)=`a_5 `AND dt_h2(Cols)=`b_3" involves some of the same columns but different models. In such cases, each ML predicate derives a set of atomic necessary conditions using Theorem 1 or Theorem 2. When combining the two sets, they mutually reinforce each other.[3]

Assuming all weights are positive in a linear regression model as an example, specifically, the necessary conditions are computed using the minimum column values to determine the upper bounds. Subsequently, if these upper bounds are used instead of the maximum values to compute the necessary conditions, the lower bounds of the column values become tighter. Hence, we propose Algorithm 1 to leverage this effect.

Furthermore, the initialization in Algorithm 1 can use existing predicates in SQL+ML query. For example, the lower bound value of column `rating_score` can be initialized using `D.rating_score>4`, if this predicate appears in the original SQL+ML query, instead of the minimum value from catalog to provide a tighter additional condition.

If any column exists in both different decision tree models, the two generated atomic necessary condition sets can also reinforce each other. Taking "`dt_h1(Col`$_1$`, Col`$_2$`) =` a_5 `AND dt_h2(Col`$_1$`, Col`$_3$`) =` b_3" as an example, the conditions of `Col`$_1$ with $\vee$ and $\wedge$ can be optimized with existing logical algebra algorithms.

Algorithm 1: Atomic Necessary Conditions for Linear Regression

```
Input: $l_{up}$, $l_{low}$, $w$, $max\_iter$ or $\epsilon$
Output: $\forall d \in \{1, \ldots, D\}$, $x_{d,low}$, $x_{d,up}$
/* Initialize with catalogs or SQL
   predicates:                                  */
1 for $d \leftarrow 1$ to $D$ do
2     $(x_{d,low}, x_{d,up}) \leftarrow (x_{d,min}, x_{d,max})$;
3 end
/* Update with Theorem 1:                       */
4 repeat
5     for $i \leftarrow 1$ to $D$ do
6         $x_{i,low} \leftarrow \frac{1}{w_i}(l_{low} - w_0 - \sum_{d=1,d\neq i}^{D} w_d \cdot x_{d,up})$;
7         $x_{i,up} \leftarrow \frac{1}{w_i}(l_{up} - w_0 - \sum_{d=1,d\neq i}^{D} w_d \cdot x_{d,low})$.
8     end
9 until $iter\_num > max\_iter$ OR $output$ $changes < \epsilon$;
```

Atomic predicates generation. From an ML predicate and column ranges, a set of atomic necessary conditions can be derived. In this set, each atomic necessary condition includes one column, a comparative operator, and a bound of value, such as $\mathbf{x_i} \leq x_{i,up}$, which can be used to generate one atomic predicate as "`Col`$_i \leq x_{i,up}$". Therefore, the atomic predicates generation function generates atomic predicates for ML predicates in an SQL+ML query. For example, by using the weight values from the *sys_model* in Fig. 2, where w_1, w_2, w_3, w_4 are 3, 2, 1, 0.5, respectively, and the column statistics from *sys_feature* in Fig. 2, with min(B.b)=2, min(C.c)=3 and min(D.d)=2 as additional conditions, we can derive an atomic predicate "A.a < 4", since:

$$w_1 \cdot \texttt{A.a} < 14 - w_2 \cdot \texttt{min(B.b)} - w_3 \cdot \texttt{min(C.c)} - w_4 \cdot \texttt{min(D.d)}$$

3.3 Composite predicate generation

Similarly, before generating composite predicates, it is crucial to derive the composite necessary conditions for ML predicates firstly.

Theorem 3 (Composite Necessary Condition for Linear Models) *When an SQL+ML includes a range ML predicate as "`lr_f([Col`$_1$`, ..., Col`$_D$`]) <` l_{up}" with the inference function as shown in Equation (1), a generated composite predicate including all columns in a subset $I \subseteq \{1, \ldots, D\}$ can be derived as*

$$\sum_{i \in I} w_i \cdot \mathbf{x_i} < l_{I,up} = l_{up} - w_0 - \sum_{d \in S_1, d \notin I} w_d \cdot x_{d,up} - \sum_{d \in S_2, d \notin I} w_d \cdot x_{d,low}$$

[3] Supporting disjunctions is similar but more verbose, and we leave it as a future work.

where $x_{d,up}$ and $x_{d,low}$ are the bounds of column $\texttt{Col}_d$ after generating atomic predicates. S_1, S_2 are described in Theorem 1.

When `Smart` selects a composite predicate with the input columns set I, the output composite predicate is "$\sum_{i \in I} w_i \cdot \texttt{Col}_i < l_{I,up}$" from the corresponding composite necessary condition. For example, by using the weight values from the *sys_model* in Fig. 2, where w_1, w_2, w_3, w_4 are 3, 2, 1, 0.5, respectively, and the column statistics from *sys_feature* in Fig. 2, with min(C.c)=3 and min(D.d)=2 as additional conditions, we can derive a composite predicate $w_1 * \texttt{A.a} + w_2 * \texttt{B.b} < 10$, since:

$$w_1 \cdot \texttt{A.a} + w_2 \cdot \texttt{min(B.b)} < 14 - w_3 \cdot \texttt{min(C.c)} - w_4 \cdot \texttt{min(D.d)}$$

Theorem 4 (Composite Necessary Condition for Decision Tree) *When an SQL+ML includes an ML predicate as "$\texttt{dt\_h([Col}_1\texttt{, ..., Col}_D\texttt{])} = v$", a generated composite predicate including all columns in a subset $I \subseteq \{1, \ldots, D\}$ can be derived as:*

$$\mathbf{x}_I : \bigvee_{j=1}^{J} \left(\bigwedge_{i \in I} c_{j,i} \right)$$

where J and $c_{j,i}$ are described in Theorem 2.

For decision tree, the generated atomic and composite predicates can be optimized further. Columns omitted in specific paths can be excluded as they do not contribute to pruning, and repeated conditions across paths need not be evaluated multiple times. For the larger decision trees, the generation time may increase. Its feasibility is maintained through optimization strategies such as consolidating conditions for the same feature across paths and amalgamating paths with identical labels during the preprocessing phase. Then, during real-time optimization, we can quickly check if all paths leading to the label share conditions on a particular feature before deriving the relevant predicates, which accelerates the process. Bitwise operations and modern compilers facilitate these optimizations. However, not all features in a decision tree can generate useful predicates. Only features that appear in all paths leading to the query label condition can produce meaningful predicates. Also, the predicate on a column or a column set needs to be the disjunction of all relevant paths, which will make the predicate filter less selective. For more complex models like random forests with thousands of trees, the generation time could become large, but its overhead is still negligible compared with the execution time. We leave optimizing the overhead for complex models as a future work.

Complex transformation. When a column undergoes transformations such as square, logarithmic, or exponential functions, we can derive the additional conditions of the transformed one by leveraging the min/max values of the original column. Then the necessary conditions can be further derived by exploiting the range of the transformed feature as in simpler models. And when generating atomic predicates for them, e.g., $log(x) < upper$, if the inverse function is straightforward to compute, the atomic predicate can be transformed to $x < exp(upper)$. If the inverse function cannot be explicitly derived, we can retain expressions like $x^x < upper$. This retention does not impose a significant computational overhead because x^x needs to be calculated even without our approach.

Supporting categorical values. Typically, categorical features are transformed into numerical ones using techniques like one-hot encoding. Our method's principles and pipeline apply directly to these encoded features, and the characteristics of one-hot encoding further tighten the generated SQL predicates.

On the one hand, since these encoded features originally come from the same column, they can be treated as multiple numerical columns and combined to form one composite predicate. Therefore, the generation process is similar to that for numerical columns. For instance, consider a dataset with two features: "Color" (a categorical feature) and "Size" (a numerical feature). After applying one-hot encoding to the "Color" feature, it is split into three binary features: "Color_Red", "Color_Green", and "Color_Blue", with corresponding weights of 2, 3, and 1, respectively. In our method, using min(Size) and max(Size) will derive a necessary condition about "Color", e.g. "2*Color_Red + 3*Color_Green + 1*Color_Blue < 1.5", which can be simplified as an atomic predicate "Color = Blue".

On the other hand, by considering the nature of one-hot encoding, we can generate tighter additional conditions of min/max ranges. In the same instance, when deriving an atomic predicate for "Size", the additional conditions are "$1 \leq$ Color ≤ 3", corresponding to the min and max weight values of "Color". These additional conditions are tighter than treating each color feature independently, where the conditions might range from min(Color) = 0 to max(Color) = 6 = 2+3+1.

3.4 Inference rewrite

To generate SQL predicates from a given ML predicate in the SQL+ML query, the inference rewrite module initially derives a set of atomic necessary conditions from Theorem 1(2) and refines them further through Algorithm 1. Each atomic necessary condition is then used to generate a corresponding atomic predicate. Finally, the inference rewrite module rewrites the query tree by appending the generated atomic predicates to the scan node.

At a certain point, the generation time could take more than actually executing the model. We can use the cost to predict the execution cost. If the execution cost is high, we can use our method to optimize the query; otherwise, we can directly execute the query. Note that as the data size grows, the execution time increases proportionally, while the generation time remains a one-time cost during the query optimization phase. This means that the relative impact of the generation overhead decreases with larger datasets, emphasizing the efficiency of our method in big data scenarios.

4 Progressive inference for composite predicates

An ML predicate may contain multiple tables, e.g., D columns from D tables, and any subset of these tables can generate a valid composite predicate[4]. Thus, the total number of valid composite SQL predicates grows exponentially, specifically $2^D - 1$, which includes D atomic predicates and $2^D - D - 1$ composite predicates.[5] It is rather expensive to utilize all composite predicates blindly because of (1) their exponential count, leading to increased execution overhead, and (2) pruning redundancy across different composite predicates, which results in redundant computational costs. Consequently, the selection of which composite predicates to retain becomes crucial in minimizing computation redundancy.

