Bulletin of the Technical Committee on

Data Engineering

December 2018 Vol. 41 No. 4

IEEE Computer Society

Letters

Special Issue on Machine Learning Life-cycle Management

Conference and Journal Notices

Farewell

It was way back in 1992 that Rakesh Agrawal, then the TCDE Chair, appointed me as Editor-in-Chief of the Data Engineering Bulletin. At the time, I saw it as a great opportunity. But it did not occur to me that it would become such an enormous part of my career. Now, 26 years later, it is time, perhaps past time, for me to pass this position on to younger hands, in this case to the capable hands of Haixun Wang. It should not come as a surprise that I am stepping down. Rather, the surprise should be "why did I stay so long?" This message is a combination of answer to that question and historical sketch of my time as EIC. These are not unrelated.

When I first became EIC, the Bulletin had already established a reputation as an industry and engineering focused publication, each issue of which was on a special topic. Won Kim, my predecessor, had very capably established that publication model. Papers are solicited by each issue editor, with the editor selecting which authors to invite. The papers are a mix of work in progress, position statements, surveys, etc. But all focused on the special topic. I was determined not to screw this up. Indeed, I accepted the EIC appointment because I believed that the role that the Bulletin played is unique in our database community. I stayed so long because I still believe that.

Over the years, the Bulletin went through several major changes. As early as 1993, the Bulletin could be accessed online as well as via print subscription. This was a major transition. Mark Tuttle, then a colleague of mine in Digital (DEC) Cambridge Research Lab designed the latex style files that enabled this. Shortly thereafter, to economize on costs, the Bulletin became one of the earliest all electronic publications in our field.

In 1995, hosting the Bulletin web site was provided by Microsoft- continuing until three years ago. Around 2010, the IEEE Computer Society became the primary host for the Bulletin. Around 2000, at the suggestion (prodding) of Toby Lehman,individual articles in addition to complete issues were served from the Bulletin web sites. Over this time, the style files and my procedures for generating the Bulletin evolved as well. Mark Tuttle again, and S. Sudarshan, who had been a Bulletin editor, provided help in evolving procedures used to generate the Bulletin and its individual articles.

The Computer Society, and specifically staff members John Daniel, Carrie Clark Walsh, and Brookes Little, provided a TCDE email membership list used to distribute issue announcements, as well as helping in myriad other ways. The existence of dbworld (one of Raghu Ramakrishnan enduring contributions) enabled wider announcement distribution to our database community. The cooperation of Michael Ley with the prompt indexing of the Bulletin at dblp both ensured wider readership and provided an incentive for authors to contribute. Over the years, I was given great support by TCDE Chairs, starting with Rakesh Agrawal, then Betty Salzberg, Erich Neuhold, Paul Larson, Kyu-Young Whang, and Xiaofang Zhou.

The most important part of being Bulletin EIC was the chance to work with truly distinguished members of the database community. It was enormously gratifying to have stars of our field (including eight Codd Award winners- so far) serving as editors. I take pride in appointing several of them as editors prior to their wider recognition. It is the editors that deserve the credit for producing, over the years, a treasure trove of special issues on technologies that are central to our data engineering field. Superlative editors, and their success in recruiting outstanding authors, is the most important part of the Bulletin's success. Successfully convincing them to serve as editors is my greatest source of pride in the role I played as Bulletin EIC.

Now I am happy to welcome Haixun to this wonderful opportunity. Haixun's background includes outstanding successes in both research and industry. He recently served ably as a Bulletin associate editor for issues on "Text, Knowledge and Database" and "Graph Data Processing". His background and prior editorial experience will serve our data engineering community well and ensure the ongoing success of the Bulletin. I wish him and the Bulletin all the best.

And so "farewell". I will always treasure having served as Bulletin EIC for so many years. It was a rare privilege that few are given. Knowing that we were reaching you with articles that you found valuable is what

has made the effort so rewarding to me personally. Thank you all for being loyal readers of the Bulletin.

David Lomet
Microsoft Corporation

Letter from the Editor-in-Chief

Thank You, David!

I know I represent the readers, the associate editors, and also the broad database community when I say we are extremely grateful to David Lomet for his distinguished and dedicated service as the Editor-in-Chief of the Data Engineering Bulletin for the last 26 years.

Since its launch in 1977, the Bulletin has produced a total of 154 issues. Reading through the topics of the past issues that spanned more than four decades makes me feel nothing short of amazing. They show not just how far the database research has come, but to a certain extent, how much the entire field of computer science and the IT industry have evolved. While important topics never fail to arise in the Bulletin in a timely fashion, it is also interesting to observe in the 154 issues many recurring topics, including query optimization, spatial and temporal data management, data integration, etc. It proves that the database research has a solid foundation that supports many new applications, and at the same time, it demonstrates that the database research is constantly reinventing itself to meet the challenges of the time. What the Bulletin has faithfully documented over the last 42 years is nothing else but this amazing effort.

Among the 154 issues since the launch of the Bulletin, David had been the Editor-in-Chief for 103 of them. This itself is a phenomenal record worth an extra-special celebration. But more importantly, David shaped the discussions and the topics in the long history of the Bulletin. I had the honor to work with David in 2016 and 2017 when I served as the associate editor for two Bulletin issues. What was most appealing to me was the opportunity of working with the top experts on a topic that I am passionate about. The Bulletin is truly unique in this aspect.

I understand the responsibility and the expectation of the Editor-in-Chief, especially after David set such a great example in the last 26 years. I thank David and the associate editors for their trust, and I look forward to working with authors, readers, and the database community on the future issues of the Data Engineering Bulletin.

The Current Issue

Machine learning is changing the world. From time to time, we are amazed at what a few dozen lines of python code can achieve (e.g., using PyTorch, we can create a simple GAN in under 50 lines of code). However, for many real-life machine learning tasks, the challenges lie beyond the dozen lines of code that construct a neural network architecture. For example, hyperparameter tuning is still considered a "dark art," and having a platform that supports parallel tuning is important for training a model effectively and efficiently. Model training is just one component in the life cycle of creating a machine learning solution. Every component, ranging from data preprocessing to inferencing, requires just as much support on the system and infrastructure level.

Joseph Gonzalez put together an exciting issue on the life cycle of machine learning. The papers he selected focus on systems that help manage the process of machine learning or resources used in machine learning. They highlight the importance of building such supporting systems, especially for production machine learning platforms.

Haixun Wang
WeWork Corporation

Letter from the Special Issue Editor

Machine learning is rapidly maturing into an engineering discipline at the center of a growing range of applications. This widespread adoption of machine learning techniques presents new challenges around the management of the data, code, models, and their relationship throughout the machine learning life-cycle. In this special issue, we have solicited work from both academic and industrial leaders who are exploring how data engineering techniques can be used to address the challenges of the machine learning life-cycle.

The machine learning life-cycle (Fig. **??**) spans not only the model development but also production training and inference. Each stage demands different skills (e.g., neural network design, data management, and cluster management) and imposes different requirements on the underlying systems. Yet there is an overwhelming need for unifying design principles and technologies to address pervasive problems including feature management, data provenance, pipeline reproducibility, low-latency serving, and prediction monitoring just to name a few.

There has been a flurry of recent progress in systems to aid in managing the machine learning life-cycle. Large industrial projects like FB Learner Flow from Facebook, Michelangelo from Uber, and TFX from Google have received considerable recent attention. In this issue, we have solicited papers from several recent industrial and academic projects that have received slightly less attention.

The first paper provides an overview of several real-world use cases and then outlines the key conceptual, data management, and engineering challenges faced in production machine learning systems. The second and third papers explores the challenges of model management and provenance across the machine learning life-cycle. They motivate the need for systems to track models and their meta-data to improve reproducibility, collaboration, and governance. The second paper introduces, ModelDB, an open-source system for model management and describe some of the functionality and design decisions. The third paper describes a related system, ProvDB, that uses a graph data model to capture and query fine-grained versioned lineage of data, scripts, and artifacts throughout the data analysis process. The fourth paper describes, MLFlow, a new open-source system to address the challenges of experimentation, reproducibility, and deployment. This work leverages containerization to capture the model development environment and a simple tracking API to enable experiment tracking. The fifth paper focuses on inference and explores the challenges and opportunities of serving white-box prediction pipelines. Finally, we solicited a summary of the recent Common Modeling Infrastructure (CMI) workshop at KDD 2018, which provides a summary of the keynotes and contributed talks.

The work covered here is only a small sample of the emerging space of machine learning life-cycle management systems. We anticipate that this will be a growing area of interest for the data engineering community.

Joseph E. Gonzalez
University of California at Berkeley
Berkeley, CA

Toward Intelligent Query Engines

Matthaios Olma Stella Giannakopoulou Manos Karpathiotakis Anastasia Ailamaki
EPFL

Abstract

Data preparation is a crucial phase for data analysis applications. Data scientists spend most of their time on collecting and preparing data in order to efficiently and accurately extract valuable insights. Data preparation involves multiple steps of transformations until data is ready for analysis. Users often need to integrate heterogeneous data; to query data of various formats, one has to transform the data to a common format. To accurately execute queries over the transformed data, users have to remove any inconsistencies by applying cleaning operations. To efficiently execute queries, they need to tune access paths over the data. Data preparation, however is i) time-consuming since it involves expensive operations, and ii) lacks knowledge of the workload; a lot of preparation effort is wasted on data never meant to be used.

To address the functionality and performance requirements of data analysis, we re-design data preparation in a way that is weaved into data analysis. We eliminate the transform-and-load cost using in-situ query processing approaches which adapt to any data format and facilitate querying diverse datasets. To address the scalability issues of cleaning and tuning tasks, we inject cleaning operations into query processing, and adapt access paths on-the-fly. By integrating the aforementioned tasks into data analysis, we adapt data preparation to each workload and thereby minimize response times.

1 Introduction

Driven by the promise of big data analytics, enterprises gather data at an unprecedented rate that challenge state-of-the art analytics algorithms [43]. Decision support systems used in industry, and modern-day analytics involve interactive data exploration, visual analytics, aggregate dashboards, and iterative machine learning workloads. Such applications, rely heavily on efficient data access, and require real-time response times irrespective of the data size. Besides the high volume of data, data analysis requires combining information from multiple datasets of various data formats which are often inconsistent [15, 25]. Therefore, satisfying these requirements is a challenge for existing database management systems.

To offer real-time support, database management systems require compute and data-intensive preprocessing operations which sanitize the data through data loading and cleaning, and enable efficient data access through tuning. These data preparation tasks rely heavily on assumptions over data distribution and future workload. However, real-time analytics applications access data instantly after its generation and often workloads are constantly shifting based on the query results [10]. Thereby, making a priori static assumptions about data or queries may harm query performance [3, 9].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

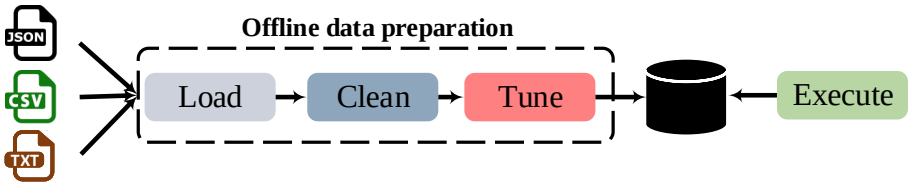

Figure 1: Data Processing pipeline.

Data preparation involves several steps of processing until raw data is transformed into a form that fits data analysis. To enable queries that combine a variety of data formats, such as relational, or semi-structured hierarchical formats which have become the state-of-the-art for data exchange, data scientists rely on database management engines which offer a broad-range of analysis operations. To overcome this heterogeneity of data formats, database management systems perform *data loading* which transforms raw data into a single relational data format to allow for more flexibility in the operations that users can execute. As the data collected by the application is often a result of combining multiple, potentially erroneous sources, it contains inconsistencies and duplicates. To return correct results, database management systems must recognize such irregularities and remove them through *data cleaning* before analyzing the data. Finally, to improve query performance and enable near real-time query responses, database management systems avoid or reduce unnecessary data access by *tuning access paths* (e.g., indexes) over the dataset. Figure 1 presents the data pipeline of a state-of-the-art data analytics framework. The data to be analyzed is collected from a variety of sources, and might appear in various formats (e.g., XML, CSV, etc.). The multiple input formats are transformed into a single uniform format by loading them into a DBMS. Then, to remove any inconsistencies cleaning operations are applied. Finally, a tuner builds access paths for efficient access. The final result is stored in a clean and tuned database, and is ready to receive query requests.

The preprocessing steps are exploratory and data-intensive, as they involve expensive operations, and highly depend on the data and the query workload. Data preparation tasks access the entire dataset multiple times: data loading results in copying and transforming the whole dataset into a common format. Cleaning tasks perform multiple passes over the data until they fix all the inconsistencies. Finally, to build indexes, an extra traversal of the dataset is needed. Therefore, the increasing data volume limits the scalability of data preparation. Furthermore, the benefits of data preparation depend highly on the to-be executed workload. Data transformation and cleaning are only useful if the queries are data intensive and access the majority of data. Finally, tuning requires a priori knowledge of queries to decide upon the most efficient physical design.

Data preparation is time consuming. Due to the influx of data, data preparation becomes increasingly expensive. Figure 2 demonstrates the breakdown of the overall execution time of a typical data analysis scenario. The breakdown corresponds to the time that a system requires to preprocess the data and execute a set of 10 queries. The execution time reported at each step is based on recent studies on loading, cleaning, and tuning [29, 31]. Specifically, assuming an optimistic scenario in which data cleaning corresponds to 50% of the analysis time, then based on [31], the rest 50% is mostly spent on loading and tuning. The loading percentage may become even higher in the presence of non-relational data formats, such as XML, because a DBMS will have to flatten the dataset in order to load it. Query execution takes 3% of the overall time. Therefore, data preparation incurs a significant overhead to data analysis.

Despite enterprises collecting and preparing increasingly larger amounts of data for analysis, often the effectively useful data is considerably smaller than the full dataset [4, 33]. The trend of exponential data growth due to intense data generation and data collection is expected to persist, however, recent studies of the data analysis workloads show that typically only a small subset of the data is relevant and ultimately used by analytical and/or exploratory workloads [10]. Therefore, having to preprocess the whole dataset results in wasting effort on data which are unnecessary for the actual analysis. Furthermore, modern-day analytics, are increasingly tolerant to result imprecision. In many cases, precision to "last decimal" is redundant for a query answer. Quick

approximation with some error guarantee is adequate to provide insights about the data [11]. Thus, using query approximation, one can execute analytical queries over small samples of the dataset, and obtain approximate results within a few percents of the actual value [32].

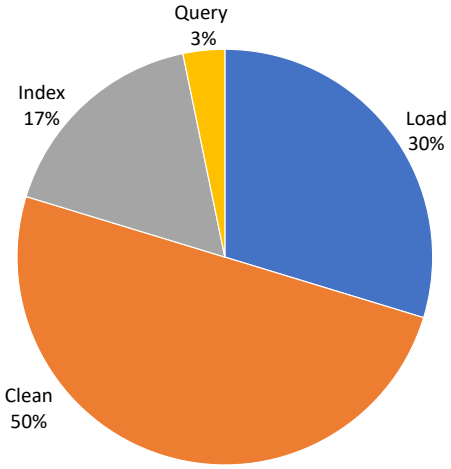

Figure 2: Cost of Data Preprocessing

Ever Changing Workload. Modern businesses and scientific applications require interactive data access, which is characterized by no or little a priori workload knowledge and constant workload shifting both in terms of projected attributes and selected ranges of the data. For example, an electricity monitoring company continuously collects information about the current and aggregate energy consumption, and other sensor measurements such as temperature. To optimize consumption, the company performs predictive analytics over smart home datasets, looking for patterns that indicate energy request peaks and potential equipment downtime [21]. Analyses in this context start by identifying relevant measurements by using range queries and aggregations to identify areas of interests. The analysis focuses on specific data regions for a number of queries, but is likely to shift across the dataset to a different subset. Due to the unpredictable nature of data analytics workloads, where queries may change depending on prior query results, applications prepare all data for data access to avoid result inconsistencies. This preparation requires investment of time and resources into data that may be useless for the workload, thereby delaying data analysis.

