GaussML: An End-to-End In-Database Machine Learning System

Guoliang Li†, Ji Sun†, Lijie Xu‡, Shifu Li‡, Jiang Wang‡, Wen Nie‡
†Tsinghua University ‡Huawei Company §ETH Zürich ¶Institute of Software, Chinese Academy of Sciences
liguoliang@tsinghua.edu.cn, {sunji11,niewen2,lishifu,wangjiang16}@huawei.com, lijie.xu@inf.ethz.ch

Abstract—In-database machine learning (In-DB ML) is appealing to database users with security and privacy concerns, as it avoids copying data out of the database to a separate machine learning system. The common way to implement in-DB ML is the ML-as-UDF approach, which utilizes the User-Defined Functions (UDFs) within SQL to implement the ML training and prediction. However, UDFs may introduce security risks with vulnerable code, and suffer from performance problems, as constrained by data access and execution patterns of SQL query operators.

To address these limitations, we propose a new in-database machine learning system, namely GaussML, which provides an end-to-end machine-learning ability with native SQL interface. To support ML training/inference within SQL query, GaussML directly integrates typical ML operators into the query engine without UDFs. GaussML also introduces an ML-aware cardinality and cost estimator to optimize the SQL+ML query plan. Moreover, GaussML leverages Single Instruction Multiple Data (SIMD) and data prefetching techniques to accelerate the ML operators for training. We have implemented a series of algorithms inside GaussML in openGauss database. Compared to the state-of-the-art in-DB ML systems like Apache MADlib, our GaussML achieves 2–6× speed-up in extensive experiments.

I. INTRODUCTION

Machine learning (ML) is now widely used for data analysis tasks. Researchers and engineers invest substantial effort into designing user-friendly machine learning interface [1], [2], constructing end-to-end machine learning pipeline [3]–[6], accelerating the model training [7]–[9], managing training data effectively [10]–[12] and developing end-to-end machine learning systems [13]–[17]. However, most of them provide Python interface that requires the extraction of training data from data storage systems, typically databases.

Nowadays, relational databases like openGauss [18] has been widely used in many large commercial businesses, including government cloud and financial services. For these businesses, the data stored in databases is one of the most valuable assets, and thus copying data out of databases for ML training leads to high risk of data leakage. Even in the same company/organization, the data infrastructure department and the application department are separated, and the data visit is strictly controlled for data security reasons. If the application team (e.g., the AI team) wants to train models, what they get is just a small sample of data from the far past. For training on fresh full dataset, a promising solution is in-database machine learning, where the database natively supports ML training and inference. In this way, customers can also leverage SQL interface for end-to-end ML tasks, which is easier for them to query data and perform ML training than writing complex Python programs.

In-database machine learning has been studied for many years [19], [20]. The most common approach is ML-as-UDF that utilizes User-Defined Functions (UDFs) within SQL to implement the computation of model training and prediction. For instance, the state-of-the-art in-DB ML tool, Apache MADlib, leverages PostgreSQL's UDFs to execute Stochastic Gradient Descent (SGD), while utilizing additional Python drivers for handling ML iterations. However, UDF-based approaches suffer from two problems. (1) Security risks: UDFs can introduce vulnerable code on data read from database to outsized. (2) Efficiency limitations: UDF-based solutions are constrained by the data access patterns and execution patterns of SQL query operators. As a result, they currently only support standard SGD and lack support for more efficient methods like mini-batch SGD because it's would be inefficient by using naïve data scan in SQL. Furthermore, UDF-based solutions cannot achieve comprehensive optimization in conjunction with query plan because it cannot touch the plan generation process.

To address these problems, we propose GaussML, which is a fully in-database machine learning system, and all the components are seamlessly integrated with native database kernel. It has three advantages. First, GaussML designs native executors to avoid data transfer, which can reduce the risk of data leakage and accelerate the data analysis, without introducing vulnerable code like UDFs. Second, GaussML can make co-optimization on traditional query execution path and machine learning operators, which can also improve the performance of end-to-end machine learning. Third, GaussML contains specific optimizations for ML operators like SIMD and data prefetching, further improving the performance.

In summary, we make the following contributions.

1) We propose a new in-database machine learning system GaussML, which supports efficient end-to-end machine learning using SQL queries. We seamlessly integrate GaussML into an open-source database openGauss 1.

2) We design an ML-aware cardinality and cost estimator in GaussML, which extends the ability of database optimizer to support complex SQL queries with machine learning (see Section III).

1https://gitee.com/opengauss-db4ai/openGauss-server
(3) We summarize the common computation patterns of widely-used ML algorithms, and then we develop native operators to accelerate model training, which can be organized as a node in the native query plan tree in database. Operations are accelerated by SIMD and data prefetching techniques, and they support distributed computing (see Section IV).

(4) We conduct extensive experiments to compare with state-of-the-art machine learning systems including MADlib and ML-A (an ML engine implemented with popular Python library), and the results show that our system outperforms existing methods by 2-6 times (see Section V).

II. System Overview

We first define the end-to-end machine learning (ML) process in Section II-A, and then introduce the architecture of GaussML from a perspective of database to show how ML is seamlessly integrated in relational database in Section II-B. Finally we introduce the native MLSQL grammar for using GaussML in Section II-C.

A. End-to-end Machine Learning

Given a relational database \( D \) with tables \( \{t_1, t_2, t_3, \ldots, t_n\} \) and a complicated data analytic problem with both database operators and ML algorithms \( P \). The steps of an end-to-end ML pipeline for solving the problem is as follows. (i) Feature engineering: creating a training view \( V \) from \( D \) including transformed features related to the problem \( P \); (ii) Model training: training a model from view \( V \) for the target on problem \( P \); (iii) Model inference: searching the proper model and predicting the labels (categories or numeric) by using it for given tuples; (iv) Data analysis: fetching and analysing tuples satisfying some constraints on model prediction results. We next show three typical scenarios for better understanding the benefits of conducting end-to-end machine learning within database.

