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*
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 outsize. (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 that it's would be inefficient by using naive 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 prefeching, 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).

[1]https://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 $\{t_1, t_2, t_3, \cdots, t_n\}$ 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 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 L1 and L2, 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

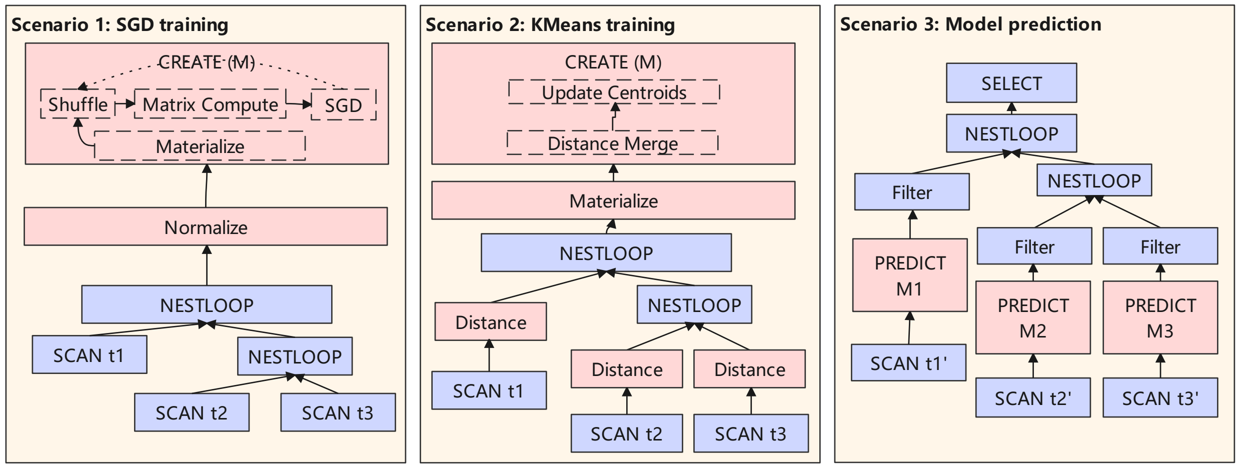


Fig. 1: Exemplary scenarios for end-to-end machine learning in database (pink boxes represent ML operators).

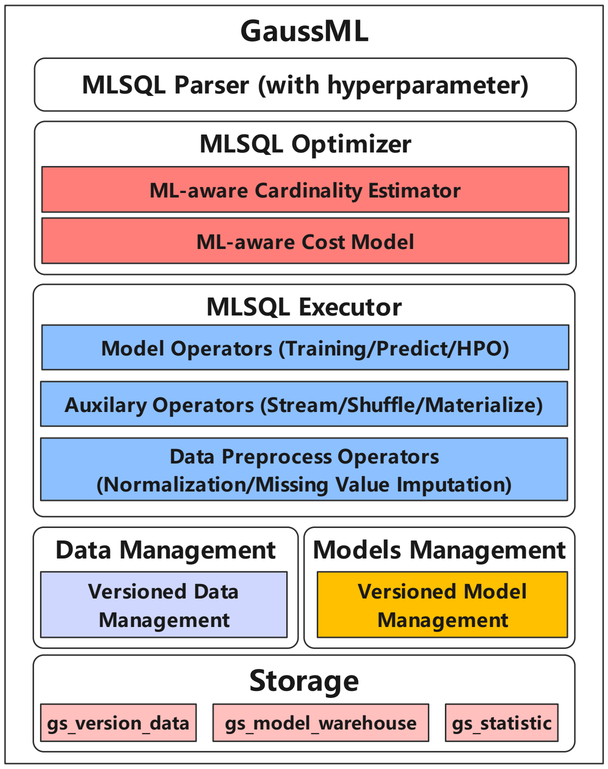

GaussML
MLSQL Parser (with hyperparameter)
MLSQL Optimizer
ML-aware Cardinality Estimator
ML-aware Cost Model
MLSQL Executor
Model Operators (Training/Predict/HPO)
Auxilary Operators (Stream/Shuffle/Materialize)
Data Preprocess Operators (Normalization/Missing Value Imputation)
Data Management
Versioned Data Management
Models Management
Versioned Model Management
Storage
gs_version_data
gs_model_warehouse
gs_statistic

Fig. 2: GaussML System Overview

Operation	MLSQL
Training	**CREATE MODEL** model_name **USING** model_type, [**AUTOML** ON — OFF] [**FEATURES** attribute_list, **TARGET** attribute_name[,attribute_name]*], **FROM** ([schema.]table — subquery — view) **WITH** (hp_name [= hp_value]) [, ...]*]

Inference	**SELECT** attribute[,attribute]*
	PREDICT BY model_name
	[(**FEATURES** attribute[,attribute]*]
	FROM ([schema.]table_name — subquery)

Explain	**EXPLAIN MODEL** model

TABLE I: Grammar of MLSQL.

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 Section IX-A.

C. GaussML SQL

In this section, we formally introduce the extended MLSQL 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. We investigate the user interface of modern DBML systems and summary their ease of usage in table **??**.

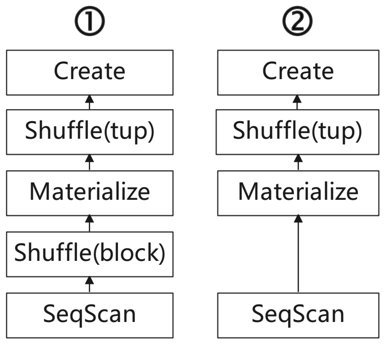
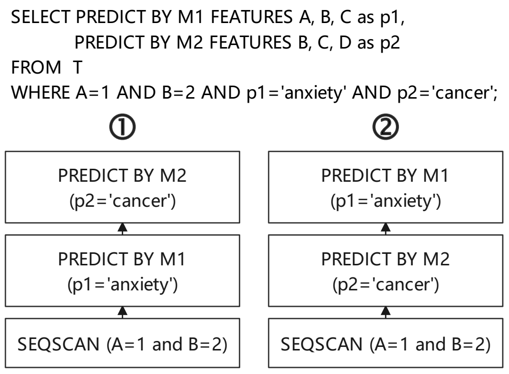


Fig. 3: Candidate plans for SGD model create.



