Bulletin of the Technical Committee on

Data Engineering

June 2019 Vol. 42 No. 2

IEEE Computer Society

Letters

Opinions

Special Issue on Machine Learning Life-cycle Management

Conference and Journal Notices

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TCDE web page is http://tab.computer.org/tcde/index.html.

The Data Engineering Bulletin

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at http://tab.computer.org/tcde/bull_about.html.

Letter from the Editor-in-Chief

One of the beauties of the Data Engineering Bulletin, with a history of 43 years and 157 issues, is that it chronicles how topics of database research evolve and sometimes reinvent themselves over time. Philip A. Bernstein's opinion piece in this issue, titled "Resurrecting Middle-Tier Distributed Transactions," is another testimony to this beauty. Bernstein tells an interesting story of transaction processing monitors running on middle-tier servers, and predicts their return to the mainstream after a 15-year decline.

Guoliang Li put together the current issue consisting of 6 papers on the interactions between database systems and AI. This is a fascinating topic. Traditional databases are heavily optimized monolithic systems designed with heuristics and assumptions. But recent work has shown that critical data structures such as database indices are merely models, and can be replaced with more flexible, faster, and smaller machine learned models such as neural networks. This opens the door to using data driven approaches for system design. On the other hand, deep learning is still facing the challenge in incorporating database accesses in end-to-end training, which hampers the use of existing structured knowledge in learning.

Haixun Wang
WeWork Corporation

Letter from the Special Issue Editor

Databases have played a very important role in many applications and been widely deployed in many fields. However, traditional database design is still based on empirical methodologies and specifications, and require heavy human involvement to tune and maintain the databases. Recently there are many attempts that use AI techniques to optimize database, e.g., learned index, learned cost estimation, learned optimizer, and learning-based database knob tuning. In this issue, we discuss (1) how to adopt AI techniques to optimize databases and (2) how to utilize database techniques to benefit AI and provide in-database capabilities.

The first paper integrates data preparation into data analysis and adapts data preparation to each workload which can minimize response times.

The second paper discusses how to provide machine learning capabilities inside a database, which is an FPGA-enabled database engine incorporating FPGA-based machine learning operators into a main memory, columnar DBMS.

The third paper presents two engineering approaches for integrating machine learning agents in a DBMS. The first is to build an external tuning controller that treats the DBMS as a black-box. The second is to integrate the machine learning agents natively in the DBMS architecture.

The fourth paper proposes the construction of an engine, a Data Alchemist, which learns how to blend fine-grained data structure design principles to automatically synthesize brand new data structures.

The fifth paper discusses the AutoML problem and proposes MILE, an environment where humans and machines together drive the search for desired ML solutions.

The last paper proposes an AI-native database. On one hand, it integrates AI techniques into databases to provide self-configuring, self-optimizing, self-monitoring, self-healing, self-diagnosis, self-security and self-assembling capabilities for databases. On the other hand, it can enable databases to provide AI capabilities using declarative languages, in order to lower the barrier of using AI.

I would like to thank all the authors for their insightful contributions. I hope you enjoy reading the papers.

Guoliang Li
Tsinghua University

Resurrecting Middle-Tier Distributed Transactions

Philip A. Bernstein
Microsoft Research, Redmond, WA 98052

1 Introduction

Over the years, platforms and application requirements change. As they do, technologies come, go, and return again as the preferred solution to certain system problems. In each of its incarnations, the technology's details change but the principles remain the same. One such technology is distributed transactions on middle-tier servers. Here, we argue that after a 15-year decline, they need to return to the mainstream.

In the 1980's, Transaction Processing (TP) monitors were a popular category of middleware product that enabled customers to build scalable distributed systems to run transactions. Example products were CICS (IBM), Tuxedo (AT&T for Unix), ACMS (DEC for VAX/VMS), and Pathway (Tandem for Guardian) [4]. Their main features were multithreaded processes (not supported natively by most operating systems), inter-process communication (usually a crude form of remote procedure call), and a forms manager (for end users to submit transaction requests). The TP monitor ran on middle-tier servers that received transaction requests from front-end processors that communicated with end-user devices, such as terminals and PC's, and with back end database servers. The top-level application code executed on the middle-tier and invoked stored procedures on the database server.

In those days, database management systems (DBMS's) supported ACID transactions, but hardly any of them supported distributed transactions. The TP monitor vendors saw this as a business opportunity and worked on adding a transaction manager feature that implemented the two-phase commit protocol (2PC). Such a feature required DBMS's to expose Start, Prepare, Commit, and Abort as operations that could be invoked by the TP monitor. Unfortunately, most of them didn't support Prepare, and even if they did, they didn't expose it to applications. They were willing to do so, but they didn't want to implement a different protocol for each TP monitor product. Thus, the XA standard was born, which defined TP monitor and DBMS interfaces (including Prepare) and protocols that allowed a TP monitor to run a distributed transaction across DBMS servers [17].

This middle-tier architecture for distributed transactions was popular for about 20 years, into the late 1990s. Then TP monitors were replaced by Application Servers, which integrated a TP monitor with web servers, so it could receive transaction requests over HTTP, rather than receiving them from devices connected by a local area or terminal network. Examples include Microsoft Transaction Server, later renamed COM+, and Java Enterprise Edition (JEE), implemented by IBM's WebSphere Application Server, Oracle's WebLogic Application Server, and Red Hat's JBoss Application Server [12]. The back end architecture was the same as before. Each transaction started executing on a middle-tier server and invoked stored procedures to read and write the database.

Although this execution model is still widely used, starting in the early 2000's it fell out of favor for new application development, especially for applications targeted for cloud computing. More database vendors offered built-in support for distributed transactions, so there was less need to control the distributed transaction from the middle tier. A larger part of database applications executed on data that was cached in the middle tier. And the NoSQL movement argued that distributed transactions were too slow, that they limited scalability, and that customers rarely needed them anyway [11]. Eventual consistency became all the rage [18].

The critics of distributed transactions had some good points. But in the end, developers found that mainstream

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

programmers really did need ACID transactions to make their applications reliable in the face of concurrent access to shared data and despite server failures. Thus, some NoSQL (key-value) stores added transaction support (e.g., CosmosDB [2], DynamoDB [8]). Google, which had initially avoided support for multi-row distributed transactions in Bigtable [5], later introduced them in Spanner [6]. There are now many cloud storage services and database products that support distributed transactions.

Like product developers, database researchers have also focused on distributed transactions for back-end database systems. Almost universally, they assume that transactions execute as stored procedures and that middle-tier applications invoke those stored procedures but do not execute the transaction logic themselves.

2 Stateful Middle-Tier Applications

This focus on stored procedures is well justified by the needs of traditional TP applications. However, stored procedures are not a good way of encapsulating application logic for a growing fraction of stateful applications that run on the middle tier. These include multi-player games, datacenter telemetry, Internet of Things, and social and mobile applications. Objects are a natural model for the entities in these applications, such as games, players, datacenter hardware, sensors, cameras, customers, and smart phones. Such applications have a large number of long-lived stateful objects that are spread over many servers and communicate via message passing. Like most new applications, these applications are usually developed to run on cloud computing platforms.

These applications typically execute on middle-tier servers, rather than as stored procedures in database servers. They do this for many reasons. They need large main memory for the state of long-lived objects. They often have heavy computation needs, such as rendering images or computing over large graphs. They use a lot of computation for message passing between objects so they can scale out. And they need computation to be elastic, independent of storage requirements. These needs are satisfied by compute servers that are cheaper than database servers because they have less storage. Hence, these apps run on compute servers in the middle tier.

2.1 Requirements for Mid-Tier Cloud Transactions

Some middle-tier applications need transactions because they have functions that read and write the state of two or more stateful objects. For example, a game may allow users to buy and sell virtual game objects, such as weapons, shields, and vehicles. A telemetry application may need to process an event exactly once by removing it from a queue and updating telemetry objects based on that event. A social application may need to add a user to a group and modify the user's state to indicate membership in that group. Each of these cases needs an ACID transaction over two or more objects, which may be distributed on different servers. Since these applications are usually developed to run on cloud computing platforms, distributed transaction support must be built into the cloud platform, a capability that is rarely supported today for cloud computing.

Distributed transactions for middle-tier applications on a cloud computing platform have four requirements that differ from those supported by the late-1990's products that run transactions on the middle-tier. First, like all previous transaction mechanisms, they need to offer excellent performance. But unlike previous mechanisms, it's essential that they be able to scale out to a large number of servers, leading to the first requirement: The system must have high throughput and low transaction latency, at least when transactions have low contention, and in addition must scale out to many servers.

To scale computation independently of storage, these applications typically save their state in cloud storage. The developers' choice of cloud storage service depends on their application's requirements (e.g., records, documents, blobs, SQL), their platform provider's offerings (e.g., AWS, Azure, Google), their employer's storage standards, and their developers' expertise. Thus, we have this second requirement: The transaction mechanism must support applications that use any cloud storage service.

The transaction mechanism needs persistent storage to track transaction state: started, prepared, committed,

or aborted. Like the apps themselves, it needs to use cloud storage for this purpose, which is the third requirement: The transaction mechanism must be able to use any of the cloud storage services used by applications.

The traditional data structure for storing transaction state is a log. The transaction manager relies on the order of records in the log to understand the order in which transactions executed. Although cloud vendors implement logs to support their database services, they do not expose database-style logging as a service for customers, leading to a fourth requirement: The transaction mechanism cannot rely on a shared log, unless it implements the log itself, in which case the log must run on a wide variety of storage services.

Due to the latency of cloud storage, requirements (2)-(4) create challenges in satisfying requirement (1).

The above requirements are a first cut, based on today's applications and platforms. It is also worth targeting variations. For example, requirement (1) could include cost/performance, which might require a tradeoff against scalability. And (4) might go away entirely if cloud platforms offer high-performance logging as a service.

3 An Implementation in the Orleans Framework

The rest of this paper sketches a distributed transaction mechanism that satisfies the above requirements [9]. Our group built it for Microsoft's actor-oriented programming framework, called Orleans, which is open source and runs on both Windows and Linux [16]. The distributed transaction project is part of a longer-term effort to enrich Orleans with other database features to evolve it into an actor-oriented database system that supports geo-distribution, stream processing, indexing, and other database features [3].

3.1 Two-Phase Commit and Locking

For ACID semantics, Orleans transactions use two-phase commit (2PC) and two-phase locking (2PL). Our first challenge was to obtain high throughput and scalability despite the requirement to use cloud storage. In our runs, a write to cloud storage within a datacenter takes 20 ms and has high variance. With 2PC, a transaction does two synchronous writes to storage. Therefore, if 2PL is used, a transaction holds locks for 40ms, which limits throughput to 25 transactions/second (TPS). Low-latency SSD-based cloud storage is faster, but still incurs over 10 ms latency, plus higher cost. To avoid this problem, we extended early lock release to 2PC [1, 7, 10, 13, 14, 15]. After a transaction T1 terminates, it releases locks before writing to storage in phase one of 2PC. This allows a later transaction T2 to read/update T1's updated objects. Thus, while T1 is writing to storage, a sequence of later transactions can update an object, terminate, and then unlock the object. To avoid inconsistency, the system delays committing transactions that directly or indirectly read or overwrite T1's writeset until after T1 commits. And if T1 aborts, then those later dependent transactions abort too. Using this mechanism, we have seen transaction throughput up to 20x that of strict 2PL/2PC.

3.2 Logging

Our initial implementation used a centralized transaction manager (TM) per server cluster [9]. It ran on an independent server and was multithreaded. Since message-passing is a potential bottleneck, it batched its messages to transaction servers. It worked well with throughput up to 100K TPS. However, it had three disadvantages: it was an obvious bottleneck for higher transaction rates; a minimum configuration required two servers (i.e., primary and backup TM) in addition to servers that execute the application; and it added configuration complexity since TM servers did not run Orleans and thus had to be deployed separately from application servers.

These disadvantages led us to redesign the system to avoid a centralized TM. Instead, we embed a TM in each application object. Each TM's log is piggybacked on its object's storage. This TM-per-object design avoids the above disadvantages and improves transaction latency by avoiding roundtrips to a centralized TM. However, it doesn't work for objects that have no updatable storage. For example, an object that performs a money transfer calls two stateful objects, the source and target of the transfer, but it has no state itself. We allow such an object to

participate in a transaction by delegating its TM function to a stateful participant in the transaction, that is, one that has updatable storage.