4.1 Composite predicate correlation

The composite predicates selection must adhere to three principles to minimize computation redundancy.
Principle 1. A composite predicate must be after joining the corresponding tables. The execution of a composite predicate must occur after joining all tables involved in the predicate. For example, a composite predicate "w_1*`A.room_num` + w_2*`B.building_age` < 10" must occur after joining tables `A` and `B`. In other words, within the query plan tree, a composite predicate must appear at higher nodes than the join nodes for tables.
Principle 2. A sub-predicate must appear before its super-predicate. Given two generated predicates P_1 and P_2 from the same ML predicate, we denote $P_1 \subseteq P_2$ (P_1 is a sub-predicate of P_2 and P_2 is a super-predicate of P_1), if each column in P_1 appears in P_2. For example, the composite predicate "w_1*`A.room_num` + w_2*`B.building_age` < 10" is a sub-predicate of the predicate "w_1*`A.room_num` + w_2*`B.building_age` + w_3*`C.hospital_num` < 14". Lemma 1 is provably true but omitted for its simplicity.

Lemma 1 *Given two predicates $P_1 \subseteq P_2$ derived from Theorem 1 and 3 for linear models or Theorem 2 and 4 for decision tree of the SQL+ML query Q, any tuple pruned by P_1 must be pruned by P_2.*

In this way, considering $P_1 \subset P_2$,[6] if P_1 appears at the same or higher node in the query plan tree than P_2, P_1 is redundant, as P_1 cannot prune more tuples than P_2. So the principle is predicate P_1 must appear before (lower in the query plan tree) P_2 if $P_1 \subset P_2$.
Principle 3. A super-predicate can reuse the result of its sub-predicate. Considering two predicates $P_1 \subset P_2$, if P_1 has already been computed in a former/deeper join operator, we do not need to compute P_2 from scratch. Instead, we can compute P_2 progressively by reusing the computed results of P_1. For example, computing "w_1* `A.room_num` + w_2* `B.building_age` + w_3* `C.hospital_num`" can reuse the computed results of "w_1* `A.room_num` + w_2* `B.building_age`".

4.2 Composite predicate selection

Composite predicate selection. Composite predicate selection aims to select high-quality composite predicates by removing the redundant predicates and satisfying the three principles. We develop a heuristic method to address the composite predicate selection problem, comprising four primary steps and Fig. 4 shows a running example.
Step 1: Generate a set of all valid composite predicates, denoted as $\mathcal{P}_{\text{all}}$, and insert them into the predicate list of the root node in the query plan tree.
Step 2: Push down each composite predicate in $\mathcal{P}_{\text{all}}$ as low as possible, adhering to **Principle 1**, with the objective of pruning the maximum intermediate tuples.
Step 3: At each node, among predicates with sub/super-predicate relationships, keep the super-predicates and remove the sub-predicates based on **Principle 2**, thereby allowing for the most effective tuple pruning.
Step 4: Progressively infer to minimize computation redundancy among the remaining predicates based on **Principle 3**. By adding the expression of sub-predicate into the target list[7], the super-predicate can reuse the previously result to infer each tuple progressively.
Algorithm analysis. Composite predicate selection selects a maximum of $D - 1$ composite predicates, where D is the number of tables in the ML predicate. It aims to prune as

[4] In scenarios where multiple columns are in the same table, we can introduce a new column as a composite of them to logically simplify the situation. When applied across multiple columns within the same table, these composite predicates can still be efficiently pushed down to the scan operator, enhancing data filtering before joins.

[5] We represent the composite predicate "w_1*`A.a` + w_2*`B.b` < 10" as "p(a,b)", with all candidate composite predicates listed in the set $\mathcal{P}_{\text{all}}$ in Fig. 4.

[6] One or more columns are in P_2 but are not in P_1.

[7] In the query plan tree, each node has a *target list* to represent its output columns.

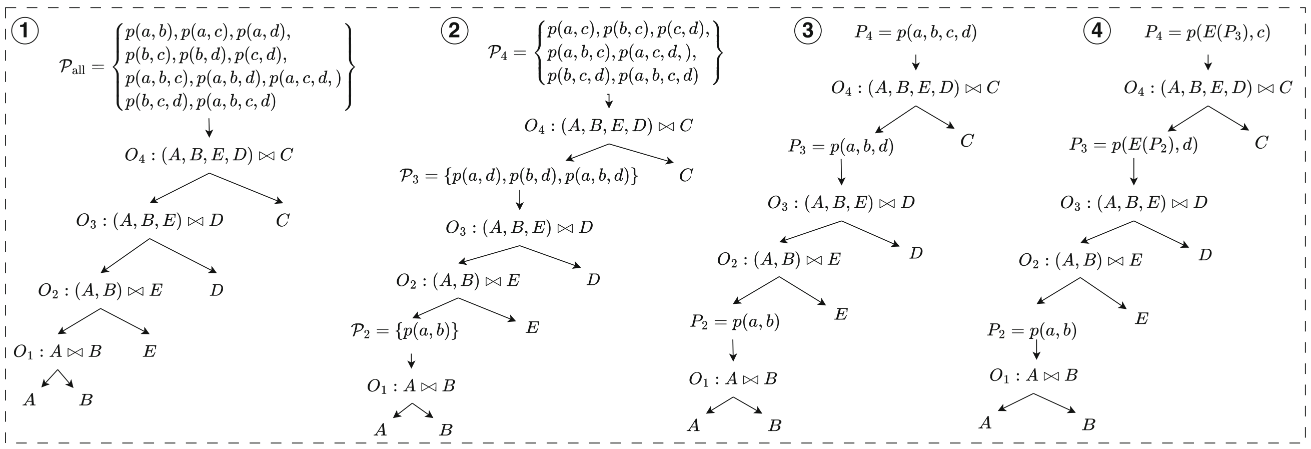

①
$\mathcal{P}_{\text{all}} = \{p(a,b), p(a,c), p(a,d), p(b,c), p(b,d), p(c,d), p(a,b,c), p(a,b,d), p(a,c,d,), p(b,c,d), p(a,b,c,d)\}$
$O_4 : (A,B,E,D) \bowtie C$
$O_3 : (A,B,E) \bowtie D$
$O_2 : (A,B) \bowtie E$
$O_1 : A \bowtie B$
②
$\mathcal{P}_4 = \{p(a,c), p(b,c), p(c,d), p(a,b,c), p(a,c,d,), p(b,c,d), p(a,b,c,d)\}$
$\mathcal{P}_3 = \{p(a,d), p(b,d), p(a,b,d)\}$
$\mathcal{P}_2 = \{p(a,b)\}$
③
$P_4 = p(a,b,c,d)$
$P_3 = p(a,b,d)$
$P_2 = p(a,b)$
④
$P_4 = p(E(P_3),c)$
$P_3 = p(E(P_2),d)$
$P_2 = p(a,b)$

Fig. 4 Example of the 4 Steps for composite predicate selection. *Step 1* ① adds all candidate predicate to the root O_4. *Step 2* ② pushes down each predicate to sets $\mathcal{P}_2$, $\mathcal{P}_3$, and $\mathcal{P}_4$. *Step 3* ③ selects the strongest predicates, P_3 and P_4, from each set based on Lemma 1, and removes the others. *Step 4* ④ reconstructs the selected predicates P_3 and P_4 using Principle 3 for progressive computation, where $E(P)$ denotes the expression of the reused result

many tuples as possible on the query plan tree while minimizing redundant computations among the generated composite predicates. Despite processing an exponential number of composite predicates with respect to D, Algorithm 2 merges generation and selection to optimize it.

The input of this reduction function CPS is the root node of the plan tree and an initially empty set, CPS($Root$, $\emptyset$). As the function executes, this set is populated with the selected composite predicates. Lines 4-5 generate the composite predicates with maximum pruning ability, and Lines 8-11 progressively infer ML predicates through passing up the intermediate results. The optimized algorithm yields a computational complexity of $\mathcal{O}(N \times D)$, where N represents the total number of operator nodes on the query plan tree.

Algorithm 2: Composite Predicate Selection CPS($Node$,$\mathcal{P}$)

```
Input: $Node$; $\mathcal{P}$ is the set of selected composite predicates.
Output: The set of tables $\mathcal{T} \subseteq \mathcal{R}$ below current node.
1  if $Node.type$ is $Scan$ then
       /* All tables in ML predicate comprise $\mathcal{R}$. */
2      $\mathcal{T} \leftarrow$ (If $Node$ scans table $t \in \mathcal{R}$ then $\{t\}$ else $\emptyset$).
3  else
       /* Gather tables in $\mathcal{R}$ below current node. */
4      $\mathcal{T} \leftarrow$ CPS($Node.left, \mathcal{P}$) $\cup$ CPS($Node.right, \mathcal{P}$);
5      $P \leftarrow$ $GeneratePredicate$(Cols in $\mathcal{T}$, ML predicate);
       /* Add the generated $P$ into the global $\mathcal{P}$. */
6      if $P \notin \mathcal{P}$ then
7          $\mathcal{P} \leftarrow \mathcal{P} \cup \{P\}$;
8          $E(P) \leftarrow ProgressivelyInfer(P, Node)$;
9          Append $E(P)$ into $Node.targets$;
10     else
11         Pass $Node.child.targets.E(P)$ to $Node.targets$;
12     end
13 end
14 return $\mathcal{T}$.
```

5 Cost-optimal inference for composite predicates placement

For each composite predicate, it can be placed in multiple positions in the query plan tree, i.e., from the lowest position (e.g., the highest table-join nodes for tables in composite predicate) to the highest position (e.g., the lowest position of its super-predicate). As an example, the predicate node P_1 in the left tree in Fig. 5 has two potential positions O_2 and O_3. Placing at different positions can lead to different costs, e.g., the computation cost of executing the predicate and the number of pruned tuples to save cost of the subsequent operators (executing higher operators in the query plan tree). Thus it is important to place each predicate at an appropriate position in the query plan tree in order to minimize the total execution cost.

