PilotScope: Steering Databases with Machine Learning Drivers

Rong Zhu*
Alibaba Group
Hangzhou, China

Lianggui Weng
Alibaba Group
Hangzhou, China

Wenqing Wei
Alibaba Group, USTC
Hangzhou, China

Di Wu#
Alibaba Group, HUST
Hangzhou, China

Jiazhen Peng#
Alibaba Group, ZJU
Hangzhou, China

Yifan Wang#
Alibaba Group, UFL
Hangzhou, China

Bolin Ding*
Alibaba Group
Hangzhou, China

Defu Lian*
USCT
Hefei, China

Bolong Zheng*
HUST
Wuhan, China

Jingren Zhou*
Alibaba Group
Hangzhou, China

ABSTRACT

Learned databases, or AI4DB techniques, have rapidly developed in the last decade. Deploying machine learning (ML) and AI4DB algorithms into actual databases is the gold standard to examine their performance in practice. However, due to the complexity of database systems, the difference between ML and DB programming paradigms, and the diversity of ML models, the tasks of developing and deploying AI4DB algorithms into databases are prohibitively difficult. Most previous works focus on specific AI4DB algorithms and ML models whose deployment requires close cooperation between ML and DB developers and heavy engineering cost.

In this paper, we design and implement PilotScope, an AI4DB middleware with a programming model that largely reduces such difficulties. With a novel abstraction of AI4DB algorithms for, *e.g.*, knob tuning and query optimization, PilotScope consists of two classes of components, *AI4DB drivers* and *DB interactors*, with different programming paradigms and roles in AI4DB tasks. ML developers focus on designing and implementing AI4DB drivers, which are algorithmic workflows that collect statistics from databases, train ML models, make decisions and optimize databases using learned models. AI4DB drivers interact with databases via DB interactors (*e.g.*, for collecting data and enforcing actions in databases). DB developers focus on implementing these interactors on one or more database engines, with the interaction details hindered from ML developers. PilotScope supports a variety of AI4DB tasks, and the implementation of an AI4DB algorithm on PilotScope can be deployed in different databases with only minimum modifications. PilotScope is effective in benchmarking these AI4DB algorithms in real-world scenarios. We hope that PilotScope could significantly accelerate iterating AI4DB research and make AI4DB techniques truly applicable in production.

PVLDB Reference Format:
Rong Zhu, Lianggui Weng,Wenqing Wei,Di Wu, Jiazhen Peng, Yifan Wang, Bolin Ding, Defu Lian, Bolong Zheng and Jingren Zhou. PilotScope: Steering Databases with Machine Learning Drivers. PVLDB, 17(5): XXX-XXX, 2024.
doi:XX.XX/XXX.XX

∗ Corresponding authors, primary contact email address red.zr@alibaba-inc.com
The three authors contribute equally to this paper, listed in alphabetic order.

Proceedings of the VLDB Endowment, Vol. 17, No. 5 doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/alibaba/pilotscope.

1 INTRODUCTION

Learned databases, or AI4DB techniques, aim at enabling automatic, fine-grained, and data-dependent characterization of the action spaces for various database tasks. To name a few, such AI-aided database tasks (AI4DB tasks) include knob tuning [25, 44, 57], index recommendation [1, 13, 16, 19, 30, 45, 49, 52], query optimization [24, 29, 39, 40, 53–55, 60, 63], index structures [21, 22, 31], monitoring [37, 59], *etc.* A considerable amount of research efforts, ranging from theoretical analysis, model and algorithm design to benchmark evaluation, have been devoted to this field.

Challenges of Deploying AI4DB Algorithms. We can fairly benchmark various AI4DB algorithms [23, 41, 46] and evaluate their values in production only by deploying them into actual databases. However, due to the complexity of database systems, the different ML and DB programming paradigms, and the diversity of ML algorithms, the tasks of developing and deploying AI4DB algorithms are prohibitively difficult. Most previous works focus on AI4DB tasks in some specific database engine. For example, [41] deploys the learned query optimizer Bao [39] on SCOPE [15] (the distributed computing system in Microsoft); [34] integrates a number of learned techniques, including query rewriter, cost model, and view advisor, into openGauss [9]; [14, 36, 42] deploy workload forecasting, behavior modeling, data collection and other functions into the NoisePage DBMS [5]. Even for one DB engine, the deployment of AI4DB techniques requires close cooperation between ML and DB developers and heavy engineering cost.

There are some practical challenges and considerations:

- *The ML and DB communities have very different programming paradigms (*e.g.*, PyTorch v.s. C).* Although we expect that the gap is closing, it is rare that one developer masters both. Thus, it is important to have an abstraction of the AI4DB system so that: 1) ML and DB developers can focus their own components; 2) it could be appliable to a wide range of database systems; and 3) their works can be re-used by others in different AI4DB tasks.

  We expect that an AI4DB solution can be easily deployed and extended for different DB engines and tasks. The hope is that, when the solution is transferred from one engine to another (*e.g.*, from PostgreSQL to Spark), only DB developers' work needs to be adapted to the new engine, while ML developers can focus on iterating algorithms and models for various AI4DB tasks.

- *An AI4DB task usually involves a long pipeline. During this pipeline, the AI4DB algorithms may need to interact with internal functions of the database multiple times (*e.g.*, collecting training data and enforcing actions in the database).* Any implementation detail may affect the end-to-end performance of a solution significantly. Thus, for a fair benchmark of different solutions, it is better for them to share as many atomic operators as possible (*e.g.*, how data is fetched from and pushed into the database).
- *Existing AI4DB solutions are designed under different technical routines to address different problems. Their requirements are very diversified and complex.* For example, knob tuning algorithms are typically applied to optimize the database performance for the entire workload, while a learned query optimizer is designed to speed up each singleton SQL query. Some AI4DB tasks ask for customized functionalities, *e.g.*, periodical model retraining. Therefore, it is very necessary to design a flexible yet easy-to-use framework that accommodates all these diversified requirements of AI4DB algorithms.

In this paper, we design and implement `PilotScope`, a desirable system aiming at resolving the problem of deploying AI4DB tasks into databases. `PilotScope` takes all the above practical issues into consideration and resolves these challenges. In the following, we first describe a sample of representative AI4DB tasks and summarize their common elements in Section 1.1 to lay a foundation for `PilotScope`. Then, we overview the main features and contributions of `PilotScope` in Section 1.2.

1.1 Sample and Elements of AI4DB Tasks

In this paper, we focus on AI4DB tasks for query processing since these methods are growing rapidly and have significant importance to database performance [23, 33, 50, 62]. As shown in Figure 1, for any database, we often tune its configurations, including knobs, indices, views and flags, to make preparations for downstream query processing. Then, for the input query Q, it is fed into the *query rewriter* to transform it to an equivalent query Q' with better execution efficiency. The query Q' is then parsed and fed into the *query optimizer* to generate an execution plan P. The query optimizer often estimates the cardinality, i.e., the number of tuples, of each sub-query of Q', predicts the cost of each candidate plan P and enumerates the join orders of tables to select the plan P^* with the minimum predicted cost for execution. Finally, the *query executor* executes plan P^* and returns a set of tuples $\mathcal{R}$ as the result.

Learned methods have been applied to replace one or more components in this workflow (marked in green in Figure 1), including but not limited to: learned knob tuning, index or view recommendation; learned query rewriter; learned cardinality estimator, cost model and join search methods; and learned end-to-end query optimizer. We describe the procedures of them as follows.

1.1.1 Learned Knob Tuning. This task is required by database users. It targets to automatically tune the values of a set of knobs K to optimize the database performance, which is measured by a metric M, on a query workload W. The method often works in iterations. Each time it executes all queries $Q \in W$ on the database, collects their execution statistics related to metric M and gradually updates the value of knobs using an ML model, such as a reinforcement learning (RL) model [57] or a Bayesian model [25, 44]. After finishing iterating, the generated value of knobs K are set accordingly into databases. The learned index recommendation and view advisor [26, 56] work in similar ways as knob tuning.

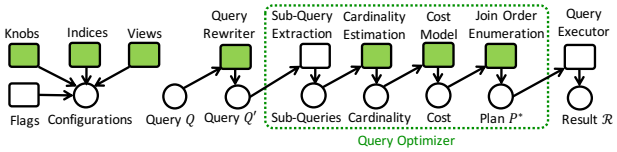


Figure 1: Workflow of query processing.

1.1.2 Learned Cardinality Estimator. It takes a (sub-)query Q as input and trains an ML model to estimate its cardinality. The ML model could be data-driven [24, 54, 63] or query-driven [29]. The data-driven models are unsupervised models building the joint distribution of all attributes while the query-driven ones are supervised models mapping featurized queries into the estimated cardinality. The models are updated periodically to keep track of data/workload changes. In similar, the learned query rewriter [12, 17, 38], cost model [28, 35] and join order search methods [18, 48] also apply ML models to replace the traditional database components.

1.1.3 Learned Query Optimizer. These methods aim at generating an execution plan P for the input query Q using ML techniques. They often train a model using collected historical data to predicate the execution time (or relative goodness) of a plan. Then, the methods generate a number of candidate plans for Q using different strategies. Specifically, some methods [40, 53] directly search the plan space by themselves and some other ones [39, 55, 60] tune the traditional query optimizer with different knobs to produce different plans. The plan with the minimum predicted time is returned for execution. Meanwhile, its execution statistics are collected to update the prediction model periodically.

Based on the workflow of these AI4DB tasks, we summarize the following elements required to deploy an AI4DB task:

1) We need a flexible data collection interface to extract different kinds of statistical data, *e.g.*, cardinality, plan cost, and execution time, from databases for model training.