Scenario 1: Data analytic makes a lot of efforts on preprocessing the dataset for easier training and better representation (i.e. feature engineering). In this process, data manipulation operations (e.g. join, projection, aggregation) and data preprocessing operations (e.g. normalization) are often used. The former operations can be optimized by optimizer in database, and the later operations can utilize the native data statistics in database. On the contrary, these operations would spend a large amount of time (even more than model training itself) in other ML systems. Moreover, as the whole process is conducted inside the database, data access is controlled by permission subsystem in database. GaussML is more secure and trusted when training on sensitive data in core business.

Similarly, in the inference phase, users can directly obtain the predicted targets from database without touching the model or the original data. This avoids the data transition overhead and guarantees the security of original data.

Scenario 2: It’s very common for real users to do the data clustering and the features are splitted on different relations, and they must get the result containing all features before training. For example, users are training K-means on data, they can write a SQL of creating model from subquery which joins all tables. In GaussML, if the distance is \( L_1 \) and \( L_2 \), we can also factorize the distance calculates on different tables, and push down to the scan node. For \( 1:n \) or \( n:n \) joins, GaussML will significantly reduce the training overhead with ML-DB co-optimization methods.

Scenario 3: In this scenario, the model predicts result are taken as the constraints for selecting desired tuples above the scan node. For example, if we want to select patients with anxiety, we should filter scanned tuples by using a well-trained anxiety models. Moreover, if we merge data from different data sources with constraints, the database will push down the predicates for better performance. As figure 1 shows, we train three models on individual tables \( t_1, t_2 \) and \( t_3 \), and then we predict the labels on new tables \( t_1', t_2' \) and \( t_3' \). In traditional relational database, tables are often joined on primary-foreign keys and take the larger side as outer table. However, GaussML should also consider the predicting cost because the cost differences of models cannot be neglected. During the process, GaussML can offer the optimal execution path by using advanced model-aware cost estimator and plan selector. Moreover, GaussML computes ML-based constraints on the fly to avoid schema change and storage overhead, and reduce the risk of information leakage.

B. GaussML Architecture

As Figure 2 shows, GaussML is composed of five major components, and they offer full-fledged machine learning ability inside relational database (openGauss in this paper).

MLSQL Parser. In this layer, GaussML extends SQL to support MLSQL by seamlessly integrating machine learning operations into SQL. It supports model training by using create model ... with ..., and model inference by using expression predict by .... The detailed usage will be introduced in Section II-C. Note that GaussML supports PBE (i.e. parser-bind-execute) protocol which is a lazy parameter binding approach to execute queries, SQLs with the same template are parsed only once.

MLSQL Optimizer. In this layer, GaussML supports optimization on MLSQL to handle both scenario 1 and scenario 2. GaussML optimizer can not only use model-aware cost estimator to find the optimal data visit path and model prediction orders, it also conducts interleaving optimizations between data visit operators and model training operators. We also design a brand-new in-database cardinality estimation component customized for machine learning operations in GaussML. The details of the optimizer will be introduced in Section III.

MLSQL Executor. In this layer, we define a set of ML operators to support high performance ML execution. GaussML supports over 20 popular machine learning algorithms, and the training executors of these algorithms are composed of 4 basic operators, including matrix computation operator, statistic operator, gradient descent operator, and distance computation operators. To increase the parallelism and reduce the latency of model training, we also develop parallel training
operators for each ML algorithm, including parallel training on single instance and distributed training on multiple nodes. We also utilize auxiliary operators to optimize model training process. For example, shuffle operator is used for accelerating convergence of stochastic gradient descent, materialized operator is used for reducing IO overhead of iterative training. GaussML also designs a set of data preprocessing operators, and these operators optimize data normalization and missing data imputation processes by leveraging native statistics in database. The detailed design and implementation of these operators are introduced in Section IV.

Training Data Management. We develop two new components in this layer to support ML executions. Firstly, we add the data redistribution operators in GaussML, which shuffle data across execution nodes to balance data labels and accelerate model convergence. Secondly, we design a new data structure RVD (Resilient Versioned Data) as training data of each model by using materialized view or lazy computation, so that users can trace the training data used by current model. We introduce this part in [21].

C. GaussML SQL

In this section, we formally introduce the extended MSQL of GaussML in Table I. For training operation, model_name sets the name of model in database after training, and model_type sets the algorithm of model (e.g. linear model, SVM), and the subquery with attribute list can select features and labels from data. The training data can be a materialized view, a subquery or a base table. Users can also offer a set of hyperparameters with knob AUTOML on, and GaussML will automatically select the optimal hyperparameters by using hyperparameter optimization (HPO) techniques (see Section IV-A). For predicting operation, the model_name in the expression designates trained model, and the projection transforms the predicted data.

III. QUERY PLANNING

Query planner is a vital component for end-to-end ML because it can effectively schedule both the ML and traditional
WHERE A=1 AND B=2 AND p1='anxiety' AND p2='cancer';
FROM T
SELECT PREDICT BY M2 FEATURES B, C, D as p2
WHERE T1.key=T2.key AND p1='anxiety'
FROM T1, T2
(p2='cancer')
SEQSCAN (A=1 and B=2)
PREDICT BY M1
(p1='anxiety')
PREDICT BY M2
(p2='cancer')

Fig. 3: Candidate plans for SGD model create.

Fig. 4: Order of model predict for scan node.

operators. However, traditional optimizer in database cannot handle end-to-end ML scenarios for four reasons [22].