SELECT PREDICT BY M1 FEATURES A, B, C as p1,
PREDICT BY M2 FEATURES B, C, D as p2
FROM T
WHERE A=1 AND B=2 AND p1='anxiety' AND p2='cancer';
①
②
PREDICT BY M2 (p2='cancer')
PREDICT BY M1 (p1='anxiety')
SEQSCAN (A=1 and B=2)
PREDICT BY M1 (p1='anxiety')
PREDICT BY M2 (p2='cancer')
SEQSCAN (A=1 and B=2)

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

III. Query Planning

Query planner is a vital component for end-to-end ML because it can effectively schedule both the ML and traditional operators. However, traditional optimizer in database cannot handle end-to-end ML scenarios for four reasons [21].

(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.

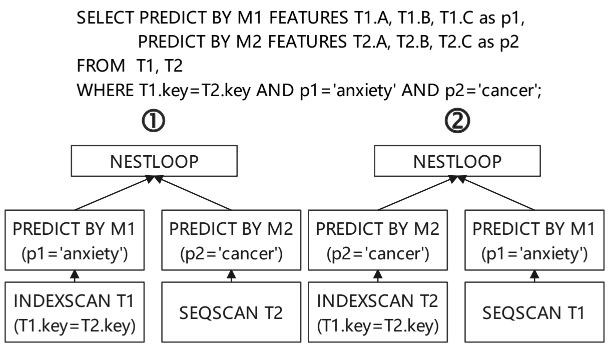
(iii) Traditional cardinality estimator cannot estimate the conjunctive predicates with both ordinary conditions and model prediction because database cannot catch correlations between model labels and data in other columns.

(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(creat) query planning

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



SELECT PREDICT BY M1 FEATURES T1.A, T1.B, T1.C as p1,
PREDICT BY M2 FEATURES T2.A, T2.B, T2.C as p2
FROM T1, T2
WHERE T1.key=T2.key AND p1='anxiety' AND p2='cancer';
①
②
NESTLOOP
PREDICT BY M1 (p1='anxiety')
PREDICT BY M2 (p2='cancer')
INDEXSCAN T1 (T1.key=T2.key)
SEQSCAN T2
NESTLOOP
PREDICT BY M2 (p2='cancer')
PREDICT BY M1 (p1='anxiety')
INDEXSCAN T2 (T1.key=T2.key)
SEQSCAN T1

Fig. 5: Inner/Outer table for NestLoop node.

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. 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 $A = 1 \wedge B = 2$ 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 M_1 and M_2 (i.e. finding patients with both anxiety and cancer in this example). The optimizer computes the total cost of these two plans and selects the cheaper one. Assume that the selectivity of predicates p_1 and p_2 are s_1 and s_2 and they are independent. N is the total number of rows in table and n rows are fetched from SEQSCAN. The cost of processing each tuple is denoted as C. Therefore the cost of the first plan is $C(SEQSCAN)*N+C(M_1)*n+C(M_2)*n*s_1$, and the cost of the second plan is $C(SEQSCAN) * N + C(M_2) * n + C(M_1) * n * s_2$.

Figure 5 shows a simple two-table join on keys, and each table has a predicate with a model prediction individually. In this case, query optimization problem is to decide which table is the inner(outer) table. GaussML sequentially scans the outer table and predicts each row by model, and if the result matches the condition, GaussML searches tuples with the same key by using index of inner table. We reuse notations s_1, s_2 and C, and denote selectivity of join condition on T_1 and T_2 as s_3 and s_4, and the total number of rows of tables T_1 and T_2 are N_1 and N_2. Therefore if all conditions are independent and $C(m1) * N_1 + C(m2) * N_1 * s_1 * N_2 * s_4$ is smaller than $C(m2)*N_2+C(m1)*N_2*s_2*N_1*s_3$, then the second plan should be selected. Besides, GaussML also conducts query optimizations for model training. For example, GaussML adds a materialized node before training node for model that need iterative data access to reduce IO overhead, and GaussML adds a redistribute node when the data distribution in each partition differs a lot (see Section IV-A).

C. Selectivity estimation for ML predicates.

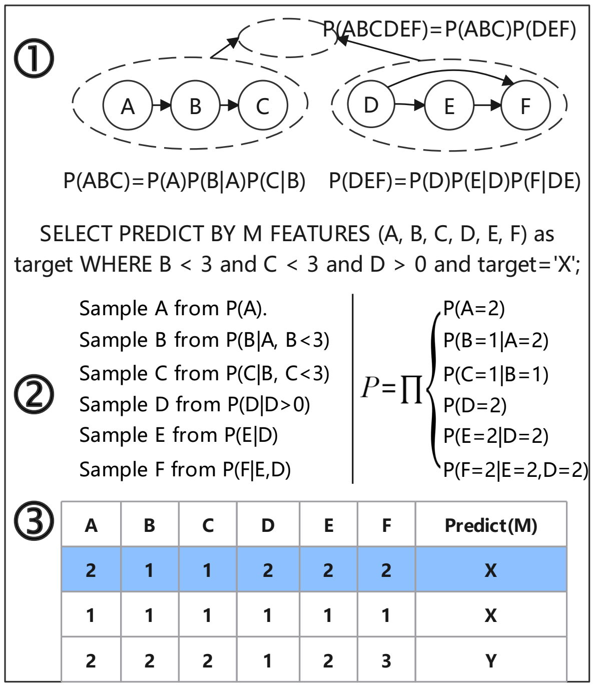
The hardest part is how to estimate the selectivity of each ML predicate before actually executing it, because the features attributes are usually high dimensional and correlated, and the model prediction results are not materialized in the table.