2) We require a mechanism to inject the AI4DB task into databases to replace the original component. The injection mechanism is responsible for invoking the AI4DB task when needed, providing the inputs to the AI4DB task and sending its output to the databases.

3) We need an ML environment to support the execution of the AI4DB algorithms and the model inference, training and updating.

4) During the execution, we should have an interface to interact with the database so that the AI4DB task could operate the databases and exchange data. For example, tuning the knobs and obtaining the produced plans in a learned query optimizer.

1.2 Features and Contributions of `PilotScope`

`PilotScope` is a middleware to deploy AI4DB tasks on top of any native database. It provides all required elements to deploy AI4DB tasks and attains a number of attractive features, including:

- `PilotScope` is easy-to-use for database users. The database user could access `PilotScope` to start any AI4DB task as needed and operate the database as usual. The execution of any AI4DB algorithm is totally transparent to the database user.

- PilotScope enables ML and DB developers to work independently to play their own strengths in developing ML and DB programs. They do not need to learn and care about the details on the other side.
- PilotScope attains high generality. It could support deploying a variety of AI4DB tasks on different database systems. Moreover, it could be easily extended to new tasks and new systems.

Due to space limits, we put a detailed comparison between PilotScope and related works, *i.e.*, composable DBMS [43] and other deployment systems [14, 34, 36, 41, 42], in Appendix A of the full paper [61]. Our contributions in PilotScope, including design, implementation and applications, are summarized as follows:

1) We propose a comprehensive framework in PilotScope to facilitate the implementation of AI4DB algorithms and their deployment into databases. PilotScope abstracts the requirements of multiple typical AI4DB algorithms and is highly user-friendly to support them in a unified manner.

2) We design a novel mechanism to allow ML and DB developers to work independently in PilotScope. This is achieved by abstracting a DB interactor interface between AI4DB algorithms and databases. The ML developers could focus on writing AI4DB algorithms using this interface without caring about the database details. The same copy of an AI4DB algorithm could steer any database in a unified manner. On the DB side, the DB interactor is implemented as lightweight patches on each steered database by its native DB developers. In different databases, the DB interactor is implemented by DB developers in different ways.

3) We highly abstract several simple but robust operators, namely push and pull, for data interaction between ML algorithms and databases. Through these two operators, PilotScope can utilize a simple flow to collect various types of data, inject AI4DB tasks and enforce actions in the databases. We have implemented a number of specific push and pull operators, such as sub-query extraction and setting cardinality, in the query optimization workflow.

4) We provide an open-source implementation of PilotScope. In our open-source repository, we implement 15 algorithms in four representative AI4DB tasks, namely knob tuning, index recommendation, cardinality estimation and end-to-end query optimizer, in PilotScope and apply them to steer two well-known databases, *i.e.*, PostgreSQL and Spark. Moreover, we show our system design and implementation in PilotScope are easily extended to support other AI4DB tasks and database systems.

5) We conduct a comprehensive evaluation to show the benefits and overheads by deploying AI4DB tasks in databases using PilotScope. The experimental results demonstrate that the overhead brought by PilotScope is negligible. Whereas, the benefits and shortcomings of different AI4DB algorithms are clearly exposed through benchmarking, which could serve as valuable insights for future research iterations.

Organization. Section 2 overviews the PilotScope system. Section 3 introduces the detailed programming model in PilotScope. Section 4 exhibits how to apply PilotScope for a number of representative AI4DB tasks. Section 5 describes the implementation details. Section 6 reports the evaluation results on PilotScope. Section 7 concludes the paper and discusses future work.

2 SYSTEM OVERVIEW

2.1 A Quick Start

PilotScope is a middleware that largely reduces the difficulties of applying and developing AI4DB tasks in databases. To DB users, PilotScope is very easy-to-use by accessing its console and typing several commands (green arrows in Figure 2(b)). First, the user types a command to ask PilotScope to start an instance to connect and log into the database. After establishing the connection, PilotScope takes control of the database. Then, the user types another command to start an AI4DB task, *e.g.*, learned query optimizer, on the database. After that, the user could operate the database with this connection as usual. PilotScope could identify the commands and send them to databases to execute if needed. Whenever the AI4DB task needs to be invoked, *e.g.*, the user executes a SQL query, the database would automatically apply the learned query optimizer to replace the traditional component. To database users, PilotScope and the underlying AI4DB algorithms are totally transparent. They just need to learn several new commands and experience the benefits of AI4DB algorithms.

In this section, we show the system architecture of PilotScope in Section 2.2. Then, we introduce its detailed workflow and working paradigm in Section 2.3 and Section 2.4, respectively.

2.2 System Architecture

The system architecture of PilotScope is shown in Figure 2(a). PilotScope provides a *console* to operate the whole system. It manages multiple AI4DB *drivers* and the *DB interactor*. The DB interactor contains an *interface* connecting AI4DB drivers with databases. Each database steered by the AI4DB drivers is attached with its specific *implementations of DB interactor*. The role of each component is described as follows.

2.2.1 AI4DB Driver. In PilotScope, each AI4DB task, which targets to replace a database component, is packaged as a *driver*. Each driver contains: 1) an *AI4DB algorithm* describing the algorithmic workflow, *e.g.*, how to learn to generate the execution plan in a learned query optimizer; 2) one or more *ML models* to be consulted in the AI4DB algorithm, *e.g.*, the execution time prediction model in a learned query optimizer; and 3) the *training and inference function* for each ML model. The algorithms and models in the driver are all written in an AI-friendly language, e.g., Python. The PilotScope integrates a *runtime environment* with the third-party dependencies and libraries to support their execution.

2.2.2 Interface of DB Interactor. This component shields the underlying details of different databases and serves as a unified bridge for drivers to interact with the databases. We abstract two operators, namely *push* and *pull*, to meet the requirements of AI4DB tasks (see Section 1.1).

The push operator allows AI4DB tasks to send requests to the databases while the pull operator extracts data from the databases. Through their cooperation, the AI4DB task could enforce actions in database and exchange data with the database. Our push and pull operators are more powerful and general than APIs provided by databases. In our design, we could apply multiple push and pull operators in a logic order, *e.g.*, we could set the database flags using push, execute a query, and then obtain its generated plan and

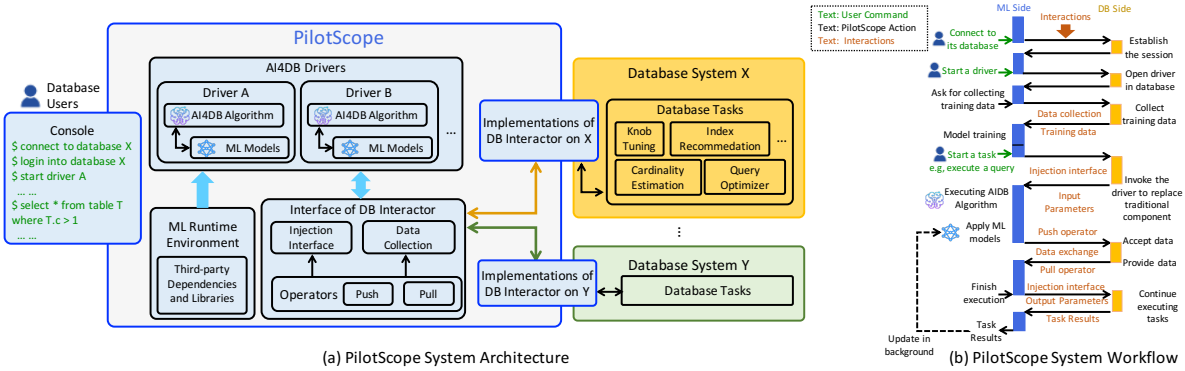


Figure 2: System architecture and workflow of `PilotScope`.

execution time using two pull operators. The details to apply these operators are introduced in Section 3.

The push and pull operators are also applied to inject AI4DB tasks into databases. Based on them, `PilotScope` abstracts the injection interface as a configurable parameter for AI4DB drivers. The driver only needs to specify the type of injection interface for itself, *e.g.*, a learned knob tuner or a learned cardinality estimator. Then, `PilotScope` would automatically invoke its AI4DB algorithm to replace the original database component when needed.

Besides, we also apply the push/pull operators to collect data, *e.g.*, estimated cardinality, plan cost and query execution time, from databases for model training. We also discuss how to apply them in Section 3 and defer the implementation details to Section 5.

2.2.3 Implementations of DB Interactor on Databases. The DB interactor is implemented in different ways on different databases. However, all of them satisfy the same interface of DB interactor and fulfill the required functions. In such a way, each driver could apply the interface to steer different databases with minimum modifications. In practice, the implementations are often developed as lightweight patches, *e.g.*, hook functions, to the database codebase so that the changes incurred to the database kennel are minimal.

2.3 Workflow of `PilotScope`

As shown in Figure 2(b), after the database user establishes the database connection and starts a driver through `PilotScope` console, the driver starts to collect the pre-defined training data from databases. For example, the query execution time would be collected for training the time prediction model in the learned query optimizer. After collecting enough data, the driver would start to train each model using the provided training function. As an option, the users could tune the hyper-parameters before model training.

After finishing model training, the AI4DB algorithm in the driver would wait to be executed. When needed, it would be invoked by `PilotScope` to replace the original database component, *e.g.*, the learned query optimizer is called whenever the database user executes a SQL query. It obtains the inputs from the injection interface (*e.g.*, the query), executes its algorithm in the runtime environment in `PilotScope`, consults the ML models using the provided model inference functions and interacts with the database through push/pull operators to fulfill its job. After it finishes executing, the injection interface sends its result (*e.g.*, the selected execution plan) to the database. For some drivers, the ML models are updated in the background to keep track of database changes.