(i) In model creating phase, the query planner should be able to mount ML operators on the plan tree, and make the performance of model training optimal.

(ii) Traditional query planner does not reorder the execution order of traditional predicates on one relation based on cost because the performance gain is little.

(iii) Traditional cardinality estimator cannot estimate the conjunctive predicates with both ordinary conditions and model prediction because database cannot catch correlations between model labels and data in other columns.

(iv) Database doesn’t have cost model for model prediction expressions. For example, traditional optimizer cannot tell the prediction cost difference between XGBoost and decision tree.

In this section, we introduce the query planner designed for end-to-end ML in GaussML. We first introduce the rules of query planning for model creating of different types of models in Section III-A, and then we show the workflow of query planning of hybrid DML query containing ML predictions in Section III-B, and next we propose the design of cardinality estimators in Section III-C, and finally we propose cost models for machine learning expressions in Section III-D.

A. ML-DB hybrid DDL (create) query planning

In Table II, we summarize four types of ML models we support in GaussML.

Firstly, stochastic gradient descent is a general model training method, and it can support large amount of data and any derivable model and loss function. In GaussML, if the data size is smaller than maximal process memory, then we materialize all the data in memory and then do the shuffle and iterative training. Otherwise, we shuffle the data blocks on disk first, and then read and materialize each block, and train in batches. These two possible plans are shown in Figure 3.

Secondly, statistic-based models generally need only a small sample of data, and go through it for several times to optimize the accuracy of prediction. The sample data can be placed in the memory, and materialize is not necessary for this training.

Thirdly, distance-based model need to compute the distance on input data iteratively when training, and thus the plan of creating such models is composed of materialize and distance operators. If the input data comes from subquery with joins, the distance computation (L1, L2) can be divided into two operators, partial distance operator and distance merge operator to reduce join overhead. As Figure 1 shows, partial distance operator can be pushed down to scan node, and for each features in the tuple of each relation, we compute the sum (for L1) or square sum (for L2) of element-wise distances between the data and given parameters (e.g., centroids for KMeans). The distance merge operator sums the distance of join results.

Fourthly, in order to pursue the best performance and deterministic model, for a small part of data, GaussML uses analytic methods (e.g., convex optimization) to solve the loss minimal problem. For this line of methods, training contains data materialize and matrix computing.

B. ML-DB hybrid DML query planning

We integrate query optimizer of GaussML into traditional optimizer in database. For each ML models, GaussML first estimates the input cardinality and computes the cost of predicting the model, and then GaussML selects an execution plan with lower execution cost. Figures 4 and 5 show two important scenarios of ML plan optimization.

In Figure 4, GaussML is able to reorder model prediction expressions above scan node according to the total cost. The original sequence scan node fetches tuples from table, and tuples are first filtered by constraint A = 1\& B = 2 because it's much cheaper than model prediction, then the left tuples are sent to next filters. The next two nodes filter results predicted by model M1 and M2 (i.e. finding patients with both anxiety and cancer in this example). The optimizer computes the total
Fig. 6: Steps of selectivity estimation for ML predicates.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>Predict(M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

The hardest part is how to estimate the selectivity of each ML predicate before actually executing it, because the features attributes are usually high dimensional and correlated, and the model prediction results are not materialized in the table.

**Problem Definition.** Selectivity is the number of tuples selected by predicates divided by number of table tuples. Selectivity estimation is vital to the optimizer in traditional database. When machine learning prediction operator is involved in GaussML, the conjunctive predicates would be more complex (refer to figure 6 as an example). We can formally define the problem as follows.

\[
    \text{sel} = \arg \min_s errf \left( s - \sigma_{1 \leq i \leq n}(op_i(f_{i,1}, f_{i,2}, \ldots, f_{i,n}))(T) \right)
\]

where \( errf \) is an arbitrary error metrics to measure the error between estimated cardinality and true cardinality. \( op \) can be any value comparing operators (\( >, <, = \), etc.), and \( f_i \) can be either a native function or ML model prediction operator. \( T \) is a table.

**Methodology.** There are two challenges for estimating such conjunctive predicates in figure 6, one is the statistics of labels produced model cannot be read directly, the other is random sampling is hard to get conditional distribution of data falling into the predicates (model predict result is correlated to other conditions). In order to estimate the selectivity precisely, GaussML progressively samples values from ordinary predicates. Several generative data distribution models are proposed to sample data in existing works [23], [24], but they have some drawbacks for ML scenarios. We propose a cardinality estimation framework to integrate the model into GaussML, and overcome these existing drawbacks.

As Figure 6 shows, step 1 builds model on original table, steps 2 & 3 are online estimation for ML queries. In step 1, we first analyze correlations between columns on data samples. We compute correlation degree as

\[
    \text{corr}(T.A, T.B) = \sum_{r \in T} P(r.A, r.B) \log \frac{P(r.A, r.B)}{P(r.A)P(r.B)}
\]

Then we group correlated columns together and select joint distribution model within each group according to the dependency topological. For example, we train a Bayesian Network for columns A, B and C such that they satisfy conditional independence assumption, and train Autoregressive model for columns D, E and F because they are correlated each other. We adopt attribute independence assumption to estimate the selectivity between different groups.

In step 2, we sample \( n \) rows of data from predicates on feature columns progressively and compute probability of each row (denoted as \( P(\text{row}_i) \)) as Figure 6 shows. In step 3, for each sample row, we obtain the result by model predicting and estimate the target distribution in samples. We denote ML predicate as \( M(\text{row}_i) \), and the selectivity of conjunctive predicates can be calculated as below.

\[
    \widetilde{\text{sel}} = \frac{\sum_{i=1}^{n} I(M(\text{row}_i) = \text{true})P(\text{row}_i)}{\sum_{i=1}^{n} P(\text{row}_i)}
\]
D. Cost estimation for ML inferences.