Adapt to Data and Workload. To address the aforementioned shortcomings, we revisit the data processing pipeline, and aim to streamline the process of extracting insights from data. We reduce the overall time of data analysis by introducing approaches which adapt online to workload and dataset, which reduce the cost of each of the steps of data analysis from data collection to result. Specifically, to reduce the cost of loading, we execute queries over raw data files [5, 25, 26, 27], to reduce the cost of data cleaning we piggy-back operations over query execution and we only sanitize data affected by the queries [17]. Finally, to reduce the cost of tuning, we take advantage of data distribution as well as relaxed precision constraints of applications and adapt access paths online and as a by-product of query execution to data and workload [31, 32]. Figure 3 demonstrates the revised data analysis process which weaves data preprocessing into query execution by adapting to the underlying data, as well as to the query workload.

At the core of our approach lies *in-situ* query processing, which allows the execution of declarative queries over external files without duplicating or "locking" data in a proprietary database format. We extend *in-situ* approaches [5, 23] by treating any data format as a first-class citizen. To minimize query response times, we build a just-in-time query engine specialized for executing queries over multiple data formats. This approach removes the need for transforming and loading, while also offering low data access cost. To reduce the cost of data cleaning, we enhance query execution by injecting data cleaning operations inside the query plan. Specifically, we introduce a query answer relaxation technique which allows repairing erroneous tuples at query execution time. By relaxing the query answer, we ensure that the query returns all entities that may belong to the query result (e.g., no missing tuples). Finally, similarly to data cleaning, building indexes over a dataset is becoming increasingly harder due to (i) shifting workloads and (ii) increasing data sizes which increase access path size as well. The decision on what access paths to build depends on the expected workload, thus, traditional database systems assume knowledge of future queries. However, the shifting workload of modern data analytics can nullify investments towards indexing and other auxiliary data structures. Furthermore, access path size increases along with input data, thus, building precise access paths over the entire dataset limits the scalability of databases systems. To address these issues, we adapt access paths to data distribution and precision requirements of the result. This enables building data structures specifically designed to take advantage of different data distributions

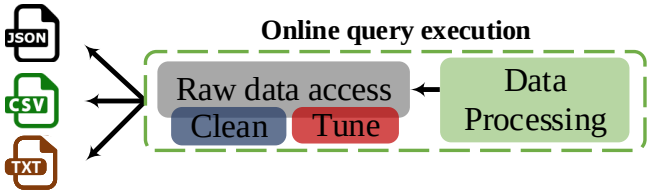

Figure 3: Integrating Cleaning and Tuning to Data Access.

and create data summaries requiring less storage space.

In this paper we describe techniques that enable instant access to data irrespective of data format, and enable data cleaning and tuning without interrupting query execution. Each technique addresses a step in the preprocessing phase of data analysis, reducing the total data-to-insight time for users. Specifically, in Section 2, we describe the design behind our just-in-time query engine which enables efficient query execution despite data heterogeneity. In Section 3, we demonstrate a novel approach to intertwine query execution with data cleaning through query answer relaxation. Our approach incrementally cleans only data that will be analyzed. In Section 4, we present our approach to adapt access paths online to data distribution and to precision requirements, as well as to available storage resources. Finally, in Section 6, we conclude by highlighting techniques and related open problems for adaptive data management systems.

2 Adapting Data Access and Query Engine to Data Format

Data analysis requires combining information from numerous heterogeneous datasets. For example, applications such as sensor data management and decision support based on web click-streams involve queries over data of varying models and formats. To support analysis workloads over heterogeneous datasets, practitioners are left with two alternatives: a) use a database engine that supports multiple operations [37], or b) execute their analysis over dedicated, specialized systems for each of their applications [35]. The first approach might hurt performance for scenarios involving non-relational data, but allows for extensive functionality and expressiveness. The second approach requires using multiple tools, as well as writing custom scripts to combine the results. Hence, performing analysis effortlessly and efficiently is challenging.

We present an approach that bridges the conflicting requirements for flexibility and performance when analyzing data of various formats. We achieve that by combining an optimizable query algebra, richer than the relational one, with on-demand adaptation techniques to eliminate numerous query execution overheads.

2.1 An Expressive Query Algebra

To support queries over heterogeneous data, we need a query algebra that treats all supported data types as first-class citizens in terms of both expressive power and optimization capabilities. Specifically, our approach is based on the monoid comprehension calculus [16]. A monoid is an algebraic construct term stemming from category theory which can be used to capture operations between both primitive and collection data types. Therefore, monoids are a natural fit for querying various data formats because they support operations over several data collections (e.g., bags, sets, lists, arrays) and arbitrary nestings of them.

The monoid calculus provides the expressive power to manipulate different data formats, and optimizes the resulting queries in a uniform way. First, the monoid calculus allows transformations across data models by translating them into operations over different types of collections, hence we can produce multiple types of output. The calculus is also expressive enough for other query languages to be mapped to it as syntactic sugar: For relational queries over flat data (e.g., binary and CSV files), our design supports SQL statements, which it translates to comprehensions. Similarly, for XML data, XQuery expressions can be translated into our internal

algebra. Thus, monoid comprehensions allow for powerful manipulations of complex data as well as for queries over datasets containing hierarchies and nested collections (e.g., JSON arrays).

For each incoming query, the first step is mapping it into the internal language that is based on monoid comprehensions. Then, the resulting monoid comprehension is rewritten to an algebraic plan of the nested relational algebra [16]. This algebra resembles the relational algebra, with the difference that it allows more complex operators, which are applicable over hierarchical data. For example, apart from the relational operators, such as selection and join, it provides the unnest and outer unnest operators which "unroll" a collection field path that is nested within an object. Therefore, the logical query plan allows for optimizations that combine the aforementioned operators.

The optimizer is responsible for performing the query rewriting and the conversion of a logical to a physical query plan. To apply the optimizations, the optimizer takes into consideration both the existence of hierarchical data, as well as that the queries might be complex, containing multiple nestings. Therefore, the optimization involves a normalization algorithm [16] which transforms the comprehension into a "canonical" form. The normalization also applies a series of optimization rewrites. Specifically, it applies filter pushdown and operator fusion. In addition, it flattens multiple types of nested comprehensions. Thus, using the normalization process, the comprehension is mapped to an expression that allows efficient query execution.

The monoid comprehension calculus is a rich model, and therefore incurs extra complexity. The more complex an algebra is, the harder it becomes to evaluate queries efficiently: Dealing with complex data leads to complex operators, sophisticated yet inefficient storage layouts, and costly pointer chasing during query evaluation. To overcome all previous limitations, we couple a broad algebra with on-demand customization.

2.2 Query Engines On-Demand

We couple this powerful query algebra with on-demand adaptation techniques to eliminate the query execution overheads stemming from the complex operators. For analytical queries over flat (e.g., CSV) data, the system must behave as a relational system. Similarly, for hierarchical data, it must be as fast as a document store. Specifically, our design is modular, with each of the modules using a code generation mechanism to customize the overall system across a different axis.

First, to overcome the complexity of the broad algebra,we avoid the use of general-purpose abstract operators. Instead, we dynamically create an optimized engine implementation per query using code generation. Specifically, using code generation, we avoid the interpretation overhead by traversing the query plan only once and generating a custom implementation of every visited operator. Once all plan operators have been visited, the system can produce a hard-coded query engine implementation which is expressed in machine code.

To treat all supported data formats as native storage, we customize the data access layer of the system based on the underlying data format while executing the query. Specifically, we mask the details of the underlying data values from the query operators and the expression generators. To interpret data values and generate code evaluating algebraic expressions, we use input plug-ins where each input plug-in is responsible for generating data access primitives for a specific file format.

Finally, to utilize the storage that better fits the current workload, we materialize in-memory caches and treat them as an extra input. The shape of each cache is specified at query time, based on the format of the data that the query accesses. We trigger cache creation i) implicitly, as a by-product of an operator's work, or ii) explicitly, by introducing caching operators in the query plan. Implicit caching exploits the fact that some operators materialize their inputs: nest and join are blocking and do not pipeline data. Explicit caching places buffering operators at any point in the query plan. An explicit caching operator calls an output plug-in to populate a memory block with data. Then, it passes control to its parent operator. Creating a cache adds an overhead to the current query, but it can also benefit the overall query workload.

Our design combines i) an expressive query algebra which masks data heterogeneity with ii) on-demand customization mechanisms which produce a specialized implementation per query. Based on this design, we

build Proteus, a query engine that natively supports different data formats, and specializes its entire architecture to each query and the data that it touches via code generation. Proteus also customizes its caching component, specifying at query time how these caches should be shaped to better fit the overall workload.

3 Cleaning Data while Discovering Insights

Data cleaning is an interactive and exploratory process which involves expensive operations. Error detection requires multiple pairwise comparisons to check the satisfiability of the given constraints [18]. Data repairing adds an extra overhead since it requires multiple iterations in order to assign candidate values to the erroneous cells until all constraints are satisfied [12, 15, 28, 38]. At the same time, data cleaning depends on the analysis that users perform; data scientists detect inconsistencies, and determine the required data cleaning operations while exploring through the dataset [40]. Therefore, the usage of offline data cleaning approaches requires long running times in order to discover and fix the discrepancies that might affect data analysis.

To address the efficiency problem, as well as the subjective nature of data cleaning, there is need for a data cleaning approach which is weaved into the data analysis process, and which also applies data cleaning on-demand. Integrating data cleaning with data analysis efficiently supports exploratory data analysis [13], and ad-hoc data analysis applications [20] by reducing the number and the cost of iterations required in order to extract insights out of dirty data. In addition, by cleaning data on the fly, one only loads and cleans necessary data thereby minimizing wasted effort whenever only a subset of data is analyzed.

We intermingle cleaning integrity constraint violations [14] with exploratory data analysis, in order to gradually clean the dataset. Specifically, given a query and a dirty dataset, we use two levels of processing to correctly execute the query by taking into consideration the existence of inconsistencies in the underlying dataset. In the first level, we map the query to a logical plan which comprises both query and cleaning operators. The logical plan takes into consideration the type of the query (e.g, Select Project, Join), and the constraints that the dataset needs to satisfy in order to optimally place the cleaning operators inside the query plan. Then, in the second level, the logical plan is executed by applying the cleaning tasks that are needed. To execute the plan, we employ a query answer relaxation technique, which enhances the answer of the query with extra information from the dataset in order to detect violations based on the output of each query operator that is affected by a constraint. Then, given the detected violations, we transform the query answer into a probabilistic answer by replacing each erroneous value with the set of values that represent candidate fixes for that value. In addition, we accompany each candidate value with the corresponding probability of being the correct value of the erroneous cell. After cleaning each query answer, the system extracts the changes made to the erroneous tuples, and updates the original dataset accordingly. By applying the changes after each query, we can gradually clean the dataset.

3.1 Logical-level Optimizations

In the first stage, the system translates the query into a logical plan involving query and cleaning operators. The cleaning operators are update operators which either operate over the underlying dataset, or over the condition that exists below them in the query plan. To place the cleaning operators, the system determines whether it is more efficient and/or accurate to integrate the query with the cleaning task, and partially clean the dataset, or to fully clean the dataset before executing the query. To decide on the cleaning strategy, we employ a cost model which exploits statistics regarding the type and frequency of the violations. To optimally place the cleaning operators, the system examines: a) the approximate number of violations that exist in the dataset, and b) how the query operators overlap with the erroneous attributes. Thus, the statistics provide an estimate of the overhead that the cleaning task adds to each query, and determine the optimal placement of the cleaning operations.

At the logical plan level, we apply a set of optimizations by pruning unnecessary cleaning checks, and unnecessary query operators. To apply the optimizations, we analyze how the input constraints that must hold

in the dataset affect the query result. For example, it is redundant to apply a cleaning task in the case of a query that contains a filter condition over a clean attribute. Therefore, the logical plan will select the optimal execution strategy of the queries given the cleaning tasks that need to be applied.

3.2 Relaxed query execution

In the final stage, the system executes the optimized logical plan, and computes a correct query answer by applying the cleaning tasks at query execution. Regardless of the type of query, we need to enhance the query answer with extra tuples from the dataset to allow the detection and repairing of errors. Executing queries over dirty data might result in wrong query answers [19]; a tuple might erroneously satisfy a query and appear in the query answer due to a dirty value, or similarly, it might be missing from the query answer due to a dirty value.

To provide correct answers over dirty data, we employ query answer relaxation [30, 36]. Query relaxation has been used successfully to address the problem of queries returning no results, or to facilitate query processing over incomplete databases. We define and employ a novel query answer relaxation technique in the context of querying dirty data, which enhances the query answer with extra tuples from the dataset that allow the detection of violations of integrity constraints. Then, given the detected errors, we propose candidate fixes by providing probabilistic answers [39]. The probabilities are computed based on the frequency that each candidate value appears - other schemes to infer the probabilities are also applicable. The purpose of the query answer relaxation mechanism is to enhance the query answer with the required information from the dataset, in order to allow correct answers to the queries.

To capture errors in query results, we first compute the dirty query answer, and then relax it by bringing extra tuples from the dataset; the extra tuples, together with the tuples of the query answer represent the candidates for satisfying the query. The set of extra tuples consist of tuples which are similar to the ones belonging to the query answer; the similarity depends on the correlation that the tuples have with respect to the integrity constraints that hold in the dataset [41]. After enhancing the query answer with the extra tuples, the cleaning process detects for violations and computes the set of candidate values for each erroneous cell together with their probabilities.

By integrating data cleaning with query execution using the aforementioned two-level process, we minimize the cost of data preparation; we efficiently clean only the part of the dataset that is accessed by the queries. In addition, by providing probabilistic answers for the erroneous entities, we reduce human effort, since users can select the correct values among the set of candidate values over the answers of the queries.

4 Adapting Data Access Paths to Workload and Resources

Apart from loading and cleaning decisions, as data-centric applications become more complex, users face new challenges when exploring data, which are magnified with the ever-increasing data volumes. Data access methods have to dynamically adapt to evolving workloads and take advantage of relaxed accuracy requirements. Furthermore, query processing systems must be knowledgeable of the available resources and maximize resource utilization thereby reduce waste. To address the variety of workloads we design different adaptive access path selection approaches depending on application precision requirements.

Adaptive indexing over raw data. To achieve efficient data access for applications requiring precise results despite dynamic workloads we propose adaptive indexing for in-situ query processing. We use state-of-the-art in-situ query processing approaches to minimize data-to-query time. We introduce a fine-grained logical partitioning scheme and combine it with a lightweight indexing strategy to provide near-optimal raw data access with minimal overhead in terms of execution time and memory footprint. To reduce the index selection overhead we propose an adaptive technique for on-the-fly partition and index selection using an online randomized algorithm.

Adapt access paths to approximation. Apart from adapting to data distribution, we need to enable scaling of access paths despite ever-increasing datasets. We take advantage of the relaxed precision requirements posed

by data scientists who tolerate imprecise answers for better query performance [11]. Existing approaches [2, 8], either require full a priori knowledge of the workload to generate the required approximate data structures or improve performance through minimizing data access at query time. We design and demonstrate an adaptive approach which generates synopses (summaries of the data, such as samples, sketches, and histograms) as a by-product of query execution and re-uses them for subsequent queries. It dynamically decides upon synopsis materialization and maintenance while being robust to workload and storage budget changes. To support interactive query performance for ever increasing datasets and dynamic exploratory workloads there is need for relaxed precision guarantees which enable the use of approximate data structures and reduce the size of stored and processed data.

These aforementioned observations serve as a platform to show the following key insights: i) Taking advantage of data characteristics in files can complement in-situ query processing approaches by building data distribution conscious access paths. Data properties such as ordering or clustering enable the construction of access paths spanning subsets of a dataset thereby reducing the cost of tuning and storage, while minimizing data access costs and further reducing the data-to-insight time. ii) Ever-increasing datasets make precise access paths prohibitively expensive to build and store. Similarly, using data synopses as a drop-in replacement for indexes limits their benefits. On the contrary, integrating synopses as a first-class citizen in query optimization and materializing synopses during query execution and re-using them across queries improves scalability and reduces preprocessing. iii) Static tuning decisions can be suboptimal in the presence of shifting exploratory workloads. On the other hand, adapting access paths online, according to the workload while adhering to accuracy requirements is key to provide high query performance in the presence of workload changes.