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 predict 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.

$$\widehat{sel} = \arg\min_{s} errf(s - \sigma_{\wedge_{1\le i\le n}(op_i(f_i(a_1,a_2,\cdots,a_n),v_i))}(T)) \quad (1)$$

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



A	B	C	D	E	F	Predict(M)
2	1	1	2	2	2	X
1	1	1	1	1	1	X
2	2	2	1	2	3	Y

Fig. 6: Steps of selectivity estimation for ML predicates.

to sample data in existing works [22], [23], 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

$$corr(T.A, T.B) = \sum_{r\in T} P(r.A, r.B) log \frac{P(r.A, r.B)}{P(r.A)P(r.B)} \quad (2)$$

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 $P(row_i)$) 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 $M(row_i)$, and the selectivity of conjunctive predicates can be calculated as below.

$$\widehat{sel} = \frac{\sum_{i=1}^{n} I(M(row_i) = true)P(row_i)}{\sum_{i=1}^{n} P(row_i)} \quad (3)$$

In this way, we can easily compute cardinality by multiplying selectivity and total number of rows in a table. For distributions in cardinality estimation, GaussML utilizes the in-database ML operators for training and evaluating.

Category	Training Pipeline	ML Models	Inference Cost
Gradient-based Training Training models with mini-batch SGD. (classification/regression/ dimension reduction)	Shuffle $\rightarrow$ [Materalize $\rightarrow$ Matrix $\rightarrow$ SGD]*	Linear model	$\alpha \cdot D \cdot card$
		Logistic model	$\alpha \cdot D \cdot card$
		Polynomial model	$\alpha \cdot D \cdot card$
		Support vector machine (SVM)	$\alpha \cdot D \cdot card$
		SVM with square kernel	$\alpha \cdot (D^2 + D + 1) \cdot card$
		SVM with Gaussian kernel	$\alpha \cdot (D + 1) \cdot D' \cdot card$
		Multi-layer perceptron (MLP)	$\alpha \cdot \lvert w \rvert \cdot card$
		Principal Components Analysis (PCA)	$\alpha \cdot D \cdot D' \cdot card$
Statistic-based Training Training model with data counting. (Tree-based/Bayes)	Sample $\rightarrow$ [Statistic]*	Decision Tree	$\alpha \cdot depth \cdot card$
		Random Forest	$\alpha \cdot depth \cdot n \cdot card$
		XGBoost	$\alpha \cdot depth \cdot n \cdot card$
		Naive Bayes	$\alpha \cdot log(N_d) \cdot D \cdot card$
		Bayes Network	$\alpha \cdot log(N_d) \cdot D \cdot card$
Distance-based Training Training models with distance functions. (clustering)	Materialize $\rightarrow$ [Distance]*	KMeans	$\alpha \cdot D \cdot K \cdot card$
		K-nearest neighbors (KNN)	$\alpha \cdot N^2 \cdot card$
		KNN (HNSW)	$\alpha \cdot log(N) \cdot card$
Analytic-based Training Training models with analytical solutions. (regression)	Materialize $\rightarrow$ Matrix	Mixture Model	$\alpha \cdot \lvert w \rvert \cdot card$

TABLE II: Details of ML Models Supported by GaussML.

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 $card$) 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 Gauss kernel will map D-dimension features into infinite dimensions, and we use $D' = max(128, 2 * D)$ 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 KNN 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, α is a coefficient which can be tuned manually or automatically.

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-

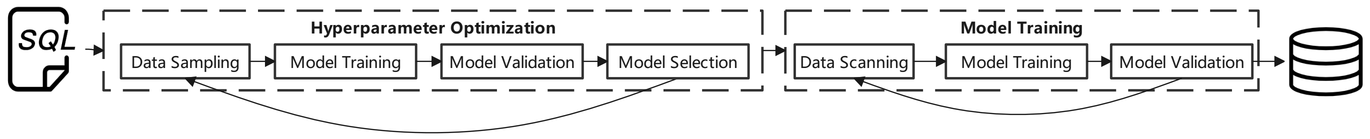


Fig. 7: Training workflow.

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 δw_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 dis_{up}, and then finds the nearest tuple whose distance to the query is smaller than dis_{up} 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, we cache the model loaded from disk into memory. If the designated model has descendant versions, *our* will organize them with version ids and users can use previous models easily.

```
-> Train Model - logistic_regression
 CTE t
  -> Select Model
   -> Streaming (dop: 1/16)
   -> Train Model - logistic_regression
    -> Materialize
     -> SampleResult
      -> Seq Scan on T1
 -> Seq Scan on T1
```

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_indexes. If the buffer is not empty, training node fetches the next tuple according to shuffled_tuple_indexes; 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 underfitting 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 $\frac{x-x_{min}}{x_{max}-x_{min}}$. In GaussML, x_{max} and x_{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 μ is the mean of the population and σ is the standard deviation of the population. GaussML saves μ and σ (by Welford's algorithm [24]) 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 deploy mode in section V-B. Next we show comparison results of different features of GaussML in section V-C-IX-C. 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).RLCP 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 people 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. CCPP 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 [25].

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.

dataset	tasks	#columns	#rows
RLCP [26]	classification	12	5,749,132
ADULT [27]	classification	14	48,842
ClickStream [28]	classification	14	165,484
UrbanGB [29]	clustering	2	360,177
Swarm [30]	dimension reduction	2,400	28,555
Workload [31]	regression	8	260,000
YearPredict [32]	regression	90	515,345
CCPP [33]	regression	4	9,568
Slice [34]	HPO	385	42,800

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 [35] 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.