Orleans transactions write object state to a log to enable undo when a transaction aborts. This is impractical for large objects and is a poor fit for concurrency control that exploits operation commutativity. We therefore developed a prototype that logs operations.

4 Summary

Many new cloud applications run their logic on the middle tier, not as stored procedures. They need distributed transactions. Thus, cloud computing platforms can and should offer scalable distributed transactions.

5 Acknowledgments

I'm grateful to the team that built the initial and final implementations of Orleans transactions: Jason Bragg, Sebastian Burckhardt, Tamer Eldeeb, Reuben Bond, Sergey Bykov, Christopher Meiklejohn, Alejandro Tomsic, and Xiao Zeng. I also thank Bailu Ding and Dave Lomet for suggesting many improvements to this paper.

References

[1] Athanassoulis, Manos ; Johnson, Ryan ; Ailamaki, Anastasia ; Stoica, Radu, Improving OLTP Concurrency through Early Lock Release, EPFL-REPORT-152158, https://infoscience.epfl.ch/record/152158?ln=en, 2009.

[2] Azure CosmosDB, https://azure.microsoft.com/en-us/services/cosmos-db/

[3] Bernstein, P.A., M., T. Kiefer, D. Maier: Indexing in an Actor-Oriented Database. CIDR 2017

[4] Bernstein, P. A., E. Newcomer: Chapter 10: Transactional Middleware Products and Standards, in Principles of Transaction Processing, Morgan Kaufmann, 2nd ed., 2009.

[5] Chang, F., J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber: Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comput. Syst. 26(2): 4:1-4:26 (2008)

[6] Corbett, J.C. et al: Spanner: Google's Globally Distributed Database. ACM Trans. Comput. Syst. 31(3): 8:1-8:22 (2013)

[7] DeWitt, D.J., R.H. Katz, F. Olken, L.D. Shapiro, M. Stonebraker, D.A. Wood: Implementation Techniques for Main Memory Database Systems. SIGMOD 1984: 1-8

[8] DynamoDB, https://aws.amazon.com/dynamodb/

[9] Eldeeb, T. and P. Bernstein: Transactions for Distributed Actors in the Cloud. Microsoft Research Tech Report MSR-TR-2016-1001.

[10] Graefe, G., M. Lillibridge, H. A. Kuno, J. Tucek, A.C. Veitch: Controlled lock violation. SIGMOD 2013: 85-96

[11] Helland, P., Life beyond Distributed Transactions: an Apostate's Opinion. CIDR 2007: 132-141

[12] Java EE documentation, http://www.oracle.com/technetwork/?java/javaee/documentation/index.html

[13] Larson, P-A, et al.: High-Perf. Concurrency Control Mechanisms for Main-Memory Databases. PVLDB 5(4): 298-309 (2011)

[14] Levandoski, L.J., D.B. Lomet, S. Sengupta, R. Stutsman, R. Wang: High Performance Transactions in Deuteronomy. CIDR 2015

[15] David B. Lomet: Using Timestamping to Optimize Two Phase Commit. PDIS 1993: 48-55

[16] Orleans, http://dotnet.github.io/orleans

[17] The Open Group, Distributed Transaction Processing: The XA Specification, http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf.

[18] Vogels W., Eventually Consistent. ACM Queue 6(6): 14-19 (2008)

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, 17].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

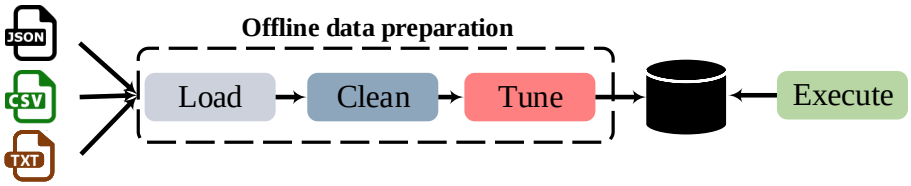

Figure 1: Data Processing pipeline.

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].

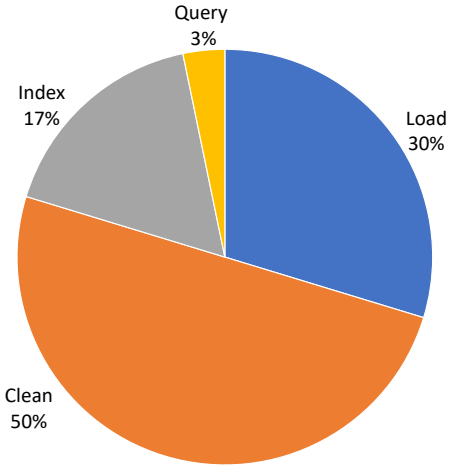

Figure 2: Cost of Data Preprocessing

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

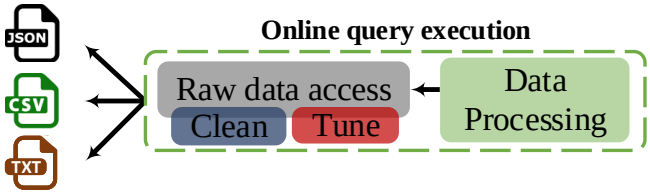

Figure 3: Integrating Cleaning and Tuning to Data Access.

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 5, 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 approach 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 workloads) and effective (i.e., standard operator models and execution patterns do not need to be modified). doppioDB runs

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

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.

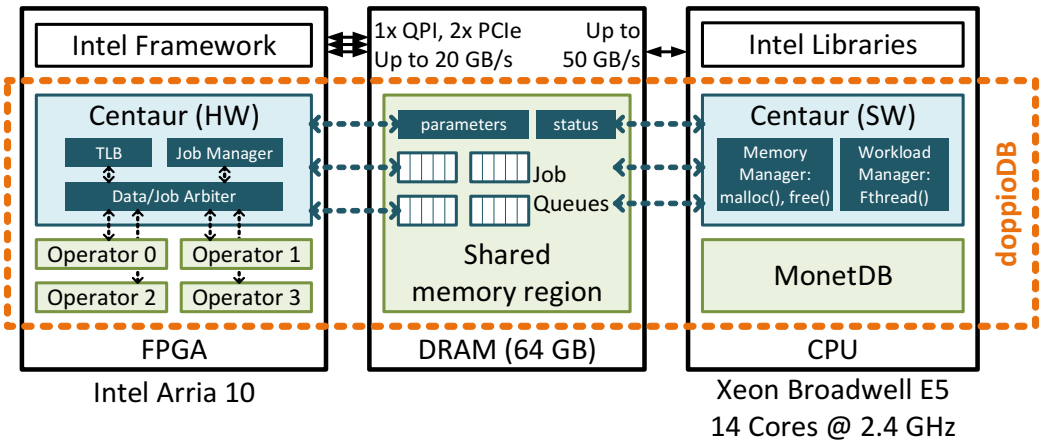

Figure 1: doppioDB using Centaur on Intel's Xeon+FPGA second generation prototype machine.

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

[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();
...
```

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.

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

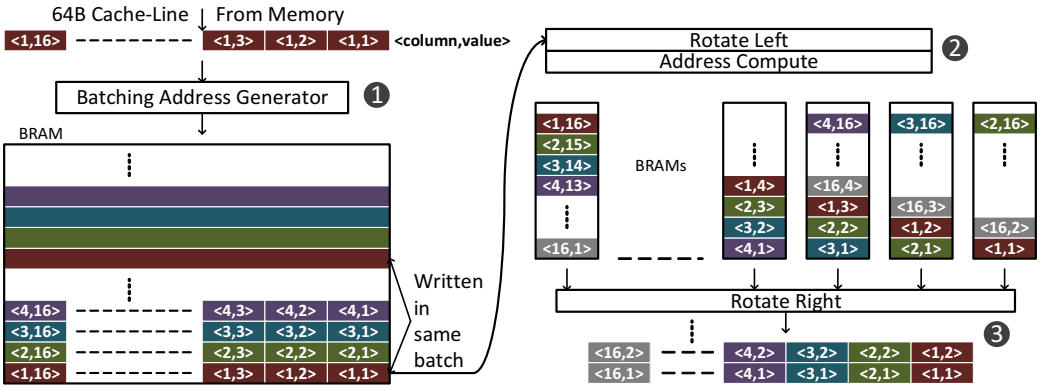

Figure 2: On the FPGA the transformation from columns to rows can be implemented as a streaming operation that introduces latency but has constant bandwidth.

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.

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

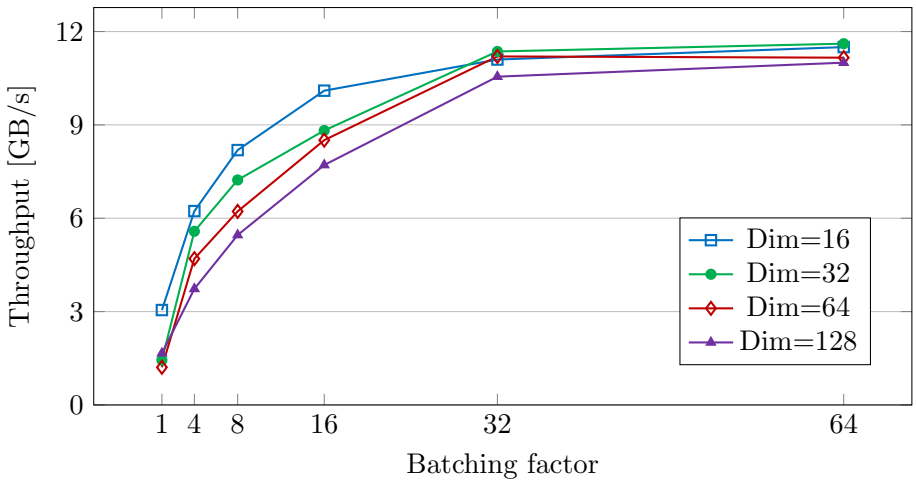

Figure 3: Streaming transformation from columns to rows reaches high throughput and is not impacted negatively by the number of dimensions to be gathered.

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;
```

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 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