Modern optimizers typically pushdown predicates as low as possible in the execution tree. This approach is based on the assumption that the calculation cost of SQL predicates is relatively low and can be ignored, therefore pushdown does not bring overhead and pruning a few data also brings benefits. This assumption is valid in many general traditional scenarios. However, in our method, the additional SQL predicates are generated from the ML predicate, which introduces a unique challenge. Firstly, the calculation costs of these predicates can be significant, particularly when they involve complex operations such as multiplication, exponential, or logarithm. Secondly, these predicates have overlapping or inclusive relationships in terms of the data they filter. Therefore, the placements of these relative SQL predicates impact the benefits of each other.

To address these challenges, we develop a novel cost model and placement strategy that optimize the positions of these additional SQL predicates based on their sub-super relations, relative calculation costs and position-aware benefits. While pushing predicates as low as possible in the execution plan tree is a common strategy, this approach could result in minimal data filtering but substantial computational overhead in some cases. In such cases, our cost-optimal placement strategy can pull up the predicates higher in the execution plan tree to effectively avoid unnecessary calculation overhead.

5.1 Cost model

We first focus on the cost of computing composite predicates, then other operators affected by them.

The cost of composite predicate. The cost of a predicate is determined by both (1) the computational cost for each tuple and (2) the total number of tuples to evaluate.

Firstly, in terms of the computational cost of a predicate for each tuple, a direct method is to evaluate the expression of the predicate itself. However, if some other sub-predicates are under the predicate, the progressive inference incrementally computes the predicate. Considering in Sect. 4.1, the calculation of predicate P_i can be computed incrementally as $Cost_P(P_i, (P_j))$ when P_j is the *closest predicate* to P_i.[8]

Definition 1 (Closest predicate) Let P_j be the closest predicate to P_i, if and only if (1) $P_j \subset P_i$ and both are derived from Theorem 1-4, (2) P_j is placed closest to P_i in the plan tree, which means no other generated predicates exist between them.

Secondly, in terms of the number of input tuples for a predicate, when a predicate P_i is located on a position O_k, the input rows of P_i are equal to the output rows from O_k. We denote the size of the output rows from the operator O_k as $Rows(O_k, (P_j))$, where P_j is the closest predicate to P_i.[9] The output rows from the operator O_k is affected by the presence of P_j, but it is independent of the location of P_j due to the previous insights, as well as independent of the presence and location of other predicates. However, computing this rows function requires an estimation of the selectivity of the predicates. We propose to obtain the size of rows in three steps:

1. Without any generated predicates in the plan tree, the size of the output rows of O_k is denoted as $Rows(O_k)$ and can be estimated by DB modules;
2. The selectivity of the generated predicate P_j is denoted as $Sel(P_j)$, which can be estimated based on the histograms of the relevant columns or more advanced methods [16];
3. Under the assumption of independence between operators, with P_j inserted below O_k, $Rows(O_k, (P_j)) = Rows(O_k) \times Sel(P_j)$.

Thus, when P_j is the closest predicate to P_i, the cost of P_i located on O_k is represented as:

$$Cost(P_i, (O_k, P_j)) = Rows(O_k, (P_j)) \times Cost_P(P_i, (P_j));$$

and if no predicate is below O_k, then

$$Cost(P_i, (O_k)) = Rows(O_k) \times Cost_P(P_i).$$

The cost of other operators. In `Smart`, the traditional DB module is able to estimate the cost of an operator O_k using the existing cost model and the size of the input rows, which is denoted as $Cost(O_k) = \mathcal{E}(O_k, (Rows(O_{k-1})))$. Introducing composite predicates below the operator affects its input rows, but only affected by the closest predicate from the above insights. When P_j is the closest predicate below O_k, the cost is:

$$Cost(O_k, (P_j)) = \mathcal{E}(O_k, (Rows(O_{k-1}, (P_j)))).$$

Impact of cardinality estimation We recognize that relying heavily on Cardinality Estimation (CE) to select among the 2^D possible predicates could lead to suboptimal join order decisions if CE is not accurate. To address this, our second technique adopts a progressive inference process, where composite predicates are chosen with a clear sub-super relationship between them. Our third technique optimizes the cost through place and merges selected predicates. This technique further mitigates the impact of inaccurate CE by requiring only relative accuracy rather than absolute precision. This relative accuracy can be reasonably achieved through existing database histograms, allowing our method to manage the risks associated with less precise CE effectively.

As a result, our method effectively manages the risks associated with less precise CE. Specifically, for atomic predicates with reliable CE, we push them down prior to determining the join order. This ensures that the join order chosen is an improvement over the original plan without

[8] The notation of $Cost(O, (\cdot))$ is consistent with DB optimizer's operator cost estimation.

[9] Note that if we only consider left-deep tree, there is only one P_j for P_i; if we consider bushy tree, there may be multiple closest predicates. Without loss of generality, we consider the left-deep tree for ease of representation.

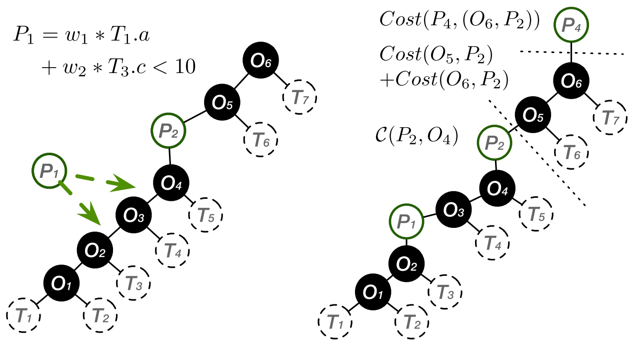


Fig. 5 The *left tree* shows an example of inserting a composite predicate. T nodes represent the scan nodes and O nodes represent other operators. P_1 includes columns from T_1 and T_3. $P_1 \subset P_2$ and P_2 is placed on O_4. The *right tree* demonstrates how $\mathcal{C}(P_4, O_6)$ accumulates costs from three components. P_2 is the closest predicate of P_4, so the first component is $\mathcal{C}(P_2, O_4)$. And the second component is the sum of the costs associated with operators O_5 and O_6 respectively. Then the third component is the cost of P_4 on the top

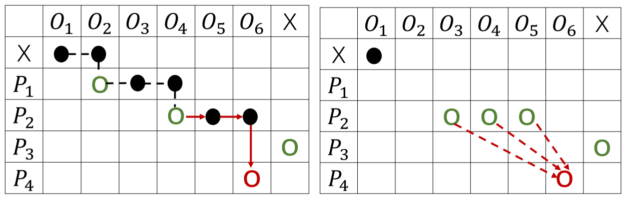


Fig. 6 The *left matrix* represents a placement P_1 on O_2, P_2 on O_4, and the absence of P_3; the three arrows correspond to accumulate components in Fig. 5 right tree. The *right matrix* shows the possible transitions to $d(P_4, O_6, P_2)$. The left matrix corresponds to the one possible transition from $d(P_2, O_4, P_1)$

predicates. For composite predicates where CE may be less accurate, we postpone the pushdown decision until after the join order is established. This helps avoid suboptimal join orders due to inaccurate CE. Even in worst-case scenarios, our optimization aims to avoid any significant overall degradation compared to the original plan without predicates. We acknowledge that accurately estimating the cardinality of composite predicates remains a challenge. As CE techniques improve, especially for composite predicates, we plan to incorporate these enhanced CE results directly into our join order selection process to further improve query performance.

5.2 Composite predicate placement

The optimization objective of composite predicate placement. We aim to place the composite predicates in the query plan tree to minimize the total execution cost. The above analysis leads to the conclusion that the cost of both the original operator and the inserted composite predicate depends on the closest composite predicate inserted under them. Therefore, we define a cost function for the (sub)plan tree with the inserted composite predicate serving as the root, and offer a recursive formula to compute the cost beginning from the subtree led by the closest predicate.

Cost Optimization in a Bottom-Up Manner. To reuse the computed results of a subtree, we propose a bottom-up manner to minimize the cost. Specifically, we note the cost function as $\mathcal{C}(P, (O))$ with two parameters, which represents the cost of the whole sub-plan tree with generated predicate P as the root node when the P is placed on the operator O. Therefore, our goal is to firstly minimize $\mathcal{C}(P_M, (O_N))$ where P_M includes all columns involved in the ML predicate and O_N is one of the possible positions of P_M. Based on these minimum values, the total execution costs of the whole plan tree can be calculated easily. And the set of composite predicates placements, corresponding to the smallest cost among the total costs, are the optimal output.