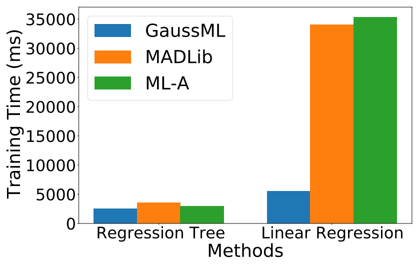
For training on datasets RLCP, GaussML outperforms MADlib by 5-400% for SVM, and the advantage of GaussML is more obvious with more iterations. The reason is that GaussML is native for both data reading and model training, while MADlib parses the SQL in database and then invokes Python functions to load data and train model. Moreover, GaussML uses materialize operator to adaptively cache min-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.

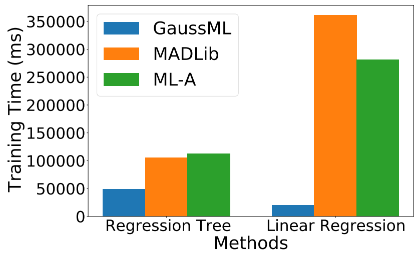
For accuracy, GaussML, MADlib and ML-A show similar final results because they use the same algorithms. However,

Method			SVM			Logistic Model			XGBoost		
dataset	#iterations	Platform	training	predict	accuracy	training	predict	accuracy	training	predict	accuracy
RLCP	10	GaussML	**21,131**	1,561	**.991**	**13,724**	1,580	**.990**	**36,182**	**2,829**	**.983**
		MADlib	50,138	**1,537**	.986	55,297	**1,534**	.981	/	/	/
		ML-A	66,751	1,873	.985	69,557	1,631	.988	311,918	3,125	.977
	100	GaussML	**95,565**	**1,568**	.996	**110,685**	1,580	.999	**131,377**	**2,822**	.999
		MADlib	485,899	1,664	.999	565,649	**1,567**	.999	/	/	/
		ML-A	579,112	1,832	.999	701,971	1,898	.999	1,487,396	2,976	.999
Adult	10	GaussML	**44.9**	**21.7**	**.743**	**113**	**16.6**	**.751**	**513**	**136**	**.803**
		MADlib	388	25	.721	430	22.8	.738	/	/	/
		ML-A	217	26.2	.717	253	24.1	.737	1,443	154	.794
	100	GaussML	**78.6**	**18.9**	.780	**589**	**22.9**	.789	**4,171**	**361**	.828
		MADlib	3,614	25.3	.781	4,338	23.2	.787	/	/	/
		ML-A	540	29.5	.775	1,142	23.5	.790	9,231	379	.825

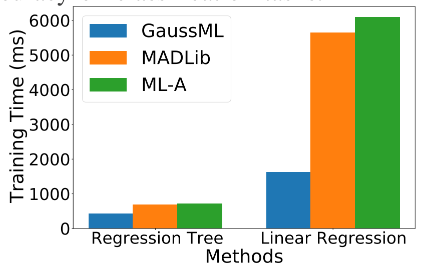
TABLE IV: Comparison training time (ms)/prediction time(ms)/validation accuracy on classification tasks.



(a) Test on dataset Workload.

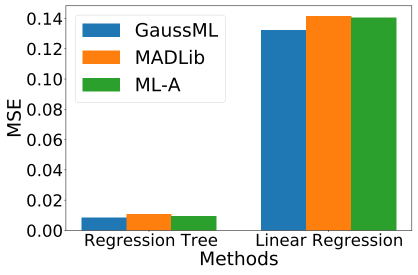


(b) Test on dataset YearPredict.

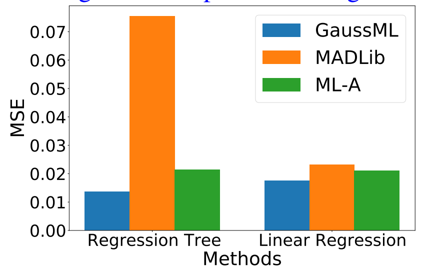


(c) Test on dataset CCPP.

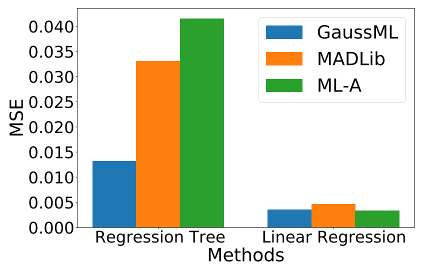
Fig. 8: Training time comparison on regression tasks.



(a) Test on dataset Workload.

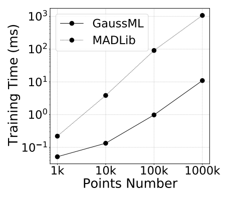


(b) Test on dataset YearPredict.

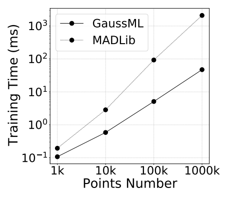


(c) Test on dataset CCPP.

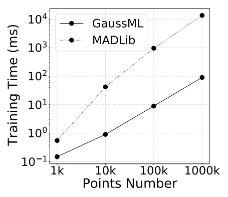
Fig. 9: Mean Square Error comparison on regression tasks.



(a) k=10,d=10



(b) k=100,d=10



(c) k=100,d=100

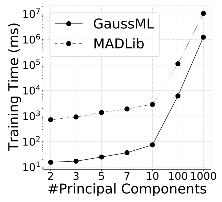
Fig. 10: Comparison on clustering tasks (d is dimension, k is #centroids, the dataset is UrbanGB, the method is kmeans).

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.