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;
```

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 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

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

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].

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

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

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 $\tau^- \to \mu^-\mu^+\mu^-$," *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*

External vs. Internal: An Essay on Machine Learning Agents for Autonomous Database Management Systems

Andrew Pavlo[1], Matthew Butrovich[1], Ananya Joshi[1], Lin Ma[1],
Prashanth Menon[1], Dana Van Aken[1], Lisa Lee[1],Ruslan Salakhutdinov[1],
[1]Carnegie Mellon University

Abstract

The limitless number of possible ways to configure database management systems (DBMSs) has rightfully earned them the reputation of being difficult to manage and tune. Optimizing a DBMS to meet the needs of an application has surpassed the abilities of humans. This is because the correct configuration of a DBMS is highly dependent on a number of factors that are beyond what humans can reason about. The problem is further exacerbated in large-scale deployments with thousands or even millions of individual DBMS installations that each have their own tuning requirements.

To overcome this problem, recent research has explored using machine learning-based (ML) agents for automated tuning of DBMSs. These agents extract performance metrics and behavioral information from the DBMS and then train models with this data to select tuning actions that they predict will have the most benefit. They then observe how these actions affect the DBMS and update their models to further improve their efficacy.

In this paper, we discuss two engineering approaches for integrating ML agents in a DBMS. The first is to build an external tuning controller that treats the DBMS as a black-box. The second is to integrate the ML agents natively in the DBMS's architecture. We consider the trade-offs of these approaches in the context of two projects from Carnegie Mellon University (CMU).

1 Introduction

Tuning a DBMS is an essential part of any database application installation. The goal of this tuning is to improve a DBMS's operations based on some objective function (e.g., faster execution, lower costs, better availability). Modern DBMSs provide APIs that allow database administrators (DBAs) to control their runtime execution and storage operations: (1) *physical design*, (2) *knob configuration*, (3) *hardware resource allocation*, and (4) *query plan hints*. The first are changes to the database's physical representation and data structures (e.g., indexes, views, partitioning). The second are optimizations that affect the DBMS's behavior through its configuration knobs (e.g., caching policies). Resource allocations determine how the DBMS uses its available hardware to store data and execute queries; the DBA can either provision new resources (e.g., adding disks, memory, or machines) or redistribute existing resources (e.g., partitioning tables across disks). Lastly, query plan tuning hints are directives that force the DBMS's optimizer to make certain decisions for individual queries (e.g., join orderings).

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Given the notorious complexity of DBMS tuning, a reoccurring theme in database research over the last five decades has been on how to automate this process and reduce the burden on humans. The first efforts in the 1970s were on building *self-adaptive* systems [30]. These were primarily recommendation tools that focused on physical database design (e.g, indexes [31, 34], partitioning [32, 43], clustering [78]). They were also external to the DBMS and meant to assist the DBA with the tuning process. In the 1990s, the database community switched to using the moniker *self-tuning* systems [16, 73]. Like their predecessors, most of the self-tuning systems targeted automated physical design [29, 17, 71]. But they also expanded their scope to include automatic DBMS knob configuration [68, 20, 70]. This was necessary because by then the more mature DBMSs had hundreds of tunable knobs and the problem of how to set them correctly because too arduous [72]. Another notable difference was that while most of the self-adaptive methods were primarily in the context of standalone recommendation tools that were external to the DBMS, some vendors added self-tuning components directly inside of the DBMS [41, 67, 19].

The current research trend is on how to use of machine learning (ML) to devise "learned" methods for automated DBMS tuning. Instead of relying on static heuristics or cost models (see Section 2), these newer approaches train models using data collected about the DBMS's runtime behavior under various execution scenarios and configurations. The tuning *agent* then predicts the expected benefit of *actions* (e.g., add an index) using these models and selects the one with the greatest expected reward. The agent then observes the affects of the deployed action and integrates this new data back into the models to improve their efficacy for future decision making. This last step removes the need for a human to make judgment calls about whether or not to make a recommended change.

There are two ways developers can integrate such ML-based tuning methods for DBMSs. The first is to use external agents that observe and manipulate a DBMS through standard APIs (e.g., JDBC, ODBC). This approach is ideal for existing DBMSs where the software engineering effort required to retrofit the architecture to support automated tuning is too onerous. The second is to integrate internal components that directly operate inside of the DBMS. Building the ML components inside of the system obviates several problems related to training data collection and modeling, as well as enables the agent to have fine-grained control of the system's behavior. But this integration requires such a tight coupling that it is often only viable for those organizations that are designing a new system from scratch (i.e., a "greenfield" project) [59].

In this paper, we discuss the trade-offs between implementing ML-based tuning agents outside of the DBMS versus designing a new DBMS around automation. Our analysis is based on our experiences developing ML-based tuning tools for existing systems [72, 79, 58] and new autonomous architectures [9, 46, 57, 60, 81]. We begin in Section 2 with an overview of how DBMS tuning algorithms work. Next, in Section 3 we describe how to use ML-based agents to automatically tune systems. We then compare the benefits of the approaches, as well as discuss some of their challenges and unsolved problems. We conclude with an overview of two DBMS projects at CMU [1] on using ML for automated tuning. The first is **OtterTune** [5], an external knob tuning service for existing DBMSs. The other is a new self-driving [57] DBMS called **NoisePage** [3] (formerly Peloton[1]) that we are designing to be completely autonomous.

2 Background

Previous researchers in the last 50 years have studied and devised several methods for automatically optimizing DBMSs [12, 18, 16, 74]. As such, there is an extensive corpus of previous work, including both theoretical [14] and applied research [24, 82, 75]. Before the 2010s, the core methodologies for automated DBMS tuning were either (1) heuristic- or (2) cost-based optimization algorithms. We now briefly discuss this prior work to motivate the transition to ML-based methods in Section 3.

[1]We are unable to continue with the Peloton [6] project due to the unholy combination of engineering, legal, and marital problems.

The most widely used approach in automated DBMS tuning is to use *heuristic* algorithms that are comprised of hard-coded rules that recommend actions [32]. For example, IBM's first release of their DB2 Performance Wizard tool in the early 2000s asked the DBA questions about their application (e.g., whether it is OLTP or OLAP) and then provides knob settings based on their answers [42]. It uses rules manually created by DB2 engineers and thus may not accurately reflect the actual workload or operating environment. IBM later released a version of DB2 with a self-tuning memory manager that again uses rules to determine how to allocate the DBMS's memory to its internal components [67, 70]. Oracle has a similar "self-managing" memory manager in their DBMS [19], but also provides a tool to identify bottlenecks due to misconfiguration using rules [21, 40]. Others have used the same approach in tuning tools for Microsoft's SQL Server [53], MySQL [2], and Postgres [7].

The other common approach for DBMS tuning is to use *cost-based* algorithms that programmatically search for improvements to the DBMS's configuration. These algorithms are guided by a cost model that estimates the benefits of design choices based on a representative sample workload trace from the application. A cost model allows a tool to identify good actions without having to deploy them for real on the DBMS, which would is prohibitively expensive and slow. Previous work in this area has evaluated several search techniques, including greedy search [17], branch-and-bound search [58, 82, 54], local search [77], genetic algorithms [54], and relaxation/approximation [13].

To avoid the problem of a cost-based algorithm making choices that do not reflect what happens in the real DBMS, the tuning tool can use the DBMS's internal components to provide it with more accurate cost model estimations. The first and most notable application of this optimization was Microsoft's AutoAdmin use of SQL Server's built-in cost models from its query planner to estimate the utility of indexes [15]. Relying on the DBMS for the cost model, however, does not work for knob configuration tuning algorithms. In the case of Microsoft's example of using the query planner to guide its search algorithms, these models approximate on the amount of work to execute a query and are intended to compare alternative query execution strategies in a fixed environment [65]. But knobs affect a DBMS's behavior in ways that are not easily reflected (if even possible) in the query planner's cost model. For example, it is non-trivial for the planner to reason for a single query about a knob that changes caching policies since the behavior can vary depending on the workload. Hence, the major vendors' proprietary knob configuration tools mostly rely on static heuristics and vary in the amount of automation that they support [21, 40, 53, 2, 7].

The critical limitation in both of the above heuristic- and cost-based tuning methods is that they tune each DBMS in isolation. That is, they only reason about how to tune one particular DBMS instance and do not leverage information collected about previous tuning sessions. Heuristic-based methods rely on assumptions about the DBMS's workload and environment that may not accurately reflect the real-world. The lack of data reuse increases the amount of time that it takes for cost-based algorithms to find improvements for the DBMS. To avoid an exhaustive search every time, developers apply domain knowledge about databases to prune the solutions that are unlikely to provide any benefit. For example, an index selection algorithm can ignore candidate indexes for columns that are never accessed in queries. But database tuning problems are NP-Complete [52, 35], and thus solving them efficiently even with such optimizations is non-trivial. This is where ML-based approaches can potentially help by providing faster approximations for optimization problems.

3 Automated Tuning with Machine Learning

ML is a broad field of study that encompasses many disciplines and is applicable to many facets of DBMSs. There are engineering and operational challenges in incorporating ML-based components into already complex DBMS software stacks [61], such as explainability, maintainability/extendability, and stability. There are also important problems on automatically provisioning resources for a fleet of DBMSs in a cloud environment. We limit the scope of our discussion to the implications of integrating ML in DBMSs for tuning in either existing or new system architectures.

ML-based agents for automated DBMS tuning use algorithms that rely on statistical models to select actions that improve the system's target objective. That is, instead of being provided explicit instructions on how to tune the DBMS, the agent extracts patterns and inferences from the DBMS's past behavior to predict the expected behavior in the future to lean how to apply it to new actions. The agent selects an action that it believes will provide the most benefit to its target objective function. It then deploys this action without having to request permission from a DBA. This deployment can either be explicit (e.g., invoking a command to build an index) or implicit (i.e., updating its models so that the next invocation reflects the change).

Agents build their models from training data that they collect from the DBMS and its environment. This data can also come from other previous tuning sessions for other DBMSs if they provide the proper context (e.g., hardware profile). The type of data that an agent collects from the DBMS depends on its action domain. Some agents target a specific sub-system in the DBMS, and thus they need training data related to these parts of the system. For example, an agent that tunes the DBMS's query optimizer [48, 56] collects information about the workload (e.g., SQL queries) and the distribution of values in the database. Another agent that targets tuning the DBMS's knob configuration only needs low-level performance metrics as this data is emblematic of the overall behavior of the system for a workload [72, 80, 24]. A holistic tuning agent that seeks to control the entire DBMS collects data from every parts of the system because they must consider latent interactions between them [57].

How an agent acquires this data depends on whether it trains its models offline or online. Offline training is where the agent replays a sample workload trace while varying the DBMS's configuration in a controlled environment. This arrangement allows the agent to guide its training process to explore unknown regions in its solution space. Offline training also ensures that if the agent selects a bad configuration that it does not cause observable problems in the production environment. With online training, the agent observes the DBMS's behavior directly as it executes the application's workload. This approach does not require the system to provide the agent a workload sample; this allows the agent to always have an up-to-date view of the workload. The agent, however, may cause the system to make unexpected changes that hurt performance and require a human to intervene. Note also that the offline versus online approaches are not mutually exclusive and a DBMS can use both of them together.