For two closest composite predicates $P_j \subset P_i$, where P_i is placed on O_k and P_j is placed on O_m, the total cost of plan tree rooted by P_i is the sum of (1) the subplan tree cost rooted by P_j, (2) the cost of the operators between P_j and P_i, and (3) the cost of P_i itself, as:

$$\mathcal{C}(P_i, (O_k)) = \mathcal{C}(P_j, (O_m)) + \mathcal{S}_{m,k} + Cost(P_i, (O_k, P_j))$$
$$\text{with, } \mathcal{S}_{m,k} := \textstyle\sum_{n \in (m,k]} Cost(O_n, (P_j))$$

where $n \in (m, k]$ represents traversal of O_n between O_m and O_k.[10]

The right tree in Fig. 5 represents an example. When $P_2 \subset P_4$, P_2 is placed on O_4 and P_4 is placed on O_6,

$$\mathcal{C}(P_4, (O_6)) = \mathcal{C}(P_2, (O_4)) + \mathcal{S}_{4,6} + Cost(P_4, (O_6, P_2))$$

This illustrates the overlapping cost calculation process within the subtrees of the whole query plan tree. Based on this, we can construct the optimal substructure for the overall cost of the optimized tree. We define the function $d(P_i, O_k, P_j)$ as the minimum $\mathcal{C}(P_i, (O_k))$ where P_j is the closest predicate to P_i. Then the **transition function** is:

$$\begin{aligned} d(P_i, O_k, P_j) &= \min_{m \in [b_j, k)} \{\mathcal{C}(P_j, (O_m)) + \mathcal{S}_{m,k}\} + Cost(P_i, (O_k, P_j)) \\ &= \min_{m \in [b_j, k), y \in [0, j)} \{d(P_j, O_m, P_y) + \mathcal{S}_{m,k}\} \\ &\quad + Cost(P_i, (O_k, P_j)) \end{aligned}$$

where O_{b_j} is the lowest position of P_j adhering to **Principle 1**. Specifically, we use $d(P_i, O_k, P_0)$ to denote the scenario where no composite predicate exists below O_k during initial-

[10] For right leaf nodes, static costs like sequential scans are ignored, while linear changes, as in index scans, are added to join node costs.

Algorithm 3: Composite Predicate Placement

```
Input: Operators $\mathcal{O}$, selected composite predicates $\mathcal{P}$.
Output: Optimal positions $\mathcal{L}$ for composite predicates in $\mathcal{P}$.
1  Initialize $D[M][N][M] \leftarrow$ MAX_INT;
2  for $i \leftarrow 1$ to $M$, $k \leftarrow 1$ to $N$ do
3      $D[i][k][0] \leftarrow d(P_i, O_k, P_0)$, compute with Eq (3);
4  end
5  for $i \leftarrow 1$ to $M$, $k \leftarrow 1$ to $N$ do
6      for $j \leftarrow 1$ to $i-1$ do
7          $cur\_min \leftarrow$ MAX_INT;
8          $back\_track \leftarrow (O_0, P_0)$;
9          for $m \leftarrow b_j$ to $k-1$, $y \leftarrow 0$ to $j-1$ do
10             $g(m, y) \leftarrow D[j][m][y] + \mathcal{S}_{m,k}$;
11             if $g(m, y) < cur\_min$ then
12                 $cur\_min \leftarrow g(m, y)$;
13                 $back\_track \leftarrow (O_m, P_y)$;
14             end
15         end
16         $D[i][j][k] \leftarrow Cost(P_i, (O_j, P_k)) + cur\_min$;
17     end
18 end
19 Back tracking $\min_{k\in[0,M)}\{D[M][N][k]\}$ to fill in $\mathcal{L}$.
```

ization,

$$d(P_i, O_k, P_0) = \sum_{z\in[1,k]} Cost(O_z) + Cost(P_i, (O_k)) \quad (3)$$

We iterate through both the closest predicate ranging from P_0 to P_{i-1} and the position of P_i. This allows us to determine the minimum costs for all possible scenarios, from which we then return the smallest value.

The right matrix in Fig. 6 gives an example for the transition of $d(P_4, O_6, P_2)$ using the state transition function: There are three choices for m and two choices for y, $d(P_2, O_3/O_4/O_5, P_0/P_1)$, if P_2 involves columns from T_1, T_3, and T_4. Then, the transition is determined by computing the minimum one.

Composite predicate placement. In Algorithm 3, we employ dynamic programming to optimize the placement of the composite predicates. It first initializes the storage and the boundary values (Lines 1-4). The core transition phase adheres to the transition function (Lines 5-18). Back tracking in Line 17 using Line 13 yields the optimal positions for the composite predicates.

Complexity analysis. Although Algorithm 3 exhibits a computational complexity of $O(M^3N^3)$, methodologies for its reduction are feasible. By pre-computing $S(n) = \sum_{x=1}^{n} Cost(O_x)$, the value of any $\mathcal{S}_{m,k} = (S(k) - S(m)) \times Sel(j)$ can be efficiently computed in constant time. Furthermore, by utilizing additional storage space to eliminate the variable y, the complexity can be further reduced to $O(M^2N^2)$.

6 Experimental results

This section presents an experimental study to evaluate our system `Smart`.

Experimental environment. `Smart` was integrated into PostgreSQL 14.4 [2]. The experiments were conducted on a server machine s6.xlarge provided by HuaweiCloud. The server had 8GB of RAM, 4 v-cores of CPU, and an HDD Disk, running on Ubuntu 18.04. We utilized the stable version of MADlib 1.18 [1], which is compatible with PostgreSQL 12.2.[11] We also employed Scikit-Learn 1.2.2 [29] with ConnectorX 0.2.3 [46].

System integration. Database administrators (DBAs) can activate the proposed three ML modules individually by adding new configuration parameters to the DBMS, which is as simple as allowing Nested Loop with enable_nestedloop. The SQL+ML optimization utility relies on two system tables: *sys_feature* and *sys_model*. ML engineers are responsible for managing the content of *sys_model*. Once a new ML model is trained, the ML engineer organizes its name, category, and weights/paths, and inserts them into *sys_model*. Data analysts manage the content of *sys_feature*. If a column has potential for inference, its name and table name are inserted into *sys_feature*. The update rule for the statistical information of each feature follows the standard catalog update strategies of columns commonly used in DBMS.
sys_model[model_name, model_category, weights/paths]
sys_feature[attribute_name, table_name, min, max]

6.1 SQL+ML benchmarks and baselines

To facilitate a fair comparison of the SQL+ML query processing efficiency of `Smart` with existing baselines, we have designed five SQL+ML benchmark kits. These benchmark kits were derived from well-known benchmarks, namely JOB [31, 39], TPC-H [40, 44], and SSB [11]. The final benchmark kit was designed based on a real-world scenario. The SQL+ML query sets include linear regression, logistic regression, and decision tree models. Table 1 provides an overview of the benchmark kits.

6.1.1 Benchmarks

Datasets. From Table 1, the benchmarks include two real-world datasets: IMDB [18] and Flight [21]. Additionally, two synthetic datasets were generated from TPC-H [40] and SSB [11]. The IMDB dataset is widely used in both DB and ML fields. The Flight dataset is derived from airport and flight information. The TPC-H and SSB datasets are generated using the data generation procedures in the corresponding

[11] We use PostgreSQL 12.2 instead of 14.4 because MADlib supports PostgreSQL up to 12.

Table 1 Statistics of datasets, model information, and query set characteristics for 5 benchmarks

Dataset	Data size	#rel	Distribution	ML model	Inferred value	#feature	Workload	#query	#table
IMDB	5.1 GB	21	Real-world	linear	Rating	4	JOB	113	4-17
TPC-H	0.1-30 GB	8	Synthetic	Linear	Price	5	TPC-H	22	6-12
SSB	10 GB	5	Synthetic	Logistic	Chosen or not	4	SSB	13	5-6
Flight	0.5 GB	4	Real-world	Logistic	Is delay	5	Delay	10	4
IMDB	5.1 GB	21	Real-world	Decision tree	Gross level	6	JOB-DT	113	6-19

benchmark kits, and they have the same structure as the original datasets. In particular, the TPC-H dataset scales from 100MB to 30GB, allowing us to explore the scalability of `Smart` in various scenarios.

ML model training. The experiments are based on linear models and decision tree, and the number of features/columns in ML predicate is shown in Table 1 as `#feature`. For the JOB and TPC-H benchmarks, we trained linear regression models. For the SSB and Flight benchmarks, we trained logistic regression models. Also, we trained decision tree for JOB. After loading all datasets into the database, we use the state-of-the-art machine learning library MADlib to train the models directly. By keeping data management and model training entirely within the database, we avoid operations like data movement and format conversion, which enhances the reproducibility of our experiments. Features and labels were selected from multiple tables connected within each schema. Without applying feature selection or normalization, our method is robust and not overly sensitive to specific column choices. Feature engineering and model tuning were intentionally not performed, as they are outside the scope of this work. In the real-world datasets, the inferred column for the JOB dataset represents the movie rating value and gross level, while for the Flight dataset, it indicates the flight delay situation. In the synthetically generated datasets, the TPC-H inferred column represents the numeric price value, and the SSB inferred column is a manually added Boolean variable that has logical relationship with the features. We evaluated our trained models using two real-world datasets. We apply an ML predicate of “inferred_rating > 8.0” on IMDB. The results show a precision of 86.8%, recall of 61.6%, and accuracy of 97.3%. We apply an ML predicate of “inferred_delay = FALSE” on Flight. The results show a precision of 80.2%, recall of 94.6%, and accuracy of 82.4%. These metrics confirm that our models are reasonable. The synthetic datasets TPCH and SSB allow us to demonstrate the broad applicability of our optimization method across diverse query types and data characteristics. Although the inference accuracy is relatively lower on synthetic datasets, they are crucial for evaluating inference efficiency, which is the primary focus of this work.