In order to select a better ML model prediction order, we need a cost model to estimate the prediction time. Cost formulas are mathematical expressions of the computation time complexity of model predictions, and they are related to 3 metrics: dimension of features for ML model (denoted as \(D\)), the estimated cardinality of precedent execution nodes (denoted as \(\text{card}\)) and model structures. Table II shows cost models for typical models we support in GaussML. The cost of linear, logistic, polynomial and SVM model is proportional to dimension of features and input cardinality. SVM with square kernel will map \(D\)-dimension features into \((D^2 + D + 1)\)-dimension. SVM with Gaussian kernel will map \(D\)-dimension features into infinite dimensions, and we use \(D' = \max(128, 2 \times D)\) centroids to compute Gaussian distances to approximate the mapping. The cost of decision tree is proportional to the depth of the tree, and the cost of ensemble models like Xgboost and random forest is proportional to the depth multiplying number of estimators. In Bayes models, we need to find the distribution from a sorted list, and the cost is proportional to \(\log(N_d)\) where \(N_d\) is the distinct value number. The naive \(k\)-NN relies on data sorting and the cost is proportional to \(N^2\), and the KNN with a pre-constructed index (HNSW) only need \(O(\log(N))\) time complexity for each query, where \(N\) is the data rows number. In each cost model, \(\alpha\) is a coefficient which can be tuned manually or automatically.

<table>
<thead>
<tr>
<th>Category</th>
<th>Training Pipeline</th>
<th>ML Models</th>
<th>Inference Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gradient-based Training</td>
<td>Shuffle (\rightarrow) [Materialize (\rightarrow) Matrix (\rightarrow) SGD]*</td>
<td>Linear model, Logistic model, Polynomial model, Support vector machine (SVM), SVM with square kernel, SVM with Gaussian kernel, Multi-layer perceptron (MLP), Principal Components Analysis (PCA)</td>
<td>(\alpha \cdot D \cdot \text{card})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\alpha \cdot \log(N_d) \cdot D \cdot \text{card})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\alpha \cdot w \cdot \text{card})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\alpha \cdot D \cdot D' \cdot \text{card})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\alpha \cdot D \cdot \text{card})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\alpha \cdot D \cdot \text{card})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\alpha \cdot \log(N_d) \cdot D \cdot \text{card})</td>
</tr>
<tr>
<td>Statistic-based Training</td>
<td>Sample (\rightarrow) [Statistic]*</td>
<td>Decision Tree, Random Forest, XGBoost, Naive Bayes, Bayes Network</td>
<td>(\alpha \cdot D' \cdot \text{card})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\alpha \cdot N^2 \cdot \text{card})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\alpha \cdot \log(N) \cdot \text{card})</td>
</tr>
<tr>
<td>Distance-based Training</td>
<td>Materialize (\rightarrow) [Distance]*</td>
<td>KMeans, K-nearest neighbors (KNN), KNN (HNSW)</td>
<td>(\alpha \cdot D \cdot K \cdot \text{card})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\alpha \cdot D \cdot \text{card})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\alpha \cdot \log(N) \cdot \text{card})</td>
</tr>
<tr>
<td>Analytic-based Training</td>
<td>Materialize (\rightarrow) Matrix</td>
<td>Mixture Model</td>
<td>(\alpha \cdot w \cdot \text{card})</td>
</tr>
</tbody>
</table>

TABLE II: Details of ML Models Supported by GaussML.

IV. QUERY EXECUTION

The implementation of query execution with ML models in GaussML relies on both native and extended operators. The extended operators include model training and predicting, and data preprocessing. In these procedures, GaussML incorporates ML algorithms with traditional execution operators. In this part, we first introduce model training workflow and operators in Section IV-A and then we describe the model persistence and model predicting in Section IV-B and Section IV-C. Next we introduce the design details of data shuffle operators in Section IV-D. Finally we introduce data preprocessing in Section IV-E.

A. Model Training

Training Workflow. The training workflow in GaussML is shown in Figure 7. For iterative training models, hyperparameter optimization engines first read a list of hyperparameters parsed from user-defined SQL, and then train models with different hyperparameters in parallel. In each iteration, executor first samples data from precedent execution nodes, and then updates model parameters according to gradients and learning rates. HPO engine stops and drops models with large validation errors in each iteration until only one model left. At last, executor trains the model on full data again until convergence, and then stores the model in the system.

Basic Training Operators. We summarize the training pattern of all the machine learning model, and implement a series of common operators in database for model training, we will present these basic operators in the following.

(1) Materialize operator is responsible for loading data from disk into memory, which will be controlled by upper operators in the training plan. For example, SGD tells the materialize operator what will be used in the next round, and materialize can trigger asynchronous data prefetch task to accelerate large scale data training.

(2) Sample operator supports two types of sampling, based on number of rows or sample ratio. GaussML offers two way to sample data, one is sample pages from file, which is faster; the other is reservoir sampling, which is more evenly, but may access more pages.

(3) Distance operator has two types, distance compute operator and distance merge operator. Distance compute operator get two lists of tuples from scan node and data from upper operator, and then it computes the distance by using SIMD. For distance computing (like L1 and L2) over features across different tables, the computing can be pushed down to each table scan, and merge later. In this way, only the distance and the join key will participate the join, and we can reduce the tuple width of join, and reduce the join overhead (data transfer, memory usage, etc.) significantly.

(4) SGD operator in GaussML updates the model parame-
...ters according to gradients and errors of functions on batches of data. It also controls the Materialize operator when to fetch the next batch.

(5) Shuffle operator has two stages, it first shuffle the blocks and get a random page for each call, and then it shuffles the tuples in the block before return the result to upper node in the execution plan.