4.1 Adaptive indexing over Raw data files

Executing queries over raw data files, despite reducing cost through avoiding the initial data loading step, it enables the access of data files by multiple applications thus it prohibits the physical manipulation of data files. Building efficient data access paths requires physical re-organization of files to reduce random accesses during query execution. To overcome this constraint we propose an online partitioning and indexing tuner for in-situ query processing which when plugged into a raw data query engine, offers fast queries over raw data files. The tuner reduces data access cost by: i) logically partitioning a raw dataset to virtually break it into smaller manageable chunks without physical restructuring, and ii) choosing appropriate indexing strategies over each logical partition to provide efficient data access. The tuner dynamically adapts the partitioning and indexing scheme as a by-product of query execution. It continuously collects information regarding the values and access frequency of queried attributes at runtime. Based on this information, it uses a randomized online algorithm to define the logical partitions. For each logical partition, the tuner estimates the cost-benefit of building partition-local index structures considering both approximate membership indexing (i.e., Bloom filters and zone maps) and full indexing (i.e., bitmaps and B + trees). By allowing fine-grained indexing decisions our proposal makes the decision of the index shape at the level of each partition rather than the overall relation. This has two positive side-effects. First, there is no costly investment for indexing that might prove unnecessary. Second, any indexing effort is tailored to the needs of data accesses on the corresponding range of the dataset.

4.2 Adapting to Relaxed Precision

State-of-the-art AQP engines are classified into two categories, depending on the assumptions they make about the query workload. *Offline AQP* engines (e.g. BlinkDB [2] and STRAT [8]) target applications where the query workload is known a priori, e.g., aggregate dashboards that compute summaries over a few fixed columns. Offline AQP engines analyse the expected workload to identify the optimal set of synopses that should be generated to provide fast responses to the queries at hand, subject to a predefined storage budget and error tolerance specification. Since this analysis is time-consuming, both due to the computational complexity of the

analysis task, as well as the I/O overhead in generating the synopses, AQP engines perform the analysis offline, each time the query workload or the storage budget changes. Offline AQP engines substantially improve query execution time under predictable query workloads, however their need for a priori knowledge of the queries makes them unsuitable for unpredictable workloads. To address unpredictable workloads *online AQP* techniques introduce approximation at query runtime. State-of-the-art online AQP engines achieve this by introducing samplers during query execution. By reducing the input tuples, samplers improve performance of the operators higher in the query plan. In this way, online AQP techniques can boost unknown query workloads. However, query-time sampling is limited in the scope of a single query, as the generated samples are not constructed with the purpose of reuse across queries – they are specific to the query, and are not saved. Thus, online AQP engines offer substantially constrained performance gains compared to their offline counterparts for predictable workloads.

In summary, all state-of-the-art AQP engines sacrifice either generality or performance, as they make static, design-time decisions based on a fixed set of assumptions about the query workload and resources. However, modern data analytics workloads are complex, far from homogeneous, and often contains a mix of queries that vary widely with respect to the degree of approximability [2].

We design a self-tuning, adaptive, online AQP engine. Our design builds upon ideas from (adaptive) database systems, such as intermediate result materialization, query subsumption, materialized view tuning and index tuning, and adapts these in the context of AQP, while also enabling a combination and extension of the benefits of both offline and online approximation engines. We extend the ideas of online AQP by injecting approximation operators in the query plan, and enabling a broad range of queries over unpredictable workloads. By performing online materialization of synopses as a byproduct of query execution, we provide performance on-par with offline AQP engines under predictable workloads, yet without an expensive offline preparation phase. The main components of our system are the enhanced optimizer which enables the use of approximate operators and matches existing synopses, and the online tuner which decides on the materialization of intermediate results.

Integrating approximation to optimizer. Our system extends a query optimizer with just-in-time approximation capabilities. The optimizer injects synopses operators into the query plan before every aggregation. Intuitively, this represents the potential to approximate at that location. Subsequently, by using transformation rules, it pushes the synopses operators closer to the raw input. The alternatives generated by rules have no worse accuracy but can have better performance. The optimizer calculates the cost of each plan using data statistics to decide a plan that adheres to user accuracy requirements and improves performance. Based on the generated query plans, the optimizer compares whether any of the already materialized synopses may be re-used. To be re-used a synopsis must (i) satisfy the accuracy guarantees requirements, and (ii) subsume the required set of data. If no existing synopses are candidates for re-use, the optimizer interacts with the online tuner to decide whether to materialize intermediate results.

Online Tuner. The optimizer feeds every prospective approximate plan to the online tuner which stores execution metadata considering historical plans (e.g., appearance frequency, execution cost). Based on the historical plans, the tuner decides whether to introduce a materializer operator to generate a summary. The tuner's decisions are driven by a cost:utility model, which leads to a formalization of the task as an optimization challenge. As the optimizer already ensures the precision of the query results, the decisions made by the tuner affect solely query performance, and not the required accuracy. Finally, the tuner keeps track of the available storage budget and decides on storage location and replication for a materialized sub-plan. The tuner based on the available storage and the cost:benefit model decides whether and which synopses to be stored or evicted.

By using approximate query processing one allows for low latency in return for relaxed precision. However, the ever-increasing data sizes introduce challenges to such systems. Specifically, offine approximation approaches in order to offer low response time, they require long pre-processing, full future workload knowledge and have high storage requirements. On the other hand, online approximation approaches although have no preprocessing, storage requirements and are workload-agnostic they have small performance gains. Our ap-

proach adaptively combines the two approaches and trades precision and storage for performance at runtime offering the best of both worlds.

5 Related Work

Our philosophy has been inspired by the omnipresent work on minimizing data-to-insight time. In-situ processing approaches, such as the work by Idreos et al. [22] propose adaptive and incremental loading techniques in order to eliminate the preparation phase before query execution. NoDB [6] advances this idea by making raw files first-class citizens. NoDB introduces data structures to adaptively index and cache raw files, tightly integrates adaptive loads, while implementing in-situ access into a modern DBMS. In the context of processing heterogeneous raw data, Spark and Hadoop-based systems [1, 42] operate over raw data, while also supporting heterogeneous data formats. RAW [27] allows queries over heterogeneous raw files using code generation. ViDa [26] envisions effortlessly abstracting data out of its form and manipulating it regardless of its structure, in a uniform way.

Work on reducing the data cleaning cost by automating and optimizing common cleaning tasks significantly reduces human effort and minimizes the preprocessing cost. BigDansing [28] is a scale-out data cleaning system which addresses performance, and ease-of-use issues in the presence of duplicates and integrity constraint violations. Tamr, the commercial version of Data Tamer [38], focuses on duplicate elimination by employing blocking and classification techniques in order to efficiently detect and eliminate duplicate pairs. In the context of adaptive and ad-hoc cleaning QuERy [7] intermingles duplicate elimination with Select Project, and Join queries in order to clean only the data that is useful for the queries. Transform-Data-by-Example [20] addresses the problem of allowing on-the-fly transformations - a crucial part of data preparation.

Work on adaptive tuning focuses on incrementally refining indexes while processing queries. Database Cracking approaches [23] operate over column-stores and incrementally sort the index column according to the incoming workload, thus reducing memory access. COLT [34] continuously monitors the workload and periodically creates new indexes and/or drops unused ones by adding an overhead to each query.

A plethora of research topics on approximate query processing is also relevant to our work. Offline sampling strategies [2, 8] focus on computing the best set of uniform and stratified samples given a storage budget by assuming some a priori knowledge of the workload. Online sampling approaches such as Quickr [24] take samples during query execution by injecting samplers inside the query plan.

6 Summary and Next Steps

The constantly changing needs for efficient data analytics combined with the ever growing datasets, require a system design which is flexible, dynamic and embraces adaptivity. We present techniques which streamline processes that constitute bottlenecks in data analysis and reduce the overall data-to-insight time. To remove data loading, we introduce a system that adapts to data heterogeneity and enables queries on variety of data formats. To reduce data cleaning overheads, we overlap cleaning operations with query execution and finally, we introduce physical tuning approaches which take advantage of data distribution as well as the reduced precision requirements of modern analytics applications.

References

[1] Apache drill. `https://drill.apache.org/`.

[2] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 29–42, 2013.

[3] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1110–1121, 2004.

[4] A. Ailamaki, V. Kantere, and D. Dash. Managing scientific data. *Communications of the ACM*, 53, 06 2010.

[5] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 241–252, 2012.

[6] I. Alagiannis, R. Borovica-Gajic, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. *Communications of the ACM*, 58(12):112–121, 2015.

[7] H. Altwaijry, S. Mehrotra, and D. V. Kalashnikov. QuERy: A Framework for Integrating Entity Resolution with Query Processing. *PVLDB*, 9(3), 2015.

[8] S. Chaudhuri, G. Das, and V. Narasayya. A Robust, Optimization-based Approach for Approximate Answering of Aggregate Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 295–306, 2001.

[9] S. Chaudhuri and V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 146–155, 1997.

[10] Y. Chen, S. Alspaugh, and R. H. Katz. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, 2012.

[11] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012.

[12] M. Dallachiesa, A. Ebaid, A. Eldawy, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. NADEEF: A Commodity Data Cleaning System. In *SIGMOD*, 2013.

[13] T. Dasu and T. Johnson. *Exploratory Data Mining and Data Cleaning*. John Wiley & Sons, Inc., New York, NY, USA, 1 edition, 2003.

[14] W. Fan. Dependencies revisited for improving data quality. In *PODS*, 2008.

[15] W. Fan. Data quality: From theory to practice. *SIGMOD Rec.*, 44(3):7–18, Dec. 2015.

[16] L. Fegaras and D. Maier. Optimizing Object Queries Using an Effective Calculus. *TODS*, 25(4):457–516, Dec. 2000.

[17] S. Giannakopoulou. Query-driven data cleaning for exploratory queries. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[18] S. Giannakopoulou, M. Karpathiotakis, B. Gaidioz, and A. Ailamaki. Cleanm: An optimizable query language for unified scale-out data cleaning. *Proc. VLDB Endow.*, 10(11):1466–1477, Aug. 2017.

[19] P. Guagliardo and L. Libkin. Making sql queries correct on incomplete databases: A feasibility study. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '16, pages 211–223, New York, NY, USA, 2016. ACM.

[20] Y. He, K. Ganjam, K. Lee, Y. Wang, V. Narasayya, S. Chaudhuri, X. Chu, and Y. Zheng. Transform-data-by-example (tde): Extensible data transformation in excel. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1785–1788, New York, NY, USA, 2018. ACM.

[21] IBM. Managing big data for smart grids and smart meters. *IBM White Paper*, 2012.

[22] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 57–68, 2011.

[23] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *Proceedings of the VLDB Endowment*, 4(9):586–597, 2011.

[24] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *SIGMOD*, 2016.

[25] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast Queries Over Heterogeneous Data Through Engine Customization. *Proceedings of the VLDB Endowment*, 9(12):972–983, 2016.

[26] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki. Just-In-Time Data Virtualization: Lightweight Data Management with ViDa. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.

[27] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive Query Processing on RAW Data. *Proceedings of the VLDB Endowment*, 7(12):1119–1130, 2014.

[28] Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and S. Yin. BigDansing: A System for Big Data Cleansing. In *SIGMOD*, 2015.

[29] S. Lohr. For Big-Data Scientists, 'Janitor Work' Is Key Hurdle to Insights, The New York Times, 2014.

[30] I. Muslea and T. J. Lee. Online query relaxation via bayesian causal structures discovery. In *AAAI*, 2005.

[31] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Slalom: Coasting Through Raw Data via Adaptive Partitioning and Indexing. *Proceedings of the VLDB Endowment*, 10(10):1106–1117, 2017.

[32] M. Olma, O. Papapetrou, R. Appuswamy, and A. Ailamaki. Taster: Self-Tuning, Elastic and Online Approximate Query Processing. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2019.

[33] S. Papadomanolakis and A. Ailamaki. AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, page 383, 2004.

[34] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. COLT: Continuous On-Line Database Tuning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 793–795, 2006.

[35] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 302–314, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[36] S. Shen. Database relaxation: An approach to query processing in incomplete databases. *Information Processing and Management*, 24(2):151 – 159, 1988.

[37] M. Stonebraker. Technical perspective - one size fits all: an idea whose time has come and gone. *Commun. ACM*, 51:76, 2008.

[38] M. Stonebraker, G. Beskales, A. Pagan, D. Bruckner, M. Cherniack, S. Xu, V. Analytics, I. F. Ilyas, and S. Zdonik. Data Curation at Scale: The Data Tamer System. In *CIDR*, 2013.

[39] D. Suciu, D. Olteanu, R. Christopher, and C. Koch. *Probabilistic Databases*. Morgan & Claypool Publishers, 1st edition, 2011.

[40] J. W. Tukey. *Exploratory data analysis*. Addison-Wesley series in behavioral science : quantitative methods. Addison-Wesley, 1977.

[41] M. Yakout, L. Berti-Équille, and A. K. Elmagarmid. Don't be scared: Use scalable automatic repairing with maximal likelihood and bounded changes. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 553–564, New York, NY, USA, 2013. ACM.

[42] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *NSDI*, 2012.

[43] M. Zwolenski, L. Weatherill, et al. The digital universe: Rich data and the increasing value of the internet of things. *Australian Journal of Telecommunications and the Digital Economy*, 2(3):47, 2014.

doppioDB 1.0: Machine Learning inside a Relational Engine

Gustavo Alonso[1], Zsolt Istvan[2], Kaan Kara[1], Muhsen Owaida[1], David Sidler[1]
[1]Systems Group, Dept. of Computer Science, ETH Zurich, Switzerland
[2]IMDEA Software Institute, Madrid, Spain

Abstract

Advances in hardware are a challenge but also a new opportunity. In particular, devices like FPGAs and GPUs are a chance to extend and customize relational engines with new operations that would be difficult to support otherwise. Doing so would offer database users the possibility of conducting, e.g., complete data analyses involving machine learning inside the database instead of having to take the data out, process it in a different platform, and then store the results back in the database as it is often done today. In this paper we present doppioDB 1.0, an FPGA-enabled database engine incorporating FPGA-based machine learning operators into a main memory, columnar DBMS (MonetDB). This first version of doppioDB provides a platform for extending traditional relational processing with customizable hardware to support stochastic gradient descent and decision tree ensembles. Using these operators, we show examples of how they could be included into SQL and embedded as part of conventional components of a relational database engine. While these results are still a preliminary, exploratory step, they illustrate the challenges to be tackled and the advantages of using hardware accelerators as a way to extend database functionality in a non-disruptive manner.

1 Introduction

Data intensive applications are often dominated by online analytic processing (OLAP) and machine learning (ML) workloads. Thus, it is important to extend the role of the database management system to a more comprehensive platform supporting complex and computationally intensive data processing. This aspiration is, however, at odds with existing engine architectures and data models. In this work, we propose to take advantage of the ongoing changes in hardware to extend database functionality without having to completely redesign the relational engine. The underlying hardware for this work, field programmable gate arrays (FPGAs), is becoming more common both in cloud deployments (e.g., Microsoft's Catapult or Amazon's F1 instances) and in conventional processors (e.g., Intel's hybrid architectures incorporating an FPGA into a CPU). FPGAs can be easily reprogrammed to provide the equivalent of a customizable hardware architecture. Thus, the FPGA can be used as a hardware extension to the database engine where additional functionality is implemented as a complement to that already available.

We have implemented this idea in a first prototype of doppioDB, identified here as doppioDB 1.0 to distinguish it from future versions, showing how to integrate machine learning operators into the database engine in a way that is both efficient (i.e., compute-intensive algorithms do not impose overhead on native database

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

workloads) and effective (i.e., standard operator models and execution patterns do not need to be modified). doppioDB runs on top of Intel's second generation Xeon+FPGA machine and it is based on MonetDB, an open source main memory columnar database.