ML methods are divided into three broad categories: (1) *supervised*, (2) *unsupervised*, and (3) *reinforcement learning*. There are existing DBMS tuning agents that use either one category of algorithms or some combination of them. We now describe these approaches in the context of DBMS tuning:

Supervised Learning: The agent builds models from training data that contains both the input and expected output (i.e., labels). The goal is for the models learn how to produce the correct answer for new inputs. This approach is useful for problems where the outcome of an action is immediately observable. An example of a supervised learning DBMS tuning method is an algorithm that predicts the cardinality of query plan operators [36, 45, 76, 37]. The training data input contains encoded vectors of each operator's salient features (e.g., type, predicates, input data sizes) and the output is the cardinality that the DBMS observed when executing the query. The objective for this agent is to minimize the difference between the predicted and actual cardinalities. Supervised learning has also been applied to tune other parts of a DBMS, including approximate indexes [39], performance modeling [26, 49], transaction scheduling [60, 63], query plan tuning [23], and knob tuning [72].

Unsupervised Learning: With this approach, the agent's training data only contains input values and not the expected output. It is up to the agent to infer whether the output from the models are correct. An example of this is an agent that clusters workloads into categories based on their access patterns patterns [51, 28]. The assigned categories have no human decipherable meaning other than the workloads in each category are similar in some way. Although not directly related to tuning, another use of unsupervised ML in DBMSs is for automatically detecting data anomalies in a database: the agent does not need to be told what are "correct" values to figure out what values do not look like the others [10].

Reinforcement Learning: Lastly, reinforcement learning (RL) is similar to unsupervised ML in that there

is no labeled training data. The agent trains a policy model that selects actions that will improve the target objective function for the current environment. RL approaches in general do not make assumptions about priors and the models are derived only from the agent's observations. This is useful for problems where the benefit or effect of an action are not immediately known. For example, the agent may choose to add an index to improve query performance, but the DBMS will take several minutes or even hours to build it. Even after the DBMS builds the index, the agent may still only observe its reward after a period of time if the queries that use do not come until later (e.g., due to workload pattern changes). Given the general purpose nature of RL, it is one of the most active areas of DBMS tuning research in the late 2010s. Researchers have applied RL for query optimization [48, 56, 50], index selection [11, 62, 23], partitioning [25, 33], and knob tuning [80].

We next discuss how to integrate agents that use the above ML approaches into DBMSs to enable them to support autonomous tuning and optimization features. We begin with an examination of strategies for running agents outside of the DBMS in Section 3.1. Then in Section 3.2 we consider the implications of integrating the agents directly inside of the DBMS. For each of these strategies, we first present the high-level approach and then list some of the key challenges that one must overcome with them.

3.1 External Agents

An external agent tunes a DBMS without requiring specialized ML components running inside of the system. The goal is to reuse the DBMS's existing APIs and environment data collection infrastructure (e.g., query traces, performance metrics) without having to modify the DBMS itself or for the DBMS to be even aware that software and not a human is managing it. Ideally a developer can create the agent in a general purpose such that one can reuse its backend ML component across multiple DBMSs.

An agent receives its objective function data either directly from the DBMS or through additional third-party monitoring tools (e.g., Prometheus, Druid, InfluxDB). The latter scenario is common in organizations with a large number of DBMS instances. Although the agent's ML algorithms are not tailored to any particular DBMS, there is DBMS-specific code to prepare the training data for consumption by the algorithm. This is colloquially known as *glue* code in ML parlance [61]. For example, the agent has to encode configuration knobs with fixed "enum" values, known as a categorical variables, as separate one-hot encoded features since ML algorithms cannot operate on strings [72]. To do this encoding correctly, the agent must obviously be aware of all possible values a knob can take; it is too difficult and a waste of time to try to infer this on its own.

Agents may also need an additional *controller* running on the same machine as the DBMS or within the same administrative domain [72]. This controller is allowed to install changes that are not accessible through the DBMS's APIs. For example, DBMSs that read configuration files at start-up on local disk will overwrite any previously set knob values. Thus, unless the agent is able to write these files, then it will not be able to persist changes. The controller may need to also restart the DBMS because many systems are not able to apply changes until after a restart.

Challenges: There are several challenges in tuning an existing DBMS that was not originally designed for autonomous operation. Foremost is that almost every major DBMS that we have examined does not support making changes to the system's configuration without periods of unavailability, either due to restarts or blocking execution [59]. Requiring the DBMS to halt execution in order to apply a change makes it difficult for agents to explore configurations in production systems and increases the time it takes to collect training data. There are methods for reducing start-up times [8, 27], but the agent still must also account for this time in their reward functions, which are often non-deterministic.

The second issue is that an agent is only able to collect performance metrics that the system already exposes. This means that if there is additional information that the agent needs, then it is not immediately available. The other issue is that is that there is often an overabundance of data that makes it difficult to separate signals from the noise [72]. Furthermore, DBMSs also do not expose information about their underlying hardware so the system

can reuse training data across operating environments. In many cases these metrics were originally meant to assist humans with diagnosing performance problems. That is, the developers added metrics assuming that they were meant for human consumption and not for enabling autonomous control. Many DBMSs do not report metrics at consistent intervals using the same unit of measurement.

Lastly, every DBMS has knobs that requires human knowledge in order to know how to set it correctly. There are obvious cases, like knobs that define file paths or port numbers, where the system will not function if they are set incorrectly. But there are other knobs where if an agent sets it incorrectly then the system will not become inoperable; instead, they will subtly affect the database's correctness or safety. The most common example of this that we found is whether to require the DBMS to flush a transaction's log records to disk before it is committed. Turning off this flushing improves performance but may lead to data loss on failures. If an agent discovers that changing this knob improves the objective function, then it will make that change. But the agent is unable to know what the right choice is because it requires a human to decide what is allowed in their organization. The agent's developers must mark such knobs as untunable in the glue code so that an agent do not modify them.

3.2 Internal Agents

An alternative to treating the DBMS like a black-box and tuning it with an external agent is to design the system's architecture to natively support autonomous operation. With this approach, the DBMS supports one or more agents running inside of the system. These agents collect their own training data, apply changes to the DBMS (ideally without restarting), and then observe how those changes affect the objective. The system does not require guidance or approval from a human for any of these steps. The benefit of running agents inside of the DBMS is that it exposes more information about the system's runtime behavior and can potentially enable more low-level control of the DBMS than what is possible with an external agent.

Most of the proposed ML tuning agents that are available today are designed to extend or replace components in existing DBMSs. One of the first of these was IBM's Learning Optimizer from the early 2000s that used a feedback mechanisms to update the query planner's cost models with data that the system observed during from query execution [66]. There are now more sophisticated proposals for changing the cost model with learned models to estimate cardinalities [36, 45, 76, 37] or even generate the query plan itself [56, 47]. These agents can also leverage the DBMS's existing components to help them "bootstrap" their models and provide them with a reasonable starting point. One notable example of augmenting an existing DBMS with ML agents is Oracle's cloud-based autonomous DBMS offering [4]. Although there is little public information about its implementation, our understanding from discussions with their developers is that it uses Oracle's previous independent tuning tools in a managed environment with limited centralized coordination.

Instead of augmenting an existing DBMS, others have looked into creating new DBMS architectures that are designed from the ground up for autonomous control [57, 62, 38]. With a new system, the developers can tailor its implementation to make its components easier to model and control. They can also customize the architecture to be more friendly to automated agents (e.g., avoiding the need to restart the system when changing knobs, having a unified action deployment framework) [59].

Challenges: The biggest problem with replacing a DBMS's existing components with new ML-based implementations is that it is hard to capture the dependencies between them. That is, if each tuning agent operates under the assumption that the other parts of the system are fixed, then their models will encapsulate this assumption. But then if each agent modifies their part of the DBMS that they control, then it will be hard to make accurate predictions. Consider a tuning agent that controls the DBMS's memory allocations. Suppose the agent initially assigns a small amount of memory for query result caching and a large amount to the buffer pool. Another index tuning agent running inside of the same DBMS then chooses to build an index because memory pressure in the buffer pool is low. But then the memory agent decides on its own to increase the result cache size and decrease the buffer pool size. With this change, there is now less memory available to store the index and

data pages, thereby increasing the amount of disk I/O that the DBMS incurs during query execution. Thus, the index that the second agent just added is now a bad choice because of change in another part of the system that it does not control.

There are three possible ways to overcome this problem but each of them have their own set of issues. The first is to use a single centralized agent rather than separate agents. This is potentially the most practical but greatly increases the dimensionality (i.e., complexity) of the models, which requires significantly more training data. The second is to have each individual agent provide a performance guarantee about what its changes in the DBMS. The agents provide this information to a central coordinator that is in charge of resource allocations. The last approach is to have a decentralized architecture where agents communicate and coordinate with each other. We suspect that this will prove to be too difficult to achieve reasonable stability or explainability.

One of the most expensive parts of these agents is when they build their models from the training data. The DBMS must prevent the agents from degrading the execution performance of the regular workload during this process. Thus, even though the agent runs inside of the DBMS, it could offload this step to auxiliary computational resources (i.e., GPUs, other machines).

4 OtterTune – Automated Knob Tuning Service for Existing DBMSs

OtterTune is an external knob configuration tuning service that works with any DBMS [72, 79, 5]. It maintains a repository of data collected from previous tuning sessions, and uses this data to build models of how the DBMS responds to different knob configurations. For a new application, it uses these models to guide experimentation and recommend optimal settings. Each recommendation provides OtterTune with more information in a feedback loop that allows it to refine its models and improve their accuracy.

As shown in Section 1, OtterTune's architecture is made up of a client-side *controller* and a server-side *tuning manager*. The controller acts as a conduit between the target DBMS and the tuning manager. It contains DBMS-specific code for collecting runtime information from the target DBMS (e.g., executing SQL commands via JDBC) and installs configurations recommended by the tuning manager. Again, the high-level operations are the same for all DBMSs but the exact commands differ for each DBMS: the controller updates the DBMS's configuration file on disk and then restarts the system using the appropriate administrative tool. The tuning manager updates its repository and internal ML models with the information provided by the controller and then recommends a new configuration for the user to try.