(6) Matrix operator computes all kinds of matrix calculations, for example, element-wise matrix add, matrix multiplication, matrix inversion, etc. In GaussML, we also use SIMD to accelerate the matrix calculation, for example, we compute 4 float simultaneously.

**Parallel/Distributed Training.** In order to accelerate model training on large dataset, we implement parallel and distributed training operators in GaussML. GaussML designs a synchronous collective model training framework, where each thread trains each iteration independently on part of training data, and they broadcast parameters/gradients by global shared memory after each iteration. An iteration ends when every thread has received information from all other threads. Before next iteration begins, each thread updates model according to synchronized parameters/gradients. One typical parallel training execution plan example is as follows.

- Merge Model (16-way)
- Streaming (type: Local dop: 1/16)
  - Train Model
  - Materialize
  - Seq Scan on T1

Different from parallel execution on single node, distributed training uses streaming operators on all data nodes to exchange parameters/gradients between executors, and each executor trains model on its local data. One typical distributed training execution plan example is as follows.

- Streaming (type: GATHER)
  Node/s: All datanodes
- Streaming (type: BROADCAST)
  Spawn on: All datanodes
  - Recursive Train Model
  - Materialize
  - Seq Scan on T1

We implement RTMController in Recursive Training operator to synchronize the behavior of each data node. Specifically, each data node should receive updated models from all other nodes before starting a new training iterations.

GaussML supports parallel/distributed training based on the architecture of GaussDB. We will introduce implementation details next.

(1) For linear model, logistic model, polynomial model and MLP, GaussML implements parallel SGD for training. Each thread firstly fetches one part of data in mini-batches and shuffles them, and then computes local gradients of model. In the distribute phase, each thread sends local model gradients \(\delta w_i\) to other threads in each iteration. In the merge phase, each thread updates the model parameters by gradients produced by all threads. The training process terminates until model convergence or iterations budget is used up.

(2) For KMeans, each thread samples \(K\) centroids first, and we guarantee the consistency of the initial centroids in different threads with the same random seed. In next iterations, each thread firstly obtains centroids from other threads and computes the mean as new centroids, and then each thread re-clusters local data and calculates centroids according to the new clusters, and finally each thread exchanges centroids with others and start the next iteration.

(3) For Naive Bayes and Bayes Network, only a few data passes are needed (typically 3 iterations, for data discretizing, data statistic, and conditional probability calculation). Specifically, each thread generates data distribution table for its local data in parallel, and aggregates the frequencies and computes the probabilities for each Bayes node in merge phase.

(4) For KNN, each thread receives the current distance upper bound \(d_{sup}\), and then finds the nearest tuple whose distance to the query is smaller than \(d_{sup}\) from its own data part. Next it shares the top-k smallest distances in each iteration. The merge node maintains \(K\) nearest tuples and updates the largest distance in \(K\) tuples.

**Hyperparameter Optimization.** In order to reduce the manual efforts of tuning models, we build an HPO engine in GaussML, which is used to find an optimal set of model hyperparameters (including learning rate, training batch size, kernel type, etc.) to achieve the lowest validation errors. HPO problem is challenging because it’s often a non-convex problem, and thus convex optimization methods cannot be utilized. Moreover, it’s hard to model the mapping between hyperparameters and validation errors, and thus the gradient-based optimization methods cannot be used. In general, three kinds of methods are used for HPO, including grid search, random search and hyperband. Grid search will try all possible combinations of hyperparameters on training data, and select the best set of hyperparameters. Grid search can guarantee to find the optimal model, but it’s too costly and it’s impossible to be deployed on cost-sensitive relational database. Random search first draws a small set of hyperparameters randomly and then train models on training data. Random search is faster but it’s more likely to miss the optimal hyperparameters. In GaussML, we implement a hyperband-based HPO engine...
in ML executor. When creating a model with AUTO-ML on, GaussML first searches candidate hyperparameters from system catalog, and then the engine trains models on sample data in parallel. The executors report model accuracy to the coordinate node after each iteration by using streaming operator. GaussML throws away hyperparameters with errors higher than the average constantly until only one model left. One model training plan example with HPO is shown as follows.

B. Model Persistence

We save the trained model in system table gs_model_warehouse in structural data including model name, training time, hyper-parameters, validation metrics and model parameters. For security and space efficiency, we store the model parameters as a binary sequence where all parameters are arranged one by one in a predefined structure. We build indexes for object id and name of each model to accelerate the model search, and we make an instance-level memory context to cache the deserialized model for avoiding repeated costly model loading process.

C. Model Prediction

The workflow of model prediction has three steps. Firstly, the ML executor searches the query model in the shared memory. If found, then the executor inputs results from precedent execution nodes into model to predict. If the model is not found in memory, executor searches the system table gs_model_warehouse by name, and then deserializes the model parameters and predicts the target labels. Finally, if the designated model has descendant versions, our will organize them with version ids and users can use previous models easily.


D. Data Shuffle

Data shuffle not only is vital to convergence speed of machine learning optimizer like stochastic gradient descent (SGD), but also balance the data distribution between different data partitions.

Local Shuffle Operator. Local shuffle aims to change the IO order for each training iteration. GaussML supports two types of local shuffle operators, tuple shuffle and block shuffle. For block shuffle, we integrate it with scan node. We shuffle block ids within each page read by heap scan function and store it into structure shuffled_block_ids, and then the upper node reads blocks according to shuffled_block_ids. For tuple shuffle, we add a shuffle node before model training. The shuffle node fetches tuples into buffer from precedent nodes, and shuffles tuple ids in buffer into an index shuffled_tuple_indices. If the buffer is not empty, training node fetches the next tuple according to shuffled_tuple_indices; otherwise, fetches tuples until the buffer is full, and then shuffles index and fetches tuples.