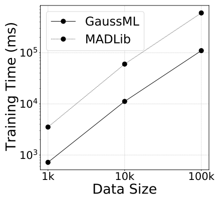
Expr2: We test regression tasks on different ML platforms on three datasets. We observe from figure 8 that GaussML outperforms MADlib and ML-A on all three datasets for training time. The reason is that the native scan operator in GaussML is the most efficient, and GaussML avoids inefficient code implemented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

Expr3: We compare the mean square test errors of GaussML, MADlib and ML-A platforms in Figure 9. We can observe that GaussML has lower test errors on three datasets, the reason are as follows. (1) Global/local shuffle in GaussML reduces the variance when model training, and makes the model less overfitting. (2) The efficiency of HPO in GaussML is faster than that in MADlib because of the early stop mechanism, and thus GaussML 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 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



(a) Varying #Components



(b) Varying Data Size

Fig. 11: Comparison PCA on Swarm dataset.

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 time 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 [25] on dataset Slice. The HPOBenchmark can simulate the process of hyperparameter selection, data sampling and

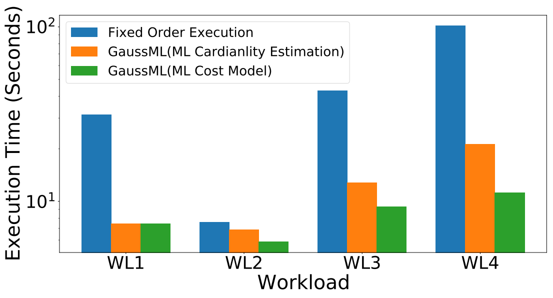


Fig. 12: Comparison on query optimization.

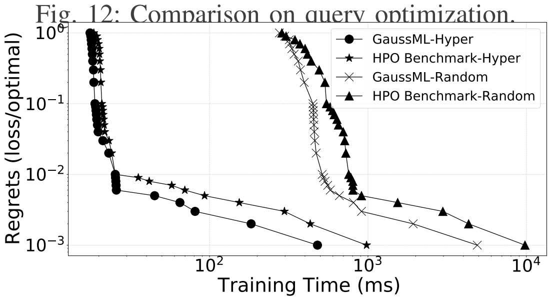


Fig. 13: Comparison training time on hyperparameters.

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 distributed environment.

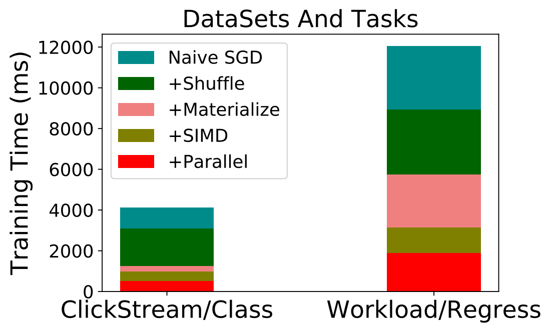


Fig. 14: Training time comparison with optimizations.

Expr10: In Figure 14, we show 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 the training time to halve further.

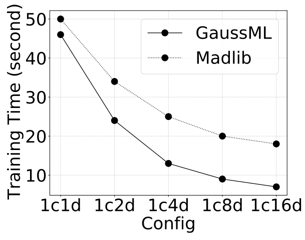
VI. Related Works

Machine learning in database. Recently, in-DB ML has been widely studied. MADlib [36] is composed of a collection of UDFs, and its performance is much lower because of the function cannot be optimized by database. SQLFlow [37] 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 [38], [39] propose optimization methods for model prediction workflow, but they do not discuss model training and they are not fully integrated in modern DBMS.

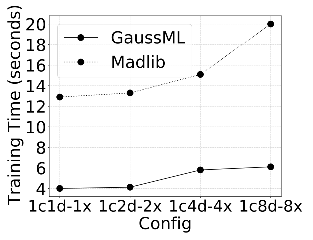
Hyperparemeter Optimization. To reduce the bars of using machine learning, HPO is becoming an inevitable technique in machine learning platform. Methods for machine learning has been proposed in papers [40], [41]. In order to accelerate it, we implement hyperband in GaussML, and GaussML stores the history performance of each configuration for warm-start. There are also some structure search methods proposed in [42], 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 [43] 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], [44]–[46], 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.

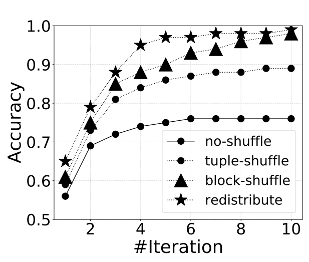


(a) Varying #Datanode

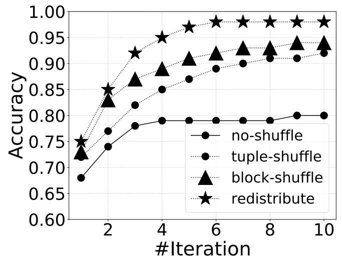


(b) Varying #Datanode and #Data

Fig. 15: Distributed training.



(a) SVM on RLCP.



(b) Logistic Regression on RLCP.

Fig. 16: Comparison on shuffle methods.

VII. 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 MLSQL 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. Moreover, 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.

References

[1] S. Fernandes and J. Bernardino, "What is bigquery?" in *Proceedings of the 19th International Database Engineering amp; Applications Symposium*, ser. IDEAS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 202–203.

[2] T. Schindler and C. Skornia, "Secure parallel processing of big data using order-preserving encryption on google bigquery," *CoRR*, vol. abs/1608.07981, 2016. [Online]. Available: http://arxiv.org/abs/1608.07981

[3] I. F. Ilyas and X. Chu, *Data Cleaning*. New York, NY, USA: Association for Computing Machinery, 2019.

[4] G. Li, "Human-in-the-loop data integration," *Proc. VLDB Endow.*, vol. 10, no. 12, p. 2006–2017, aug 2017.