Notably, for all drivers designed for the same task, *e.g.*, different learned cardinality estimation methods, `PilotScope` allows to open at most one driver at any time or else they would make conflicts. For different tasks, `PilotScope` supports opening multiple drivers for them at the same time, *e.g.*, a learned knob tuner and a learned cardinality estimator in Section 6.4. Due to the stability and unpredictable behaviors of ML models [23, 50], `PilotScope` also provides a *validation mode* to safely tune and test AI4DB tasks. In this mode, the AI4DB task could be applied as usual. However, when quitting this mode, `PilotScope` would automatically recover the database settings (*e.g.*, knobs or indices) as original.

2.4 A Collaborative Working Paradigm

The whole working paradigm to integrate an AI4DB algorithm into the codebase of a database includes the designing, developing and maintenance phases. In existing systems [34, 41, 46], ML and DB developers need to work closely and know about each other in the whole paradigm. Whereas, `PilotScope` allows ML and DB developers to work more independently and focus on their own fields, especially in the developing phase.

Designing Phase. As shown in Figure 3, at the initial stage, the ML researchers design the AI4DB algorithm and ML models to solve the database task. During this phase, they need to learn about the background knowledge of the database task to formulate it into an ML problem and select proper features, models and loss functions to solve it. We note that the ML researcher often just requires high-level knowledge, *e.g.*, the task inputs, outputs, traditional approaches and constraints, but does not need to know about the implementation details. Thus, the ML researchers could consult the DB developers in a conceptual view or refer to a database seminar book to acquire such knowledge by themselves. The connections between the ML researchers and the DB developers are not tight.

Developing Phase. Then, the ML developers (possible the ML researchers themselves) define the AI4DB driver for the target database task and configure its injection type. After that, they write the AI4DB algorithm, inference and training functions for each model in the AI-friendly language. In this process, the existing interface of DB interactor may not meet their requirements. For example, there does not exist an injection interface for such a task or a push/pull operator to transfer a new type of data. The ML

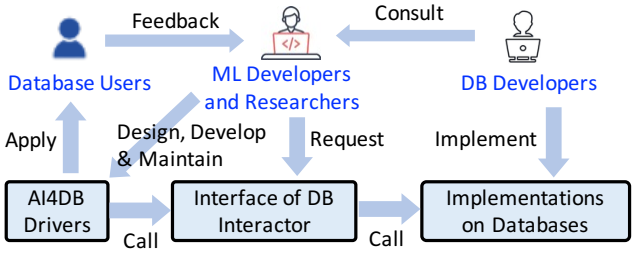


Figure 3: The collaborative working paradigm in PilotScope w.r.t. different roles.

developers just define the inputs and outputs of these new interfaces and send the requests to the DB developers.

On the DB side, the DB developers for a native database are responsible for implementing the requested interfaces on databases. These developers are familiar with the underlying codes of databases, so they could find the best way to implement the interfaces as light-weight patches using the database language. Notably, implementing these patches is much easier than integrating a complete ML program into databases. What they do is just like their ordinary work on developing new features on the native database.

In this paradigm, the ML developers are liberated from the database details. The only connection between ML and DB developers is the interface of DB interactor. In our open-source PilotScope, we have defined the interface for some representative tasks (see Section 4) and implemented the interface on PostgreSQL and Spark (see Section 5). In the future, the interface of DB interactor and related implementations could be gradually enriched by the open-source community and commercial database companies.

Maintenance Phase. After an AI4DB driver is deployed, the ML researchers could also update models and monitor their performance according to user feedback through PilotScope. For model updating, we provide commands in PilotScope console to allow researchers (or users) to tune the model hyper-parameters and retrain the model manually. The researchers could also pre-define some events in the AI4DB algorithm to update the model automatically. For example, in Section 4.4, for the learned query optimizer, we apply a timer to retrain the model periodically to keep track of data and workload changes. For performance monitoring, the researchers are required to bury some functions in the AI4DB algorithm to export necessary statistical data. Then, users could access such data for monitoring. In our future work, we would integrate more off-the-shelf tools with PilotScope to improve the convenience of performance monitoring.

3 PROGRAMMING MODEL

We introduce the programming APIs for DB interactor operators, ML model and driver. Figure 4 and Figure 5 list all programmable APIs and some examples. We describe the details as follows.

DB Interactor Operator APIs. In PilotScope, we use the concept of a *session* to define the interaction process between the AI4DB algorithm and the database. In each session, the PilotScope creates and maintains a new connection *e.g.*, a database session, to enforce actions in the database. Each session is created with the API session_start(), which returns a session_id to mark this session,

```
def session_start():
    # create a new session and return its session_id
    ... ...
def session_end(session_id):
    # close the session specified by session_id
    ... ...
def session_exec(session_id):
    # execute all operators registered before it
    ... ...
def push(session_id, data_type, data_name, data_value)
    # send data from ML side to the database side
    # data_type: type of the data, including FLAG, INDEX, VIEW,
    # QUERY, SUB_Q, CARD, COST, PLAN, TIME
    #  FLAG, INDEX, VIEW: database flags, indices and views
    #  QUQEY: SQL query, SUB_Q: sub-queries of input query
    #  CARD, COST, PLAN, TIME: estimated cardinality, plan cost, execution plan
    ↪  and plan execution time of the input query
    # data_name: only appliable for FLAG, null for others
    # data_value: the value of the data
    ... ...
def pull(session_id, data_type, data_name)
    # acquire data from database side to the ML side
    ... ...
def interaction_example(x, y, Q, C)
    sid = session_start()
    push(sid, FLAG, x, y)
    push(sid, QUERY, null, Q)
    push(sid, CARD, null, C)
    P = pull(sid, PLAN, null)
    t = pull(sid, TIME, null) # P and t are placeholder noew
    sesssion_exec(sid) # execute the above push and pull operators
    # The value of P and t are obtained only after sesssion_exec
    Do sth. using P, t
    session_end(sid)
```

Figure 4: Programming APIs of DB interactor operators.

and is released by session_end(). In each session, we could enforce the actions to databases, *e.g.*, updating its configurations, sending or acquiring data, using multiple push() and pull() operators.

The parameters of the two operators include session_id, data_type, data_name and data_value. Here, data_type refers to the type of data we send to or acquire from the database. We define a variety of data types to support the AI4DB tasks in the query processing workflow (see details in the comments in Figure 4). The parameter data_name is only applicable for flags, which refers to the specific flag name, *e.g.*, max_memory_size. It is set to null for other data types. For the push operator, the parameter data_value indicates the value of the data we send to the database.

Notably, in each session, we could apply multiple push() and and pull() operators. However, they are not executed immediately but just define the logic order of the operators. We apply the API session_exec() with a session_id to execute all operators registered in front of it. In such a way, we could exchange multiple types of data and/or enforce different actions for the same query.

For example, in the interaction_example() function in Figure 4, we create a new session sid. We first use push(sid, FLAG, x, y) to set the database flag x to value y. Then, we push a query Q to execute and apply push(sid, CARD, null, C) to replace its estimated cardinality to C. After that, we could acquire the newly generated plan P and its execution time t of query Q using two pull() operators. Notably, at this time, P and t are just two placeholders. Their value would be instantiated after invoking the session_exec() API. Finally, we close this session using the session_end() API.

Model and Driver APIs. For each new driver class, it needs to override the init() and algo() functions. The init function initializes the object of this driver. Specifically, we need to do the following necessary jobs in it: 1) specify the injection type, *e.g.*, a member variable, of the driver class; 2) initialize the models to be used in the AI4DB algorithm; and 3) collect training data and train ML

```
def collect_data(data_type, data_size, data_addr):
    # collect raw data from database for model training
    # data_type: CARD, COST, PLAN, TIME
    # data_size: the size of data to be collected
    # data_addr: the address to store the collected data
    ... ...
class model(args):
    def model_train(train_data, args): # model training
        ... ...
    def model_inference(inference_data, args): # model inference
        ... ...
class driver(injection_type, args):
    # inference_type: the injection type for the task,
    # including INJ_KNOB, INJ_INDEX, INJ_CARD, INJ_E2EQO
    def init():
        self.injection_type = injection_type
        Collect data to perform model training
        Add customized functions here
        ... ...
    def algo(args):
        Do sth. according to the AI4DB task... ...
```

Figure 5: Programming APIs of model and driver.

models. Developers could also add customized codes to implement other functions,*e.g.*, periodic model updating using a timer. For the injection type, we provide four options for the sample applications in Section 4, namely INJ_KNOB, INJ_INDEX, INJ_CARD and INJ_E2EQO to inject the learned knob tuning, index recommendation, cardinality estimation and end-to-end query optimizer, respectively.

The algo() function, which describes the AI4DB algorithm, applies ML models and DB interactor operators to accomplish the AI4DB task and is called by `PilotScope` when needed.

For training data collection, we provide the API collect_data() to obtain statistical data from the databases. It has three parameters: 1)data_type for the type of data to be collected. We provide the option CARD, COST PLAN and TIME to obtain estimated cardinality, plan cost, execution plan and query execution time, respectively; 2)data_size indicates the size of data to be collected; and 3)data_addr refers to the address to store the collected data. Users could filter or process the collected raw data for model training. Both our injection interface and training data collection are implemented using the DB interactor operators. We discuss the implementation details and the extension method in Section 5.