Combining machine learning (ML) tasks with database management systems (DBMS) is an active research field and there have been many efforts exploring this both in research [1, 2, 3, 4, 33] and industry [5, 6]. This combination is attractive because businesses have massive amounts of data in their existing DBMS and there is a high potential for using ML to extract valuable information from it. In addition, the rich relational operators provided by the DBMS can be used conveniently to denormalize a complex schema for the purposes of ML tasks [7].

2 Prototyping Platform

2.1 Background on FPGAs

Field Programmable Gate Arrays (FPGAs) are reconfigurable hardware chips. Once configured, they behave as application-specific integrated circuits (ASIC). Internally they are composed of programmable logic blocks and a collection of small on-chip memories (BRAM) and simple arithmetic units (DSPs) [8]. Their computational model is different from CPUs: instead of processing instruction by instruction, algorithms are laid out spatially on the device, with different operations all performed in parallel. Due to the close proximity of logic and memory on the FPGA, building pipelines is easy, and thanks to the flexibility of the on-chip memory, custom scratch-pad memories or data structure stores can be created.

Current FPGA designs usually run at clock-rates around 200-400 MHz. To be competitive with a CPU, algorithms have to be redesigned to take advantage of deep pipelines and spatial parallelism.

2.2 Intel Xeon+FPGA Platform

While the use of FPGAs for accelerating data processing has been studied in the past, it is the emergence of hybrid CPU+FPGA architectures that enables their use in the context of a database with a similar overhead as NUMA architectures. In the past, FPGAs and other hardware accelerators, such as GPUs, have been placed "on the side" of existing database architectures much like an attachment rather than a component [9, 10, 11]. This approach requires data to be moved from the main processing unit to a detached accelerator. As a result, system designs where whole operators (or operator sub-trees) are offloaded to the accelerator are favored compared to finer integration of the accelerated operators into the query plan. We have designed doppioDB for emerging heterogeneous platforms where the FPGA has direct access to the main memory of the CPU, avoiding data copy. These platforms have also opened up opportunities of accelerating parts of operators such as partitioning or hashing [12] instead of full operations or even entire queries.

We use the second generation Intel Xeon+FPGA machine[1](Figure 1) that is equipped with an Intel Xeon Broadwell E5 with 14 cores running at 2.4 GHz and, in the same package as the Xeon, an Intel Arria 10 FPGA. The machine has 64 GB of main memory shared with the FPGA. Communication happens over 1 QPI and 2 PCIe links to the memory controller of the CPU. These are physical links as the FPGA is not connected via the PCIe bus. The resulting aggregated peak bandwidth is 20 GB/s.

On the software side, Intel's *Accelerator Abstraction Layer* (AAL) provides a memory allocator to allocate memory space shareable between the FPGA and the CPU. This allows data to be accessed from both the CPU and the FPGA. Apart from the restrictions of the operating system, such as not supporting memory mapped files for use in the FPGA, the rest of the memory management infrastructure has remained untouched.

[1]Results in this publication were generated using pre-production hardware and software donated to us by Intel, and may not reflect the performance of production or future systems.

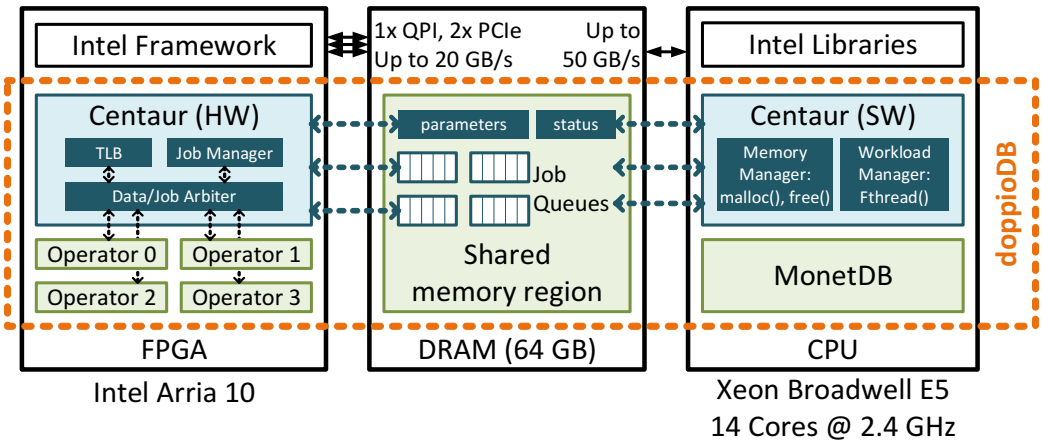

Figure 1: doppioDB using Centaur on Intel's Xeon+FPGA second generation prototype machine.

2.3 MonetDB

We use MonetDB as the basis of doppioDB. MonetDB is an open source columnar read-optimized database designed for fast analytics. It stores relational tables as a collection of columns. A column consists of a memory heap with values and a non-materialized positional identifier. Because of this design, the values of a column are always stored in consecutive memory (or a memory mapped file) and they are addressable by simple memory pointers. MonetDB follows the operator-at-a-time paradigm and materializes the intermediate results of each operator. These design features make MonetDB suitable for integrating a hardware accelerator as they often require well-defined memory and execution boundaries per column and per operator.

3 doppioDB: Overview

Integrating machine learning operators in an OLAP-oriented database such as MonetDB requires to tackle several challenges. Many ML operators are iterative and scan the data multiple times. For example, to train a model on an input relation, the relation should be materialized before training starts. In a tuple-at-a-time execution model of the operator tree, every operator is invoked once per input tuple. As a result, iterative ML operators cannot fit in this execution model without changing the query execution engine. On the other hand, in an operator-at-a-time execution model, an operator processes all the input tuples at once before materializing its result and passing it to the next operator in the tree. In this execution model, the iterative nature of an ML operator is hidden inside the operator implementation and does not require to be exposed to the query execution engine. In addition, an operator-at-a-time execution model eliminates the cost of invoking the FPGA operator for every tuple.

Another challenge is the row-oriented data format required for ML operators. Column-oriented data fits OLAP workloads well but most ML algorithms work at tuple-level and therefore require row-oriented data. This puts databases in a difficult position: if data is stored in a row format, OLAP performance suffers. Keeping two copies of the data, one in each format, would introduce storage overhead and would significantly slow down updates. An alternative is to introduce a data transformation step to convert column-oriented data to a row-oriented format. Transforming data on-the-fly using a CPU is possible, but leads to cache-pollution and takes away computation cycles from the actual algorithm. However, on the FPGA, the transformation step can be performed using extra FPGA logic and on-chip memory resources without degrading processing throughput or adding overhead on the query runtime as we discuss in Section 4.2.

When adding new functionality to the database engine, a constant challenge is how to expose this functionality at the SQL level. The use of user defined functions (UDFs) is a common practice, but there are some significant drawbacks with this approach. First, UDFs limit the scope of applicability of the operator, e.g., they do not support updates or they are applied to one tuple at a time. Second, usually a query optimizer perceives UDFs as black boxes that are pinned down in the query plan, missing optimization opportunities. We believe extending SQL with new constructs and keywords allows a better exposure of the functionality and the applicability of complex operators. However, this is not easy to achieve, since the order of execution and the rules of the SQL language has to be respected. Later in the paper we discuss different SQL extensions for ML operators.

Since FPGAs are not conventional compute devices, there is no general software interface for FPGA accelerators. Typically, every accelerator has its own software interface designed for its purposes. However, in the database environment where many different operators will use the FPGA, the customizable hardware needs to be exposed as part of the platform, with general purpose communication and management interfaces. We have implemented the communication between MonetDB and the FPGA using the open-source Centaur[2] [15] framework, which we modify to provide better memory access and extend with a data transformation unit that can be used by operators that require a row-oriented format instead of the default columnar format of MonetDB.

4 Database integration of FPGA based operators

4.1 Communication with the FPGA

There have been many efforts in the FPGA community to generalize FPGA accelerators through software abstractions and OS-like services for CPU-FPGA communication. Examples of these efforts include hThreads [13], ReconOS [14], and Centaur [15]. Since Centaur is developed for the Intel Xeon+FPGA prototype machine and it is open source, we decided to use it in developing doppioDB. In this work, we port Centaur to Intel's second generation Xeon+FPGA (Broadwell+Arria10) platform.

Centaur abstracts FPGA accelerators as hardware threads and provides a clean thread-like software interface, called *FThread*, that hides the low level communication between FPGA and CPU. Its *Workload Manager* (Figure 1) allows for concurrent access to different operators. It guarantees concurrency by allocating different synchronous job queues for different operators types. Overall, this makes it possible to share FPGA resources between multiple queries and database clients. Centaur's *FThread* abstraction allows us to express FPGA operators as separate threads which can be invoked from anywhere in doppioDB. For example, we can use data partitioning on the FPGA as shown in Listing 1. We create an *FThread* specifying that we want to perform partitioning on relation R with the necessary configuration, such as the source and destination pointers, partitioning fanout, etc. After the *FThread* is created, the parent thread can perform other tasks and finally the *FThread* can be joined to the parent similar to C++ threads.

By creating the *FThread* object, we communicate a request to the FPGA to execute an operator. Internally, the request is first queued in the right concurrent job queue allocated in the CPU-FPGA shared memory region, as shown in Figure 1. Then, Centaur's *Job Manager* on the FPGA, monitoring the queues continuously, dequeues the request and starts the execution of the operator. In case all operators of the requested type are already allocated on the FPGA by previous requests, the *Job Manager* has to wait until an operator becomes free before dispatching the new request. The Job Manager scans the different job queues in the shared memory concurrently and independent of each other such that a job queue that has free operator is not blocked with another queue waiting on a busy operator.

On the FPGA, Centaur partitions the FPGA into four independent regions each hosting an accelerator (examples shown in Figure 1). Centaur's *Job Manager* facilitates CPU-FPGA communication and enables concurrent

[2] `https://github.com/fpgasystems/Centaur`

Listing 1: An FPGA operator representation in Centaur.

```
relation *R, *partitioned_R;
...
// Create FPGA Job Config
PARTITIONER_CONFIG config_R;
config_R.source = R;
config_R.destination = partitioned_R;
config_R.fanout = 8192;
...
// Create FPGA Job
FThread R_fthread(PARTITIONER_OP_ID, config_R);
...
// Do some other work
...
// Wait for FThreads to finish
R_fthread.join();
...
```

access to all accelerators. In addition, the *Data Arbiter* multiplexes the access to the memory interface from multiple operators using a round-robin mechanism.

Porting Centaur to the target platform. Centaur's *Memory Manager* implements a memory allocator that manages the CPU-FPGA shared memory region. However, we discovered through our experiments that this custom memory allocator incurs a significant overhead in certain workloads. This is mostly due to a single memory manager having to serve a multi-threaded application from a single memory region. To overcome this, we allow the database engine to allocate tables in the non-shared memory region using the more sophisticated operating system memory allocation. Then, we perform memory copies from the non-shared to the shared memory region only for columns used by the FPGA operators. This is not a fundamental requirement and is only caused by the limitations of the current FPGA abstraction software stack: In future iterations of the Xeon+FPGA machine, we expect that the memory management for FPGA abstraction libraries will be integrated into the operating system, thus giving Centaur the ability to use the operating system memory allocation directly. The FPGA can then access the full memory space, without memory copies.

Beyond the modification to the memory allocator, we changed the following in Centaur: First, we clocked up Centaur FPGA architecture from 200 MHz to 400 MHz to achieve a 25 GB/s memory bandwidth. Operators can still be clocked at 400 or 200 MHz. In addition, we replaced the FPGA pagetable, which is limited to 4 GB of shared memory space, with the Intel's MPF module which implements a translation look-aside buffer (TLB) on the FPGA to support unlimited shared memory space. We also added a column to row conversion unit to support operators which require row-oriented data format.

4.2 On-the-fly Data Transformation

In doppioDB we support machine learning operators on columnar data by adding a "transformation" engine that converts data on-the-fly to a row-oriented representation (Figure 2). The engine is part of the *Data Arbiter* in Figure 1 which is plugged in front of the operator logic. The design can be generalized and such transformations can be done across many different formats (data encodings, sampling, compression/decompression, encryption/decryption, summarization, etc.), a line of research we leave for future work. Such transformations are essentially "for free" (without impacting throughput and using a small part of the resources) in the FPGA and, as such, will change the way we look at fixed schemas in database engines.

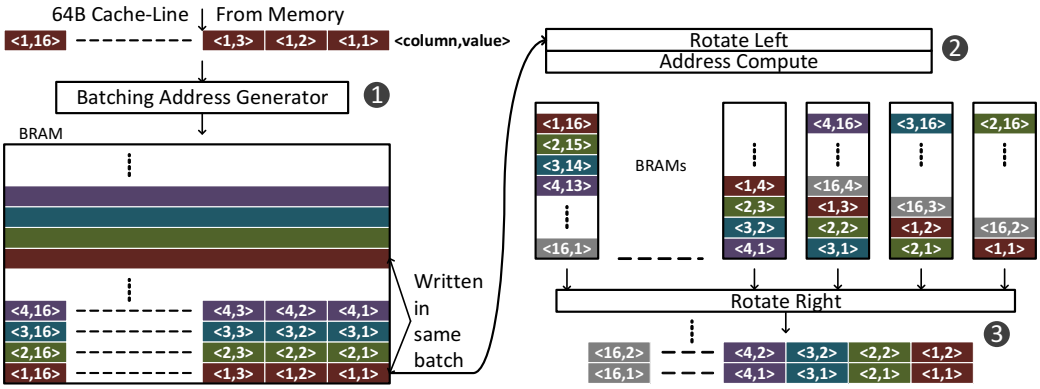

Figure 2: On the FPGA the transformation from columns to rows can be implemented as a streaming operation that introduces latency but has constant bandwidth.

When designing this engine we made several assumptions. First, data belonging to each dimension resides in its own column, and the ordering of tuples per column is the same (there are no record-IDs, tuples are associated by order instead) . This allows us to scan the different columns based on a set of column pointers only. While the actual type of the data stored in each column is not important for this unit, our current implementation assumes a data width of 4 Bytes per dimension. This is, however, not a fundamental limitation and the circuit could be extended to support, for instance, 8 Byte values as well.

The gather engine, as depicted in Figure 2, requests data belonging to different columns in batches to reach high memory bandwidth utilization. A batch is a number of successive cache lines requested from a single dimension column before reading from the next dimension column. Each cache line contains 16 entries of a column (16*4 B=64 B). The incoming data is scattered across a small reorder memory such that, when read sequentially, this memory returns one line per dimension (1). In the next step these lines are rotated and written into smaller memories in a "diagonal" fashion (2). This means that if the first 4 bytes are written to address 0 of the first memory, the second 4 bytes will go to address 1 of the second memory and so on. This layout ensures that when reading out the same address in each of these small memories, the output will contain one 4 Byte word from each dimension. With an additional rotate operation, we obtain a cache-line having values from each column in their respective positions (3). Thus, the flexibility of the FPGA allows us to build a "specialized cache" for this scatter-gather type of operation that would not be possible on a CPU's cache.

The nominal throughput of this unit is 12.8 GB/s at 200 MHz and is independent of the number of dimensions. As Figure 3 shows, throughput close to the theoretical maximum can already be achieved when batching 32 cache lines. The effect of having multiple dimensions is visible because DRAM access is scattered over a larger space, but with a sufficiently large batch size, all cases converge to 11.5 GB/s.

In terms of resource requirements, the number of BRAMs needed to compose the scatter memory depends on the maximum number of dimensions and the maximum batching factor, since at least one batch per dimension has to be stored. The choice for both parameters is made at compile-time. At runtime it is possible to use less dimensions and, in that case, the batching factor can be increased correspondingly. The number of the smaller memories is fixed (16), but their depth depends on the maximum number of dimensions. Even when configured for up to 256 dimensions with a batching factor of 4, only 128 kB of the on-chip BRAM resources are needed for this circuit.