(a) An SQL+ML query example of benchmark `Flight`

```
SELECT  flights.id, is_delay
FROM    flights, airlines, airports AS start_ap, airports AS dest_ap
WHERE   flights.airline = airlines.iata_code
   AND  flights.origin_airport = start_ap.iata_code
   AND  flights.destination_airport = dest_ap.iata_code
   AND  lr_flight([flights.distance, flights.dep_delay, airlines.airlineid,
          start_ap.latitude, dest_ap.longitude]) = False AS is_delay;
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Model name:     lr_flight
Model category: logistic regression
Model weights:  w = [1.2779, 0.0002, -0.1575, 0.021, -0.0015, 0.005]
Inference Function: g(x) = FALSE,  if w*x < 0
```

(b) An SQL+ML query example of benchmark `IMDB`

```
SELECT  t.title AS movie_title, inferred_rating
FROM    keyword AS k, movie_info AS mi, movie_keyword AS mk,
        title AS t, mi_votes, mi_budget, mi_gross
WHERE   k.keyword LIKE '%sequel%'  AND mi.info IN ('Bulgaria')
   AND  t.production_year > 2010     AND  . . --(omit join conditions)
   AND  lr_rating([t.production_year, mi_votes.votes,
        mi_budget.budget, mi_gross.gross]) > 8.0  AS inferred_rating;
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Model name:     lr_rating
Model category: linear regression
Model weights:  w = [-0.009, -6.0e-4, 2.2e-7, 0.0211, -0.0035]
Inference Function: f(x) = w*x
```

(c) A UDF query example of benchmark `IMDB`

```
CREATE FUNCTION infer_rating (x_1 numeric, x_2 numeric,
                              x_3 numeric, x_4 numeric)
RETURNS numeric AS $$
    SELECT -0.009 - 6.0e-4*x_1 + 2.2e-7*x_2 + 0.0211*x_3 - 0.0035*x_4;
$$ LANGUAGE SQL;
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
SELECT  t.title AS movie_title, inferred_rating
FROM    keyword AS k, movie_info AS mi, movie_keyword AS mk,
        title AS t, mi_votes, mi_budget, mi_gross
WHERE   k.keyword LIKE '%sequel%'  AND mi.info IN ('Bulgaria')
   AND  t.production_year > 2010     AND  . . --(omit join conditions)
   AND  infer_rating(t.production_year, mi_votes.votes,
        mi_budget.budget, mi_gross.gross) > 8.0  AS inferred_rating;
```

Fig. 7 Two SQL+ML example cases and a UDF example case. UDF is identical to inlined inequality, due to the pull-up strategy of PostgreSQL. Model parameters are embedded in the function, but `Smart` manages them in the system table

Workload. Each SQL+ML query template was adapted from the corresponding original SQL query template in JOB, TPC-H, and SSB benchmarks. The adaptations involved two parts: appending relational tables with features to the `FROM` clauses and inserting ML predicates with inference functions into the `WHERE` clauses. As an example in Fig. 7, in the query set of the JOB benchmark, each query selects movies based on their “title" column. We added rating inference for each selected movie and filtered the inferred ratings within a specific range, such as “rating $>$ 8.0." The design of the query sets maintains the multi-table join requirements of the original JOB queries. Similarly, the TPC-H and SSB queries include the ML predicates before the aggregation requirements. As the Flight dataset does not include pre-existing queries, we designed 10 queries that consider the flight delay under the influence of multiple factors, including 4-way joins. Figure 7 shows an example.

Metric. To measure the efficiency, we utilized the execution time obtained from the “Explain Analyze" tool in PostgreSQL. We executed each SQL+ML query three times and reported the shortest execution time to mitigate the impact of cache memory. The generation and optimization overhead was included in the reported end-to-end times. The additional planning time added by `Smart` is minor. For example, PostgreSQL takes 3006 ms to generate the query plans for 113 JOB queries, while `Smart` requires only 41 ms in addition.

6.1.2 Baselines

PostgreSQL 12 + MADlib. MADlib [1] is a popular in-DB ML library that provides a user-friendly interface for inference functions with well-trained models stored in DB. When using MADlib to process SQL+ML queries, we modify each query by adding the model table to the "FROM" clause and replacing the inference function in ML predicates with the MADlib function.

PostgreSQL14 + UDF. Inference functions can be created with UDFs, aligning with the database design philosophy. Additionally, the utility of UDFs can only be minimally optimized by the DB optimizer, such as through pushdown. Unlike MADlib, creating UDFs is an offline process that does not impose the requirement for users to design SQL+ML query templates.

PostgreSQL 14 + ConnectorX + Scikit-Learn. Scikit-Learn [29] is a popular ML library in Python used for traditional ML algorithms. ConnectorX [46] is a state-of-the-art data transformation channel that facilitates the transformation of relational data from DBMS to ML systems. Hence, we integrate these platforms with PostgreSQL 14.4 to implement the DB+ML strategy. First, PostgreSQL retrieves tuples using joins and selections, and then utilizes ConnectorX to fetch the tuples. ConnectorX enables us to handle result tuples without exporting them to external storage or reloading them. Instead, it manages the result tuples in memory, and controls the data transport to Scikit-Learn for ML predicates processing. However, this baseline has limited processing capability. If ML predicate is not the final operator in the query plan, the inferred results need to be transformed back into relational data and loaded into the DBMS. Unfortunately, ConnectorX is currently unable to handle this for the DB+ML strategy.

Table 2 End-to-end comparison of `Smart` and baselines

Workload	DB+ML	MADlib	UDF	`Smart`
JOB(113)	19634.8 s	3642.4 s	1384.0 s	309.7 s
TPCH(22)	—	4782.3 s	2591.6 s	1521.7 s
SSB(13)	—	2800.6 s	1619.7 s	665.9 s
Flight(10)	466.6 s	228.1 s	185.6 s	67.6 s
DT(113)	—	705.1 s	1140.0 s	144.3 s

6.2 End-to-end comparison

To compare the efficiency of `Smart` with the three mentioned baselines, we conducted a set of experiments using above benchmarks. The queries within each benchmark's query set are ordered by query ID, then sequentially executed using the four different methods. The total execution time, which is the sum of the individual execution times of all queries in each benchmark, is then reported. The server and DB configurations are consistent across all methods.

The experimental results are presented in Table 2. When comparing the total execution time horizontally, `Smart` consistently exhibits the highest efficiency. For JOB, `Smart` outperforms all three baselines by 67.6×, 12.5×, and 7.4×, respectively. For TPC-H and SSB, `Smart` provides speedups of 3.1× and 4.2× over MADlib, and 1.7× and 2.4× over UDF, respectively. In the case of the Flight dataset, `Smart` achieves speedups of 6.9×, 3.4×, and 2.7× compared to the baselines. Lastly, for JOB-DT, `Smart` a performance boost of 4.9× over MADlib and 7.9× over UDF.

Despite ConnectorX reducing the overhead of transferring data from a relational data model to a dataframe model, the overall latency still remains relatively high. This set of experiments demonstrates that extending the entire ML pipeline and using an additional processing flow to complete the ML predicates is disadvantageous. It is preferable to avoid exporting data from the DBMS and instead integrate the data processing pipeline inside DBMS.

By installing the MADlib library in Postgres for ML functions, SQL+ML query processing can be performed inside DBMS, eliminating the need for data copying. However, MADlib still has its own limitations. Firstly, in order to perform the inference, the DBMS needs to access the integrated interface of MADlib and uses the returned value from

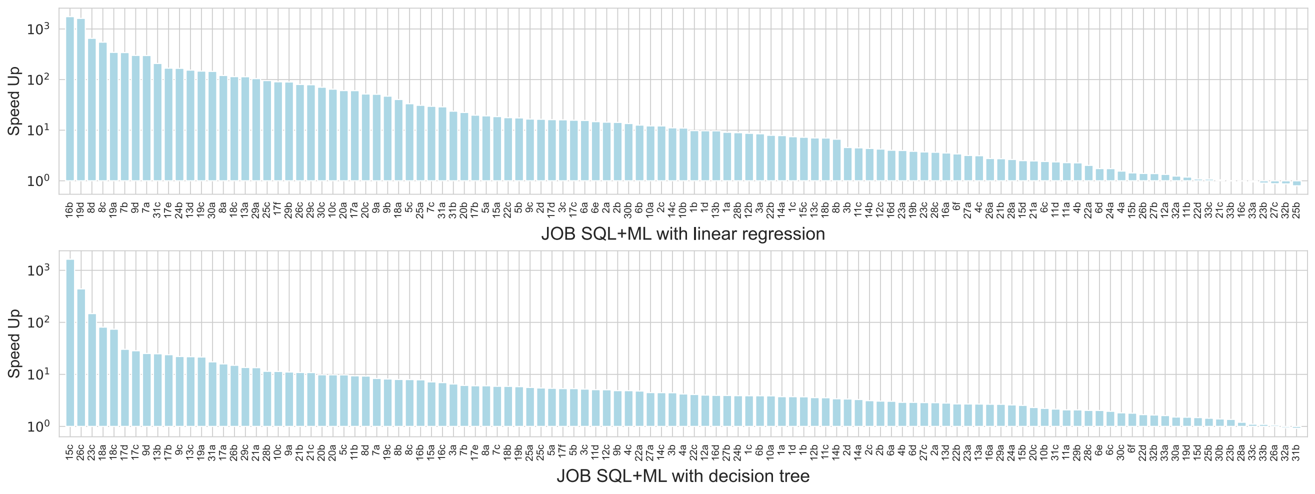


Fig. 8 Query to query speed up of `Smart` on JOB and JOB-DT

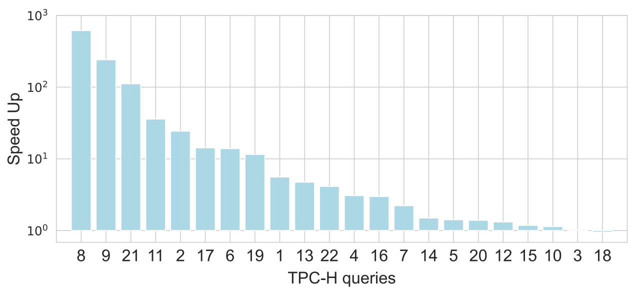


Fig. 9 Query to query speed up of `Smart` on TPC-H

the function, which introduces additional overhead. Furthermore, potential type conversions may occur during this process, resulting in additional execution time. Therefore, when using UDF to declare inference in SQL+ML queries, the execution is faster compared to using MADlib.

By moving the ML process from outside the database to inside the database, we gain the opportunity to utilize specially designed algorithms for optimizing computation. In `Smart`, we exploit the necessary conditions for ML predicates to achieve early data pruning. This involves excluding tuples that will not yield selected results from the pipeline as early as possible during the join process. This approach significantly accelerates the SQL+ML queries.