4 SAMPLE APPLICATIONS

In this section, we exhibit how to apply `PilotScope` for specific AI4DB tasks. In our open-source repository, we implement 15 ML-based algorithms for four representative tasks mentioned in Section 1.1. Notably, the AI4DB drivers for knob tuning and index recommendation work on the query workload level. They aim at configuring the databases to optimize their overall performance on the workload and are invoked upon user requests, *e.g.*, the query workload characteristics have significant changes. Whereas, the AI4DB drivers for cardinality estimation and query optimizer work on each singleton SQL query. They accept each input SQL query and inject the cardinality or execution plan generated by the AI4DB methods to improve its performance. Aside from these well studied problems, we also show `PilotScope` could support some new AI4DB tasks. Due to space limits, in Appendix B of the full paper [61], we apply our `PilotScope` to deploy an AI4DB method proposed in [51] to eliminate regressions on learned query optimizers. In the future, `PilotScope` will be continuously updated to stay aligned with the state-of-the-art literature in AI4DB.

```
class model_knob():
    model_train(train_data):
        ... ...
    model_inference(X, v, M):
        ... ...
        return predicted_metic # predicting metric M when applying value v of X

class driver_knob(K, S): # opened and invoked by user request
    def init():
        self.injection_type = INJ_KNOB
        self.knob_model = model_knob() # pre-trained offline
        ... ...
    def algo(X, W, M): # tune knobs X in K with workload W to optimize metric M
        Set the initial value v of X
        sid = session_start()
        for i in range(MAX_ITER):
            push(sid, FLAG, X, v)
            session_exec(sid)
            for each Q in W:
                push(sid, QUERY, null, Q)
                exec_time = pull(sid, TIME, null) # by metric M
                session_exec(sid)
            Update self.knob_model using W, v, exec_time
            for each new value n_v:
                predict.add(nv, self.knob_model.inference(X, n_v, M))
            v = argmin (nv, predict)  # update the new value to v
        session_end(sid)
        return v
```

Figure 6: The `PilotScope` model and driver for knob tuning.

4.1 Knob Tuning

Let $K = \{k_1, k_2, \ldots, k_n\}$ be a set of tuneable system knobs in the databases, e.g., work memory size and maximum connections. For the knob k_i, let S_i denote its domain size. Let $S = S_1 \times \cdots \times S_n$. The knob tuning problem asks to find a combination of knob values $(s_1, s_2, \ldots, s_n) \in S$ such that an objective performance metric M, e.g., query execution time or system throughput, is maximized.

`PilotScope` could well support the general workflow of knob tuning algorithms discussed in Section 1.1.1. As shown in Figure 6, we define a driver driver_knob with the parameters K and S on all tuneable knobs and their domain. It applies the injection type INJ_KNOB. In `PilotScope`, the user could apply the command "auto knob tune -X, -W -M" to tune knobs $X \subseteq K$ on databases using workload W to optimizer metric M. The query workload W could be regarded as the training data for the driver. The injection interface would pass the parameters to the algo() function in driver_knob and execute it immediately.

In the init() function of driver_knob, it builds an ML model model_knob to predict the metric M when applying value v for knobs $X \subseteq K$. This model is often pre-trained offline and would be gradually updated in executing the algorithm. In the algo() function, it first sets an initial value v for knobs in X. Then, we start a database interaction session to set the value of knobs X to v, execute each query $Q \in W$ and collect its metric M, e.g., the query execution time, by the push() and pull() operators. After finishing execution, we use the exact value of metric M to refine the model parameters, predict the metric value of each candidate and update the knob value v to the new value with the minimum predicted number. Finally, the knob value v of X is returned. The injection interface is responsible for setting the knob value in databases (see implementation details in Section 5). Notably, such drivers are not frequently invoked and the iteration over the training workload W could be done in the background in databases. Therefore, it would not consume too many resources to affect the normal service of the databases.

This framework could support a variety of advanced knob tuning algorithms mentioned in [27]. We implement two Bayesian optimization methods SMAC [25] and GP-BO [44] and an RL-based

```
class driver_index(): # opened and invoked by user request
    def init():
        self.injection_type = INJ_INDEX
        ... ...
    def algo(C, W, B): # select indices from C with workload W under budget B
        Set the initial D of candidate columns
        sid = session_start()
        for i in range(MAX_ITER):
            push(sid, INDEX, null, D)
            session_exec(sid)
            for each Q in W:
                push(sid, QUERY, null, Q)
                plan_cost = pull(sid, COST, null)
                session_exec(sid)
            Update D according to plan cost with some strategies
        session_end(sid)
        return D
```

Figure 7: The PilotScope driver for index recommendation.

method DDPG [57]. They have been shown to have better performance on DBMS tuning in recent studies [27, 58].

4.2 Index Recommendation

Let $C = \{c_1, c_2, \ldots, c_m\}$ denote a number of columns where we could build an index. For column c_i, we denote its index size as $s(c_i)$. For any query Q, let $\text{cost}(Q, D)$ denote the cost of executing Q with indices on all columns in $D \subseteq C$. Given a workload W, the index recommendation problem asks to find the D such that: 1) the index size $\sum_{c \in D} s(c) \leq B$ is no greater than the budget B; and 2) the cost $\sum_{Q \in W} \text{cost}(Q, D)$ of executing W is minimized.

Similar to knob tuning, PilotScope could also build a driver driver_index for index recommendation. It applies the injection type INJ_INDEX. In similar, the user applies the PilotScope command "auto index recom -C, -W -B" to recommend indices on columns C using the training workload W under budget B.

As shown in Figure 7, the AI4DB algorithm in driver_index also works in iterations from an initial set $D \subseteq C$. Then, we start a session and use the push() operator to set the database index to D before each loop. We execute all queries $Q \in W$ and collect their estimated cost. After that, the candidate columns D are updated using some strategies and returned in the final. We implement multiple algorithms with different index updating strategies in our open-source repository, including: 1) AutoAdmin [16], DB2Advisor [49], DTA [1] and Extend [45] starting with the empty set and adding indices each time; 2) Drop [52] and Relaxation [13] starting with a large set and reducing indices each time; and 3) linear programming, such as CoPhy [19]. AutoAdmin (also DTA) and DB2Advisor have been applied for commercial databases SQL Server and DB2, respectively. Some work [20] also applies learned models to search and update the index in each iteration. We could also deploy the learned view advisor task in PilotScope in a very similar way to index recommendation. We omit the details due to space limits.

4.3 Learned Cardinality Estimation

Unlike with knob tuning and index recommendation, cardinality estimation needs to be called whenever the database executing a SQL query. As shown in Figure 8, in the driver class driver_card, we set its injection type to INJ_CARD in the init() function. We build an ML model model_card for cardinality estimation. For query-driven and data-driven methods, the models are trained using different data. Specifically, for query-driven models, we apply the API collector_data() to collect the estimated cardinality of queries for model

```
class model_card():
    model_train(train_data):
        ... ...
    model_inference(Q): # return estimated cardinality for query Q
        ... ...
        return est_card

class driver_card(): # opened by user, invoked for each input SQL query
    def init():
        self.injection_type = INJ_CARD
        train_data = collect_data(CARD, MAX_SIZE, data_addr)
        # for data-driven models, access the tables for train_data
        self.card_model = model_card()
        self.card_model.model_train(train_data)
        Collect train_data when needed
        Set timer to call model_train() periodicallly when needed
        ... ...
    def algo(Q): # input query Q
        sid = session_start()
        push(sid, QUERY, null, Q)
        SQ = pull(sid, SUB_Q, null)
        session_exec(sid)
        new_cards = self.card_model.model_inference(SQ)
        session_end(sid)
        return SQ, new_cards
```

Figure 8: The PilotScope model and driver for cardinality estimation.

training. For data-driven models, we could directly access the data in database tables to build the unsupervised models. Sometimes, the model needs to be updated. At this time, we could start a background thread to continuously collect training data and set a timer to invoke the model_train() function for updating in periodical.

In the algo() function, for the input query Q (passed by the injection interface), we estimate the cardinality of each sub-query Q' of Q. To this end, our pull operator provides the function to obtain all sub-queries SQ of Q in a batch manner. This would save the communication cost but need to adapt the traditional databases a bit. After estimating the cardinality of all queries in SQ, we return them together with the new cardinality. Notably, our injection interface also supports replacing the cardinality of all sub-queries in a batch manner. We defer the implementation details in Section 5.

Our driver_card could seamlessly support any learned cardinality estimation method. In our open-source repository, we integrate several representative methods, namely MSCN [29], NeuroCard [54] and DeepDB [24] using supervised deep neural networks, unsupervised auto-regression model and sum-product networks, respectively. They have exhibited superior performance in some benchmark evaluations [23, 47]. Other learned components in query optimizer, such as query rewriter, cost model or join order search method, could also be deployed using PilotScope in a similar way.