[5] S. Krishnan, J. Wang, M. J. Franklin, K. Goldberg, T. Kraska, T. Milo, and E. Wu, "Sampleclean: Fast and reliable analytics on dirty data," *IEEE Data Eng. Bull.*, vol. 38, no. 3, pp. 59–75, 2015.

[6] S. Krishnan and E. Wu, "Alphaclean: Automatic generation of data cleaning pipelines," *CoRR*, vol. abs/1904.11827, 2019.

[7] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging ai applications," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 561–577.

[8] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan, "Mlbase: A distributed machine-learning system," in *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013.

[9] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su, "Scaling distributed machine learning with the parameter server," in *OSDI*. USENIX Association, 2014, pp. 583–598.

[10] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, "Data management challenges in production machine learning," in *SIGMOD Conference*. ACM, 2017, pp. 1723–1726.

[11] P. Agrawal, R. Arya, A. Bindal, S. Bhatia, A. Gagneja, J. Godlewski, Y. Low, T. Muss, M. M. Paliwal, S. Raman, V. Shah, B. Shen, L. Sugden, K. Zhao, and M. Wu, "Data platform for machine learning," in *SIGMOD Conference*. ACM, 2019, pp. 1803–1816.

[12] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, F. Xie, and C. Zumar, "Accelerating the machine learning lifecycle with mlflow," *IEEE Data Eng. Bull.*, vol. 41, no. 4, pp. 39–45, 2018.

[13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *OSDI*. USENIX Association, 2016, pp. 265–283.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

[15] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *NeurIPS*, 2019, pp. 8024–8035.

[16] A. Yakovlev, H. F. Moghadam, A. Moharrer, J. Cai, N. Chavoshi, V. Varadarajan, S. R. Agrawal, T. Karnagel, S. Idicula, S. Jinturkar, and N. Agarwal, "Oracle automl: A fast and predictive automl pipeline," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3166–3180, 2020.

[17] Y. Li, Y. Shen, W. Zhang, J. Jiang, B. Ding, Y. Li, J. Zhou, Z. Yang, W. Wu, C. Zhang, and B. Cui, "Volcanoml: Speeding up end-to-end automl via scalable search space decomposition," vol. 14, no. 11, 2021.

[18] G. Li, X. Zhou, J. Sun, X. Yu, Y. Han, L. Jin, W. Li, T. Wang, and S. Li, "opengauss: An autonomous database system," *Proc. VLDB Endow.*, vol. 14, no. 12, pp. 3028–3041, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p3028-li.pdf

[19] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar,

"The madlib analytics library or MAD skills, the SQL," *CoRR*, vol. abs/1208.4165, 2012. [Online]. Available: http://arxiv.org/abs/1208.4165

[20] X. Feng, A. Kumar, B. Recht, and C. Ré, "Towards a unified architecture for in-rdbms analytics," in *SIGMOD Conference*. ACM, 2012, pp. 325–336.

[21] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proc. VLDB Endow.*, vol. 9, no. 3, pp. 204–215, 2015.

[22] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica, "Deep unsupervised cardinality estimation," *Proc. VLDB Endow.*, vol. 13, no. 3, pp. 279–292, 2019.

[23] Z. Wu, A. Shaikhha, R. Zhu, K. Zeng, Y. Han, and J. Zhou, "Bayescard: Revitilizing bayesian frameworks for cardinality estimation," *arXiv preprint arXiv:2012.14743*, 2020.

[24] B.P.Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.

[25] A. Klein and F. Hutter, "Tabular benchmarks for joint architecture and hyperparameter optimization," *CoRR*, vol. abs/1905.04970, 2019. [Online]. Available: http://arxiv.org/abs/1905.04970

[26] "Record linkage comparison patterns data set," 2011.

[27] "Adult data set," 1996.

[28] "Clickstream data for online shopping," 2021.

[29] "Urbangb, urban road accidents coordinates labelled by the urban center," 2019.

[30] "Swarm behaviour data set," 2020.

[31] "Very big and important dataset," 2020.

[32] "Yearpredictionmsd data set," 2011.

[33] "Combined cycle power plant data set," 2014.

[34] "Relative location of ct slices on axial axis data set," 2011.

[35] W. McKinney, "pandas: a foundational python library for data analysis and statistics," 2011.

[36] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar, "The madlib analytics library or MAD skills, the SQL," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1700–1711, 2012.

[37] Y. Wang, Y. Yang, W. Zhu, Y. Wu, X. Yan, Y. Liu, Y. Wang, L. Xie, Z. Gao, W. Zhu, X. Chen, W. Yan, M. Tang, and Y. Tang, "Sqlflow: A bridge between SQL and machine learning," *CoRR*, vol. abs/2001.06846, 2020. [Online]. Available: https://arxiv.org/abs/2001.06846

[38] Y. Lu, A. Chowdhery, S. Kandula, and S. Chaudhuri, "Accelerating machine learning inference with probabilistic predicates," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, G. Das, C. M. Jermaine, and P. A. Bernstein, Eds. ACM, 2018, pp. 1493–1508. [Online]. Available: https://doi.org/10.1145/3183713.3183751

[39] F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas, "Opening the black boxes in data flow optimization," *CoRR*, vol. abs/1208.0087, 2012. [Online]. Available: http://arxiv.org/abs/1208.0087

[40] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, 2012.

[41] L. Li, K. G. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *J. Mach. Learn. Res.*, vol. 18, pp. 185:1–185:52, 2017.

[42] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *CoRR*, vol. abs/1611.01578, 2016.

[43] L. Xu, S. Qiu, B. Yuan, J. Jiang, C. Renggli, S. Gan, K. Kara, G. Li, J. Liu, W. Wu, J. Ye, and C. Zhang, "In-database machine learning with corgipile: Stochastic gradient descent without full data shuffle," in *ACM SIGMOD International Conference on Management of Data (SIGMOD 2022)*, June 2022.