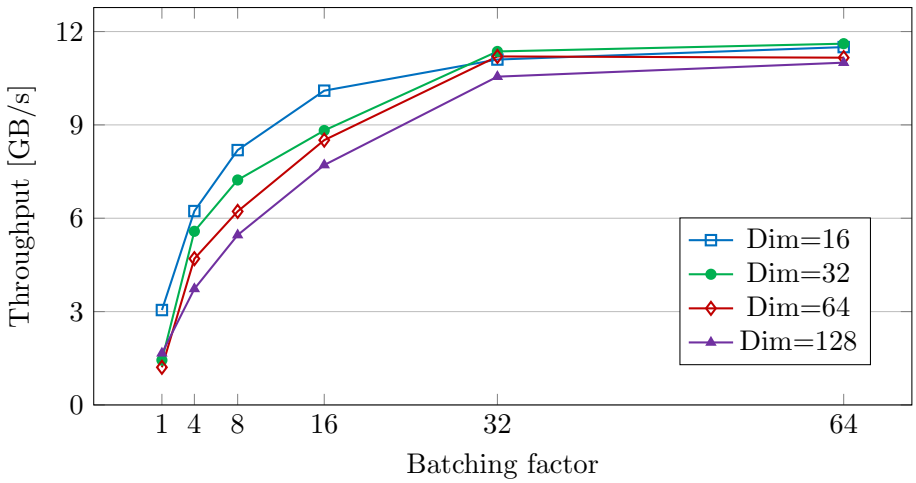

Figure 3: Streaming transformation from columns to rows reaches high throughput and is not impacted negatively by the number of dimensions to be gathered.

Listing 2: Template queries to train models on relations and do inference with trained models

```
Q_train: CREATE MODEL model_name ON
            (SELECT attr1, attr2, ..., label FROM data_set WHERE ...)
         WITH model_type USING training_algorithm(algorithm_parameters);

/*Infer without modifying the table*/
Q_infer1: SELECT new_data_set.id,
          INFER('model_name') FROM new_data_set;

/*Infer and save results into a table*/
Q_infer2: INSERT INTO inferred_data_set(label)
          SELECT INFER('model_name') FROM new_data_set;
```

5 Stochastic Gradient Descent

Overview By including a stochastic gradient descent (SGD) in doppioDB, our goal is to show that the FPGA-enabled database is capable of efficiently handling iterative model-training tasks. The SGD operator enables us to *train* linear regression models and support vector machines (SVM) on the FPGA using relational data as input. There has been many studies showing the effectiveness of FPGA-based training algorithms [16, 17, 18, 33, 35]. We based our design on open-sourced prior work by Kara et al. [18], which performs both gradient calculation and model update on the FPGA using fine grained parallelism and a pipelined design. We integrated the SGD training algorithm into a DBMS in two steps: We extended SQL to enable a user's *declarative* interaction with ML operators, followed by the physical integration of FPGA-accelerated ML operators into the DBMS.

SQL Integration There has been many efforts to enable the usage of ML operators directly from SQL. While most efforts (MADlib [1], SAP HANA [5] and Oracle Data Miner [6]) expose ML operators as user defined functions (UDFs), some recent work considered extending SQL with new keywords to make ML operators in SQL more transparent. For instance, Passing et al. [2] propose to introduce the "ITERATE" keyword to SQL to represent the iterative nature of ML training algorithms. To accomplish the same goal, Cai et al. [4] propose the "FOR EACH" keyword. In this work, we argue that the interaction with ML operators in a DBMS should be

Listing 3: Queries to train various models on any desired projection using SGD.

```
Q1: CREATE MODEL proteins_model ON
      (SELECT attr1, attr2, ..., attr15, label FROM human_proteome)
    WITH LINREG USING SGD(num_iterations, learning_rate);

Q2: CREATE MODEL detect_fraud ON
      (SELECT Name, ..., IsFraud FROM transactions
      WHERE IsFraud IS NOT NULL)
    WITH SVM USING SGD(num_iterations, learning_rate);

Q3: CREATE MODEL stock_predictor ON
      (SELECT Region, ..., OpenCloseValues.Close FROM Transactions, Actors,
          Stocks, OpenCloseValues
      WHERE Stocks.Region = 'Europe' AND YEAR(OpenCloseValues.Date) > 2010 AND
          Actors.Position = 'Manager')
    WITH SVM USING SGD(num_iterations, learning_rate);
```

Listing 4: Inference queries to make predictions on tuples with empty labels.

```
Q1: SELECT human_proteome.id, INFER('proteins_model') AS prediction
    FROM human_proteome WHERE label IS NULL;

Q2: SELECT transactions.name, INFER('detect_fraud')
    FROM transactions WHERE IsFraud IS NULL;

Q3: SELECT Stocks.Name, INFER('stock_predictor')
    FROM Transactions, Actors, Stocks, OpenCloseValues
    WHERE OpenCloseValues.Close IS NULL;
```

done in a more simple and intuitive way than previously proposed. To accomplish this, we propose a new SQL structure as shown in Listing 2.

With the structure shown in Listing 2, the user specifies (1) the model name after **CREATE MODEL**, (2) the attributes and the label that the model should be trained on after **ON**, (3) the type of the ML model after **WITH** (e.g., support vector machine, logistic regression, decision trees, neural networks etc.), (4) the training algorithm along with the parameters after **USING** (e.g., SGD, ADAM etc.). After the model is created, we can use it for inference on new tables using the **INFER** keyword, passing the model name. A model in doppioDB contains besides the actual ML model parameters also meta-parameters, specifying which attributes it was trained on. The **INFER** function ensures during query compilation that it receives all the attributes necessary according to the meta-parameters of the model. Otherwise, the SQL compiler raises a compile time error.

Listing 3 shows how the syntax we introduce can be used on realistic scenarios. For instance, in *Q3*, the ability to perform multiple joins and selections on four relations and then to apply a training algorithm on the projection is presented. This is a very prominent example showing the convenience of declarative machine learning. In Listing 4, three inference queries with **INFER** are presented, using the models created by **CREATE MODEL** queries, again showing the convenience of performing prediction on tuples with an empty label.

Physical Integration The trained model is stored as an internal data structure specific to given relational attributes –similar to an index– in the database. Inside the **CREATE MODEL** query, the training (iterative reading) happens over the resulting projection of the subquery inside **ON**(...). The operator-at-a-time execution

Table 1: Stochastic Gradient Descent Training Time

Data set	#Tuples	#Feat.	#Epochs	doppioDB (CPU)	doppioDB (FPGA)
proteome	38 Mio.	15	10	10.55 s	3.44 s
transactions	6.4 Mio.	6	100	7.35 s	2.54 s
stocks	850 K.	5	100	0.92 s	0.32 s

of MonetDB fits well here: The training-related data is materialized once and is read multiple times by the SGD engine. In case of FPGA-based SGD, the pointers to the materialized data (multiple columns) are passed to the FPGA, along with training related parameters such as the number of iterations and the learning rate. The FPGA reads the columns corresponding to different attributes with the help of the gather engine as described in Section 4.2, reconstructing rows on-the-fly. The reconstruction is needed, because SGD requires all the attributes of a sample in the row-format to compute the gradient.

The tuples created by the subquery are read as many times as indicated by the number of iterations. For each received tuple, the SGD-engine computes a gradient using the model that resides on the FPGA-local on-chip memory. The gradient is directly applied back to the model on the FPGA, so the entire gradient descent happens using only the on-chip memory, reserving external memory access just for the training data input. After the training is complete, the model is copied from on-chip memory to the main memory of the CPU, where doppioDB can use it to perform inference.

Evaluation We use the following data sets in our evaluation: (1) A human proteome data set [19], consisting of 15 protein-related features and 38 Million tuples (Size: 2.5 GB); (2) A synthetic financial data set for fraud detection [20], consisting of 6 training-related features and 6.4 Million tuples (Size: 150 MB); and (3) A stock exchange data set, consisting of 5 training-related features and 850 Thousand tuples (Size: 17 MB).

In Table 1, we present the time for training a linear SVM model on the data sets, using either the CPU or FPGA implementation. In both cases, the number of epochs (one epoch is defined as a full iteration over the whole data set) and learning rates are set to be equal. Therefore, the resulting models are statistically equal as well. Achieving multi-core parallelism for SGD is a difficult task because of the algorithm's iterative nature, especially for lower dimensional and dense learning tasks. Therefore, we are using a single-threaded and vectorized implementation for the CPU execution. We observe that the FPGA-based training is around 3x faster for both data sets, providing a clear performance advantage. The FPGA-based implementation [18] offers finer grained parallelism, allowing the implementation of specialized vector instructions just for performing SGD, and also puts these instructions in a specialized pipeline, thereby providing higher performance. It is worth noting that the models we are training are linear SVM models, which are relatively small and on the lower compute intensive side compared to other ML models such as neural networks. For larger and more complex models, the performance advantage of specialized hardware will be more prominent [21, 22].

6 Decision Tree Ensembles

Overview A *decision tree* is a supervised machine learning method used in a wide range of classification and regression applications. There have been a large body of research considering the use of FPGAs and accelerators to speedup decision tree ensemble-based inference [23, 24, 25, 26, 27]. The work of Owaida et al. [23, 24] proposes an accelerator that is parameterizable at runtime to support different tree models. This flexibility is necessary in a database environment to allow queries using different models and relations to share the same accelerator. In doppioDB we base our FPGA decision tree operator on the design in [23].

Table 2: Runtime for Decision tree ensemble inference.

Query	#Tuples	CPU-1	CPU-28	doppioDB (FPGA)
Qinfer-1	1,000,000	47.62 s	2.381 s	0.481 s
Qinfer-2	855,819	8.63 s	0.428 s	0.270 s

Integration in doppioDB The original implementation works on row-oriented data, so as a first step, we replaced the data scan logic with the gather engine described in Section 4.2. As a result our implementation operates on columnar data in doppioDB without the need for any further changes to the processing logic. To integrate the decision trees into doppioDB, we use the same SQL extensions proposed for SGD to create models and perform inference as in Listing 2. In Listing 5 we show two examples of training and inference queries for the *higgs* and *physics* relations. Since currently we do not implement decision tree training in doppioDB, the training function *DTree('filename')* imports an already trained model from a file. The trained model can be obtained from any machine learning framework for decision trees such as XGBoost [28]. Inside doppioDB, the model is stored as a data structure containing information about the list of attributes used to train the model, the number of trees in the ensemble, the maximum tree depth, the assumed value of a missing attribute during training, and a vector of all the nodes and leaves of all the ensemble trees.

The **INFER** function invokes the FPGA decision tree operator by passing the model parameters and pointers to all the attribute columns to the FPGA. The FPGA engine then loads the model and stores it in the FPGA local memories. Then, the gather engine scans all the attribute columns and constructs tuples to be processed by the inference logic.

Listing 5: Training and inference queries for decision trees on the Higgs and Physics relations.

```
Qtrain-1: CREATE MODEL higgs_model ON
              (SELECT attr1, ..., attr28, label FROM higgs)
          WITH DECTREE USING DTree('higgs_xgboost.model');

Qtrain-2: CREATE MODEL physics_model ON
              (SELECT attr1, ..., attr74, label FROM physics)
          WITH DECTREE USING DTree('physics_xgboost.model');

Qinfer-1: SELECT particles_new.EventId,
          INFER('higgs_model') AS higgs_boson
          FROM particles_new;

Qinfer-2: SELECT physics_new.id,
          INFER('physics_model') AS prediction
          FROM physics_new;
```

Evaluation To evaluate the decision tree operator we used the 'Higgs' data set from [29] and the 'Physics' data set from [30]. The 'Higgs' data set is collected from an experiment simulating proton-proton collisions using the ATLAS full detector simulator at CERN. A tuple consists of 28 attributes (floating point values) which describe a single particle created from the collisions. The experiment objective is to find the Higgs Boson. The training produces a decision tree ensemble of 512 trees, each 7 levels deep. The 'Physics' data set is collected from simulated proton-proton collisions in the LHCb at CERN. The data set consists of 74 attributes. The attributes describe the physical characteristics of the signal decays resulting from the collisions. The objective of the trained model on the data is to detect lepton flavour decay in the proton-proton collisions. If such a decay is detected this indicates physics beyond the standard model (BSM). The trained model consists of 200 trees, each 10 levels deep.

For training, we use XGBoost to train both data sets offline, then we import the trained models using the queries *Qtrain-1* and *Qtrain-2*. Once the models are created and imported into the the database, we run the two inference queries in Listing 5. For comparisons with CPU performance, we use multi-threaded XGBoost implementation as a baseline. Table 2 summarizes the runtime results for inference on FPGA and CPU. The evaluation results demonstrate the superiority of the FPGA implementation over single threaded CPU implementation (CPU-1). Using the full CPU compute power (CPU-28) brings the CPU runtime much closer to the FPGA runtime. However, in a database engine typically there are many queries running at the same time sharing CPU resources, which makes it inefficient to dedicate all the CPU threads to a compute intensive operator such as decision trees inference. The FPGA achieves its superior performance by parallelizing the processing of large number of trees (256 trees in our implementation are processed simultaneously) and eliminating the overhead of random memory accesses through specialized caches on the FPGA to store the whole trees ensemble and data tuples being processed.

7 Conclusions

In this paper we have briefly presented doppioDB, a platform for future research on extending the functionality of databases with novel, compute-intensive, operators. In this work we demonstrate that it is possible to include machine learning functionality within the database stack by using hardware accelerators to offload operators that do not fit well with existing relational execution models. As part of ongoing work, we are exploring more complex machine learning operators more suitable to column store databases [31] and data representations suitable for low-precision machine learning [32]. Apart from machine learning, more traditional data analytics operators such as large scale joins, regular expression matching and skyline queries (pareto optimality problem) also benefit from FPGA-based acceleration, as we have demonstrated in previous work [34].

Acknowledgements We would like to thank Intel for the generous donation of the Xeon+FPGA v2 prototype. We would also like to thank Lefteris Sidirourgos for feedback on the initial design of doppioDB and contributions to an earlier version of this paper.

References

[1] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, *et al.*, "The MADlib analytics library: or MAD skills, the SQL," *PVLDB*, vol. 5, no. 12, pp. 1700–1711, 2012.

[2] L. Passing, M. Then, N. Hubig, H. Lang, M. Schreier, S. Günnemann, A. Kemper, and T. Neumann, "SQL-and Operator-centric Data Analytics in Relational Main-Memory Databases.," in *EDBT'17*.

[3] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn, "Design and implementation of the LogicBlox system," in *SIGMOD'15*.

[4] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine, "Simulation of database-valued Markov chains using SimSQL," in *SIGMOD'13*.

[5] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, "The SAP HANA Database–An Architecture Overview.," *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 28–33, 2012.

[6] P. Tamayo, C. Berger, M. Campos, J. Yarmus, B. Milenova, A. Mozes, M. Taft, M. Hornick, R. Krishnan, S. Thomas, M. Kelly, D. Mukhin, B. Haberstroh, S. Stephens, and J. Myczkowski, *Oracle Data Mining*, pp. 1315–1329. Boston, MA: Springer US, 2005.

[7] A. Kumar, J. Naughton, and J. M. Patel, "Learning Generalized Linear Models Over Normalized Data," in *SIGMOD'15*.

[8] J. Teubner and L. Woods, *Data Processing on FPGAs*. Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2013.

[9] E. A. Sitaridi and K. A. Ross, "GPU-accelerated string matching for database applications," *PVLDB*, vol. 25, pp. 719–740, Oct. 2016.

[10] IBM, "IBM Netezza Data Warehouse Appliances," 2012. `http://www.ibm.com/software/data/netezza/`.

[11] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankorn, M. King, S. Xu, and Arvind, "BlueDBM: An Appliance for Big Data Analytics," in *ISCA'15*.

[12] K. Kara, J. Giceva, and G. Alonso, "FPGA-Based Data Partitioning," in *SIGMOD'17*.

[13] D. Andrews, D. Niehaus, R. Jidin, M. Finley, *et al.*, "Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link," *IEEE Micro*, vol. 24, July 2004.

[14] E. Lübbers and M. Platzner, "ReconOS: Multithreaded Programming for Reconfigurable Computers," *ACM TECS*, vol. 9, Oct. 2009.

[15] M. Owaida, D. Sidler, K. Kara, and G. Alonso, "Centaur: A Framework for Hybrid CPU-FPGA Databases," in *25th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM'17)*, 2017.

[16] D. Kesler, B. Deka, and R. Kumar, "A Hardware Acceleration Technique for Gradient Descent and Conjugate Gradient," in *SASP'11*.