4.4 Learned Query Optimizer

PilotScope also supports deploying learned end-to-end query optimizers to directly generate plans for SQL queries. As discussed in Section 1.1.3, different methods apply different strategies for candidate plan search and different models for plan selection. Specifically, Neo [40] and Balsa [53] search the plan search by themselves using a (bounded) best-first search strategy and an ML model to predict the execution time of each (partial) plan. Therefore, their training cost and query optimization time are very long. Alternatively, Bao [39], HyperQO [55] and Lero [60] tune the native query optimizer with different knobs to generate a set of candidate plans. Their performance is shown to be much better than others [39, 60]. We present the pseudocodes to implement Bao in Figure 9. Lero and HyperQO could be implemented in a similar way.

```
class model_prediction():
    model_train(train_data):
        ... ...
    model_inference(P): # return estimated execution time for plan P
        ... ...
        return est_exec_time

class driver_bao(H): # opened by user, invoked for each input SQL query
# parameter H: set of hints for the database
    def init():
        self.injection_type = INJ_E2EQO
        train_data = collect_data(TIME, MAX_SIZE, data_addr)
        self.predict_model = model_prediction()
        self.predict_model.model_train(train_data)
        Collect train_data
        Set timer to call model_train() periodicallly
        ... ...
    def algo(Q): # input query Q
        for each h in H:
            sid = session_start()
            push(sid, FLAG, hint, h)
            push(sid, QUERY, null, Q)
            Ph = pull(sid, PLAN, null)
            session_exec(sid)
            session_end(sid)
            PS.add(Ph)
        plan_time = self.predict_model.model_inference(PS)
        P = argmin(PS, plan_time)
        return P
```

Figure 9: The PilotScope model and driver for end-to-end query optimizer Bao in [39].

In our driver_bao class, we set its injection type to INJ_E2EQO. We apply an ML model to predict the execution time of each plan and collect query execution time for model training. According to its requirements, we collect data in the background and retrain the model in periodical. For the query Q, Bao tunes the native query optimizer with different flags of hints to enable/disable certain operators, *e.g.*, hash_join or index_scan, to generate different candidate plans. By enabling or disabling certain operators, we could enforce the native database to try different code paths and produce possible better plans. To this end, we require to pass the set of all possible hints H as an input parameter for the driver_bao class. This allows us to apply different hints for different databases.

In the algo() function, for each hint $h \in H$, we start a new session to set hint flags to h, send the query Q to the database and obtain the generated candidate plan Ph. Notice that, unlike knobs and indices, tuning the flags of hint would only affect the current database session. After obtaining all candidate plans, we consult the prediction model to find and return the best plan P. The injection interface is responsible for sending P to a database for execution.

The HyperQO method applies flags of leading hints on join orders. It could be implemented by just replacing the hint flag h with the leading hint flag. Lero has two differences with Bao. First, it applies a learning-to-rank model to produce the relative order of plans for the same query. We also collect the query execution time but perform model training and inference in different ways (see details in [60]). Second, Lero scales the estimated cardinality of sub-queries with different factors s to produce different candidate plans. To deploy Lero, we just need to pull the estimated cardinality of all sub-queries of Q, scale the estimated cardinality and push the new cardinality back to the database to produce different candidate plans. We omit their pseudocodes due to space limits.

5 SOME IMPLEMENTATION DETAILS

In this section, we discuss implementation details in PilotScope, including the DB interactor operators (in Section 5.1) and the related injection interface and data collection (in Section 5.2). Based on them, we show how PilotScope could support to simultaneously apply multiple AI4DB tasks in Section 5.3. Finally, we introduce how to implement the validation execution mode and fault tolerance mechanism in PilotScope (in Section 5.4).

5.1 DB Interactor Operators

5.1.1 The General Approach. We attach the original database with some lightweight patches to enable the interaction with AI4DB algorithms. We describe our approach to support the AI4DB tasks for the query processing workflow (listed in Section 1.1). As shown in Figure 10, we insert a number of blue nodes, called *anchors* and some new flags, into the original query processing workflow. They represent the new patches added to the database codebase.

Specifically, we add two anchors before and after each database component. When the AI4DB algorithm starts a new database interaction session using session_start(), we prepare an empty *prefix*. This prefix records: 1) all variables in the AI4DB algorithm sent to the database component (in push operators) and 2) all intermediate data produced by the database component acquired by the variables in the algorithm (in pull operators). When executing session_exec(), this prefix is sent to the database side. Then, the database starts to invoke the query processing workflow to call each database component, *i.e.*, checking the configurations, query rewriter, query optimizer and executor, one by one. Before each component, the anchor marked with push examines whether there exists a request in the prefix to replace its input. If so, we replace its original input data with the value provided in the prefix. After each component, the anchor marked with pull checks whether we need to pull its output. If so, we fill the value of the corresponding variable into the prefix. After executing all required components, the database returns the prefix to the ML side to instantiate the value of the variables in the AI4DB algorithm. The prefix is reset to be empty.

For the interaction_example shown in Figure 4, it registers three push and two pull operators before session_exec(). We illustrate the prefix in Figure 10. In execution, the database first updates the flag x in configurations to y and then starts executing query Q (see the two dash arrows). Before the cardinality estimator, it replaces the estimated cardinality with variable C provided in the prefix. Then, the database generates the query plan and fills the value of variable P (see the dash arrow to t). After that, it executes the plan, obtains its execution time and fills in the value of variable t. P and t are returned to the ML side to instantiate the value of the two variables.

5.1.2 Our Implementation Strategy on PostgreSQL and Spark. Later, we could implement the DB interactor interfaces in multiple databases by its developers. The interfaces consist of three modules: the query parser module, the data transmission module, and the anchor module. The query parser module is responsible for adding and resolving the prefix before each SQL query. We could simply add its code before the SQL query processing codes in the database. In our implementation, we store all information in the prefix in the common JSON format. The data transmission module receives the prefix from the database to instantiate the value of the variables in the AI4DB algorithm. We implement it using a standardized HTTP transmission method. In terms of the code complexity, the query parser and data transmission modules are relatively easy.

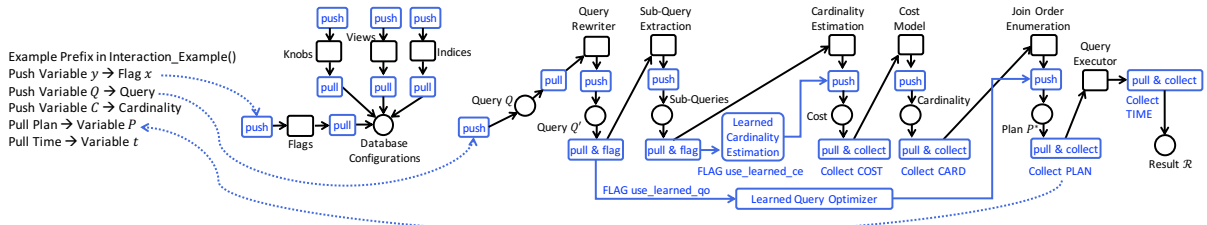


Figure 10: Workflow of query processing with anchors added by `PilotScope`.

The anchor module demands a proficient knowledge of the database codebase. However, in comparison to injecting a whole AI4DB algorithm into the database codebase, it is much easier as it just embeds a number of functions into the query execution process. On each specific database, *e.g.*, PostgreSQL and Spark, we could even implement some anchors with off-the-shelf tools provided by the database to incur minimal changes to its kernel.

First, some anchors could be achieved by directly using database commands. On PostgreSQL, we directly use the "SET" command to set knobs, indices and views and the "EXPLAIN" command to pull the execution plan and its cost. On Spark, for a Spark session, we directly set the values of knobs in "SparkSession.config". For hint flags, we could set them together with the SQL queries in "SparkSession.sql". To get a plan, we could apply "DataFrame.explain("plan")".

Second, some anchors are implemented using the database hooks, which are interfaces provided by the database itself to inject codes. In PostgreSQL, we apply its hook mechanism provided by the pg_hint_plan plug-in unit in [7] to set hint flags. We also use the two hook functions ExecutorStart_hook and ExecutorEnd_hook to get plan execution time.

Finally, we write the functions of all remaining anchors, *e.g.*, sub-query extraction, join order setting and other push/pull operators on PostgreSQL and Spark, as patches and insert them into the right place of the database codebase. The code complexity of each anchor depends on the complexity of setting or collecting the specific data. For example, collecting the execution time is relatively easy by deploying timers before and after the execution. However, collecting all sub-queries in a batch manner can be a bit complex. In the query processing workflow, each sub-query is proceeded by the cardinality estimation function one by one. Therefore, the developer needs to embed codes into the cardinality estimation function to collect all sub-queries together.

5.2 Injection Interface and Data Collection

The injection interface and data collection are all built upon DB interactor operators. The code complexity to implement them is very low. We describe the details as follows.

5.2.1 Injection Interface. Recall that, in `PilotScope`, we maintain a connection for each user to operate the database (see Section 2.1). Unlike the database sessions that are created by AI4DB algorithms, this database session is created by a user. `PilotScope` automatically records its identifier of this session, denoted as mid. Based on it, we could affect its behaviors using the DB interactor operators to inject AI4DB algorithms.

We divide all AI4DB tasks into two types. First, for tasks like knob tuning and index recommendation, they are invoked upon user request using `PilotScope` commands. We only need to send their results, *e.g.*, knob value and indices, into the database. This is fairly easy. For knob tuning discussed in Section 4.1, after the user types the command "auto knob tune -X, -W -M", the AI4DB algorithm in the knob tuning driver is executed. When it finishes, `PilotScope` obtains the value v for knobs X. Then, it executes push(mid, FLAG, X, v) and session_exec(mid). The knobs in the databases would be tuned according to the AI4DB algorithm. For the index recommendation task in Section 4.1, the injection approach is similar. For similar tasks, such as learned view advisor [26, 56], they could also be supported by `PilotScope` in this manner.