[44] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *NIPS*, 2012, pp. 1232–1240.

[45] X. Chen, T. Chen, H. Sun, Z. S. Wu, and M. Hong, "Distributed training with heterogeneous data: Bridging median- and mean-based algorithms," in *NeurIPS*, 2020.

[46] M. Li, D. G. Andersen, A. J. Smola, and K. Yu, "Communication efficient distributed machine learning with the parameter server," in *NIPS*, 2014, pp. 19–27.

IX. Appendix

A. Data Management

Data management supports the executions of ML operations like distributed training and training data tracing. In this section, we introduce versioned dataset management techniques in GaussML.

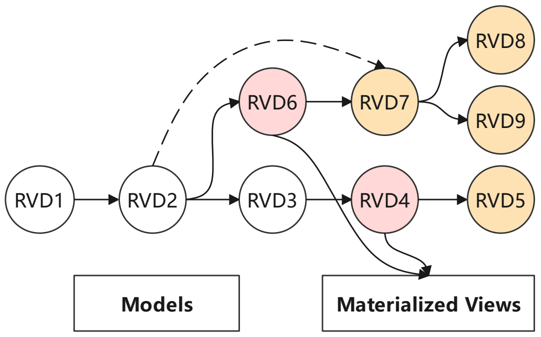


Fig. 17: Resilient Versioned Dataset Link.

B. Versioned Data Management

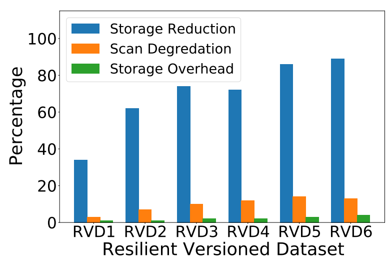
Training data tracing is necessary for model updating, model reproducing and model selection. However, training data tracing is challenging for two reasons. Firstly, training data may be transformed by complicated data preprocess queries, such as data merge, data projection, data aggregation and data cleansing. Secondly, it's costly to materialize all data versions. Considering that training data for a model may be derived from another existing training data, we propose a resilient versioned dataset (RVD) with two modes, materialized and lazy computation, so that GaussML could decide the RVD mode based on the space budget and materializing benefit. For example, if a training data version is transformed by a workflow with extremely high cost computations, it would be better to materialize the RVD so that the derived versions do not need to replay the computations. RVD supports three actions, they are replay, drop and materialize, GaussML will balance the space budget and replay time in the whole system and judiciously select which RVDs should be materialized.

Resilient Versioned Dataset (RVD). RVD is composed of two parts. One is the materialized view in database (which exists only when the RVD mode is *Materialization*), the other is metadata. The view is a relational table in GaussML managed by system user and backend process. The metadata stores following variables, *rvd_id* is the version id of the dataset, *parents* stores the list of versions where current dataset derived, *ref_count* is the number of models trained from this version, *mode* indicates the mode of this RVD (true is materialized, false is lazy computation), *query* stores the computation query for this version in plain text. *view_id* refers to the materialized table in GaussML, and it should be set as 0 if the version mode is lazy computation.

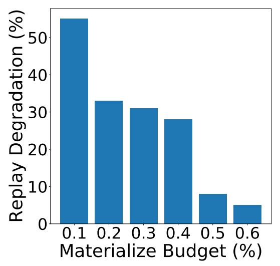
RVD Replay. All RVDs in GaussML is managed by a linked structure. When users designate the model name and replay the training data, the model searches for the RVD metadata based on the *rvd_id* attached in the model. If the RVD for the model is in materialized mode, then GaussML can conduct heap scan on the corresponding materialized table and return tuples to the user. Otherwise, GaussML searches for the nearest materialized precedent RVDs by links and replays the computation queries by backtracking, and finally returns the result tuples. As figure 17 shows, RVD_4 and RVD_6 are materialized, and if users want to get data in RVD_8, then the yellow ones RVD_7 and RVD_8 should be replayed.

RVD Drop. Models could be dropped by users, and RVDs that do not have model references anymore will be cleaned up. Specifically, when GaussML finds a RVD x with *ref_count* 0 in metadata, GaussML drops the corresponding materialized view (if exists), and then merges query to its descendent RVDs (i.e. replace one of the subquery reference notation in the descendent RVDs by current query), and next merges the parents into derived RVDs. In each descendent RVD, GaussML removes *rvd_id* of RVD x from the parents list. Finally, GaussML destroys the RVD x and finishes the vacuum. As figure 17 shows, if RVD_6 is removed, metadata of RVD_6 would be merged into RVD_7 and RVD_2 would connect to RVD_7.

RVD Materialize. There are two approaches to materialize RVDs in GaussML. Users can materialize RVD manually by running SQL to update the mode in certain RVD. Also, GaussML can automatically materialize RVD with high computation cost, and remove materialzied views with high storage cost. GaussML uses a simple and robust strategy to maintain the snapshots of RVDs. Firstly, GaussML reads the space expansion ratio from configurations. Space expansion ratio decides the upper bound of total size of all materialized data (default by 50% of original database). Secondly, GaussML records the replay time for each running RVD, and sorts RVDs in descending order by replay time. Thirdly, GaussML picks one RVD in lazy computation mode with the largest replay time. If the total snapshot size does not exceed the space budget, then GaussML materializes it. Otherwise, GaussML turns some materialized RVDs with less computation time to lazy mode and drops the materialized data until the selected RVD can fit in the space budget. In an extreme case, if current RVD requires larger storage than the space budget, GaussML picks the next RVD and repeats this step.



(a) columns dropping, columns adding, and rows appending.



(b) rows filtering, data transformation.