[17] M. Bin Rabieah and C.-S. Bouganis, "FPGASVM: A Framework for Accelerating Kernelized Support Vector Machine," in *BigMine'16*.

[18] K. Kara, D. Alistarh, C. Zhang, O. Mutlu, and G. Alonso, "FPGA accelerated dense linear machine learning: A precision-convergence trade-off," in *FCCM'15*.

[19] M. Wilhelm, J. Schlegl, H. Hahne, A. M. Gholami, M. Lieberenz, M. M. Savitski, E. Ziegler, L. Butzmann, S. Gessulat, H. Marx, *et al.*, "Mass-spectrometry-based draft of the human proteome," *Nature*, vol. 509, no. 7502, pp. 582–587, 2014.

[20] E. Lopez-Rojas, A. Elmir, and S. Axelsson, "PaySim: A financial mobile money simulator for fraud detection," in *EMSS'16*.

[21] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–74, ACM, 2017.

[22] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. O. G. Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, *et al.*, "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?," in *FPGA*, pp. 5–14, 2017.

[23] M. Owaida, H. Zhang, C. Zhang, and G. Alonso, "Scalable Inference of Decision Tree Ensembles: Flexible Design for CPU-FPGA Platforms," in *FPL'17*.

[24] M. Owaida and G. Alonso, "Application Partitioning on FPGA Clusters: Inference over Decision Tree Ensembles," in *FPL'17*.

[25] J. Oberg, K. Eguro, and R. Bittner, "Random decision tree body part recognition using FPGAs," in *Proceedings of the 22th International Conference on Field Programmable Logic and Applications (FPL'12)*, 2012.

[26] B. V. Essen, C. Macaraeg, M. Gokhale, and R. Prenger, "Accelerating a Random Forest Classifier: Multi-Core, GP-GPU, or FPGA?," in *20th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM'12)*, 2012.

[27] Y. R. Qu and V. K. Prasanna, "Scalable and dynamically updatable lookup engine for decision-trees on FPGA," in *HPEC'14*.

[28] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *KDD'16*.

[29] T. Salimans, "HiggsML," 2014. `https://github.com/TimSalimans/HiggsML`.

[30] LHCb Collaboration, "Search for the lepton flavour violating decay $\tau^- \rightarrow \mu^-\mu^+\mu^-$," *High Energy Physics*, vol. 2015, Feb. 2015.

[31] K. Kara, K. Eguro, C. Zhang, and G. Alonso, "ColumnML: Column Store Machine Learning with On-the-Fly Data Transformation", in *PVLDB'19*

[32] Z. Wang, K. Kara, H. Zhang, G. Alonso, O. Mutly, and C. Zhang, "Accelerating Generalized Linear Models with MLWeaving: A One-Size-Fits-All System for Any-Precision Learning", in *PVLDB'19*

[33] D. Mahajan, J. K. Kim, J. Sacks, A. Ardalan, A. Kumar, and H. Esmaeilzadeh, "In-RDBMS Hardware Acceleration of Advanced Analytics", in *PVLDB'18*

[34] D. Sidler, M. Owaida, Z. Istvan, K. Kara, and G. Alonso, "doppioDB: A Hardware Accelerated Database", in *SIGMOD'17*

[35] Z. He, D. Sidler, Z. István, G. Alonso, "A flexible K-means Operator for Hybrid Databases", in *FPL'18*

XuanYuan: An AI-Native Database

Guoliang Li, Xuanhe Zhou, Sihao Li
Department of Computer Science,Tsinghua University, Beijing, China
Gauss Database Group, Huawei Company
liguoliang@tsinghua.edu.cn

Abstract

In big data era, database systems face three challenges. Firstly, the traditional empirical optimization techniques (e.g., cost estimation, join order selection, knob tuning) cannot meet the high-performance requirement for large-scale data, various applications and diversified users. We aim to design learning-based techniques to make database more intelligent. Secondly, many database applications require to use AI algorithms, e.g., image search in database. We can embed AI algorithms into database, utilize database techniques to accelerate AI algorithms, and provide AI capability inside databases. Thirdly, traditional databases focus on using general hardware (e.g., CPU), but cannot fully utilize new hardware (e.g., ARM, AI chips). Moreover, besides relational model, we can utilize tensor model to accelerate AI operations. Thus, we need to design new techniques to make full use of new hardware.

To address these challenges, we design an AI-native database. On one hand, we integrate AI techniques into databases to provide self-configuring, self-optimizing, self-healing, self-protecting and self-inspecting capabilities for databases. On the other hand, we can enable databases to provide AI capabilities using declarative languages, in order to lower the barrier of using AI.

In this paper, we will introduce the five levels of AI-native databases and provide the open challenges of designing an AI-native database. We will also take autonomous database knob tuning, deep reinforcement learning based optimizer, machine-learning based cardinally estimation, and autonomous index/view advisor as examples to showcase the superiority of AI-native databases.

1 INTRODUCTION

Databases have played a very important part in the recent evolution of computers and been widely deployed in many applications. Over the past fifty years, databases have undergone three revolutions.

The first generation is stand-alone databases, which address the problems of data storage, data management and query processing [2]. The representative systems include PostgreSQL and MySQL.

The second generation is cluster databases, which aim to provide high availability and reliability for critical business applications. The representative systems include Oracle RAC, DB2 and SQL server.

The third generation is distributed databases (and cloud-native databases), which aim to address the problems of elastic computing and dynamic data migration in the era of big data [5]. The representative systems include

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Aurora [1] and GaussDB [2].

However, the traditional databases still have several limitations in the big data era, due to the large-scale data, various applications and diversified users.

(1) Traditional database design is still based on empirical methodologies and specifications, and require human involvement (e.g., DBAs) to tune and maintain the databases. We use several examples to show that databases can be improved using AI techniques. First, databases have hundreds of knobs and it requires DBA to tune the knobs to adapt to different scenarios. Recently the database committee attempts to utilize machine learning techniques [1, 11, 20] to automatically tune the knobs, which can achieve better results than DBAs. Second, database optimizer relies on cost and cardinality estimation but traditional techniques cannot provide estimation. Recently deep learning based techniques [8, 16] are proposed to estimate the cost and cardinality which also achieve better results. Moreover, learning-based optimizers [10, 14], learning-based index recommendation [17], learning-based automatic views [12] provide alternative optimization opportunities for database design. Third, traditional database is designed by database architects based on their experiences. Recently some learning based self-design techniques are proposed, e.g., learned indexes [9] and learned NoSQL database design [7]. Thus we should utilize more AI techniques to enhance database and make database more intelligent.

(2) Traditional databases focus on *relational data model* and provides relational data management and analysis ability. However, in the big data era, there are more and more diverse data (e.g., graph data, time-series data, spatial data, array data) and applications (e.g., machine learning and graph computing). It calls for the database systems to support multiple models (e.g., relational model, graph model, tensor model) to support diversified applications (e.g., relational data analysis, graph computing and machine learning). We can embed AI algorithms into databases, design in-database machine learning framework, utilize database techniques to accelerate AI algorithms, and provide AI capability inside databases.

(3) Transitional databases only consider general-purpose hardware, e.g., CPU, RAM and disk, but cannot make full use of new hardware, e.g., ARM AI chips, GPU, FPGA. It calls for a heterogeneous computing framework that can efficiently utilize diversified computing powers to support data management, data analysis, and in-database machine learning.

To address these problems, we propose an AI-native database (XuanYuan), which not only integrates AI techniques into database to make database more intelligent but also provides in-database AI capabilities. In particularly, on one hand, XuanYuan integrates AI techniques into databases to provide self-configuring, self-optimizing, self-healing, self-protecting and self-inspecting capabilities for databases, which can improve the database's availability, performance and stability, and reduce the burden of intensive human involvement. On the other hand, XuanYuan enables databases to provide AI capabilities using declarative languages, in order to lower the barrier of using AI. XuanYuan also fully utilizes diversified computing power to support data analysis and machine learning.

An AI-native database can be divided into five stages. The first is AI-advised database, which takes an AI engine as a plug-in service and provides offline database suggestions, e.g., offline index advisor, offline knob tuning. The second stage is AI-assisted database, which takes an AI engine as a built-in service and provides online monitoring and suggestions, e.g., online index advisor, online statistics collection, and online diagnosis. The third is AI-enhanced database. One one hand, it provides AI based database components, e.g., learned index, learned optimizer, learned cost estimation, learned storage layout. On the other hand, it provides in-database AI algorithms and accelerators. The fourth is AI-assembled database, which provides multiple data models (e.g., relational model, graph model, tensor model) and fully utilizes the new hardware to support heterogeneous computing. It can provide multiple options for each component, e.g., learned optimizer, cost-based optimizer, and rule-based optimizer, and thus can automatically select the best component for different scenarios. The fifth is AI-designed database, which integrates AI into the life cycle of database, including design, development,

[1] https://aws.amazon.com/cn/rds/aurora/

[2] https://e.huawei.com/en/solutions/cloud-computing/big-data/gaussdb-distributed-database

evaluation, and maintenance, which provides the best performance for every scenario.

In this paper, we first present the details of AI-native databases and then provide the research challenges and opportunities for designing an AI-native database.

2 AI-Native Database

2.1 Motivation

In this section, we explain how AI and DB can benefit from each other. Firstly, we discuss how AI technique can benefit databases (AI4DB). Secondly, we present how database techniques can help AI (DB4AI). Thirdly, we discuss how to fully utilize the new hardware to support both AI and DB.

2.1.1 AI4DB

AI techniques can benefit databases from six aspects.

Self-configuration. All databases have hundreds of tuning knobs, which are vital to nearly every aspect of databases, such as performance, availability, robustness, etc. However, these configurations require to be adjusted manually, which not only is time consuming but also cannot find optimal configurations. AI based methods, e.g., deep reinforcement learning, can automatically tune the database knobs. Moreover, other configurations (e.g., software bugs, partition scheme) can also be optimized by AI-based methods.

Self-optimizing. AI can optimize the database performance from many aspects. First, cost/cardinality estimatio is vital to query plan selection. However, database mainly estimates cardinality based on raw statistics (e.g., read/write blocks, backends, deadlocks) and is poor in estimating the resulting row number of each query operator (e.g., hash join, aggregate, filter), with histograms simply. AI-based methods, e.g., Tree-LSTM, can learn data distribution in depth and provide more accurate cardinality estimation. Second, join order selection. Different join schemes have a great impact on query efficiency. And finding the best plan is a NP-hard problem. With static algorithms (e.g., dynamic programming, heuristics algorithm), the performance of join order selection in databse is limited by the effect of the estimator. AI based methods can better choose between different join order plans by taking one-step join as the short-term reward and the execution time as the long-term reward.

Self-healing. High reliability within database has become a critical requirement. But database can crash down for many reasons (e.g., poor performance, hardware/software failovers). So to keep database running is a tough thing. Attempts of traditional database include two aspects. First, set rule-based monitors and alert people when trigger the threshold. But the standards are coarse-grained and cannot effectively prevent accidents. Second, back up data or image periodically. Although that can help in recovery, it cannot assure high reliability. AI-based methods, e,g, CNN, can automatically detect, diagnose, alert and recover from database problems. For example, for distributed database, it supports live data migration when nodes crash or load leans with minimum cost.

Self-protecting. High availability is another important requirement. Database needs to protect the health of each transaction, such as the waiting time and the allocated resources. Without dynamic protecting mechanisms, database performance can be affected. First, unhealthy resource competition can severely affect database stability. AI-based methods can automatically balance the relation between workspace of a single query and the overall concurrency. Second, anomaly query can cause great loss to database. For example, run-away queries take much longer time than estimated in query plan, which may be hanging. And database will execute such queries for very long time before kill them. With AI techniques, we can not only solve the problems in query processing, but avoid wasting resources for run-away queries.

Self-inspecting. Data consistency is vital to data validation. Database maintains data consistency with fixed rules. For example, for single database, data in memory is written back to disk every a number of write operations; for distributed database, data is synchronized between replication nodes using writing logs. With AI

techniques, database can automatically learn how to check and assure data consistency and database health. To a higher level, AI models should translate those experiences into publication of instrumentation to help human beings understand.

Self-assembling. Each processing layer (e.g., optimizer, executor, storage engine) in a database has several alternatives, each of which has its own advantages. But in traditional database, query is processed in fixed path and cannot take advantage of optional components dynamically. With AI techniques, it can learn how to select proper component in each layer and assemble the complete executing path based on the query and workload characters.

2.1.2 DB4AI

AI techniques rely on data heavily. With database storing, managing and manipulating data, the training and learning procedure can be more efficient. But to support AI techniques on database, there are several problems.

Firstly, AI has different user interfaces to DB. Generally, people write AI models in Python or R languages, but manipulate data from relational database with SQL statements. There are two problems when call AI-related services. 1) People need frequently switch between different systems when writing AI-related applications. 2) It is not easy for traditional data analysts, who only know some SQL knowledge, to write AI code. So if we can extend parsing rules to make SQL support both DB and AI, it would be very convenient to support AI-related services.

Secondly, DB can optimize AI algorithms. Database can not only provide AI with data, but better support AI services with its mechanisms. We can see from model building training, reusing. 1) With unified SQL interface, user can easily build an AI model using user defined functions or stored procedures. 2) Training AI models consume time with a large number of tensor calculation. Database can better support tensor operators with extended relational algebra and execute them in the executor, which help in model training. 3) Well-trained AI models can be persisted with materialized views or query tables and be reused easily and efficiently.

2.1.3 Heterogeneous Computing Framework

With Moore's law on the verge of failure, database can no longer rely on single processor to improve processing ability. Now heterogeneous computing framework brings about new potential. Firstly, it incorporates different computing powers (e.g., GPU, NPU, FPGA) and accelerators. Secondly, with dissimilar co-processors handling different tasks, it can gain great improvement in performance and efficiency. But to support heterogeneous computing framework, database needs to solve three main problems.

• Single resource scheduling in system level. The theoretical architecture and operators in traditional DB are not suitable to computing-intensive chips. That is, simple computing in structured data analysis dose not need chips with high parallel computing power. DB needs to design special heterogeneous computing units, considering of computing and storing resources.

• Single accelerator architecture. Traditional accelerator architecture only supports OLAP type, For OLTP and HTAP workloads, caused by data consistency between system memory and accelerator's local memory, it is inefficient. DB needs to incorporate new techs, such as CXL released by Intel, to provide efficient memory access methods for accelerators.

• Single relational algebra. Firstly, to provide heterogeneous computing, we need to define data models for different operating. For example, relational data model is good for data management, but cannot fit TPU/NPU processing model. Besides, We want different computing powers to work together. For example, with extended relational algebra, we can use tensor computing model to accelerate relational operators, such as join and aggregation.

To alleviate the issues talked above, in this section we propose a design of AI-native database, which better provides DB and AI services in five levels.