To initialize a new tuning session, the user first selects which metric should be the *target objective* for OtterTune to optimize when selecting configurations. OtterTune retrieves this information either from (1) the DBMS itself via its query API, (2) a third-party monitoring service, or (3) a benchmarking framework [22]. OtterTune also requests other necessary information from the user about the target DBMS at this time, such as the DBMS's version and connection information.

OtterTune then begins the first *observation period* where the controller connects to the the DBMS and runs the sample workload. Once the observation period is over, OtterTune collects the DBMS's runtime metrics and configuration knobs, and then delivers this information to the tuning manager. The result from the first observation period serves as a baseline since it reflects the DBMS's performance using its original configuration.

OtterTune's tuning manager receives the result from the last observation period from the controller and stores it in its repository. Next, the tuning manager selects the next configuration to try on the target DBMS. This process continues until the user is satisfied with the improvements over the original configuration.

4.1 Machine Learning Pipeline

OtterTune's ML pipeline uses a combination of supervised and unsupervised methods. It processes, analyzes, and builds models from the data in its repository. Both the Workload Characterization and Knob Identification

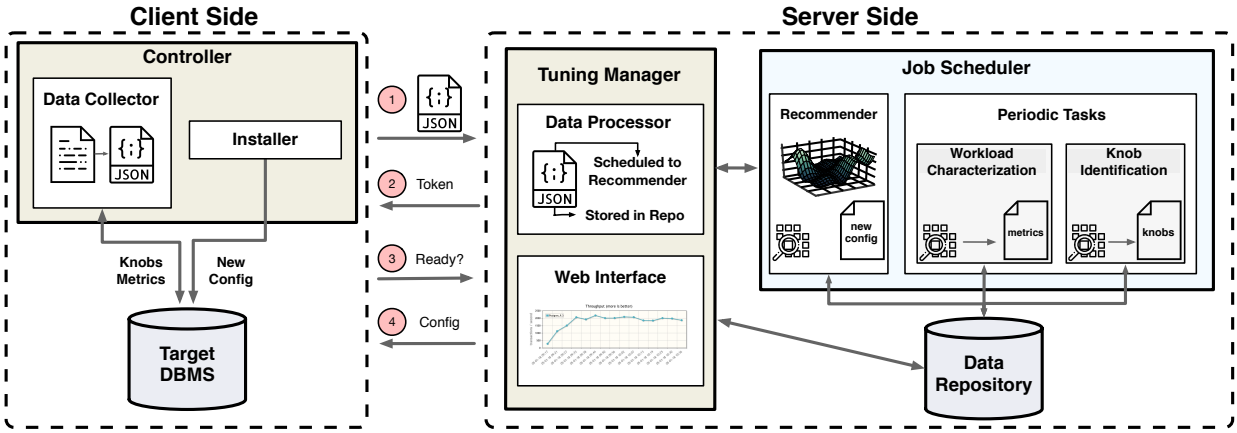

Figure 1: **OtterTune Architecture Overview** – The controller connects to the target DBMS, collects its knob/-metric data, transforms the collected information into a JSON document, and sends it to the server-side tuning manager. The tuning manager stores the information in the data repository and schedules a new task with the job scheduler to compute the next configuration for the target DBMS to try. The controller (1) sends the information to the tuning manager, (2) gets a token from the tuning manager, (3) uses this token to check status of the tuning job, and (4) gets the recommended configuration when the job finishes and then the agent installs it in the DBMS.

modules execute as a background task that periodically updates their models as new data becomes available in the repository. The tuning manager uses these models to generate new knob configurations for the target DBMS. OtterTune's ML pipeline has three modules:

Workload Characterization: This first component compresses all of the past metric data in the repository into a smaller set of metrics that capture the distinguishing characteristics for different workloads. It uses *factor analysis* (FA) to model each internal runtime metric as linear combinations of a few factors. It then clusters the metrics via k-means, using their factor coefficients as coordinates. Similar metrics are in the same cluster, and it selects one representative metric from each cluster, namely, the one closest to the cluster's center.

Knob Identification: The next component analyzes all past observations in the repository to determine which knobs have the most impact on the DBMS's performance for different workloads. OtterTune uses a popular feature-selection technique, called *Lasso* [69], to determine which knobs have the most impact the system's overall performance. Lasso is similar to the least-squares model, except that it uses L1 regularization. It forces certain coefficients to be set to zero. The larger weight of L1 penalty is, the more coefficients become zero.

Automated Tuner: In the last step, the tuner analyzes the results it has collected so far in the tuning session to decide which configuration to recommend next. It performs a two-step analysis after each observation period. First, the system uses the performance data for the metrics identified in the Workload Characterization component to identify the workload from a previous tuning session that best represents the target DBMS's workload. It compares the metrics collected so far in the tuning session with those from previous workloads by calculating the Euclidean distance, and finds the previous workload that is most similar to the target workload, namely, the one with smallest Euclidean distance.

5 NoisePage – A Self-Driving DBMS Architecture

NoisePage is a new DBMS that we are developing at CMU to be self-driving [57, 3]. This means that the system is able to tune and optimize itself automatically without any human intervention other than selecting the target objective function on start-up. The DBMS's core architecture is a Postgres-compatible HTAP system. It uses HyPer-style MVCC [55] over Apache Arrow in-memory columnar storage [44]. We chose an in-memory

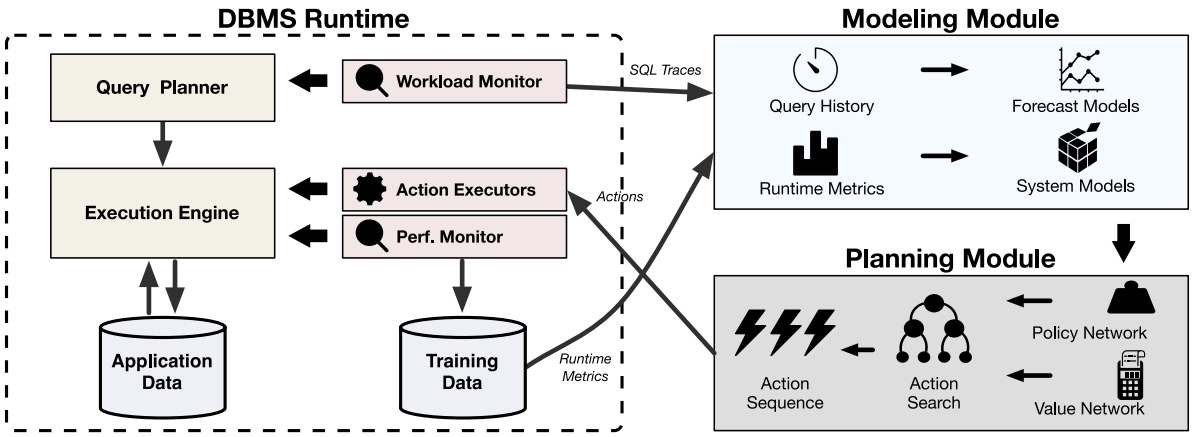

Figure 2: **NoisePage Architecture Overview** – The DBMS contains workload and performance monitors that collect runtime data about the system. It then stores this information in a separate training data repository. The Modeling module retrieves this information and builds forecasting models. These models are then used to guide the Planning module's search process for selecting actions that it believes will improve the DBMS's objective function.

architecture to enable the system to apply actions in minutes (instead of hours) with minimal impact on the application's performance. Faster action deployment enables the system to quickly explore solutions and collect new training data. This enables the system to react better to changes in the application's workload and its operating environment.

The DBMS's control agent runs in a continuous loop where it selects actions to deploy that it estimates will improve the target objective. NoisePage supports actions that affect (1) database physical design (e.g., add/drop indexes), (2) knob configuration (e.g., memory allocation), and (3) hardware resources (e.g., scaling up/down). We are not currently investigating how to automatically support query plan tuning as these requires more fine-grained models that can reason about individual sessions.

Because the DBMS is built from scratch for autonomous control, we designed the architecture in a modular manner to allow agents to collect training data offline more efficiently. That is, one can start just a single component in the DBMS (e.g., the transaction manager) and then perform a parameter sweep across many configurations without having to go through the DBMS's entire execution path. This reduces the number of redundant configurations that the system observes to (1) improve data collection efficiency and (2) reduce model over-fitting. The DBMS then combines this offline data with data collected online during query execution to improve it accuracy.

5.1 Machine Learning Pipeline

We next provide an overview of NoisePage's self-driving pipeline. Section 2 illustrates the overall architecture of the DBMS with its modeling and planning modules. The DBMS is instrumented with a workload and performance monitors that collect information about the entire system during both query execution and action deployment.

Modeling: This first module is responsible for training prediction models using data that the monitors collect from observing its runtime operations. There are two categories of models. The first are forecast models that predict the application's future workload and database state. Forecasting is necessary so that DBMS can prepare itself accordingly, much like a self-driving car has to predict the road condition up ahead. But instead of using cameras and LIDAR like a car, a self-driving DBMS uses workload traces and database statistics to generate forecast models [46]. These models are independent of the DBMS's configuration since they are determined by the application. The second category of models predict how the DBMS's internal sub-systems will respond to

configuration changes made by actions. The DBMS trains these models from its internal metrics collected by its performance monitors. It then computes how changes in these models affect the target objective function. This is known as the "value function" in ML algorithms.

Planning: In the second module, the DBMS use the models generated in the previous step to select actions that provide the best reward (i.e., objective function improvement) given the current state of the system. This reward includes an estimation the cost of deploying each action. The system estimates the application's behavior for some finite prediction horizon using its workload forecast models. It then searches for an action that achieves the best reward without violating human-defined constraints (e.g., hardware budgets, SLOs). This search can use either (1) tree-based optimization methods, such as a Monte Carlo search tree [64] or (2) RL methods using deep neural networks for policy and value functions. The search can be weighted so that it is more likely to consider the actions that provide the most benefit for the current state and avoid recently reversed actions.

To avoid having to consider all possible actions at each iteration (e.g., indexes for every possible combination of columns), there is a large corpus of previous work on pruning less effective or useless actions [16]. Since the set of relevant candidate actions is dependent on the DBMS environment, it can change multiple times during the day. Thus, one key unsolved problem, however, is how to represent this dynamic action set in the DBMS's models' fixed-length feature vectors.

Deployment: For a given action selected in the planning module, the next step is for the DBMS to deploy it. This part includes the mechanisms to efficiently execute the action's sub-tasks, as well as the ability to observe the action's effect on its performance both during and after the deployment. The DBMS's planning modules use the data that it collects during this phase to update their models and improve their decision making.

6 Conclusion

Autonomous DBMSs will enable organizations to deploy database applications that are more complex than what is possible today, and at a lower cost in terms of both hardware and personnel. In this paper, we surveyed the approaches for adding automatic tuning agents based on ML to DBMSs. We discussed the high-level differences of external versus and internal agents, as well as the separate challenges in these approaches. We then discussed two examples of these architectures from CMU: (1) OtterTune [5] and (2) NoisePage [3]. Although there is still a substantial amount of research needed in both systems and ML before we achieve fully autonomous (i.e., self-driving) DBMSs, we contend that the field has made several important steps towards this goal in recent years.

One final point that we would like to make is that we believe that autonomous DBMSs will not supplant DBAs. We instead envision these systems will emancipate them from the burdens of arduous low-level tuning and allow them to pursue higher minded tasks, such as database design and development.

7 Acknowledgments

This work was supported (in part) by the National Science Foundation Intel Science and Technology Center for Visual Cloud Systems, Google Research Grants, AWS Cloud Credits for Research, and the Alfred P. Sloan Research Fellowship program.

References

[1] Carnegie Mellon Database Group. `https://db.cs.cmu.edu`.

[2] MySQL Tuning Primer Script. `https://launchpad.net/mysql-tuning-primer`.

[3] NoisePage. https://noise.page.

[4] Oracle Self-Driving Database. https://www.oracle.com/database/autonomous-database/index.html.

[5] OtterTune. https://ottertune.cs.cmu.edu.

[6] Peloton. https://pelotondb.io.

[7] PostgreSQL Configuration Wizard. https://pgtune.leopard.in.ua.

[8] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. L. Wiener, and O. Zed. Scuba: Diving into data at facebook. *Proc. VLDB Endow.*, 6(11):1057–1067, Aug. 2013.

[9] J. Arulraj, A. Pavlo, and P. Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 583–598, 2016.

[10] P. Bailis, E. Gan, S. Madden, D. Narayanan, K. Rong, and S. Suri. Macrobase: Prioritizing attention in fast data. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 541–556, 2017.

[11] D. Basu, Q. Lin, W. Chen, H. T. Vo, Z. Yuan, P. Senellart, and S. Bressan. *Cost-Model Oblivious Database Tuning with Reinforcement Learning*, pages 253–268. 2015.

[12] P. Bernstein, M. Brodie, S. Ceri, D. DeWitt, M. Franklin, H. Garcia-Molina, J. Gray, J. Held, J. Hellerstein, H. Jagadish, et al. The asilomar report on database research. *SIGMOD record*, 27(4):74–80, 1998.

[13] N. Bruno and S. Chaudhuri. Automatic physical database tuning: a relaxation-based approach. In *SIGMOD*, pages 227–238, 2005.

[14] S. Ceri, S. Navathe, and G. Wiederhold. Distribution design of logical database schemas. *IEEE Trans. Softw. Eng.*, 9(4):487–504, 1983.

[15] S. Chaudhuri and V. Narasayya. Autoadmin "what-if" index analysis utility. *SIGMOD Rec.*, 27(2):367–378, 1998.

[16] S. Chaudhuri and V. Narasayya. Self-tuning database systems: a decade of progress. In *VLDB*, pages 3–14, 2007.

[17] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft SQL server. In *VLDB*, pages 146–155, 1997.

[18] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning RISC-style database system. In *VLDB*, pages 1–10, 2000.

[19] B. Dageville and M. Zait. Sql memory management in oracle9i. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 962–973, 2002.

[20] B. Debnath, D. Lilja, and M. Mokbel. SARD: A statistical approach for ranking database tuning parameters. In *ICDEW*, pages 11–18, 2008.

[21] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood. Automatic performance diagnosis and tuning in oracle. In *CIdR*, 2005.

[22] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[23] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. Narasayya. Ai meets ai: Leveraging query executions to improve index recommendations. In *Proceedings of the 2019 ACM International Conference on Management of Data*, SIGMOD '19, 2019.

[24] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with iTuned. *VLDB*, 2:1246–1257, August 2009.

[25] G. C. Durand, M. Pinnecke, R. Piriyev, M. Mohsen, D. Broneske, G. Saake, M. S. Sekeran, F. Rodriguez, and L. Balami. Gridformation: Towards self-driven online data partitioning using reinforcement learning. aiDM'18, pages 1:1–1:7, 2018.

[26] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *International Conference on Data Engineering*, pages 592–603. IEEE, 2009.

[27] G. Graefe, W. Guy, and C. Sauer. *Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, and Media Restore*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2014.

[28] C. Gupta, A. Mehta, and U. Dayal. PQR: Predicting Query Execution Times for Autonomous Workload Management. In *ICAC*, pages 13–22, 2008.