Cross-node Shuffle Operator. The distributed training in GaussML performs well if the training label distributed evenly across all data nodes. However, this condition is not always satisfied due to the distribution key bias. For example, if training labels are correlated to the time, and the data is distributed by time, then the labels are likely to be bias. In GaussML, we provide two strategies to redistribute the data, they are full redistribution and fine tuning.

The query optimizer in GaussML will check the bias of training labels according to the partition-level statistics in system table gs_statistic. Specifically, query optimizer checks the MCV of discrete training labels, or it checks the histogram of continuous columns on all data partitions, and if the distribution differs a lot between partitions, query optimizer inserts a redistribution operator in the plan to shuffle training data across different data nodes.

However, full shuffle across all data nodes is not always possible in considering of efficiency. We offer another fine tune strategy for training label redistribution in GaussML. Instead of adding a new operator, the fine tune strategy redistributes a small set of tuples by using streaming (broadcast) operator when training. Specifically, in each iteration, a data node receives models from other nodes. The node then tests these models on a sample of local data, and collect those under-fitting training labels. In next broadcast phase, each data node sends both the updated model and some training tuples. In this way, each node can utilize the samples from other nodes to calibrate the distribution of data itself.

E. Data Preprocessing

In most cases, data preprocess for model training and prediction can be done by simple Select-Projection-Join (SPJ) queries. However, some data preprocessing operators need to know the data distribution in an aggregation subquery. For example, min-max scaling depends on the maximum and minimum statistics on each column. In this case, we integrate two new operators for normalization and missing value imputation into GaussML. These operators can reuse the system statistics table in relational database, and avoid additional costly aggregation subqueries when training models.

Normalization. GaussML offers min-max scaling and z-score normalization for ML queries. For linear scaling, each value \( x \) in an attribute is transformed as \( \frac{x - \mu}{\sigma} \). In GaussML, \( x_{\text{max}} \) and \( x_{\text{min}} \) are collected in real-time, and all statistics are maintained in system table gs_statistic. For z-score normalization, each value \( x \) is transformed as \( \frac{x - \mu}{\sigma} \) where \( \mu \) is the mean of the population and \( \sigma \) is the standard deviation of the population. GaussML saves \( \mu \) and \( \sigma \) (by Welford’s algorithm [25]) in system table gs_statistic.

Missing Value Imputation. To support missing value imputation, we utilize the distribution of the column or the
whole dataset for missing value inference. It’s costly to model the distribution of dataset before training the data, and thus GaussML makes full use of the system statistics including the most common values (MCV) on single column, histogram on single column, MCV on multi-columns, and machine learning model on multi-columns. Next we introduce how GaussML uses these statistics to infer the missing value. If machine learning models on multi-columns exist, we predict the conditional probability of each possible value given values in other columns. For example, we have a bayesian model for table $T$ with three attributes $a_1, a_2, a_3$, and attribute $a_3$ is missing for a tuple. We predict $a_3$ by using probability $P(a_3|a_1, a_2)$ given by model. Similarly, if MCV on multi-column exists, GaussML matches other attributes ($a_1$ and $a_2$ in the example) with the most common values, and $a_3$ in matched tuples are highly possible to be the missing value. When the multi-column statistics does not exist (due to resource limitation), GaussML uses single-column statistics, including MCV and histogram. Specifically, GaussML samples missing values from MCV according to the frequency or from buckets in histogram according to the height of buckets.

V. EXPERIMENTS

A. Experimental Settings

In this section, we first introduce the datasets, methods and hardware we use for evaluating GaussML in section V-A. Then we analyze the overall evaluation results for standalone deployment mode in section V-B. Next we show comparison results of different features of GaussML in section V-C-V-D. At last, we summarize all experiments.

Datasets. We adopt six datasets to evaluate the efficiency and performance of different machine learning platforms (as table III shows). RLCF contains data pairs of student information in the course of several years and each pair is marked as ‘match’ or ‘nonmatch’. ADULT contains staff members and their incomes, the classification task is to find whether income of each person exceeds 50K per year. ClickStream contains information on clickstream from online store. UrbanGB contains coordinates of locations in urban and each location belongs to a partition. Each record in Swarm represents a boid, including positions, velocity, alignment, etc. Workload contains query ranges and 3 kinds of aggregation results, and we use the coordinator and count to predict the average result. YearPredict represents description of songs and their published years. CCPF contains 4 features of the Combined Cycle Power Plant, we use them to predict the energy output of the plant. These datasets cover classification, clustering, dimension reduction and regression tasks, and they are challenging and general for all machine learning platforms. Slice is retrieved from CT images, it is adopted by HPO benchmark proposed by Klein [26].

Methods. (1) GaussML is a system build in native OpenGauss database, it offers MLSQL interface for end-to-end machine learning applications. There are three deployment approaches including single-thread, multiple-thread and multiple-node.

<table>
<thead>
<tr>
<th>dataset</th>
<th>tasks</th>
<th>#columns</th>
<th>#rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLCF [27]</td>
<td>classification</td>
<td>12</td>
<td>5,749,132</td>
</tr>
<tr>
<td>ADULT [28]</td>
<td>classification</td>
<td>14</td>
<td>48,842</td>
</tr>
<tr>
<td>ClickStream [29]</td>
<td>classification</td>
<td>14</td>
<td>165,484</td>
</tr>
<tr>
<td>UrbanGB [30]</td>
<td>clustering</td>
<td>2</td>
<td>360,177</td>
</tr>
<tr>
<td>Swarm [31]</td>
<td>dimension reduction</td>
<td>2,400</td>
<td>28,555</td>
</tr>
<tr>
<td>Workload [32]</td>
<td>regression</td>
<td>8</td>
<td>260,000</td>
</tr>
<tr>
<td>YearPredict [33]</td>
<td>regression</td>
<td>90</td>
<td>515,345</td>
</tr>
<tr>
<td>CCPF [34]</td>
<td>regression</td>
<td>4</td>
<td>9,568</td>
</tr>
<tr>
<td>Slice [35]</td>
<td>HPO</td>
<td>385</td>
<td>42,800</td>
</tr>
</tbody>
</table>

TABLE III: Datasets

It also supports automatic hyperparameters optimization in database.