Fig. 18: Resilient versioned data with various operations.

C. *Experiments of Versioned Data Management*

Expr9: We compare the resilient versioned dataset in GaussML with fully materialized dataset. We conduct the experiments on a dataset with 9 columns and 64M rows. V_0 is the initial dataset. From V_1 to V_7, we modify the dataset incrementally by a sequence of operations. All operations are as the same structure, they drops two columns, adds two new columns and appends a copy of V_0 (with same columns). In figure 18a, the storage reduction is the total space saving by using RVD compared to traditional materialized datasets. The scan degradation shows the growth rate of scanning time for visiting each versioned dataset compared to reading materialized dataset directly. The storage overhead is the space overhead for storing those meta information compared to the added data. From these metrics, we can observe that the resilience versioned dataset can effectively save storage space by 30%-90%, because it only stores the incremental part. However, the cost of replaying certain data increases by 3%-15% because operations like union and merge should be done when recovering the original data from different partitions.

Expr10: Compared to few unions and merges, replaying complex rows filtering and data transformation may cost more time to recover the data. GaussML allows to materialize some hot resilience versioned datasets to reduce the data replay cost. We conduct experiments on 7 RVDs, and operations between every two RVDs are either filtering according to certain column or conducting mathematical expression on rows.V_0 is the initial data. Figure 18b shows the degradation of sequentially replaying all RVDs with different materialize budgets, and we can observe that if GaussML allows materializing 50% of full dataset (complete materialized copies of 7 RVDs), the replay degradation can be reduced below 10%.

D. *Examples of ML interfaces*

For interfaces, GaussML provides extended SQL with ML statements. Snowpark uses SQL + in-memory structure (e.g., Pandas or DataFrame) + Python. MADlib uses SQL + UDF. SparkML uses in-memory structure + Python/Scala/Java. The usage of GaussML is much easier than Python-API based Snowpark and UDF-based Madlib.

Training Interface. In Snowpark, some basic data preprocess operations can be compiled to SQL queries and executed in SQL engine, but users still need to load the intermediate data in a local in-memory structure (e.g. Pandas Dataframe) before training the model, and this restricts the amount of training data and model scalability. The typical usage of Snowpark can be found in website [2], and the following case shows a series separated steps for training a model.

```
# Step 1: Create train and test dataframes
all_data = session.sql("select *, IFF(CLASS
    = 'g', 1.0, 0.0) as LABEL from
    Gamma_Telescope_Data").drop("CLASS")
```

[2]https://docs.snowflake.com/en/developer-guide/snowpark-ml/snowpark-ml-modeling

```
train_data, test_data = all_data.
    random_split(weights=[0.9, 0.1], seed
    =0)

# Step 2: Construct training pipeline with
    preprocessing and modeling steps
FEATURE_COLS = [c for c in train_data.
    columns if c != "LABEL"]
LABEL_COLS = ["LABEL"]

pipeline = Pipeline(steps = [
    ("impute", SimpleImputer(input_cols=
    FEATURE_COLS, output_cols=FEATURE_COLS)
    ),
    ("scaler", StandardScaler(input_cols=
    FEATURE_COLS, output_cols=FEATURE_COLS)
    ),
    ("model", XGBClassifier(input_cols=
    FEATURE_COLS, label_cols=LABEL_COLS))
])

# Step 3: Train
pipeline.fit(train_data)
```

In MADlib, users can take the table name as training input, and then the UDF scans the data for model training, finally the model is written back to a table in database. Madlib doesn't have a uniform interface for different models, sometimes the users may be confused of the meaning of parameters. For example, the logistic training function doesn't have return values, but KMeans training function returns centroids as model. Also, if users want to preprocess the data, a materialized table or view must be created in advance.

```
SELECT logregr_train(source_table,
                     out_table,
                     dependent_varname,
                     independent_varname,
                     grouping_cols,
                     max_iter,
                     optimizer,
                     tolerance,
                     verbose);
);
```

```
SELECT km_model FROM madlib.kmeanspp(
                           rel_source,
                           expr_point,
                           k,
                           fn_dist,
                           agg_centroid,
                           max_num_iterations,
                           min_frac_reassigned,
                           seeding_sample_ratio);
```

GaussML uses one clean SQL with model create and subqueries for all training methods with different data requirement as Section II-C shows.

```
CREATE MODEL xgboost_model USING XGBoost
FEATURES (attr1, attr2)
FROM (SELECT IMPUTE(A, avgA) as attr1,
    IMPUTE(B, avgB) attr2
      FROM (SELECT A, B, avg(A) as meanA,
    avg(B) as avgB
              FROM T1 Join T2
              WHERE C != "LABEL");
```

Inference Interface. In Snowpark, the model predict is similar to python ML library sklearn, users need to call the predict function on the trained model class with materialized data. Snowpark has no way to integrate the model prediction in SQL, and thus it does not support features like prediction push down, and ML&DB co-optimization.

```
# Use the model to make predictions.
predictions = xgboost_model.predict(
    features_df)
predictions[OUTPUT_COLS].show()
```

In `MADlib`, users can apply model prediction for each tuple generated from table or subqueries, this is similar to `GaussML`, but it cannot optimize the execution path of the SQL.

```
SELECT p.id, madlib.logregr_predict(coef,
    ARRAY[1, treatment, trait_anxiety]),
       p.second_attack::BOOLEAN
FROM patients p, patients_logregr m
ORDER BY p.id;
```

`GaussML` still has the most simple and efficient interface for model inference and complex pipeline, for example,

```
SELECT *,
       PREDICT BY xgboost_model as label
          (FEATURES T1.A, T1.B) as l1,
       PREDICT BY svm_model as label
          (FEATURES T2.A, T2.B) as l2,
FROM T1, T2
WHERE l1='K' AND l2 in ['s','w'];
```