Second, for tasks like cardinality estimation and query optimizer, they need to be invoked by the database when needed. To this end, we add two flags, namely use_learned_ce and use_learned_qo, as patches in the database (see Figure 10). These flags are closed by default. If the user opens a driver, `PilotScope` applies push() and session_exec() with mid to open the corresponding flag in the database. Our anchors added before the cardinality estimator and query optimizer (marked as flags in Figure 10) would check the flags. If it is opened, it would ask `PilotScope` to invoke the AI4DB algorithm and pass the input parameters to it. After the algorithm finishes, `PilotScope` sends the result, *e.g.*, the new cardinality or plan, to the anchor. The anchor would skip the original component and apply the new result for the downstream procedures. Similarly, we could extend `PilotScope` to support tasks such as learned query rewriter [12, 17, 38], cost model [28, 35], join order search methods [18, 48] and learned index [21, 22, 31].

5.2.2 Data Collection. For each type of collected data, *e.g.*, query execution time, we could collect it using the pull operator. Specifically, when we start the collect_data() API, we add a pull() operator on the collected type of data before executing each SQL query. Our anchor (marked as pull & collect in Figure 10) would check the pull() operator in the prefix and send the data to `PilotScope` if it is opened. `PilotScope` would write the data into the address specified in the collect_data() API. If the collected data reaches the desired size, the collect_data() API closes the flag. Following this approach, we could extend `PilotScope` to collect more statistical information from the databases.

5.3 Applying Multiple AI4DB Tasks in `PilotScope`

`PilotScope` supports applying multiple AI4DB tasks on one database at the same time. For AI4DB tasks having no overlaps in the execution workflow, `PilotScope` could execute them in order. For example, we could first apply a learned knob tuning method and then execute a query using a learned cardinality estimator. Besides, in query execution, we could apply a learned query rewriter and a learned query optimizer at the same time.

However, for AI4DB tasks having overlapped functions, *e.g.*, cardinality estimator and query optimizer, the situation is a bit complex. In developing each injection interface, PilotScope requires the ML developers to specify whether it has overlaps w.r.t. other injection interfaces. Based on these, PilotScope could identify the conflicts between different drivers. The user could tune the *priority* of each driver in PilotScope. In execution, PilotScope could automatically apply the AI4DB task in the driver with higher priority but ignore the others. In such a way, the user could select to apply a learned cardinality estimator or an end-to-end learned query optimizer in databases.

Note that, updating the priority of tasks only affects the database session (marked with mid) where the user operates the database. For tasks running in different sessions, we could apply different priorities. We show this by the example of Bao [39], the learned query optimizer discussed in Section 4.4. We could set the learned query optimizer with higher priority in the database user session (with identifier mid), so Bao would be executed for each SQL query. In its algorithm, it iteratively creates a session (with identifier sid) to invoke the traditional query optimizer to generate candidate plans. In this session, we could replace the traditional cardinality estimator with the learned one by opening its flag, *i.e.*, adding push(sid, FLAG, use_learned_ce, true) after session_start() in Figure 9. In this case, the candidate plans are generated using learned cardinality while the execution plan is obtained using Bao.

5.4 Validation Mode and Fault Tolerance

PilotScope supports the validation execution mode which could run AI4DB tasks and automatically recover the original settings of databases when quitting it. Meanwhile, it could tolerate faults caused by network failures or system blocks. We apply different strategies to implement them for the two types of AI4DB tasks defined in Section 5.2.1.

For the first type of AI4DB tasks like knob tuning, PilotScope records the original settings, *e.g.*, knobs or indices, when entering the validation mode. When quitting it, PilotScope reset these settings. For fault tolerance, PilotScope sets a pre-defined wait_time for each driver. If the algorithm exceeds its waiting time, PilotScope restarts this job to run once again.

For the second type of AI4DB tasks like cardinality estimation, PilotScope just closes all flags, *e.g.*, use_learned_ce, when quitting the validation mode. For fault tolerance, if the algorithm exceeds the waiting time, the anchors in the database would continue to run this job with the original database component.

6 SYSTEM EVALUATION

We conduct extensive experiments to evaluate the performance of PilotScope. We describe our experimental setups in Section 6.1. The conducted experiments target to answer the most crucial questions as follows:

1) How about the performance of AI4DB algorithms when deployed into actual database systems using PilotScope (in Section 6.2);

2) How much overhead does PilotScope bring in the deployment and its impact on the AI4DB algorithms (in Section 6.3);

3) What about the performance of databases when applying multiple AI4DB algorithms together by PilotScope (in Section 6.4).

We emphasize that PilotScope only serves as a middleware to facilitate the implementation of AI4DB algorithms and support deploying them into databases. Thus, the performance of the AI4DB algorithms is independent of our PilotScope. The problems to analyze their detailed performance and design better methods are beyond the scope of this paper. However, our PilotScope could truly benchmark the actual performance of AI4DB algorithms in real-world scenarios and expose their shortcomings. This would be very helpful to iterate AI4DB research in the future.

6.1 Experimental Setups

AI4DB Algorithms and Settings. We deployed all of the 15 algorithms mentioned in Section 4 on both PostgreSQL and Spark[1] by PilotScope. Due to space limits, we report the evaluation results of the most representative algorithm for each AI4DB task, namely the SMAC [25] for knob tuning, Extend [45] for index recommendation, DeepDB [24] for learned cardinality estimation and Bao [39] for learned query optimizer. The reasons for selecting these algorithms are described as follows:

1) SMAC is shown to perform better than other algorithms in the benchmark evaluation [27, 58]. For PostgreSQL and Spark, we obtain the set of tuneable knobs in the configuration files, including all numerical (*e.g.*, autovacuum_vacuum_threshold) and Boolean knobs (*e.g.*, enable_indexonlyscan). We tune knobs using the training workload to iterate 50 times to optimize the execution time.

2) Extend is shown to provide the best combination of runtime and solution quality in the benchmark [30]. We use it to build indices on all feasible columns with a budget of 250MB on memory.

3) DeepDB is verified to perform better than the other two algorithms on real-world databases. The benchmark evaluation in [23] shows that, on some datasets, the traditional query optimizer could obtain near optimal plans when applying DeepDB for cardinality estimation.

4) Bao and Lero [60] follow similar technical routines. We select the set of hint flags in the same way as [2]. Its prediction model is updated after executing every 100 queries.

For all algorithms, we use their original implementations in [3, 4, 10, 11] and set all other (hyper-)parameters to the default values.

Benchmarks. We apply three benchmarks widely used for query optimization: 1) IMDB dataset on 21 tables of movies and stars with its JOB workload [32] containing 113 realistic queries; 2) STATS dataset on 8 tables of user-contributed content on the Stats Stack Exchange network with its STATS-CEB workload [8, 23] having 146 queries varying in join size and forms; and 3) TPC-DS benchmark [6] on synthetic data. For IMDB and STATS, we apply its JOB and STATS-CEB workload for testing, respectively. We generate a training workload of 1,000 queries where each time we randomly sample a query template from JOB or STATS-CEB, fetch its join template and attach some randomly generated predicates to it. For TPC-DS, we set the scaling factor to 10 and generate a training and testing workload having 4,900 and 490 queries, respectively.

[1] All index recommendation algorithms are not deployed on Spark since it does not support index.

Environments. We deploy PilotScope and all databases on a cluster of Linux machines. Each one is equipped with an Intel(R) Xeon(R) Platinum 8163 CPU running at 2.5 GHz, 96 cores, 512GB DDR4 RAM, 1TB SSD and one NVIDIA RTX-2080TI GPU for model training and inference. We install PostgreSQL 13.1 and Spark 3.3.2.

6.2 Performance of AI4DB Algorithms When Deployed using PilotScope

We evaluate the performance of each database with and without each deployed AI4DB algorithm. Specifically, for learned knob tuning (or index recommendation), we record the end-to-end time of executing the test query workload with the knobs (or indices) generated by the AI4DB algorithm and in the default setting, respectively. For knob tuning, the default values of all knobs are provided by the original PostgreSQL in its configuration file. For index recommendation, the default indices are set in the same way to the original benchmark. For learned cardinality estimation (or query optimizer), we record the end-to-end execution time of applying the AI4DB algorithm and traditional component, respectively. Notably, since DeepDB does not support join queries across foreign keys in the JOB workload, we just apply another workload called JOB-light [54, 63] having 70 large queries on the IMDB dataset to evaluate its performance. Notice that, JOB-light is not a subset of JOB workload. Their queries are totally different. The results are shown on Figure 11. In each sub-figure, the first red bar and second light blue bar indicate the performance of the database with the AI4DB algorithm and the default method, respectively. We observe that:

1) The generality of PilotScope is verified, which could support to apply different AI4DB algorithms in different databases using one framework.

2) In most cases, the AI4DB algorithms could bring remarkable performance improvement to databases. For example, on STATS dataset, the end-to-end execution time is 2.71×, 1.82×, 1.32× and 1.46× faster than the original PostgreSQL using learned knob tuning, index recommendation, cardinality estimation and query optimizer, respectively. Specifically, for the knob tuning, the SMAC algorithm produces different values for almost all knobs, *e.g.*, set autovacuum_vacuum_scale_factor to 56.86 rather than its default value 0.2. For the index recommendation, the Extend algorithm removes all non-unique indices on 25 columns in the IMDB dataset and recommends building indices on only four columns. On the STATS dataset, the Extend algorithm removes all non-unique indices on 12 columns and does not recommend the database building any additional indices. Although it builds fewer indices than the original benchmarks, it attains better performance in terms of the query execution time.

3) In some cases, the performance of AI4DB algorithms is even worse than the traditional database components. This is due to the shortcomings of the AI4DB algorithms. For example, on PostgreSQL, Bao needs to generate several dozens of candidate plans and predict their execution time through a large model. The DB interaction and model inference time degrade its overall performance, so Bao performs worse than the original database query optimizer on the IMDB even its pure plan execution time is better. On Spark, the improvement of learned knobs is not significant due to the large domain size of the knobs.