[29] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for olap. In *ICDE*, pages 208–219, 1997.

[30] M. Hammer. Self-adaptive automatic data base design. In *National Computer Conference*, AFIPS '77, pages 123–129, 1977.

[31] M. Hammer and A. Chan. Index selection in a self-adaptive data base management system. In *SIGMOD*, pages 1–8, 1976.

[32] M. Hammer and B. Niamir. A heuristic approach to attribute partitioning. In *SIGMOD*, pages 93–101, 1979.

[33] B. Hilprecht, C. Binnig, and U. Röhm. Towards learning a partitioning advisor with deep reinforcement learning. In *aiDM@SIGMOD*, pages 6:1–6:4, 2019.

[34] S. E. Hudson and R. King. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Trans. Database Syst.*, 14(3):291–321, Sept. 1989.

[35] M. Y. L. Ip, L. V. Saxton, and V. V. Raghavan. On the selection of an optimal set of indexes. *IEEE Trans. Softw. Eng.*, 9(2):135–143, 1983.

[36] O. Ivanov and S. Bartunov. Adaptive cardinality estimation. *CoRR*, abs/1711.08330, 2017.

[37] 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.

[38] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. Sagedb: A learned database system. In *CIDR*, 2019.

[39] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504, 2018.

[40] S. Kumar. Oracle Database 10g: The self-managing database, Nov. 2003. White Paper.

[41] E. Kwan, S. Lightstone, A. Storm, and L. Wu. Automatic configuration for IBM DB2 universal database. Technical report, IBM, jan 2002.

[42] E. Kwan, S. Lightstone, A. Storm, and L. Wu. Automatic configuration for IBM DB2 universal database. Technical report, IBM, jan 2002.

[43] K. D. Levin. Adaptive structuring of distributed databases. In *National Computer Conference*, AFIPS '82, pages 691–696, 1982.

[44] T. Li, M. Butrovich, A. Ngom, W. McKinney, and A. Pavlo. Mainlining databases: Supporting fast transactional workloads on universal columnar data file formats. 2019. *Under Submission*.

[45] H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte. Cardinality estimation using neural networks. CASCON '15, pages 53–59, 2015.

[46] 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 ACM International Conference on Management of Data*, SIGMOD '18, 2018.

[47] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer. *CoRR*, abs/1904.03711, 2019.

[48] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In *aiDM@SIGMOD*, pages 3:1–3:4, 2018.

[49] R. Marcus and O. Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *CoRR*, abs/1902.00132, 2019.

[50] R. Marcus and O. Papaemmanouil. Towards a hands-free query optimizer through deep learning. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research*, 2019.

[51] B. Mozafari, C. Curino, A. Jindal, and S. Madden. Performance and resource modeling in highly-concurrent oltp workloads. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 301–312, 2013.

[52] R. Mukkamala, S. C. Bruell, and R. K. Shultz. Design of partially replicated distributed database systems: an integrated methodology. *SIGMETRICS Perform. Eval. Rev.*, 16(1):187–196, 1988.

[53] D. Narayanan, E. Thereska, and A. Ailamaki. Continuous resource monitoring for self-predicting DBMS. In *MASCOTS*, pages 239–248, 2005.

[54] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *SIGMOD*, SIGMOD, pages 1137–1148, 2011.

[55] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. SIGMOD, 2015.

[56] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning state representations for query optimization with deep reinforcement learning. DEEM'18, pages 4:1–4:4, 2018.

[57] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-driving database management systems. In *CIDR 2017, Conference on Innovative Data Systems Research*, 2017.

[58] A. Pavlo, C. Curino, and S. Zdonik. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *SIGMOD*, pages 61–72, 2012.

[59] A. Pavlo et al. Make Your Database Dream of Electric Sheep: Engineering for Self-Driving Operation. 2019. *Under Submission*.

[60] A. Pavlo, E. P. Jones, and S. Zdonik. On predictive modeling for optimizing transaction execution in parallel oltp systems. *Proc. VLDB Endow.*, 5:85–96, October 2011.

[61] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, and M. Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[62] A. Sharma, F. M. Schuhknecht, and J. Dittrich. The case for automatic database administration using deep reinforcement learning. *CoRR*, abs/1801.05643, 2018.

[63] Y. Sheng, A. Tomasic, T. Zhang, and A. Pavlo. Scheduling OLTP transactions via learned abort prediction. In *aiDM@SIGMOD*, pages 1:1–1:8, 2019.

[64] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[65] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *SIGMOD*, pages 953–966, 2008.

[66] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 19–28, 2001.

[67] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in DB2. In *VLDB*, pages 1081–1092, 2006.

[68] D. G. Sullivan, M. I. Seltzer, and A. Pfeffer. Using probabilistic reasoning to automate software tuning. SIGMETRICS, pages 404–405, 2004.

[69] T.Hastie, R.Tibshirani, and J.Friedman. *The Elements of Statistical Learning*. Springer, 2001.

[70] W. Tian, P. Martin, and W. Powley. Techniques for automatically sizing multiple buffer pools in DB2. In *CASCON*, pages 294–302, 2003.

[71] G. Valentin, M. Zuliani, D. Zilio, G. Lohman, and A. Skelley. DB2 advisor: an optimizer smart enough to recommend its own indexes. In *ICDE*, pages 101–110, 2000.

[72] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1009–1024, 2017.

[73] G. Weikum, C. Hasse, A. Mönkeberg, and P. Zabback. The COMFORT automatic tuning project. *Information Systems*, 19(5):381–432, July 1994.

[74] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: From wishful thinking to viable engineering. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 20–31, 2002.

[75] D. Wiese, G. Rabinovitch, M. Reichert, and S. Arenswald. Autonomic tuning expert: A framework for best-practice oriented autonomic database tuning. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, pages 3:27–3:41, 2008.

[76] L. Woltmann, C. Hartmann, M. Thiele, D. Habich, and W. Lehner. Cardinality estimation with local deep learning models. In *aiDM@SIGMOD*, pages 5:1–5:8, 2019.

[77] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *WWW*, pages 287–296, 2004.

[78] C. T. Yu, C.-m. Suen, K. Lam, and M. K. Siu. Adaptive record clustering. *ACM Trans. Database Syst.*, 10(2):180–204, June 1985.

[79] B. Zhang, D. V. Aken, J. Wang, T. Dai, S. Jiang, J. Lao, S. Sheng, A. Pavlo, and G. J. Gordon. A demonstration of the ottertune automatic database management system tuning service. *PVLDB*, 11(12):1910–1913, 2018.

[80] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 ACM International Conference on Management of Data*, SIGMOD '19, 2019.

[81] T. Zhang, A. Tomasic, Y. Sheng, and A. Pavlo. Performance of OLTP via intelligent scheduling. In *ICDE*, pages 1288–1291, 2018.

[82] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: integrated automatic physical database design. In *VLDB*, pages 1087–1097, 2004.

Learning Data Structure Alchemy

Stratos Idreos Kostas Zoumpatianos Subarna Chatterjee Wilson Qin Abdul Wasay
Brian Hentschel Mike Kester Niv Dayan Demi Guo Minseo Kang Yiyou Sun

Harvard University

Abstract

We propose a solution based on first principles and AI to the decades-old problem of data structure design. Instead of working on individual designs that each can only be helpful in a small set of environments, we propose the construction of an engine, a Data Alchemist, which learns how to blend fine-grained data structure design principles to automatically synthesize brand new data structures.

1 Computing Instead of Inventing Data Structures

What do analytics, machine learning, data science, and big data systems have in common? What is the major common component for astronomy, biology, neuroscience, and all other data-driven and computational sciences? Data structures.

Data structures are one of the most fundamental areas of computer science. They are at the core of all subfields, including data systems, compilers, operating systems, human-computer interaction systems, networks, and machine learning. A data structure defines how data is physically stored. **For all algorithms that deal with data, their design starts by defining a data structure** that minimizes computation and data movement [1, 10, 37, 31, 63, 65, 97, 98, 22]. For example, we can only utilize an optimized sorted search algorithm if the data is sorted and if we can maintain it efficiently in such a state.

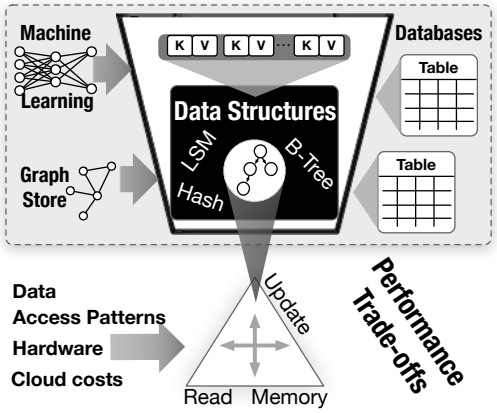

Figure 1: Design trade-offs.

A Hard, Open, and Continuous Problem. Since the early days of computer science dozens of new data structures are published every year. The pace has increased over the last decade, with 50-80 new data structures yearly according to data from DBLP [25]. This is because of 1) the growth of data, 2) the increasing number of data-driven applications, 3) more fields moving to a computational paradigm where data collection, storage, and analysis become critical, and 4) hardware changes that require a complete redesign of data structures and algorithms. Furthermore, for the past several decades, the hardware trend is that computation (e.g., CPUs, GPUs) becomes increasingly faster relative to data movement (e.g, memory, disks). This makes data structure design ever more critical as the way we store data dictates how much data an algorithm moves.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

There Is No Perfect Data Structure Design. Each design is a compromise among the fundamental tradeoffs [10]: read, update, and memory amplification. This is depicted visually in Figure 1. Read amplification is how much excess data an algorithm is forced to read; due to hardware properties such as page-based storage, even when reading a single data item, an algorithm has to load all items of the respective page. Write amplification is how much excess data an algorithm has to write; maintaining the structure of the data during updates typically causes additional writes. Memory amplification is how much excess data we have to store; any indexing that helps navigate the raw data is auxiliary data. Overall, this complex three-way tradeoff causes each design to be effective for only specific workloads [10]. For example, a Log-Structured Merge-tree (LSM-tree) relies on batching and logging data without enforcing a global order. While this helps with writes, it hurts reads since now a single read query (might) have to search all LSM-tree levels. Similarly, a sorted array enforces an organization in the data which favors reads but hurts writes, e.g., inserting a new item in a sorted (dense) array requires rewriting half the array on average. In this way, there is no universally good design. To get good performance, the design of a data structure has to be tailored to the data, queries, and hardware of the target application.

The Vast Design Space Slows Progress. The design of a data structure consists of 1) a data layout, and 2) algorithms that support basic operations (e.g., put, get, update). The data layout design itself may be further broken down into 1) the base data layout, and 2) an index which helps navigate the data, i.e., the leaves of a B^+tree and its inner nodes, or the buckets of a hash table and the hash-map. We use the term data structure design throughout the proposal to refer to the overall design of the base data layout, indexing, and the algorithms together. A data structure can be as simple as an array or as arbitrarily complex as using sophisticated combinations of hashing, range and radix partitioning, careful data placement, and encodings. There are so many different ways to design a data structure and so many moving targets that it has become a notoriously hard problem; it takes several months even for experts to design and optimize a new structure. Most often, the initial data structure decisions for a data-intensive system remain intact; it is too complex to redesign a data structure, predict what its impact would be as workload and hardware change, implement it, and integrate it within a complex system or pipeline. Along the same lines, it is very hard to know when and why the core data structure within a complex system will "break", i.e., when performance will drop below an acceptable threshold.

The Problem In Sciences. For data-driven fields without computer science expertise, these low-level choices are impossible to make. The only viable solution is using suboptimal off-the-shelf designs or hiring expensive experts. Data science pipelines in astronomy, biology, neuroscience, chemistry and other emerging data-driven scientific fields, exhibit exactly those characteristics [88, 89]. Recognizing these needs, new systems with novel storage schemes appear continuously for targeted problems [28, 75, 82, 83, 96] and exploratory workloads [54, 92, 39, 45]. However, given the vast design space (we provide more intuition on that later), the likelihood that a single off-the-shelf data structure fits an evolving and complex scientific scenario with unpredictable and exploratory access patterns, is extremely small. We include a relevant quote from Efthimios Kaxiras, Prof. of Pure and Applied Physics at Harvard University: *"In chemistry and materials science, there exist a huge number of possible structures for a given chemical composition. Scientists are struggling to find ways to sort through these structures in an efficient way. The development of tailored database platforms would open great opportunities for new research in materials and molecular design."*

The Problem In Business. For both established companies and data-driven startups, the complexity leads to a slow design process and has severe cost side-effects [14, 18]. Time to market is of extreme importance, so data structure design stops when a design "is due" and only rarely when it "is ready". We include a quote from Mark Callahan, a data systems architect with more than two decades of experience: *"Getting a new data structure into production takes years. Assumptions made about the workload and hardware are likely to change. Decisions today are frequently based on expert opinions, and these experts are in short supply."*

The Problem In Clouds. In today's cloud-based world even slightly sub-optimal designs, e.g., by 1%, translate to a massive loss in energy utilization [61] for the cloud provider and cloud expenses for the user. This implies two trends. First, getting as close to the optimal design is as critical as ever. Second, the way a data structure design translates to cost needs to be embedded in the design process, i.e., being able to trade among