6.3 Query-by-query comparison

To assess the effectiveness of `Smart` for individual queries, we compare the execution times of each query in the JOB, JOB-DT, and TPC-H experiments for MADlib and `Smart`. The results are presented in Figs. 8 and 9, which show the number of queries that can benefit from `Smart` across multiple query templates.

We executed queries in JOB and TPC-H separately with MADlib and `Smart`. For JOB-DT queries, we use UDF and `Smart`, as MADlib lacks a decision tree inference interface for temporal/CTE tables in SQL+ML. We recorded the execution time using the aforementioned approach. The speed-up ratio is calculated as $\text{SpeedUp} = T_{\text{baseline}}/T_{\texttt{Smart}}$, where T_{baseline} is the total execution time with MADlib or UDF and $T_{\texttt{Smart}}$ is the total execution time with `Smart`.

Out of the 113 evaluated queries in JOB, the fastest query demonstrated an impressive speed-up of over 1000×, and more than half of the queries achieved a speed-up of over 10×. In the JOB-DT set, one query achieved a 1000× speed-up, and 23 queries exhibited speed-ups exceeding 10×. In the TPC-H experiment, among the 22 SQL+ML queries in the TPC-H dataset, the performance of 21 queries improved. Notably, 8 queries achieving a speed-up greater than 10×, and 3 queries experienced a speed-up exceeding 100×. Query 25b and 32b in JOB experienced a slight slowdown, but the impact was negligible. This can be attributed to the influence of various factors on the PostgreSQL query optimizer when generating query plans. As a result of inference rewrite, queries with added predicates are pushed down to the scan operators of the query plan tree by DB optimizer. This leads to substantial changes in the overall query plan, and then the cardinality estimation by DB optimizer, due to its relative inaccuracy, might result in a plan with a slower execution speed.

The majority of queries experienced improved performance after applying `Smart`. In these queries, the ML predicates filtered out a significant number of tuples, resulting in smaller output tuple results. As a result of this characteristic, most of the incompletely joined tuples in `Smart` are pruned by generated SQL predicates before being fully joined. Furthermore, some queries produced empty

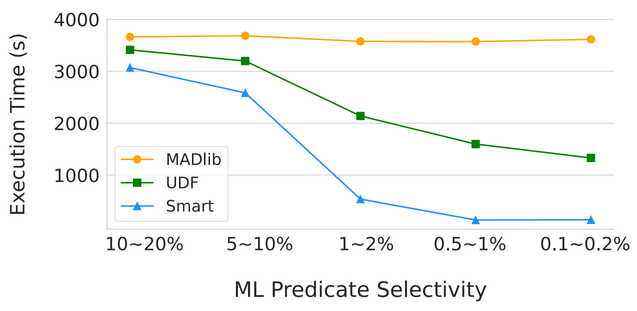


Fig. 10 Effects of ML predicate selectivity

Table 3 Execution time (ms) of some JOB queries to show the performance of atomic predicates and composite predicates

JOB query	7c	13b	14c	28b	30c
UDF	2067.0	805.9	749.7	7060.0	1283.5
Atomic preds	797.2	512.9	630.0	2049.8	823.5
Atomic+comp	238.6	389.7	318.2	934.2	297.7

results and inference rewrite prunes many tuples, thus significantly improving the performance.

Even on uniformly generated data and SQL+ML queries adapted from the TPC-H query set, `Smart` can accelerate queries by minimizing the number of intermediate results during joins. Owing to the substantial differences in table sizes within the dataset and the complexity of the queries, the inference rewrite in this set of experiments did not result in significant changes to the query plan tree, and as a result, no queries exhibited a noticeable performance decline. This also indirectly corroborates the validity of our observations from the JOB experiments.

From an end-to-end and query-wise experimental standpoint, `Smart` effectively optimizes the majority of SQL+ML queries, leading to significant performance improvements. Excluding the occasional suboptimal query plans caused by imprecise cardinality estimation, `Smart` consistently delivers speed-ups from 1$\times$ to 1000$\times$.

6.4 Selectivity of ML predicate

The optimization efficacy of `Smart` is significantly influenced by the selectivity of ML predicates in SQL+ML. We conducted experiments using three different methods on JOB to explore the trends in execution time with varying selectivity. We maintained the same hardware and PostgreSQL configuration while modifying the ML predicates in SQL+ML to obtain various selectivity values.

The results are presented in Fig. 10. The horizontal axis represents different ranges of the selectivity of ML predicates. As the selectivity of ML predicate decreases, the execution time of the MADlib group exhibits minor fluctuations within a narrow range. This behavior arises because the inference inequality is consistently positioned at the top of the query plan tree, and changes in ML predicate do not significantly impact the main portion of the query execution. Conversely, the UDF strategy demonstrates a slight decline in outcomes as the ML predicates are pushed down, resulting in the processing of fewer tuples by above operators when the ML predicate selectivity decreases. The execution time of `Smart` decreases significantly as the selectivity of ML predicate decreases, eventually approaching zero when the label selectivity approaches zero. This phenomenon is closely tied to the utilization of the ML predicate. In MADlib, the ML predicate is solely employed as a comparison value, and altering the ML predicate does not affect the number of tuples to be processed. The UDF strategy exhibits similar behavior, but operators above the ML predicates benefit from reduced row processing when the selectivity of ML predicate decreases. In contrast, `Smart` fully capitalizes on the ML predicate by introducing a series of SQL predicates in the query plan tree to expedite query execution. Consequently, as the label selectivity decreases, `Smart` becomes increasingly efficient, resulting in reduced execution time. In real-world scenarios, users often require query results within specific ranges rather than retrieving all the tuples. This characteristic aligns with the performance advantage of `Smart`.

As selectivity increases, the optimization effect gradually diminishes, and the performance approaches that of the baseline. However, our method does not introduce additional overhead at higher selectivities such as above 20%. When selectivity is high, most of the data meets the ML predicate conditions. This leaves little opportunity for data pruning before the join, resulting in a smaller optimization. The third technique proposed in our paper, the cost-optimal placement strategy, addresses this challenge by dynamically adjusting the positions of atomic and composite predicates. Based on the cost model, if any additional SQL predicate has less pruning effect but introduces overhead, it will be moved up in the execution plan tree until merged with the super predicate to avoid redundant calculations. For instance, when selectivity reaches 100%, all generated SQL predicates will disappear since they are moved up and merged into the original ML predicate. Then, the final execution plan tree generated by our method is identical to the original plan tree, ensuring no additional overhead.

6.5 Ablation study

To examine the individual impact of two kinds of predicates and three proposed optimization techniques on query efficiency, we conducted ablation experiments on the JOB. We evaluate the importance of both atomic predicates and composite predicates, with the comparison results of some queries listed in Table 3. Then, to evaluate the three proposed

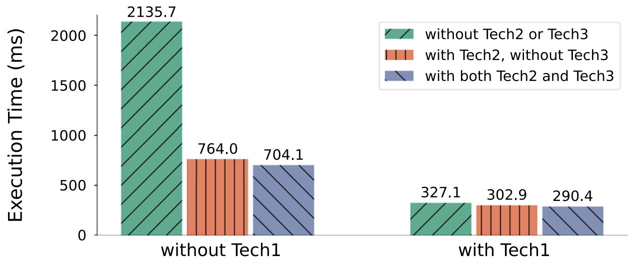


Fig. 11 Ablation study of `Smart` with JOB

techniques, we split the experiments into two groups depending on whether inference rewrite was used. Each group includes three scenarios: without using progressive inference and cost-optimal inference, using progressive inference without cost-optimal inference, and utilizing both of them. We report the total running time of the SQL+ML query set on JOB. The analyzed results are presented in Fig. 11.

The findings from both Table 3 and Fig. 11 reveal that atomic predicates employed by inference rewrite plays a significant role in enhancing overall query performance. However, the performance improvement resulting from composite predicates from progressive inference and cost-optimal inference becomes less pronounced after the application of atomic predicates from inference rewrite. This can be attributed to the fact that atomic predicates and composite predicates contain tuples prune overlapping from Fig. 11. Meanwhile, as shown in Table 3, composite predicates are vital that they further reduce the execution time significantly. It is noteworthy that inference rewrite has the potential to introduce changes to the plan tree structure due to its impact on the query optimizer. These changes can have either positive or negative effects on query performance. In contrast, progressive inference and cost-optimal inference do not influence the structure.

Overall, our ablation experiments demonstrate that atomic predicates with inference rewrite bring noticeable performance improvements, and the composite predicates with progressive inference and cost-optimal inference can further enhance performance.

6.6 Scalability

To investigate the trends of benefits when I/O cost becomes the bottleneck factor, we varied the sizes of the dataset and executed the SQL+ML workloads of TPC-H. The results are depicted in Fig. 12, where the orange line represents the baseline running the UDF strategy, and the blue line represents `Smart` with inference rewrite and progressive inference only. The horizontal axis represents the dataset size, ranging from 100MB to 30GB. The vertical axis represents the execution time of each SQL+ML query with significant I/O overhead, measured in seconds.

Our findings indicate that approximately two-third of the queries exhibited a significant performance improvement, while the remaining one-third did not demonstrate much improvement.

As the data set size increased, `Smart` effectively reduced the number of tuples in the intermediate results through inference rewrite and progressive inference thereby decreasing the need for subsequent operators, e.g. `index_scan`, and consequently significantly reducing I/O overhead. This is reflected by the shallow slope of the blue line in the first row of Fig. 12. Conversely, in queries where inference rewrite and progressive inference did not provide effective tuples pruning, the slope of the blue line in the third rows of Fig. 12 resembled that of the orange line.