These results indicate the effectiveness of AI4DB algorithms on different databases and also verify the necessity of developing PilotScope. It could benchmark the actual performance of AI4DB algorithms in real-world scenarios and expose their shortcomings. This would be very helpful to iterate AI4DB research in the future. For example, in a learned query optimizer, we need to balance the plan quality and exploration cost to optimize the end-to-end time.

6.3 Impact and Overhead of PilotScope

Next, we examine the impact and overhead brought by PilotScope to the databases and AI4DB algorithms.

The third bar in each sub-figure in Figure 11 reflects the execution time of query workload on PostgreSQL and Spark without installing PilotScope. In comparison to the second bar, where the execution time is recorded on the databases with PilotScope, the two execution time is very close. This indicates that PilotScope makes no difference on the native database systems. This is because all implementations of the DB interactor are developed as lightweight patches into the database codebase. They do not affect the original database behaviors when no AI4DB algorithms are applied.

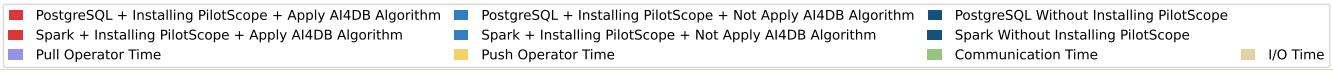
The right part (corresponding to the right y-axis) in each sub-figure in Figure 11 represents the overhead brought by PilotScope in executing query workload with each AI4DB algorithm. Specifically, the fourth to seventh bars represent the time cost of all pull operators, all push operators, communication and I/O, respectively. We have two observations:

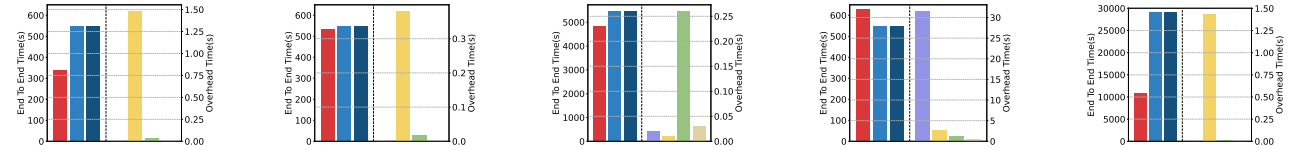
1) In almost all cases, the overall overhead is negligible in comparison to the end-to-end execution time. Specifically, the overhead occupies at most 0.27% and 0.066% for most AI4DB algorithms on PostgreSQL and Spark, respectively. This verifies the success of the design choices of our programming model, which enables the AI4DB algorithms to transfer only a small amount of data to fulfill their jobs. Meanwhile, we adopt some techniques, such as batch of sub-queries, to decrease the data transfer time. The only exception is Bao, whose interaction cost is very high as it needs to fetch a large number of plans from databases.

2) In different AI4DB algorithms, the time cost of each type of overhead is different. For example, DeepDB consumes the most time on communication to pass sub-queries from databases to the AI4DB algorithms and estimated cardinality from AI4DB algorithms to databases. Bao costs a large time, especially on the IMDB, to invoke traditional query optimizer to pull candidate plans. This detailed analysis could further help us to optimize the bottlenecks of AI4DB algorithms in actual databases. For example, how to decrease the number of generating candidate plans in learned query optimizer while still ensuring the plan quality.

6.4 Deploying Multiple AI4DB Algorithms using PilotScope

Finally, we exhibit the ability of PilotScope to flexibly deploy multiple AI4DB algorithms on the same database. Notice that, these experiments have never been performed in the literature. The results are shown in Figure 12. We present three settings on PostgreSQL: 1) applying SMAC to tune knobs and then applying Bao to learn to generate execution plans (the left two sub-figures); 2) applying the learned query optimizer Bao, which calls the learned cardinality





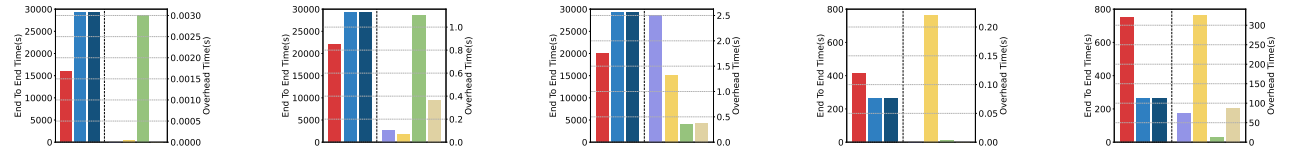
(a) SMAC (Knob Tuning) on PostgreSQL, IMDB dataset, JOB workload

(b) Extend (Index Recommendation) on PostgreSQL, IMDB dataset, JOB workload

(c) DeepDB (Cardinality Estimation) on PostgreSQL, IMDB dataset, JOB-light workload

(d) Bao (Query Optimizer) on PostgreSQL, IMDB dataset, JOB workload

(e) SMAC (Knob Tuning) on PostgreSQL, STATS dataset, STATS-CEB workload



(f) Extend (Index Recommendation) on PostgreSQL, STATS dataset, STATS-CEB workload

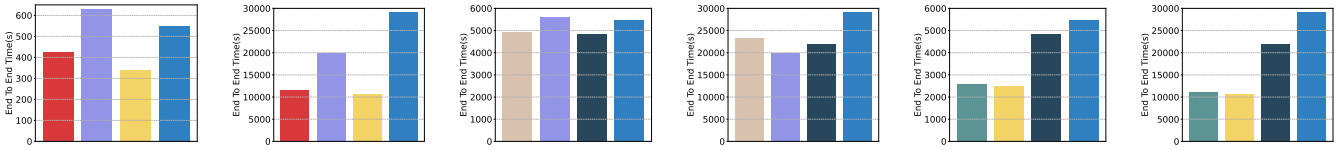
(g) DeepDB (Cardinality Estimation) on PostgreSQL, STATS dataset, STATS-CEB workload

(h) Bao (Query Optimizer) on PostgreSQL, STATS dataset, STATS-CEB workload

(i) SMAC (Knob Tuning) on Spark, TPC-DS dataset and workload

(j) Bao (Query Optimizer) on Spark, TPC-DS dataset and workload

Figure 11: End-to-end execution time of query workloads and overhead of `PilotScope` on PostgreSQL and Spark.



(a) SMAC + Bao on IMDB dataset, JOB workload

(b) SMAC + Bao on STATS dataset, STATS-CEB workload

(c) DeepDB + Bao on IMDB dataset, JOB-light workload

(d) DeepDB + Bao on STATS dataset, STATS-CEB workload

(e) DeepDB + SMAC on IMDB dataset, JOB-light workload

(f) DeepDB + SMAC on STATS dataset, STATS-CEB workload

Figure 12: End-to-end execution time of multiple AI4DB algorithms on PostgreSQL.

estimation DeepDB for generating each candidate plan (the middle two sub-figures, the detailed method is discussed in the last paragraph in Section 5.3); 3) applying SMAC to tune knobs and DeepDB to estimate cardinality since these two methods provide the most benefits by Figure 11 (the right two sub-figures)

We observe that the effect of applying multiple AI4DB algorithms is complex. In some cases, such as Figure 12(b), (c) (e) and (f), the performance of applying two algorithms is comparable to applying only one. In other cases, such as Figure 12(a) and (d), the performance of applying two algorithms is even worse than applying any single algorithm. We consider that it is possible the turned knob value (or new estimated cardinality) affects the behaviors of the learned query optimizer, so it explores bad plans. However, it still needs more in-depth analysis on their correlations. Note that, until now, there exists very little research on the impact of applying multiple AI4DB algorithms together. `PilotScope` could serve as a powerful tool to conduct research towards this direction. It is very attractive to design AI4DB algorithms that could work together to optimize the overall performance of databases.

7 CONCLUSIONS AND FUTURE WORK

We propose `PilotScope`, an AI4DB middleware with a programming model to deploy and develop AI4DB algorithms in databases. `PilotScope` abstracts the interactions between typical AI4DB algorithms and databases into purely a simple interface that hinders the underlying details of databases. With such abstraction, the same AI4DB algorithm in `PilotScope` could be applied to steer multiple different databases. Meanwhile, ML and DB developers could work more independently and play to their own strengths in developing AI4DB tasks in databases. The abstract interface is shown to be expressive and general. It has been applied to support a variety of AI4DB tasks and algorithms and could be easily extended to new AI4DB tasks. Extensive experiments have verified the effectiveness of `PilotScope` in terms of research and application fields.

In the future, we hope to inspire and iterate AI4DB research with `PilotScope`. We target to build an open-source community for `PilotScope` to attract developers to: 1) develop and evaluate more AI4DB tasks and algorithms, including but not limited to, learned index, database monitor and diagnosis; and 2) implement the DB interactor to steer more databases, *e.g.*, MySQL and MongoDB. More over, we would try to push forward to set up a standard for the interface of DB interactor and deploy `PilotScope` in real-world production scenarios.

Acknowledgements. This work was supported by grants from the National Natural Science Foundation of China (No. 62022077 and 62372194) and the program of Alibaba Innovative Research.

REFERENCES

[1] 2020. Anytime Algorithm of Database Tuning Advisor for Microsoft SQL Server. https://www.microsoft.com/en-us/research/publication/anytime-algorithm-of-database-tuning-advisor-for-microsoft-sql-server/.