read, write, and memory as the relative costs of these resources change. Furthermore, cost policies can vary significantly among cloud providers which implies that for the same data, queries, and hardware, the optimal data structure can be different across different cloud providers. In sciences, for universities and labs that maintain their own cluster and thus are affected by energy costs, or use cloud infrastructure and thus are affected by operating costs, it is critical to manage those costs.

The Research Challenge. The long-term challenge is whether we can easily or even automatically find the optimal storage design for a given problem. This has been recognized as an open problem since the early days of computer science. In his seminal 1978 paper, Turing award winner Robert Tarjan includes this problem in his list of the five major challenges for the future (which also included P vs. NP) [90]: *"Is there a calculus of data structures by which one can choose the appropriate data representation and techniques for a given problem?"*. This is exactly the problem we attack. We identify the source of the problem to be that **there is currently no good way to predict the performance impact** of new workloads, hardware, and data structure designs; we need a full implementation and extensive testing. Similarly, **there is no good way to enumerate the possible designs so we can choose among them**. We make three critical observations.

1. Each data structure design can be described as a set of design concepts. These are all low-level design decisions, such as using partitioning, pointers, or direct addressing.
2. Each new data structure can be classified in two ways: it contains a) a new combination or tuning of existing design concepts, or b) at least one new design concept.
3. By now the research community has invented so many fundamental design concepts that most new designs are combinations or tunings of existing concepts.

Thesis: *If we knew the possible design space of data structures, i.e., the exhaustive set of fundamental design principles, and the performance properties we get when we synthesize them in arbitrary (valid) permutations, then we can create an engine that allows us to reason about every design, no matter how complex. The challenge then translates to being able to search the massive design space.*

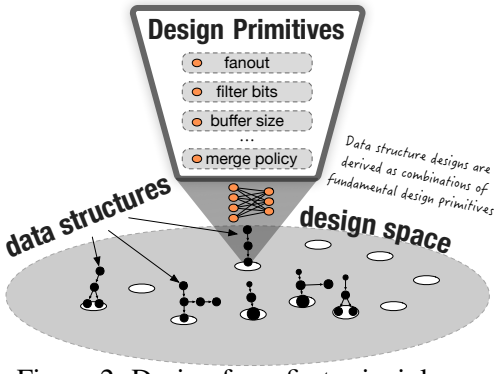

Figure 2: Design from first principles.

Data Structure Alchemy. We set out to discover the first principles of data structure design and develop algorithms to search through the massive design space they form. Our goal is to be able to reason about their combinations, tunings, and the performance properties they bring. Effectively, the principles and their structure form a "grammar" with which we can describe any data structure in a systematic way, including designs that have not been discovered yet. New designs can be "calculated" from the design space given constraints such as the expected workload, and hardware environment. If we focus on understanding and managing the core design elements, then new designs can be derived and existing designs can be transformed as if by magic - thus "alchemy".

The vision and components of the engine we propose are captured in Figure 2. We call this engine the Data Alchemist. Its functionality is to design a close-to-optimal data structure given a workload and hardware. The Data Alchemist takes four inputs: 1) the workload (queries and data) for which we would like to design an effective data structure, 2) a hardware profile where the data structure is expected to live, 3) optional performance and budget restrictions (e.g., point queries should be answered $\leq$ .3 seconds), and 4) optional constraints for the search algorithms such as acceptable distance to the estimated optimal, a cap for the search time, and an initial design to bootstrap the search. Then, the Data Alchemist outputs the resulting data structure design (abstract syntax tree) and its implementation in C++ code.

The architecture of the Data Alchemist gives us a clear path and methodology to make a dent in the decades-old problem of data structure design. Specifically, it boils down to the following contained challenges: 1)

identifying the fundamental design principles, 2) creating methods that can estimate the behavior of full designs that blend more than one design primitives, and 3) creating practical search algorithms that utilize the previous two components and input constraints to generate designs automatically. In the rest of this paper, we summarize our existing work towards the first two goals and we focus on sketching the main research challenges and describe ideas on how to approach the goal of automated design.

2 Design Space and Cost Calculation via Model Synthesis.

Blending Design Principles. The core idea is that the Data Alchemist contains a library of first principles which can be synthesized in arbitrary ways to give full data structure designs within a massive design space. We define the design of a data structure as the set of all decisions that characterize its data layout and algorithms, e.g., "Should data nodes be sorted?", "Should they use pointers?", and "How should we scan them exactly?". We define a first principle as a design concept that is not possible to break into more fine-grained concepts. For example, consider the design choice of linking two nodes of a data structure with a pointer. While there are potential tuning options (e.g., the size of the pointer), it is not possible to break this decision further; we either introduce a pointer or not. We separate design principles that have to do with the layout of a data structure from those that have to do with how we access the data. The bottom part of Figure 2 shows examples of data layout and access primitives for the key-value model. Overall, a core part of our effort is in analyzing how fine-grained data structure principles behave in terms of critical performance metrics: read, write, concurrency, and storage size. We build models for those behaviors, and then develop methods that synthesize more complex data structure designs by putting together the models of multiple fine-grained design principles.

We made a step toward towards this goal by introducing the **design space** of data structures supporting the key-value model [47]. The design space of data structures is defined by all designs that can be described as combinations and tunings of the "first principles of data layout design". As an analogy consider the periodic table of elements in chemistry; it sketched the design space of existing elements based on their fundamental components, and allowed researchers to predict the existence of unknown, at the time, elements and their properties, purely by the structure of the design space. In the same way, we created the **periodic table of data structures** [46] which describes more data structure designs than stars on the sky and can be used as a design discovery guide.

Naturally, a design space does not necessarily describe "all possible data structures"; a new design concept may be invented and cause an exponential increase in the number of possible designs. However, after 50 years of computer science research, the chances of inventing a fundamentally new design concept have decreased exponentially; many exciting innovations, in fact, come from utilizing a design concept that, while known, it was not explored in a given context before and thus it revolutionizes how to think about a problem. Using Bloom filters as a way to filter accesses in storage and remote machines, scheduling indexing construction actions lazily [41], using ML models to guide data access [63], storage [55] and other system components [62], can all be thought of as such examples. Design spaces that cover large fundamental sets of concepts can help accelerate progress with figuring out new promising directions, and when new concepts are invented they can help with figuring out the new possible derivative designs. For example, using models as part of the data structure design is an exciting recent idea, e.g., to replace the index layer or part of it [63, 62]. Such design options can become part of the design space for a more holistic design [43].

Algorithm and Cost Synthesis from Learned Cost Models. To fully utilize the knowledge of the vast design space we need to be able to compare different designs in terms of expected performance. Complexity analysis explains how the properties of a design scale with data but does not give a precise performance computation given a workload and hardware. On the other hand, full implementation and testing on the target workload and hardware provide the exact performance. The combination of both complexity analysis and implementation gives the complete picture. The problem is that with so many designs possible in the design space, it is not feasible to analyze, implement, and test the numerous valid candidates. A less explored option is building

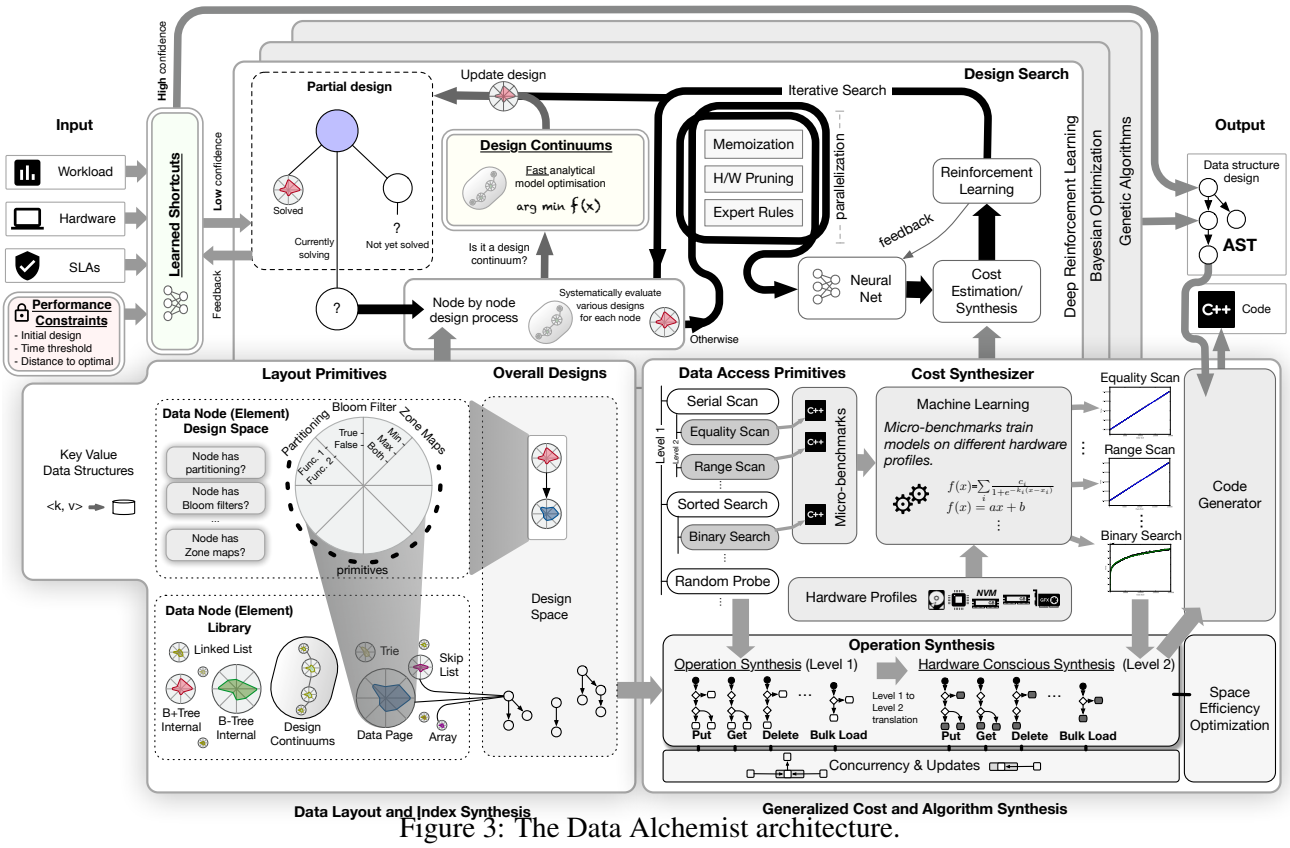

Figure 3: The Data Alchemist architecture.

generalized hardware-conscious cost models [70, 97]. Similar to a full implementation, such models provide results very close to ground truth, and similar to complexity analysis they provide hints on how performance scales. However, building generalized models is nearly equally hard to a full implementation given the level of detail and hardware-conscious properties needed [56]. In addition, any change in hardware properties requires retuning the model. Overall, arguing formally about the performance of diverse data structures designs, especially as workload and hardware properties change, is a notoriously hard problem [16, 56, 70, 91, 97, 99, 100]. In our case we want to perform this task for a massive number of candidate designs concurrently.

We have shown that given a data structure definition that describes its layout, it is possible to automatically design the data access algorithms and compute their cost by **synthesizing operations from fundamental access primitives** on blocks of data [47, 48]. There always exists a one-to-one mapping from a basic data layout to a high-level access pattern, e.g., if a data block is sorted we can utilize a sorted search. However, the performance that we would get out of a sorted search depends on 1) the exact algorithm (binary search, interpolation search, etc.), 2) the specific hardware, and 3) the exact engineering (e.g., using a new set of SIMD instructions). The Data Alchemist will synthesize the detailed algorithm and compute the cost via a hybrid of analytical models, implementation, and machine learning. We call these **learned cost models**. For each generic class of access pattern (e.g., sorted search) on a block of data there exist many instances of algorithms and implementations. These are short implementations that capture the exact access pattern. For example, for a binary search, the implementation consists of an array of sorted integers where we perform a series of searches. During a training phase, this code runs on the desired hardware and we learn a model as data scales (across the memory hierarchy). Each model captures the subtle performance details of diverse hardware settings and the exact engineering used to build the code for each access pattern. To make training easier, our models start as analytical models since we know how these access patterns will likely behave. Using learned cost models, it is possible to compute the accurate performance even in the presence of skew, capturing caching effects and other hardware and workload properties [47, 48]. The bottom right part of Figure 2 depicts examples of access principles and cost synthesis.

3 Learning to Design Data Structures

Once we know what performance we get when we blend two or more design principles, then the next big challenge is to design algorithms that can try out different combinations to reach a good result in terms of performance (e.g., response time, storage space, budget). The design space, however, is not enough. The problem is that the design space is truly vast. For example, only for the key-value model [47, 48] and without considering the extended design space needed for updates, concurrency control, and ML models, we estimate that the number of possible valid data structure designs explodes to $\gg 10^{32}$ even if we limit the overall design to only two different kinds of nodes (e.g., as is the case for B$^+$tree).

Thus, it is hard even for experts to "see" the optimal solutions even if we give them the design space. We need **search algorithms that navigate the possible design space to automatically design data structures** which are close to the best option (if not the best) given a desired workload and hardware. Using brute force and even dynamic programming variations leads to either impractical search times or we can only search through a small part of the design space. We study the properties of the design space as well as the properties of the applications using data structures to **turn the search process into a tractable problem in practice**. Our solutions lie in machine learning based algorithms that utilize model synthesis, hardware-conscious termination triggers, expert knowledge to lead the search algorithms in the right direction, and when possible closed-form models to quickly search within sub-spaces of the larger space. Our agenda involves designing four components:

1. **Design Continuums** that allow us to search fast within pockets of the design space.
2. **Performance Constraints** that provide application, user, and hardware based bounds.
3. **Learned Shortcuts** to accelerate a new search by learning from past results.
4. **Practical Search Algorithms** that utilize continuums, constraints, and shortcuts.

The overall search functionality is depicted at the top of Figure 2 and allows building and experimenting with many variations of search algorithms, constraints, and continuums. The high-level idea is that these algorithms treat the design space components of the Data Alchemist (bottom part of Figure 2) as a black box to try out different combinations of design principles. For each combination, the estimated performance feeds into the search algorithm's policy. Next we describe each one of four components in more detail to discuss the vision and preliminary results.