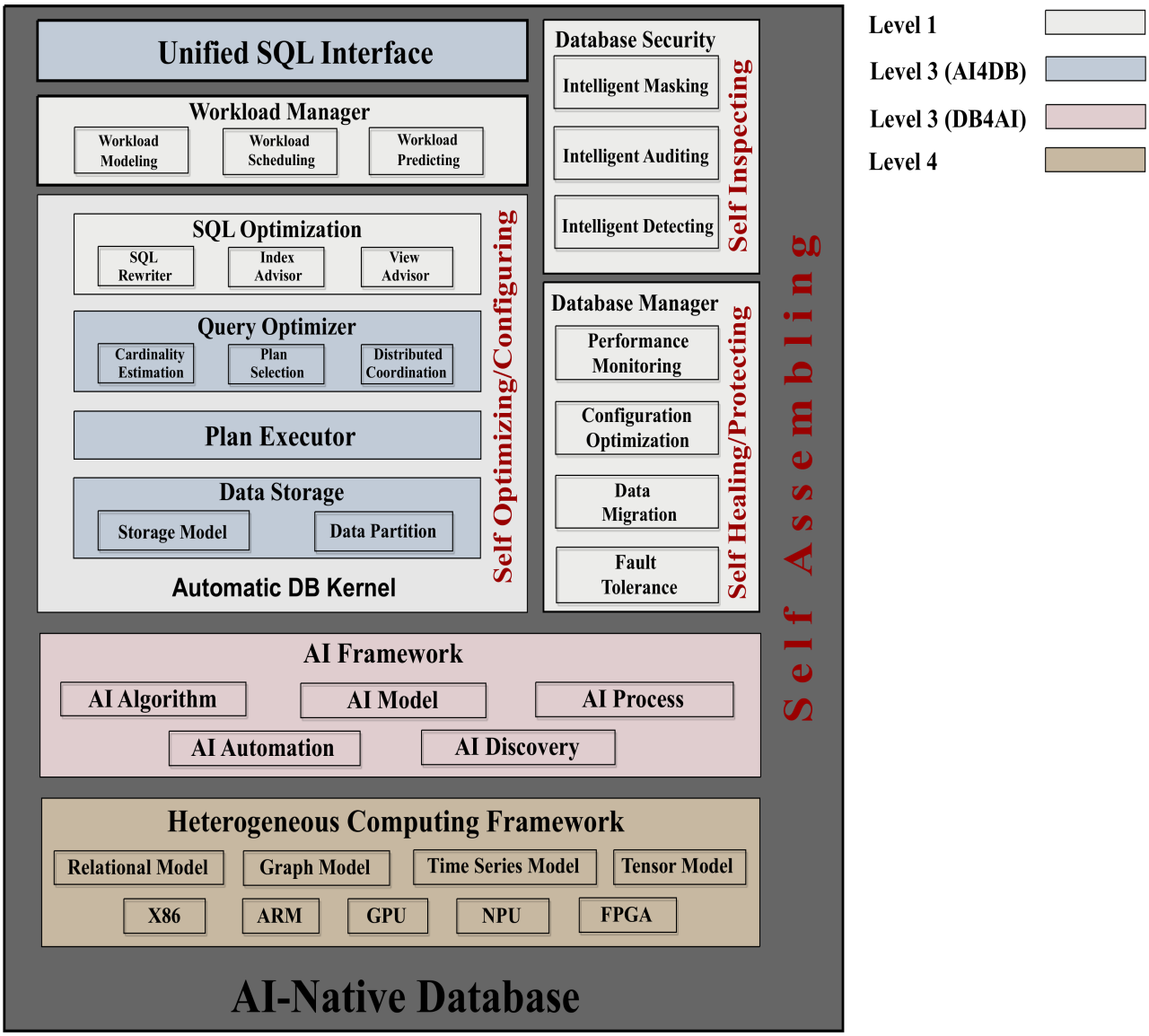

Figure 1: The architecture of AI-Native databases

2.2 Level 1: AI-Advised Database

As shown in Table 4, in the first level, our database is AI-Advised, which provides auxiliary optimization of the database through suggestions [1, 15, 11]. It has an AI engine packing AI tools, which are loosely coupled with database in the form of plug-in. Limited by available resources, the AI engine mainly provides auxiliary tools from four aspects.

◦ **Workload Management**. It has two functions: to schedule user workloads and summary workload characters for the other modules. For `workload modeling`, traditional cost models are mostly based on empirical formulas, which have poor adaptability to different physical environments. Therefore, the machine learning model can be used to analyze and evaluate the current workload situation and future overhead, and dynamically adapt to the external environment based on gradient changes. For `workload scheduling`, in traditional databases, resource scheduling depends heavily on parameters related to resource control, such as workspace

size, maximum concurrent IO number, and etc. But these parameters are static and need to be manually configured. So we can use reinforcement learning to learn relations between database state (physical/logical), workload and database resources, and provide a reasonable and robust scheduling mechanism. For `workload predicting`, workload usually dose not change in a fixed pattern. Traditional workload predicting methods are usually grasped by database experts according to statistical data, which can not guarantee high accuracy. So we proposed to use machine learning method for workload forecasting [13], which can have better adaptability to different workloads.

◦ **SQL Optimization**. It is to optimize database in SQL level. For `SQL rewriter`, SQL rewriter helps optimizer to choose efficient query plans by changing the way SQL is written. This level is often completed manually, which is impractical for heavy workload. So we provide a rewriting tool to learn the principles of SQL writing (e.g., avoiding full table scanning, selecting indexed columns as joins, and using table variables) and optimize the SQL structure. For `index advisor`, index is very important to improve the efficiency of retrieval tasks on complex data sets [6]. However, the commonly used indexes are universal data structures. They do not analyze and utilize the distribution of data. So through machine learning, we can learn a model that reflects data patterns, and can automatically synthesize a special index structure at a lower cost. For `view advisor`, given a set of queries, extracting high-frequency sub-queries and establishing materialized views to improve the performance of the database is very helpful. The traditional method caches at the whole query level, and the hit rate is low. We construct the ML algorithm of sub-query selection according to the actual constraints so as to improve the query efficiency of batch SQL tasks.

◦ **DB Manager**. It provides services (e.g., performance monitoring, tuning, fault tolerance) to ensure the stable operation of the database. In distributed scenario, high load pressure will lead to a sharp increase in failure rate, which brings great challenges to database maintenance. For `performance monitoring`, it automatically monitors the status of the database system (e.g., the number of batch requests per second, the number of user connections, network transceiver efficiency). Then it analyzes those indicators using machine learning algorithms. For `tuning`, database configuration involves hundreds of tunable system parameters, which control the database components in many aspects. Traditional tuning methods [3, 22] either relies on human beings (DBA-based), or cannot utilize history information (search-based). So we use a deep-reinforcement-learning-based tool [11] to learn relations between database state, query and parameters, which can adapt to environment changes dynamically. For `fault tolerance`, it includes a variety of strategies for resolving failures of hardware, transactions and system. In case of data error, the system needs to cancel the corresponding transactions in time. So we use a ML-based tool to monitor the changes caused by typical failures and realize pre-alerts.

◦ **DB Security**. It is to combine information security, cryptography technology with AI tech to ensure the security of database. For `intelligent masking`, it is to hide privacy data such as ID number. Intelligent masking technology only uses the high-dimensional, non-linear data inside the neural network to help improve the effect of data hiding. For example, in the research project conducted by Stanford University and Google, satellite images have been converted into high-frequency signals that are not easily detected. For `intelligent auditing`, it optimizes the auditing work from two aspects: data preprocessing and dynamic analysis. Traditional audit work often requires auditors to obtain a large number of business data. This part of the data understanding work is a waste of human resources. Intelligent auditing not only saves manpower cost, but also helps auditors to make better decisions by providing useful information from massive data. For `intelligent detecting`, it can automatically detect system vulnerabilities. Known intelligent detecting technology is mainly to retrieve through security scanning, while unknown security vulnerabilities need to do a lot of retrieval and testing work. Using machine learning algorithm to discover security vulnerabilities [23], it can not only detect most known vulnerabilities in national vulnerability database, but also predict and evaluate potential vulnerabilities.

2.3 Level 2: AI-Assisted Database

In the second level, our database is AI-Assisted, which is embedded into the database kernel for runtime optimization. AI components (e.g., tuning model, workload scheduling, view advisor) can be merged into the corresponding database components [12]. This way, AI processes are integrated into the working procedure of the database. For example, if embedding the tuning model into the query optimizer, every time we generate the query plan, we can first conduct query tuning (adjust user-level parameters to better adapt to the incoming query characters), and then normally generate and execute query plan. The advantage of AI-Assisted database is that 1) it can provide more fine-grained optimization; 2) by embedding AI engine into the kernel, it can reduce much overhead such as communication cost.

However, neither Level 1 or 2 is a real AI-native database. Because the main components and organizing mode in those databases are still based on traditional methods. There are huge gap between AI technologies and databases. Starting from the third level, we introduce how to integrate intelligence integration, heterogeneity into database design to achieve the AI-native database.

2.4 Level 3: AI-Enhanced Database

In the third level, our database is AI-Enhanced, which embeds intelligence and integration into database design. Firstly, we propose an intelligent database kernel, which implants AI into the design of database kernel. Secondly, we propose a unified engine for both AI and DB, with which clients can use AI as easily as they use a DB.

2.4.1 Intelligent Database Kernel

We present the design of an intelligent database kernel that enables self-configuring, self-healing, self-optimizing, self-protecting, self-inspecting and finally achieves self-organizing.

◦ **Self-configuring**. Database can automatically adjust their own configuration to adapt to environment changes, including tuning parameters, upgrading software,adjusting partitioning/replication scheme, reorganizing tablespace and etc. Each function is embedded into particular DB modules to work as part of the processing. For example, to realize tuning in query level, we embed tuner into optimizer. Each time a parse tree is input into our optimizer, it first configures parameters based on the query features and then actually generates the query plan. This way, it not only helps to generate better query plan, but prepares suitable environment for executing the query plan.

◦ **Self-healing**. Database can automatically detect, diagnose, alert and recover from DB problems (e.g., poor performance, hardware/software failovers). It adopts advantages of tools in AI-Advised DB to fully save humans from the failure-recovery loop. Firstly, it takes protective actions ahead of time (e.g., backup, data sharding, reosurce scheduling). Secondly, it can recover services even if emergency occurs (e.g., live migration).

◦ **Self-optimizing**. Database can automatically collect statistics of queries and optimize database performance in multiple granularities. Firstly, it collect statistics used in optimizer to learn current workload, which can benefit from cardinality estimation and workload scheduling. Secondly, it automatically design index, materialized query table (MQT) and data partition, which can directly optimize performance of single query. Thirdly, it automatically controls the flow of queries with query patroller to achieve overall optimization.

◦ **Self-protecting**. Database can automatically monitor processing procedure of each query and prevent potential damage in time. For example, it can throttle service requests which compete resources to cut down deadlock and average waiting time; and it can kill / lower the priority of run-away queries (execution time is far longer than that estimated by the optimizer) to prevent from using up system resources.

◦ **Self-inspecting**. Database can automatically monitor database state and conclude operating rules by itself. It can check database state (e.g., data consistency, DB health) all the time. And by concluding all the failover

conditions and solutions, it can translate the experience in the form of publication of instrumentation data in favor of human understanding.

◦ **Self-organizing**. For current databases, each level of service (e.g., parser, optimizer and executor) has multiple standardized components to choose from. How to choose the appropriate processing path for one or a batch of queries becomes meaningful.

So we propose a self-organizing module. For different queries, we can dynamically select the appropriate components in each layer of service and assemble the appropriate execution path. The execution path can be seen as a natural language sequence (NLS), such as $\langle SQL_i$, parser_pg, optimizer_RBO, storage row, accelerator $\rangle$, in which each position has only discrete token options. So the problem is how to generate NLS in query level. One method is to use RL algorithm. It takes the whole path sequence as an episode and a single action as an epoch. Under each epoch, the agent chooses the next component (action), which executes the query, leading to a state transition (e.g., the query status changes into syntax tree). In this problem, action is discrete, so DDQN algorithm [18] can be used. Compared with other RL algorithms with discrete output, DDQN eliminates the problem of overestimation (deviate greatly from the optimal solution) by choosing decoupling actions and calculating the target Q value.

However, RL-based routing algorithm has two problems. Firstly, Q network will not score until the last node is generated. It is insensitive to the choice of intermediate nodes. However, for the entire path, it is necessary to give action a comprehensive score on current and future impacts. Secondly, when training with epoch as a unit, each node of a path is scattered in a training sample, instead of being used as a whole to calculate the gradient.

Generating antagonistic network (GAN) [21] can better solve those problems of end-to-end path selection. We use G network to generate path vectors based on workload, database state and component characteristics, and we use D network as the performance model. But the traditional GAN network model is not fully applicable to our problem. Because it is mainly used to generate data with continuous range of values, and it is difficult to generate path vectors with discrete tokens. That is because G-networks need to be fine-tuned based on gradient descent and regress towards the expectation. But when the data is discrete tokens, fine-tuning is often meaningless. So we choose to combine RL algorithm with GAN [19]. Firstly, G network is used as agent in RL algorithm: action is the next service node; state includes not only query status, database status, but also generated node information. Each iteration generates a complete path. Secondly, unlike pure RL algorithm, each action is scored by D network to guide the generation of the whole path sequence.

2.4.2 Unified Engine for AI and DB

After achieving the intelligent database kernel, we gain real integration by providing unified engine for DB and AI. AI and DB are closely connected. For DB, as talked above, AI can make DB smarter. For AI, DB stores large scale of data for training the ML models. So it becomes easy when we can call ML model with SQL statements. Firstly, SQL is easy to use, with which technicians with basic SQL knowledge can complete most of traininig and predicting tasks. Secondly, it saves people from frequently switching between different languages of systems, when we need to write SQL to fetch data and run python scripts to run ML algorithms.

For this work, there have been some products such as BigQuery ML [4], SQL for DL and SQLFlow. Considering the Pros and Cons of these works, we propose an extended DB engine to achieve the following aims:

◦ **AI Support**. Firstly, SQL Parser should extend SQL syntax to utilize AI methods, including model creating, training and predicting. Besides, it should minimize the use of other scripting languages (e.g., Python and R). Secondly, we extend the relational algebra theory to support both relational and tensor data models. This way, DB can support tensor processing, which is largely used in the training of AI models.

◦ **General Purpose**. SQL parser should contain an abstract layer which maintains a loose couple with lower components. This way, it can be easily used on different SQL engines (e.g., PostgreSQL, Flink and Hive) or machine learning toolkits (e.g., TensorFlow, Keras and scikit-learn).

◦ **Custom Style**. Besides general operators such as SELECT and WITH, we also allow user to define how

to train, evaluate and predict the ML models with user-defined functions. And they also can cache the trained model with materialized views.

The general workflow is as follows. Firstly, SQL Parser parses a SQL statement and produces a general-purpose query plan. Based on the operators in the plan, it decides whether it is for manipulating data or calling ML models. If for manipulating data, the query plan is sent to the DB executor and be executed. Otherwise, it is sent to the AI executor and be executed.

This way, database can also provide AI services in an end-to-end way. As table 3 shows, Database supports AI services in five levels.

• **DB-based AI Algorithm**. Database provides ML tool-kits and libraries (e.g., TensorFlow, Scikit-learn). Clients can write AI algorithms in UDFs or stored procedures. And they can call them repeatedly from database.

• **DB-based AI Model**. Database provides classic AI models (e.g., random forest, RNN, DDQN). Clients can directly call AI models from database and only focus on the training part.

• **DB-based AI Process**. Database further provides AI processes, which not only include the mature AI models but the training methods (e.g., SGD, Adam, reinforcement learning). Clients can directly call AI processes from database, only with data as input. Database automatically fine-tunes AI model using corresponding training method, and utilizes the model to produce results.

• **DB-based Automation**. Database can automatically parse the requirements in AI and call proper process to solve it. Clients only need to declare their requirement. For example, if we hope to search some images, we just declare the conditions and data sources with SQL. And database automatically chooses the CNN model to conduct the work, which is also a part of the query plan.

• **DB-based Discovery**. Database offers an insight into the data schema and application queries, and optimizes itself with AI techniques. For example, if a new workload comes, database can call a DRL-based tuner to adjust the system parameters based on the query features, including cost estimation, resource allocation, concurrency control and etc.

Table 3: Five levels of machine learning consumability in DBMS

Level	Consumability	Description	Target Users	ML Skill Level
1	Algorithm	Algorithms as UDFs and stored procedures	Data scientists, experienced developers	High
2	Model	Models as first-class object with DDL, DCL, DML capability	App. developers and DBAs	Medium
3	Process	Processes as first-class object with DDL, with DCL, DML capability	App. developers and DBAs	Low
4	Automation	Problem specifications as first-calss object with DDL, DCL, DML capability	Business specialists, DBAs app. developers	Low
5	Discovery	Discover ML opportunities based on data schema and application queries	Business specialists, DBAs app. developers	Very Low

2.5 Level 4: AI-Assembled Database

In the fourth level, our database is AI-Assembled, which embeds heterogeneity into database design. That is ,we assemble AI with database by supporting multiple leading computing powers. We know traditional database is only based on CPU. However, DB and AI technology usually require different computing power and hardware. For DB, traditional optimizer processes queries with CPU. While AI technology requires new chips to support parallely processing (e.g., GPU and NPU) and self-scheduling. And now many applications need to use both DB and AI technology, especially in large data analysis scenarios.