[2] 2020. Bao appendix. https://rmarcus.info/appendix.html.

[3] 2020. Implementation of DeepDB: Learn from Data, not from Queries! https://github.com/DataManagementLab/deepdb-public.

[4] 2020. A prototype implementation of Bao for PostgreSQL. https://github.com/learnedsystems/BaoForPostgreSQL.

[5] 2021. NoisePage – Database Management System Project. https://noise.page/.

[6] 2021. Transaction Processing Performance Council(TPC): Version 2 and Version 3. https://github.com/Nathaniel-Han/End-to-End-CardEst-Benchmark.

[7] 2023. Give PostgreSQL ability to manually force some decisions in execution plans. https://github.com/ossc-db/pg_hint_plan.

[8] 2023. A new CardEst Benchmark to Bridge AI and DBMS. https://github.com/Nathaniel-Han/End-to-End-CardEst-Benchmark.

[9] 2023. openGauss. https://github.com/opengauss-mirror.

[10] 2023. Platform to evaluate index selection algorithms. https://github.com/hyrise/index_selection_evaluation.

[11] 2023. SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. https://automl.github.io/SMAC3/v2.0.1/.

[12] Christopher Baik, H. V. Jagadish, and Yunyao Li. 2019. Bridging the Semantic Gap with SQL Query Logs in Natural Language Interfaces to Databases. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 374–385.

[13] Nicolas Bruno and Surajit Chaudhuri. 2005. Automatic Physical Database Tuning: A Relaxation-based Approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, Fatma Özcan (Ed.). ACM, 227–238.

[14] Matthew Butrovich, Wan Shen Lim, Lin Ma, John Rollinson, William Zhang, Yu Xia, and Andrew Pavlo. 2022. Tastes Great! Less Filling! High Performance and Accurate Training Data Collection for Self-Driving Database Management Systems. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 617–630.

[15] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.* 1, 2 (2008), 1265–1276.

[16] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld (Eds.). Morgan Kaufmann, 146–155.

[17] Mahendra Chavan, Ravindra Guravannavar, Karthik Ramachandra, and S. Sudarshan. 2011. DBridge: A program rewrite tool for set-oriented query execution. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 1284–1287.

[18] Jin Chen, Guanyu Ye, Yan Zhao, Shuncheng Liu, Liwei Deng, Xu Chen, Rui Zhou, and Kai Zheng. 2022. Efficient Join Order Selection Learning with Graph-based Representation. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*, Aidong Zhang and Huzefa Rangwala (Eds.). ACM, 97–107.

[19] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. 2011. CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads. *Proc. VLDB Endow.* 4, 6 (2011), 362–372.

[20] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1241–1258.

[21] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 969–984.

[22] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1189–1206.

[23] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *Proc. VLDB Endow.* 15, 4 (2021), 752–765.

[24] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proc. VLDB Endow.* 13, 7 (2020), 992–1005.

[25] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *Learning and Intelligent Optimization - 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers (Lecture Notes in Computer Science)*, Carlos A. Coello Coello (Ed.), Vol. 6683. Springer, 507–523.

[26] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc T. Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation Reuse in Analytics Job Service at Microsoft. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 191–203.

[27] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. 2022. LlamaTune: Sample-Efficient DBMS Configuration Tuning. *Proc. VLDB Endow.* 15, 11 (2022), 2953–2965.

[28] Johan Kok Zhi Kang, Gaurav, Sien Yi Tan, Feng Cheng, Shixuan Sun, and Bingsheng He. 2021. Efficient Deep Learning Pipelines for Accurate Cost Estimations Over Large Scale Query Workload. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1014–1022.

[29] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org.

[30] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *Proc. VLDB Endow.* 13, 11 (2020), 2382–2395.

[31] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 489–504.

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

[33] Guoliang Li, Xuanhe Zhou, and Lei Cao. 2021. AI Meets Database: AI4DB and DB4AI. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2859–2866.

[34] Guoliang Li, Xuanhe Zhou, Ji Sun, Xiang Yu, Yue Han, Lianyuan Jin, Wenbo Li, Tianqing Wang, and Shifu Li. 2021. openGauss: An Autonomous Database System. *Proc. VLDB Endow.* 14, 12 (2021), 3028–3041.

[35] Yan Li, Liwei Wang, Sheng Wang, Yuan Sun, and Zhiyong Peng. 2022. A Resource-Aware Deep Cost Model for Big Data Query Processing. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 885–897.

[36] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. 2021. MB2: Decomposed Behavior Modeling for Self-Driving Database Management Systems. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1248–1261.

[37] Minghua Ma, Zheng Yin, Shenglin Zhang, Sheng Wang, Christopher Zheng, Xinhao Jiang, Hanwen Hu, Cheng Luo, Yilin Li, Nengjun Qiu, Feifei Li, Changcheng Chen, and Dan Pei. 2020. Diagnosing Root Causes of Intermittent Slow Queries in Large-Scale Cloud Databases. *Proc. VLDB Endow.* 13, 8 (2020), 1176–1189.

[38] Nantia Makrynioti and Vasilis Vassalos. 2021. Declarative Data Analytics: A Survey. *IEEE Trans. Knowl. Data Eng.* 33, 6 (2021), 2392–2411.

[39] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 1275–1288.

[40] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (2019), 1705–1718.

[41] Parimarjan Negi, Matteo Interlandi, Ryan Marcus, Mohammad Alizadeh, Tim Kraska, Marc T. Friedman, and Alekh Jindal. 2021. Steering Query Optimizers: A Practical Take on Big Data Workloads. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2557–2569.

[42] Andy Pavlo, Matthew Butrovich, Lin Ma, Prashanth Menon, Wan Shen Lim, Dana Van Aken, and William Zhang. 2021. Make Your Database System Dream

of Electric Sheep: Towards Self-Driving Operation. *Proc. VLDB Endow.* 14, 12 (2021), 3211–3221.

[43] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. 2023. The Composable Data Management System Manifesto. *Proc. VLDB Endow.* 16, 10 (2023), 2679–2685.

[44] Bin Xin Ru, Ahsan S. Alvi, Vu Nguyen, Michael A. Osborne, and Stephen J. Roberts. 2020. Bayesian Optimisation over Multiple Continuous and Categorical Inputs. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event (Proceedings of Machine Learning Research)*, Vol. 119. PMLR, 8276–8285.

[45] Rainer Schlosser, Jan Kossmann, and Martin Boissier. 2019. Efficient Scalable Multi-attribute Index Selection Using Recursive Strategies. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 1238–1249.

[46] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 99–113.

[47] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation: A Design Space Exploration and A Comparative Evaluation. *Proc. VLDB Endow.* 15, 1 (2021), 85–97.

[48] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1153–1170.

[49] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, David B. Lomet and Gerhard Weikum (Eds.). IEEE Computer Society, 101–110.

[50] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready For Learned Cardinality Estimation? *Proc. VLDB Endow.* 14, 9 (2021), 1640–1654.

[51] Lianggui Weng, Rong Zhu, Di Wu, Bolin Ding, Bolong Zheng, and Jingren Zhou. 2024. Eraser: Eliminating Performance Regression on Learned Query Optimizer. *Proc. VLDB Endow.* (2024).

[52] Kyu-Young Whang. [n.d.]. Index Selection in Relational Databases. In *Foundations of Data Organization, Proceedings of the International Conference on Foundations of Data Organization, May 22-24, 1985, Kyoto, Japan*, Sakti P. Ghosh, Yahiko Kambayashi, and Katsumi Tanaka (Eds.). 487–500.

[53] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 931–944.

[54] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *Proc. VLDB Endow.* 14, 1 (2020), 61–73.

[55] Xiang Yu, Chengliang Chai, Guoliang Li, and Jiabin Liu. 2022. Cost-based or Learning-based? A Hybrid Query Optimizer for Query Plan Selection. *Proc. VLDB Endow.* 15, 13 (2022), 3924–3936.

[56] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. 2020. Automatic View Generation with Deep Learning and Reinforcement Learning. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1501–1512.

[57] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 415–432.

[58] Xinyi Zhang, Zhuo Chang, Yang Li, Hong Wu, Jian Tan, Feifei Li, and Bin Cui. 2022. Facilitating Database Tuning with Hyper-Parameter Optimization: A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 15, 9 (2022), 1808–1821.

[59] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries using Graph Embedding. *Proc. VLDB Endow.* 13, 9 (2020), 1416–1428.

[60] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. *Proc. VLDB Endow.* 16, 6 (2023), 1466–1479.

[61] Rong Zhu, Lianggui Weng, Wenqing Wei, Di Wu, Jiazhen Peng, Yifan Wang, Bolin Ding, Defu Lian Bolong Zheng, and Jingren Zhou. [n.d.]. PilotScope: Steering Databases with Machine Learning Drivers (Full Version). In *https://github.com/duoyw/PilotScope/blob/main/paper/fullversion.pdf*.

[62] Rong Zhu, Ziniu Wu, Chengliang Chai, Andreas Pfadler, Bolin Ding, Guoliang Li, and Jingren Zhou. [n.d.]. Learned Query Optimizer: At the Forefront of AI-Driven Databases. In *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*, Julia Stoyanovich, Jens Teubner, Paolo Guagliardo, Milos Nikolic, Andreas Pieris, Jan Mühlig, Fatma Özcan, Sebastian Schelter, H. V. Jagadish, and Meihui Zhang (Eds.). 1–4.

[63] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. *Proc. VLDB Endow.* 14, 9 (2021), 1489–1502.