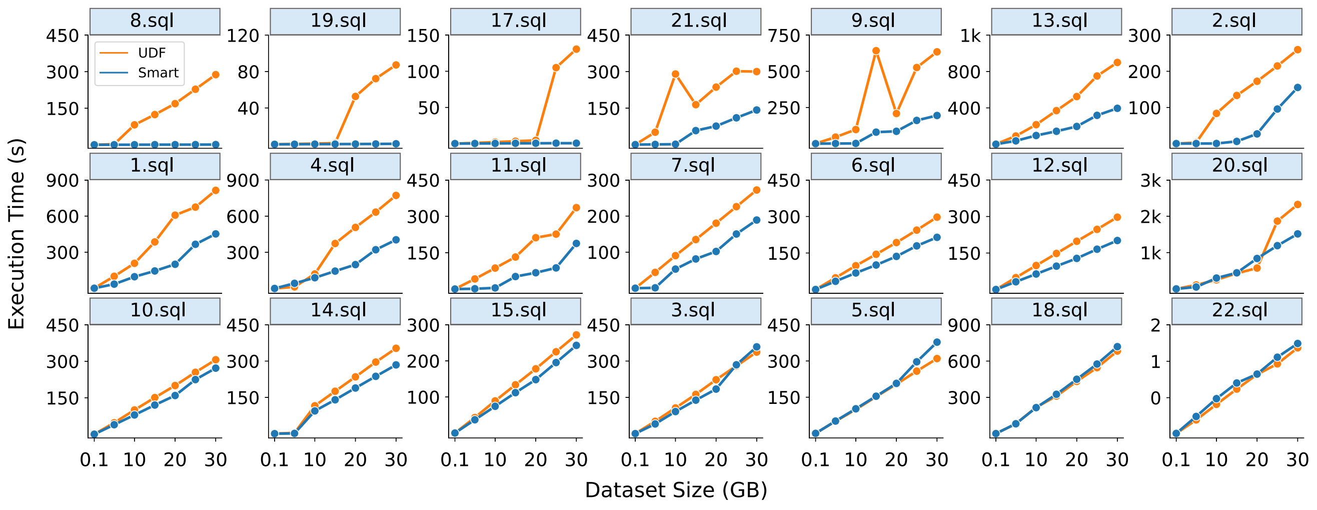


Fig. 12 Scalability on TPC-H. Queries are ordered by the speedup

In summary, Smart demonstrated comparable or superior scalability performance compared to the baseline, outperforming the baseline in most cases.

7 Related works

In-DB ML inference. Inference (or prediction) is becoming an official statement of extended SQL in many DBMS products, such as Redshift [5], Big Query ML [13], SQL Server ML [36], MADlib [9, 12, 15, 45, 50] on PostgreSQL, and Spark MLlib [35, 42]. Recently, openGauss [32, 37, 52] designs *predict by* statement as well, to support in-DB ML inference. However, the in-DB optimization for inference predicate has not been widely studied.
Inference on ML platform. The bottleneck comes from ML inference [19] when processing SQL+ML. Clipper [10], Willump [25, 26] and PRETZEL [30] provide inference optimization methods for ML systems, but none of them works for DBMS queries with inference predicates. Raven [23] designs an intermediate representation for prediction queries, then separates operators into DB part and ML part with serious cross-optimizations [38]. The DB parts are compiled as a new SQL to process by DBMS, and the ML parts are executed with ML systems. For unstructured data like images and videos, PP [34], CORE [49] and FiGO [6] optimize inferred labels in video DBMS specifically. Among them, PP generates simpler predicates from models as proxy models to rewrite queries. A recent work, ConnectorX [46] accelerates the data loading from DBMS to ML systems. However, how to transfer inferred results back to DBMS to support complex applications is omitted. Thus, existing DB+ML strategies suffer from data copy and low performance, and the optimization techniques can not adapt to in-DB inference.
In-DB ML training. In-DB training acceleration has been studied [14, 48]. Lara [28] and LaraDB [17], LevelHeaded [3], and Froid [41] explored how to co-optimize linear algebra and relational algebra for model training. FAQ-AI [24] represents model training tasks as additive inequalities in SQL, then rewrites the complex input query into multiple simple sub-queries. However, they cannot be used for inference [7] because during training, complete computations are performed iteratively for each tuple, while inference computes only once and can terminate early based on query conditions.

8 Conclusion

We proposed Smart, an in-DB query optimization for SQL+ML queries with ML predicates. The main idea is employing SQL predicates to prune irrelevant tuples. To this end, Smart firstly generates tight and valid SQL predicates for logical optimization. Then Smart selects proper SQL predicates to prune irrelevant tuples. Moreover, Smart uses a cost model to optimize the query plan with selected SQL predicates. We implement Smart into PostgreSQL, which outperforms baselines on all four benchmarks.

Acknowledgements This paper was supported by National Key R&D Program of China (2023YFB4503600), National Science and Technology Major Project of China (CJGJZD20230724093403007), NSF of China (61925205, 62232009, 62102215), Zhongguancun Lab, Huawei, and Beijing National Research Center for Information Science and Technology (BNRist). Guoliang Li is the corresponding author.

References

1. 1.18.0, M.: Madlib 1.18.0 documentation (2023). https://madlib.apache.org/docs/v1.18.0/index.html
2. 14.4, P.: Postgresql 14.4 release notes (2023). https://www.postgresql.org/docs/release/14.4/
3. Aberger, C.R., Lamb, A., Olukotun, K., Ré, C.: Levelheaded: A unified engine for business intelligence and linear algebra querying. In: 34th IEEE international conference on data engineering, ICDE 2018, Paris, France, April 16-19 pp. 449–460. IEEE Computer Society (2018). https://doi.org/10.1109/ICDE.2018.00048
4. Agrawal, A., Chatterjee, R., Curino, C., Floratou, A., Godwal, N., Interlandi, M., Jindal, A., Karanasos, K., Krishnan, S., Kroth, B., Leeka, J., Park, K., Patel, H., Poppe, O., Psallidas, F., Ramakrishnan, R., Roy, A., Saur, K., Sen, R., Weimer, M., Wright, T., Zhu, Y.: Cloudy with high chance of DBMS: a 10-year prediction for enterprise-grade ML. In: 10th conference on innovative data systems research, CIDR 2020, Amsterdam, The Netherlands, January 12-15 Online Proceedings. www.cidrdb.org (2020). http://cidrdb.org/cidr2020/papers/p8-agrawal-cidr20.pdf
5. Amazon.com: Redshift ml (2023). https://aws.amazon.com/redshift/features/redshift-ml/
6. Cao, J., Sarkar, K., Hadidi, R., Arulraj, J., Kim, H.: Figo: Fine-grained query optimization in video analytics. In: Ives, Z.G., Bonifati, A., Abbadi, A.E., (eds.) SIGMOD '22: International conference on management of data, Philadelphia, PA, USA, June 12 - 17 pp. 559–572. ACM (2022). https://doi.org/10.1145/3514221.3517857
7. Chai, C., Wang, J., Luo, Y., Niu, Z., Li, G.: Data management for machine learning: a survey. IEEE Trans. Knowl. Data Eng. **35**(5), 4646–4667 (2023). https://doi.org/10.1109/TKDE.2022.3148237
8. Chen, L., Kumar, A., Naughton, J.F., Patel, J.M.: Towards linear algebra over normalized data. Proc. VLDB Endow. **10**(11), 1214–1225 (2017)
9. Cohen, J., Dolan, B., Dunlap, M., Hellerstein, J.M., Welton, C.: MAD skills: new analysis practices for big data. Proc. VLDB Endow. **2**(2), 1481–1492 (2009). https://doi.org/10.14778/1687553.1687576
10. Crankshaw, D., Wang, X., Zhou, G., Franklin, M.J., Gonzalez, J.E., Stoica, I.: Clipper: A low-latency online prediction serving system. In: Akella, A., Howell, J., (eds.) 14th USENIX symposium on networked systems design and implementation, NSDI 2017, Boston, MA, USA, March 27-29 pp. 613–627. USENIX Association (2017). https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw
11. eyalroz: Ssb kit (2023). https://github.com/eyalroz/ssb-dbgen
12. Feng, X., Kumar, A., Recht, B., Ré, C.: Towards a unified architecture for in-rdbms analytics. In: Candan, K.S., Chen, Y., Snodgrass, R.T., Gravano, L., Fuxman, A., (eds.) Proceedings of the ACM SIGMOD international conference on management of data, SIG-