(2) MADlib is a set of ML UDFs customized for relational database, and it’s widely adopted by Postgres, GreenPlum, etc. We also integrate these functions into openGauss for fair evaluation.

(3) ML-A is implemented by using Python script with widely-used machine learning libraries including sklearn [14], pandas [36] and openGauss connector. They offer Python programming interface for ML users and users need to load batches of dataset from database by using database connector in each training iteration.

In order to guarantee the fairness of evaluations, we set the same batch size and iteration number for GaussML, MADlib, and ML-A.

Hardware Environment. We conduct experiments on 4 machines each with 72 Intel 3.00GHz CPU cores, 64GB memory (which is enough for datasets we use) and 2TB disk.

B. Overall Comparison

Expr1: In table IV, we compare GaussML with MADlib and ML-A on classification tasks, and the results show that GaussML significantly outperform MADlib and ML-A.

For training on datasets RLCF, GaussML outperforms MADlib by 5-400% for SVM, and the advantage of GaussML is more obvious with more iterations. The reason is that GaussML is native for both data reading and model training, while MADlib parses the SQL in database and then invokes Python functions to load data and train model. Moreover, GaussML uses materialize operator to adaptively cache mini-batches when loading data iteratively. ML-A is the worst for training performance because it need to transfer data on network by connecting to database server. Similar result can be observed for Logistic model and XGBoost. On dataset ADULT, GaussML outperforms MADlib by 4-9 times for 10 iterations training, and 1-2 orders of magnitude for 100 iterations training. This is because that the materialize operator in GaussML can cache nearly all data for small dataset. The ML-A outperforms MADlib, but still much slower than GaussML.

Experiment results also show that GaussML and MADlib spend similar time for model prediction because they all need to load data once in database. However, ML on ML-A is slower than GaussML, because the data loading by Python driver is more time consuming.

For accuracy, GaussML, MADlib and ML-A show similar final results because they use the same algorithms. However,
GaussML convergence faster than MADlib and ML-A, because GaussML uses advanced online shuffle technique which can change the blocks order read in different iterations.

Expr2: We test regression tasks on different ML platforms on three datasets. We observe from figure 8 that GaussML outperforms MADlib and ML-A on all three datasets for training time. The reason is that the native scan operator in GaussML is the most efficient, and GaussML avoids inefficient code implemented by Python. The results show that ML methods implemented on MADlib and ML-A have similar training time because MADlib only puts algorithms in

We compare the mean square test errors of ML-A because of the early stop mechanism, and thus MADlib could explore more hyper-parameters than MADlib.

Expr4: Figure 10 shows training time of K-means with varying number of centroids and dimensions. We can observe that GaussML outperforms MADlib by 1 order of magnitude.

### Table IV: Comparison training time (ms)/prediction time(ms)/validation accuracy on classification tasks.

<table>
<thead>
<tr>
<th>Method</th>
<th>SVM</th>
<th>Logistic Model</th>
<th>XGBoost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>training</td>
<td>predict</td>
<td>accuracy</td>
</tr>
<tr>
<td>RLCP</td>
<td>GaussML</td>
<td>21,131</td>
<td>1,561</td>
</tr>
<tr>
<td></td>
<td>MADlib</td>
<td>50,138</td>
<td>1,537</td>
</tr>
<tr>
<td></td>
<td>ML-A</td>
<td>66,751</td>
<td>1,873</td>
</tr>
<tr>
<td>Adult</td>
<td>GaussML</td>
<td>95,565</td>
<td>1,568</td>
</tr>
<tr>
<td></td>
<td>MADlib</td>
<td>485,299</td>
<td>1,564</td>
</tr>
<tr>
<td></td>
<td>ML-A</td>
<td>579,112</td>
<td>1,832</td>
</tr>
</tbody>
</table>

Fig. 10: Comparison on clustering tasks (d is dimension, k is #centroids, the dataset is UrbanGB, the method is kmeans).
for different hyperparameters and data volumes. With the total number of points increasing, training time increases faster on MADlib than GaussML, because GaussML uses a more efficient data visit path. When we increase the number of centroids by 10 times, training time also increases around 1-10 times, because distance computation cost between points and centroids increases. When we increase dimensions of dataset by 10 times, training time also increases 1-8 times, because it increases the distance computation cost.

Expr5: Because the PCA methods on GaussML and MADlib use iterative training approach for supporting large datasets, and we only compare them in figure 11a and figure 11b. We can observe that with the number of principal components increasing, the training time drastically increases because of computation cost increasing. The training time of GaussML is always less than MADlib significantly because that GaussML adopts native data scan method with data materializing. We also observe that training time on both GaussML and MADlib increases with data size increasing, because of computation cost and IO cost growing.

In a word, single-thread model training in GaussML is faster than other systems for all machine learning tasks, because of it has more efficient data visit path with materialize operator and local shuffle operators.

C. Query Planning

Expr6: In figure 12, we test different query optimization techniques on 4 workloads. The WL1 is a workload with complex data preprocessing pipeline in SQL, it contains operations like union, merge and join which would be very expensive if a bad execution plan is executed, and the results show that query performance significantly decreases by 10 times by using query optimizer in openGauss. The WL2 is a workload with single relation and multiple filter conditions with different machine learning model predictions, the execution plans generated from ML-aware optimizer outperform traditional optimizer by 1-2 times because GaussML is able to reorder the filter conditions according to their selectivity. The WL3 and WL4 are two hybrid workloads with multiple relations, and constraints with machine learning constraints exist on each relation, we can observe that the ML-aware optimizer outperforms traditional optimizer by 1-2 times because it can optimize the join orders of different relations with model prediction predicates.