AI-assembled database mainly has two contributions. Firstly, it can support ARM architecture. 1) It effectively utilizes ARM array with multi-core mode, inter-node parallelism, intra-node parallelism, instruction-level parallelism, compiler parallelism and other super-parallel technologies. 2) For the problem of competition brought by multi-core, it provides cross-chip access optimization and resource scheduling optimization. Secondly, it can utilize different computing powers to better serve the AI components. It is able to switch computing powers based on the processing types. For example, for the tuning module in optimizer, when training the tuning model, we fetch training data into memory with ARM and conduct backward propagation (training the neural network) with NPU. And database should flexibly schedule computing powers, e.g., when NPUs are overburdened but most ARM resources are idle, ARM array can share some burden.

The ultimate objective of this database is to fully unleash the power of diversified computing, which includes x86, ARM, GPU, NPU, and accelerators. We aim to continuously push our AI strategy forward and foster a complete computing ecosystem.

2.6 Level 5: AI Designed Database

In the last level, AI technology is integrated into the whole database life cycle (DBLC) of our database so as to achieve the ultimate goal of scenario-aware and AI-as-a-Service. The whole life cycle includes five phases: 1) Database initialization; 2) Database design; 3) Database implementation; 4) Database evaluation; 5) Database operation and maintenance.

Table 4: Five levels of AI-native database

Level	Feature	Description	Example
1	AI-advised	plug-in AI engine	○ Workload Manager (e.g., workload scheduling/predicting) ○ SQL Optimization (e.g., SQL rewriter, index/view advisor) ○ DB Maintenance (e.g., knob tuner, fault tolerance) ○ Security (e.g., intelligent masking/auditing/detecting)
2	AI-assisted	Built-in AI engine	○ Embed workload scheduling as job-queue mechanism ○ Embed index advisor as an optional database indexing ○ Embed knob tuner as a self-adaptive module ○ Replace database auditing with intelligent auditing
3	AI-enhanced	Hybrid DB&AI engine	○ Intelligent Database Kernel • Self-configuring (e.g., self tune/upgrade/data partition) • Self-healing (e.g., self failover/alert/recovery) • Self-optimizing (e.g., collect stats, design index/MQT) • Self-protecting (e.g., throttle run-away queries) • Self-inspecting (e.g., consistency/health check) • Self-organizing (e.g., form executing path in query level) ○ Unified Engine for AI+DB
4	AI-assembled	Heterogeneous computing	○ Support new hardware (e.g., ARM, GPU, NPU) ○ Extend relational algebra to support tensor model
5	AI-designed	The life cycle is AI-based	○ Database initialization ○ Database design ○ Database implementation and loading ○ Database testing and evaluation ○ Database operation and maintenance

3 Challenges and Opportunities

It brings new research challenges and opportunities to design an AI-native database, which aims to support data management, data analysis, machine learning together in the same system.

3.1 From one-size-doesn't-fit-all to one-stack-fits-all

Michael stonebraker argues that one size does not fit all, due to various applications (e.g., OLTP, OLAP, stream, graph) and diversified hardware (e.g., CPU, ARM, GPU, FPGA, NVM). Note that the database components and their variants are limited, but the number of possible combinations for these components to assemble a database is huge. So the database architects design the database architectures by combining different variants of techniques based on their empirical rules and experience for adapting to different scenarios. Thus these human-designed databases may not be optimal because they may fall into local optimum. It calls for automatically designing a database using AI techniques.

We argue that one stack fits all. The basic idea is to first implement the database components, e.g., indexes, optimizers, storage, where each component has multiple variants/options, then use AI techniques to assemble these components to form some database candidates, and finally select a database that best suits a given scenario. In this way, we can automatically verify different possible databases (i.e., different combinations of components), explore many more possible database designs than human-based deign, and could design more powerful database. This is similar to AlphaGO, where the learning-based method beats humans, because the machines can explore more unknown spaces.

There are several challenges in one-stack-fits-all. First, each component should provide standard interfaces such that different components can be integrated together. Second, each component should have different variants or implementations, e.g., different indexes, different optimizers. Third, it calls for a learning-based component to assemble different components. Fourth, the assembled database can be evaluated and verified before the database is deployed in real applications. Fifth, each component should be run on different hardware, e.g., learned optimizers should be run on AI chips and traditional cost-based optimizers should be run on general-purpose chips.

3.2 Next Generation Analytic Processing: OLAP 2.0

Traditional OLAP focuses on relational data analytics. However, in the big data era, many new data types have emerged, e.g., graph data, time-series data, spatial data, it calls for new data analytics techniques to analyze these multi-model data. Moreover, besides traditional aggregation queries, many applications require to use machine learning algorithms to enhance data analytics, e.g., image analysis. Thus it is rather challenging to integrate AI and DB techniques to provide new data analytics functionality. We call hybrid DB and AI online analytic processing on multi-model data *OLAP 2.0*.

There are several challenges in supporting OLAP 2.0. First, different data types use different models, e.g., relational model, graph model, KV model, tensor model, and it calls for a new model to support multi-model data analytics. Second, OLAP 2.0 queries may involve both database operations and AI operations, and it needs to design new optimization model to optimize these heterogeneous operations across different hardware.

3.3 Next Generation Transaction Processing: OLTP 2.0

Traditional OLTP mainly uses general-purpose hardware, e.g., CPU, RAM and disk, but cannot make full use of new hardware, e.g., AI chips, RDMA, and NVM. Actually, we can utilize new hardware to improve transaction processing. First, based on the characteristics of NVM, including non-volatile, read-write asymmetry speed, and wear-leveling, we need to reconsider the database architecture. For example, we can utilize NVM to replace RAM and replace page-level storage with record-level storage on NVM. Second, we can utilize RDMA to

improve the data transmission in databases. Moreover, we can use the programmable feature of intelligent Ethernet card to enable filtering on RDMA and avoid unnecessary processing in RAM and CPU. Third, there are some AI chips which are specially designed hardware, and it is also promising to design database-oriented chips that are specially defined hardware for databases.

There are several challenges in supporting OLTP 2.0. First, it is challenging to fully utilize new hardware to design a new generation database. Second, it is hard to evaluate and verify whether the new hardware can benefit the database architecture.

3.4 AI4DB

There are several challenges that embed AI in databases.

Training Samples. Most AI models require large-scale, high-quality, diversified training data to achieve good performance. However, it is rather hard to get training data in databases, because the data either is security critical or relies on DBAs. For example, in database knob tuning, the training samples should be gotten based on DBAs' experiences. Thus it is hard to get large-scale training samples. Moreover, the training data should cover different scenarios, different hardware environments, and different workloads.

Model Selection. There are lots of machine learning algorithms and it is hard to automatically select an appropriate algorithm for different scenarios. Moreover, the model selection is affected by many factors, e.g., quality, training time, adaptability, generalization. For example, deep learning may be the best choice for cost estimation while reinforcement learning may be the best choice for join order selection. The training time may also be important, because some applications are performance critical and cannot tolerate long training time.

Model Convergence. It is very important that whether the model can be converged. If the model cannot be converged, we need to provide alternative ways to avoid making bad decisions. For example, in knob tuning, if the model is not converged, we cannot utilize the model for knob suggestion.

Adaptability. The model should be adapted to different scenarios. For example, if the hardware environments are changed, the model can adapt to the new hardware.

Generalization. The model should be generalized different settings. For example, if the workloads are changed, the model should support the new workloads. If the data are updated, the model should be generalized to support new data.

3.5 DB4AI

Accelerate AI algorithms using indexing techniques. Most of studies focus on the effectiveness of AI algorithms but do not pay much attention to the efficiency, which is also very important. It calls for utilizing database techniques to improve the performance of AL algorithms. For example, self-driving requires a large number of examples for training, which is rather time consuming. Actually, it only requires some *important examples*, e.g., the training cases in the night or raining day, but not many redundant examples. Thus we can index the samples and features for training.

Discover AI Models. Users may only know their requirements, e.g., classification, but do not know which AI algorithms should be used. Thus it is important to automatically discover AI algorithms. Moreover, it is also very challenging to reuse the trained AI models by different users.

3.6 Edge Computing Database.

Most databases are designed to be deployed on servers. With the development of 5G and IOT devices, it calls for tiny database embedded in small devices. There are several challenges. The first is security to protect the date. The second is real-time data processing. The small device has low computing power, and it is rather challenging

to provide high performance on such small devices. The third is data migration among different devices. Some devices have small storage and it is challenging to migrate the data across different devices.

4 CONCLUSION

We proposed an AI-native database XuanYuan, which not only utilizes AI techniques to enable self-configuring, self-optimizing, self-inspecting, self-healing, but also provides in-database AI capabilities to lower the burden of using AI. We categorized AI-native databases into five levels, AI-advised, AI-assisted, AI-enhanced, AI-assembled, AI-designed. We also discussed the research challenges and provided opportunities in designing an AI-native database.

References

[1] D. V. Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*, pages 1009–1024, 2017.

[2] C. J. Date. *An introduction to database systems (7. ed.)*. Addison-Wesley-Longman, 2000.

[3] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with ituned. *PVLDB*, 2(1):1246–1257, 2009.

[4] S. Fernandes and J. Bernardino. What is bigquery? In *Proceedings of the 19th International Database Engineering & Applications Symposium, Yokohama, Japan, July 13-15, 2015*, pages 202–203, 2015.

[5] G. Figueiredo, V. Braganholo, and M. Mattoso. Processing queries over distributed XML databases. *JIDM*, 1(3):455–470, 2010.

[6] A. Gani, A. Siddiqa, S. Shamshirband, and F. H. Nasaruddin. A survey on indexing techniques for big data: taxonomy and performance evaluation. *Knowl. Inf. Syst.*, 46(2):241–284, 2016.

[7] S. Idreos, N. Dayan, W. Qin, M. Akmanalp, S. Hilgard, A. Ross, J. Lennon, V. Jain, H. Gupta, D. Li, and Z. Zhu. Design continuums and the path toward self-designing key-value stores that know and learn. In *CIDR*, 2019.

[8] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR*, 2019.

[9] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *SIGMOD*, pages 489–504, 2018.

[10] S. Krishnan, Z. Yang, K. Goldberg, J. M. Hellerstein, and I. Stoica. Learning to optimize join queries with deep reinforcement learning. *CoRR*, abs/1808.03196, 2018.

[11] G. Li, X. Zhou, B. Gao, and S. Li. Qtune: A query-aware database tuning system with deep reinforcement learning. In *VLDB*, 2019.

[12] X. Liang, A. J. Elmore, and S. Krishnan. Opportunistic view materialization with deep reinforcement learning. *CoRR*, abs/1903.01363, 2019.

[13] L. Ma, D. V. Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 631–645, 2018.

[14] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, pages 3:1–3:4, 2018.

[15] M. Mitzenmacher. A model for learned bloom filters and related structures. *CoRR*, abs/1802.00884, 2018.

[16] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *SIGMOD*, pages 4:1–4:4, 2018.

[17] W. G. Pedrozo, J. C. Nievola, and D. C. Ribeiro. An adaptive approach for index tuning with learning classifier systems on hybrid storage environments. In *HAIS*, pages 716–729, 2018.

[18] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *AAAI*, pages 2094–2100, 2016.

[19] L. Yu, W. Zhang, J. Wang, and Y. Yu. Seqgan: Sequence generative adversarial nets with policy gradient. In *AAAI*, pages 2852–2858, 2017.

[20] J. Zhang, Y. Liu, K. Zhou, and G. Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *SIGMOD*, 2019.

[21] Z. Zhou, J. Liang, Y. Song, L. Yu, H. Wang, W. Zhang, Y. Yu, and Z. Zhang. Lipschitz generative adversarial nets. *CoRR*, abs/1902.05687, 2019.

[22] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *SoCC*, pages 338–350, 2017.

[23] S. Zong, A. Ritter, G. Mueller, and E. Wright. Analyzing the perceived severity of cybersecurity threats reported on social media. *CoRR*, abs/1902.10680, 2019.

Call for Papers

35th IEEE International Conference on Data Engineering

8-12 April 2019, Macau SAR, China

The annual IEEE International Conference on Data Engineering (ICDE) addresses research issues in designing, building, managing and evaluating advanced data-intensive systems and applications. It is a leading forum for researchers, practitioners, developers, and users to explore cutting-edge ideas and to exchange techniques, tools, and experiences. The 35th ICDE will take place at Macau, China, a beautiful seaside city where the old eastern and western cultures as well as the modern civilization are well integrated.

Topics of Interest

We encourage submissions of high quality original research contributions in the following areas. We also welcome any original contributions that may cross the boundaries among areas or point in other novel directions of interest to the database research community:

- Benchmarking, Performance Modelling, and Tuning
- Data Integration, Metadata Management, and Interoperability
- Data Mining and Knowledge Discovery
- Data Models, Semantics, Query languages
- Data Provenance, cleaning, curation
- Data Science
- Data Stream Systems and Sensor Networks
- Data Visualization and Interactive Data Exploration
- Database Privacy, Security, and Trust
- Distributed, Parallel and P2P Data Management
- Graphs, RDF, Web Data and Social Networks
- Database technology for machine learning
- Modern Hardware and In-Memory Database Systems
- Query Processing, Indexing, and Optimization
- Search and Information extraction
- Strings, Texts, and Keyword Search
- Temporal, Spatial, Mobile and Multimedia
- Uncertain, Probabilistic Databases
- Workflows, Scientific Data Management

Important Dates

For the first time in ICDE, ICDE2019 will have two rounds' submissions. All deadlines in the following are 11:59PM US PDT.

First Round:

Abstract submission due: May 25, 2018
Submission due: June 1, 2018
Notification to authors (Accept/Revise/Reject): August 10, 2018
Revisions due: September 7, 2018
Notification to authors (Accept/Reject): September 28, 2018
Camera-ready copy due: October 19, 2018

Second Round:

Abstract submission due: September 28, 2018
Submission due: October 5, 2018
Notification to authors (Accept/Revise/Reject): December 14th, 2018
Revisions due: January 11th, 2019
Notification to authors (Accept/Reject): February 1, 2019
Camera-ready copy due: February 22, 2019

General Co-Chairs
Christian S. Jensen, Aalborg University
Lionel M. Ni, University of Macau
Tamer Özsu, University of Waterloo

PC Co-Chairs
Wenfei Fan, University of Edinburgh
Xuemin Lin, University of New South Wales
Divesh Srivastava, AT&T Labs Research

For more details: http://conferences.cis.umac.mo/icde2019/

It's FREE to join!

TCDE

tab.computer.org/tcde/

The Technical Committee on Data Engineering (TCDE) of the IEEE Computer Society is concerned with the role of data in the design, development, management and utilization of information systems.

- Data Management Systems and Modern Hardware/Software Platforms
- Data Models, Data Integration, Semantics and Data Quality
- Spatial, Temporal, Graph, Scientific, Statistical and Multimedia Databases
- Data Mining, Data Warehousing, and OLAP
- Big Data, Streams and Clouds
- Information Management, Distribution, Mobility, and the WWW
- Data Security, Privacy and Trust
- Performance, Experiments, and Analysis of Data Systems

The TCDE sponsors the International Conference on Data Engineering (ICDE). It publishes a quarterly newsletter, the Data Engineering Bulletin. If you are a member of the IEEE Computer Society, you may join the TCDE and receive copies of the Data Engineering Bulletin without cost. There are approximately 1000 members of the TCDE.

Join TCDE via Online or Fax

ONLINE: Follow the instructions on this page:

www.computer.org/portal/web/tandc/joinatc

FAX: Complete your details and fax this form to **+61-7-3365 3248**

Name _____

IEEE Member # _____

Mailing Address _____

_____

Country _____

Email _____

Phone _____

TCDE Mailing List

TCDE will occasionally email announcements, and other opportunities available for members. This mailing list will be used only for this purpose.

Membership Questions?

Xiaoyong Du
Key Laboratory of Data Engineering and Knowledge Engineering
Renmin University of China
Beijing 100872, China
duyong@ruc.edu.cn

TCDE Chair

Xiaofang Zhou
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, QLD 4072, Australia
zxf@uq.edu.au

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398