MOD 2012, Scottsdale, AZ, USA, May 20-24 pp. 325–336. ACM (2012). https://doi.org/10.1145/2213836.2213874
13. Google: Big query ml (2023). https://cloud.google.com/bigquery/docs/inference-overview
14. Guo, Y., Zhang, Z., Jiang, J., Wu, W., Zhang, C., Cui, B., Li, J.: Model averaging in distributed machine learning: a case study with apache spark. VLDB J. **30**(4), 693–712 (2021). https://doi.org/10.1007/s00778-021-00664-7
15. Hellerstein, J.M., Ré, C., Schoppmann, F., Wang, D.Z., Fratkin, E., Gorajek, A., Ng, K.S., Welton, C., Feng, X., Li, K., Kumar, A.: The madlib analytics library or MAD skills, the SQL. Proc. VLDB Endow. **5**(12), 1700–1711 (2012). https://doi.org/10.14778/2367502.2367510
16. Hu, X., Liu, Y., Xiu, H., Agarwal, P.K., Panigrahi, D., Roy, S., Yang, J.: Selectivity functions of range queries are learnable. In: Ives, Z.G., Bonifati, A., Abbadi, A.E., (eds.) SIGMOD '22: international conference on management of data, Philadelphia, PA, USA, June 12 - 17 pp. 959–972. ACM (2022). https://doi.org/10.1145/3514221.3517896
17. Hutchison, D., Howe, B., Suciu, D.: Laradb: A minimalist kernel for linear and relational algebra computation. In: Afrati, F.N., Sroka, J., (eds.) Proceedings of the 4th ACM SIGMOD workshop on algorithms and systems for mapreduce and beyond, BeyondMR@SIGMOD 2017, Chicago, IL, USA, May 19 pp. 2:1–2:10. ACM (2017). https://doi.org/10.1145/3070607.3070608
18. IMDB: The csv files of imdb dataset (2023). http://homepages.cwi.nl/~boncz/job/imdb.tgz
19. Jassy, A.: Aws re:invent 2018 keynote (2023). https://www.youtube.com/watch?v=ZOIkOnW640A&t=5316s
20. Jr., J.F.R., Pépin, J.L., Goeuriot, L., Amer-Yahia, S.: An extensive investigation of machine learning techniques for sleep apnea screening. In: d'Aquin, M., Dietze, S., Hauff, C., Curry, E., Cudré-Mauroux, P., (eds.) CIKM '20: The 29th ACM international conference on information and knowledge management, virtual event, Ireland, October 19-23 pp. 2709–2716. ACM (2020). https://doi.org/10.1145/3340531.3412686
21. Kaggle: Flight dataset (2023). https://www.kaggle.com/usdot/flight-delays
22. Kaggle: State of data science and machine learning 2021 (2023). https://www.kaggle.com/kaggle-survey-2021
23. Karanasos, K., Interlandi, M., Psallidas, F., Sen, R., Park, K., Popivanov, I., Xin, D., Nakandala, S., Krishnan, S., Weimer, M., Yu, Y., Ramakrishnan, R., Curino, C.: extending relational query processing with ML inference. In: 10th conference on innovative data systems research, CIDR 2020, Amsterdam, The Netherlands, January 12-15 Online Proceedings. www.cidrdb.org (2020). http://cidrdb.org/cidr2020/papers/p24-karanasos-cidr20.pdf
24. Khamis, M.A., Curtin, R.R., Moseley, B., Ngo, H.Q., Nguyen, X., Olteanu, D., Schleich, M.: Functional aggregate queries with additive inequalities. ACM Trans. Database Syst. **45**(4), 17:1-17:41 (2020). https://doi.org/10.1145/3426865
25. Kraft, P., Kang, D., Narayanan, D., Palkar, S., Bailis, P., Zaharia, M.: A demonstration of willump: a statistically-aware end-to-end optimizer for machine learning inference. Proc. VLDB Endow. **13**(12), 2833–2836 (2020). https://doi.org/10.14778/3415478.3415487
26. Kraft, P., Kang, D., Narayanan, D., Palkar, S., Bailis, P., Zaharia, M.: Willump: A statistically-aware end-to-end optimizer for machine learning inference. In: Dhillon, I.S., Papailiopoulos, D.S., Sze, V., (eds.) Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4. mlsys.org (2020). https://proceedings.mlsys.org/book/297.pdf
27. Kumar, A., Naughton, J.F., Patel, J.M.: Learning generalized linear models over normalized data. In: Sellis, T.K., Davidson, S.B., Ives, Z.G., (eds.) Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4 pp. 1969–1984. ACM (2015). https://doi.org/10.1145/2723372.2723713
28. Kunft, A., Katsifodimos, A., Schelter, S., Breß, S., Rabl, T., Markl, V.: An intermediate representation for optimizing machine learning pipelines. Proc. VLDB Endow. **12**(11), 1553–1567 (2019). https://doi.org/10.14778/3342263.3342633
29. scikit learn: scikit-learn 1.2.2 (2023). https://scikit-learn.org/1.2/
30. Lee, Y., Scolari, A., Chun, B., Santambrogio, M.D., Weimer, M., Interlandi, M.: PRETZEL: opening the black box of machine learning prediction serving systems. In: Arpaci-Dusseau, A.C., Voelker, G., (eds.) 13th USENIX symposium on operating systems design and implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10 pp. 611–626. USENIX Association (2018). https://www.usenix.org/conference/osdi18/presentation/lee
31. Leis, V., Gubichev, A., Mirchev, A., Boncz, P.A., Kemper, A., Neumann, T.: How good are query optimizers, really? Proc. VLDB Endow. **9**(3), 204–215 (2015). https://doi.org/10.14778/2850583.2850594
32. Li, G., Zhou, X., Sun, J., Yu, X., Han, Y., Jin, L., Li, W., Wang, T., Li, S.: opengauss: an autonomous database system. Proc. VLDB Endow. **14**(12), 3028–3041 (2021). https://doi.org/10.14778/3476311.3476380
33. Li, S., Chen, L., Kumar, A.: Enabling and optimizing non-linear feature interactions in factorized linear algebra. In: Boncz, P.A., Manegold, S., Ailamaki, A., Deshpande, A., Kraska, T., (eds.) Proceedings of the 2019 international conference on management of data, SIGMOD Conference, Amsterdam, The Netherlands, June 30 - July 5 pp. 1571–1588. ACM (2019). https://doi.org/10.1145/3299869.3319878
34. Lu, Y., Chowdhery, A., Kandula, S., Chaudhuri, S.: Accelerating machine learning inference with probabilistic predicates. In: Das, G., Jermaine, C.M., Bernstein, P.A., (eds.) Proceedings of the 2018 international conference on management of data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15 pp. 1493–1508. ACM (2018). https://doi.org/10.1145/3183713.3183751
35. Meng, X., Bradley, J.K., Yavuz, B., Sparks, E.R., Venkataraman, S., Liu, D., Freeman, J., Tsai, D.B., Amde, M., Owen, S., Xin, D., Xin, R., Franklin, M.J., Zadeh, R., Zaharia, M., Talwalkar, A.: Mllib Machine learning in apache spark. J. Mach. Learn. Res. **17**(34), 1–7 (2016)
36. Microsoft: Sql server ml (2023). https://learn.microsoft.com/en-us/sql/t-sql/queries/predict-transact-sql?view=sql-server-ver16
37. openGauss: Native db4ai engine (2023). https://docs.opengauss.org/en/docs/5.0.0/docs/AIFeatureGuide/native-db4ai-engine.html
38. Park, K., Saur, K., Banda, D., Sen, R., Interlandi, M., Karanasos, K.: End-to-end optimization of machine learning prediction queries. In: Ives, Z.G., Bonifati, A., Abbadi, A.E., (eds.) SIGMOD '22: international conference on management of data, Philadelphia, PA, USA, June 12 – 17 pp. 587–601. ACM (2022). https://doi.org/10.1145/3514221.3526141
39. Rahn, G.: Job kit (2023). https://github.com/gregrahn/join-order-benchmark
40. Rahn, G.: Tpc-h kit (2023). https://github.com/gregrahn/tpch-kit
41. Ramachandra, K., Park, K., Emani, K.V., Halverson, A., Galindo-Legaria, C.A., Cunningham, C.: Froid: optimization of imperative programs in a relational database. Proc. VLDB Endow. **11**(4), 432–444 (2017). https://doi.org/10.1145/3186728.3164140
42. Spark, A.: Mllib (2023). https://spark.apache.org/docs/latest/ml-pipeline.html#estimators
43. SQLFlow: Sqlflow: extends sql to support ai (2023). https://sql-machine-learning.github.io
44. TPC: Tpc-h (2023). https://www.tpc.org/tpch/default5.asp
45. Wang, D.Z., Franklin, M.J., Garofalakis, M.N., Hellerstein, J.M., Wick, M.L.: Hybrid in-database inference for declarative information extraction. In: Sellis, T.K., Miller, R.J., Kementsietsidis,

A., Velegrakis, Y., (eds.) In Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12–16 pp. 517–528. ACM (2011). https://doi.org/10.1145/1989323.1989378

46. Wang, X., Wu, W., Wu, J., Chen, Y., Zrymiak, N., Qu, C., Flokas, L., Chow, G., Wang, J., Wang, T., Wu, E., Zhou, Q.: Connectorx: accelerating data loading from databases to dataframes. Proc. VLDB Endow. **15**(11), 2994–3003 (2022)
47. Wu, Y., Lentz, M., Zhuo, D., Lu, Y.: Serving and optimizing machine learning workflows on heterogeneous infrastructures. Proc. VLDB Endow. **16**(3), 406–419 (2022)
48. Xu, L., Qiu, S., Yuan, B., Jiang, J., Renggli, C., Gan, S., Kara, K., Li, G., Liu, J., Wu, W., Ye, J., Zhang, C.: In-database machine learning with corgipile: stochastic gradient descent without full data shuffle. In: Ives, Z.G., Bonifati, A., Abbadi, A.E., (eds.) SIGMOD '22: international conference on management of data, Philadelphia, PA, USA, June 12 - -17, pp. 1286–1300. ACM (2022). https://doi.org/10.1145/3514221.3526150
49. Yang, Z., Wang, Z., Huang, Y., Lu, Y., Li, C., Wang, X.S.: Optimizing machine learning inference queries with correlative proxy models. Proc. VLDB Endow. **15**(10), 2032–2044 (2022)
50. Zhang, Y., Kumar, A., McQuillan, F., Jayaram, N., Kak, N., Khanna, E., Kislal, O., Valdano, D.: Tech report of distributed deep learning on data systems: a comparative analysis of approaches. Tech. rep., Pivotal (now VMware) (2021). https://adalabucsd.github.io/papers/TR_2021_Cerebro-DS.pdf
51. Zhou, X., Chai, C., Li, G., Sun, J.: Database meets artificial intelligence: a survey. IEEE Trans. Knowl. Data Eng. **34**(3), 1096–1116 (2022). https://doi.org/10.1109/TKDE.2020.2994641
52. Zhou, X., Jin, L., Sun, J., Zhao, X., Yu, X., Li, S., Wang, T., Li, K., Liu, L.: Dbmind: a self-driving platform in opengauss. Proc. VLDB Endow. **14**(12), 2743–2746 (2021). https://doi.org/10.14778/3476311.3476334