In summary, ML-aware query optimizer in GaussML can generate better execution plan for end-to-end machine learning. This is because GaussML not only makes full use of cost-based optimization in openGauss, but judiciously designs the model computation cost estimator.

D. Query Execution

Expr7: We test HPO engines in GaussML with HPOBenchmark [26] on dataset Slice. The HPOBenchmark can simulate the process of hyperparameter selection, data sampling and model training for any plugin algorithm. In order to compare the performance of HPO methods, we connect HPOBenchmark as a controller in GaussML. Figure 13 shows that the HPO strategies implemented in GaussML produce similar training loss curves with HPOBenchmark. We can also observe that Hyperband strategy outperforms Random in training efficiency by 1 order of magnitude. This is because Hyperband can discard non-promising hyperparameter configurations instantly, while the Random need to train models completely.

Expr8: We deploy GaussML on multiple nodes, and if the number of data nodes exceeds 4, we simulate multiple nodes by using different processes and data directories. From figure 15, we can observe that with the number of data nodes increasing, the training time decreases significantly because distributed training accelerates data processing. However, training time becomes stable after 8 data nodes, and the reason is as two folds. First it increases communication cost for broadcasting parameters. Second the difference between distributions of data nodes causes overfitting within each node and more iterations are needed for distributed consensus.

In figure 15b, we observe that training time of MADlib increases more than GaussML when the data size and parallelism increase simultaneously, because GaussML has better scalability.

Expr9: We compare the effect of different shuffle methods on the parallel training curve. From figure 16, we can observe that training without shuffle can only achieve 77% accuracy for SVM and 80% for logistic model, while training with global shuffle can achieve 99% accuracy. It’s worth noting that the tuple-shuffle and block shuffle on single node is fast, while the data redistribution at global scale takes much more time because of data transfer. Therefore, the recommended shuffle method is block-shuffle unless the label distribution is very skew on different nodes.

We designs the framework to incorporate these operators in GaussML with high efficiency, and these operators further improve the performance of model training in parallel and
Fig. 14: Training time comparison with optimizations.

**Expr10:** Figure 14 shows the training time decreases after GaussML adds individual optimization approach. We observe that shuffle technique decreases training time by 15-45%, data materialize decreases training time by another 40-70%, and SIMD/parallel technique makes training time to halve further.

VI. RELATED WORKS

**Machine learning in database.** Recently, in-DB ML has been widely studied. MADlib [37] is composed of a collection of UDFs, and its performance is much lower because of the function cannot be optimized by database. SQLFlow [38] builds an SQL engine on database and external machine learning platforms, and it interprets the SQL to Python program language, fetches data from database, and trains models in existing platform. SQLflow is also lack of holistic optimization with database and the data transferring is very time consuming. Papers [39], [40] propose optimization methods for model prediction workflow, but they do not discuss model training and they are not fully integrated in modern DBMS.

**Hyperparameter Optimization.** To reduce the bars of using machine learning, HPO is becoming an inevitable technique in machine learning platform. Methods for machine learning has been proposed in papers [41], [42]. In order to accelerate it, we implement hyperband in GaussML and GaussML stores the history performance of each configuration for warm-start. There are also some structure search methods proposed in [43], but they are customized for complex neural network which is not the scope of this paper.

**Data Shuffle.** Model training approaches like SGD rely on data shuffle for fast convergence, and researchers propose some partial shuffle methods to reduce IO [20]. GaussML supports both fully shuffle across data nodes and local shuffle by integrating the shuffle method [44] into openGauss.

**Distributed Training.** Distributed model training can be classified as data parallelism and model parallelism. On large scale data, data parallelism is the most efficient way to accelerate training. Many techniques have been adopted in machine learning products [9], [45]–[47], but they are not designed for distributed relational database and mostly rely on a parameter server. In order to avoid the coordinate node in openGauss becomes the bottleneck, GaussML makes the data nodes exchange parameters directly.

**Future Work**

We can extend GaussML to support GPU usage in future work, as GaussML has abstracted the ML computation of diverse algorithms as typical ML operators such as SGD operator, statistic operator, distance operator, as illustrated in Table II. In GaussML, these operators are essentially implemented based on the matrix computation, which can be accelerated by GPUs such as the CUDA library. However, although GPU can reduce the computation time, especially for large batch size, it adds overhead for transferring data between RAM and GPU memory. Therefore, the key factor of efficiency by using GPU is batch size, this is generally tuned by the programmer (or platform user). In database, we can easily computes the computing cost of each ML operators the computing power of GPU, and the optimizer in GaussML can choose a proper batch size and whether to use GPU for ML operators. In summary, GaussML could support a more adaptive GPU usage than Python-based ML system.

VIII. CONCLUSION

This paper proposes an end-to-end in-database machine learning system GaussML. GaussML is seamlessly integrated with openGauss database. GaussML offers native MSQL interface for data analysts, and supports holistic optimization for end-to-end machine learning queries, including model training and model prediction. GaussML introduces an ML-aware cardinality and cost estimator to optimize the SQL+ML query plan. GaussML leverages SIMD and data prefetching techniques to accelerate the ML operators for training. Extensive results on real datasets show that GaussML outperforms counterparts significantly for typical machine learning queries.

**Acknowledgement.** This paper was supported by National Key R&D Program of China (2023YFB4503600), NSF of China (61925205, 62232009, 62102215).