Design Continuums. A key ingredient in any algorithm is to induce domain-specific knowledge. Our insight is that there exist "design continuums" in the design space of data structures which can accelerate the search algorithms. An intuitive way to think of design continuums is as a performance hyperplane that connects a subset of data structure designs. It can be thought of as a super-structure that encapsulates all those designs by taking advantage of the notion that those designs are synthesized from the same set of fundamental principles.

A design continuum contains a set of rules to instantiate each one of its members and crucially a cost model with a single closed-form equation for each one of the core performance metrics (read, write, memory size) which applies across all member designs. In turn, closed-form models can be computed instantly to reason about this part of the design space. Thus we can search that part of the design space quickly to augment the search algorithms.

We introduced the first design continuum that connects a set of key-value structures [40]. Specifically, we have shown that it is possible to connect diverse designs including Tiered LSM-tree [50, 20, 21, 24, 64], Lazy Leveled LSM-tree [23], Leveled LSM-tree [73, 20, 27, 30], COLA [13, 52], FD-tree [66], B$^\epsilon$tree [15, 8, 13, 51, 52, 76], and B$^+$tree [12].

Our goal is to discover and formalize as many design continuums as possible and connect them to our search algorithms such that when a search algorithm "hits" a continuum, it can instantaneously get the best design within that space using the closed-form models as shown at the top part of Figure 2. One of the most exciting challenges here is to formalize design continuums that are based as much as possible on average case analysis as opposed to

worst case as in [40]. This requires building unifying models that take into account properties such as data and query distribution as well as the state of the target data structure in a workload that contains a sequence of steps.

Additional opportunities in creating "searchable" pockets of the design space include the use of integer solvers. This is for small parts of the design space which are potentially hard to formalize in a continuum but can be modeled as an integer problem. For example, consider partitioning of an array for optimal performance of a mix of point read, range read, update, delete, and insert operations. This is a small space that needs to be navigated by most data structures which contain blocks of data and partitioning can be a good option for each block. Each operation benefits or is hurt by partitioning in a different way, and so finding the right layout depending on the workload is a delicate balance. This problem can be mapped as a binary integer optimization problem and assigned directly to an integer solver. The Data Alchemist may contain numerous "searchable pockets" in the design space each one relying either on closed-form formulas or integer solvers.

Performance Constraints. The larger the number of continuums we manage to create, the more tractable the search process becomes. However, due to the complexity of the design space, we expect that we will be able to generate many small continuums rather than few big ones (in terms of the number of designs they contain). As such, given the astronomical size of the design space, search algorithms will intuitively still have a massive number of candidates to consider. To provide the next level of speed-up we are working towards performance constraints that bound search algorithms based on application, hardware and user context.

First, we observe that **the ultimate possible performance is limited by the underlying hardware** in any given scenario. For example, if an application or a researcher/engineer enter a query to the Data Alchemist to find a good data structure design on hardware H, then immediately we can consult the learned models which were trained on H for the best read, and write performance possible, e.g., reading or writing a single page from disk for a point read or write. Once a search algorithm finds a design within $k\%$ of these hardware imposed bounds, it can stop trying to improve as no further substantial improvements are possible. Parameter $k\%$ can be exposed as input as it is an application/user level decision.

Another performance constraint is to use a data structure design (full or partial) that the user suggests as starting point. This allows to induce expert and application knowledge in the search algorithm. The search process then can **auto-complete** the design without having to reconsider the initial decisions (this feature can also be used to detect a "bad" original design). Other examples of constraints include a time bound on the overall time a search algorithm should run, i.e., returning a best effort result once this time passes as well as returning top-k results which includes classes of promising design decisions it did not have time to consider. The top left and central part of Figure 2 shows examples of constraints and how they can be used as part of a search algorithm to accelerate the search.

Learned Shortcuts. Another critical component to speed up the search algorithms is learning from past results. We design and employ diverse neural network ensamples which are fed by an embedding generated using the input query, data, and hardware of every request and then it is "labeled" by the resulting output data structure design. Our goal is to create shortcuts through supervised learning that can practically instantly answer future queries without going through the search process, or alternatively the output of the neural network can work as a good starting point for a search. The top left part of Figure 2 shows how this fits in a search algorithm. Our work also includes accelerating the training and inference of neural network ensamples to achieve interactive speed and quick reaction to new workloads [93].

Algorithms. We design black box optimization algorithms that utilize continuums, constraints, and shortcuts to find a good design as fast as possible given a request. This includes Genetic Algorithms (GA) [44], Bayesian Optimization (BO), and Deep Reinforcement Learning (DRL). There are numerous challenges. For example, let us consider reinforcement learning. First, we must formalize the space of actions and rewards in data structure design. We experiment with both policy-based approaches that design the entire data structure at a time, and action-based approaches that design individual nodes at a time as well as hybrids. One approach is to model the design of a data structure as a multi-armed bandit problem. That is, we have a design space of design decisions that can be synthesized to a massive number of "design elements" as shown in Figure 2. Each element can be

seen as a bandit that can be chosen for any node of a data structure. Then, the problem is bounded by the number of different types of nodes we would like to include in the final data structure design. For example, the majority of the published key-value data structures consist of two node types, the index and the data, with Masstree [71] and Bounded Disorder [67] being exceptions that consist of three types of node. With the Data Alchemist we have the opportunity to explore designs with an arbitrary number of node types.

We expect that **no single algorithm will be a universal solution**. This is because of the core principles of how these families of algorithms behave in terms of convergence and response time. In the case of GA, exploration happens through mutation, exploitation is a result of the crossover and the selection process, whereas history is maintained implicitly through the surviving population. When it comes to BO, exploration happens by selecting solutions with high variance (i.e., solutions we are less certain about), exploitation, on the other hand, happens through selecting solutions with high mean (i.e., solutions we are quite sure about). History is maintained by a combination of the acquisition function and the probabilistic model. Finally, DRL makes conditional choices to explore the solution space, picks solutions with a high expected reward to exploit existing knowledge, and history is maintained in a deep neural network. As such, they all behave differently depending on the complexity of the problem at hand, desired span of the search through the design space, convergence speed, and the desired properties of the resulting data structure design (robustness vs. ultimate performance). The Data Alchemist incorporates several algorithms as shown at the top part of Figure 2 and choose the right one depending on the context.

Learned Models for Complex Patterns. The Data Alchemist constructs the cost of data structure designs out of fine-grained primitives for which it knows learned models. However, a complex design inevitably loses accuracy when synthesized out of many models. A solution is to generate the code for sets of design primitives and learn a single compound model for all of them via on-the-fly experiments during the search algorithm (there are too many possible compound models to train for all of them a priori). Results can also be cached in the learned models library for future use. Such compound models can increase the accuracy of the search algorithms, leading to better data structure designs. We build a compiler to generate this code by directly utilizing the fact that we already have the code of the individual learned models in the library. This compiler can also be used to output starter code for the resulting design of a search using the abstract syntax tree of the design and the code of the learned models that were chosen by the search algorithm. This makes it easier to adopt, extend, and fine-tune designs.

4 Inspiration

Our work is inspired by numerous efforts that also use first principles and clean abstractions to understand a complex design space. John Ousterhout's project Magic allows for quick verification of transistor designs so that engineers can easily test multiple designs synthesized by basic concepts [74]. Leland Wilkinson's "grammar of graphics" provides structure and formulation on the massive universe of possible graphics [95]. Timothy G. Mattson's work creates a language of design patterns for parallel algorithms [72]. Mike Franklin's Ph.D. thesis explores the possible client-server architecture designs using caching based replication as the main design primitive [29]. Joe Hellerstein's work on Generalized Search Trees makes it easy to design and test new data structures by providing templates which expose only a few options where designs need to differ [37, 6, 7, 58, 57, 59, 60]. S. Bing Yao's [97] and Stefan Manegold's [70] work on generalized hardware conscious cost models showed that it is possible to synthesize the costs of complex operations from basic access patterns. Work on data representation synthesis in programming languages enables synthesis of representations out of small sets of (3-5) existing data structures [79, 80, 19, 86, 84, 35, 34, 69, 87]. Work on tuning [49, 17] and adaptive systems is also relevant as conceptually any adaptive technique tunes along part of the design space. For example, work on hybrid data layouts and adaptive indexing automates selection of the right layout [9, 3, 33, 41, 26, 81, 4, 68, 20, 78, 32, 77, 101, 42, 53, 85]. Similarly works on tuning via experiments [11],

learning [6], and tuning via machine learning [2, 36] can adapt parts of a design using feedback from tests.

5 Summary and Future Steps

We describe the path toward automating data structure invention and design from first principles and AI. The secret sauce is in finding the first principles of design, mapping the design space that they form, being able to reason about the expected performance of designs, and finally building practical AI algorithms that can navigate this space to design new data structures. Searching the whole massive space is not likely possible so the key is in translating as much design and application knowledge into the search algorithms. Our ongoing efforts include applying this same methodology of first principles and AI beyond traditional key-value structures, focusing on forming the design space of statistics computation [94], neural networks [93], and sketches [38].

References

[1] D. J. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[2] D. V. Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. ACM SIGMOD, 2017.

[3] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A Hands-free Adaptive Store. ACM SIGMOD, 2014.

[4] V. Alvarez, F. M. Schuhknecht, J. Dittrich, and S. Richter. Main Memory Adaptive Indexing for Multi-Core Systems. DAMON, 2014.

[5] M. R. Anderson, D. Antenucci, V. Bittorf, M. Burgess, M. J. Cafarella, A. Kumar, F. Niu, Y. Park, C. Ré, and C. Zhang. Brainwash: A Data System for Feature Engineering. CIDR, 2013.

[6] P. M. Aoki. Generalizing "Search" in Generalized Search Trees (Extended Abstract). IEEE ICDE, 1998.

[7] P. M. Aoki. How to Avoid Building DataBlades That Know the Value of Everything and the Cost of Nothing. SSDBM, 1999.

[8] L. Arge. The Buffer Tree: A Technique for Designing Batched External Data Structures. *Algorithmica*, 2003.

[9] J. Arulraj, A. Pavlo, and P. Menon. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. ACM SIGMOD, 2016.

[10] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing Access Methods: The RUM Conjecture. EDBT, 2016.

[11] S. Babu, N. Borisov, S. Duan, H. Herodotou, and V. Thummala. Automated Experiment-Driven Management of (Database) Systems. HotOS, 2009.

[12] R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indexes. ACM SIGFIDET Workshop on Data Description and Access, 1970.

[13] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-Oblivious Streaming B-trees. SPAA, 2007.

[14] P. A. Bernstein and D. B. Lomet. CASE Requirements for Extensible Database Systems. *IEEE Data Engineering Bulletin*, 10(2):2–9, 1987.

[15] G. S. Brodal and R. Fagerberg. Lower Bounds for External Memory Dictionaries. SODA, 2003.

[16] A. F. Cardenas. Evaluation and Selection of File Organization - a Model and System. *Communications of the ACM*, (9):540–548, 1973.

[17] S. Chaudhuri, V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. VLDB'97.

[18] A. Cheung. Towards Generating Application-Specific Data Management Systems. CIDR, 2015.

[19] D. Cohen and N. Campbell. Automating Relational Operations on Data Structures. *IEEE Software*, 1993.

[20] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal Navigable Key-Value Store. ACM SIGMOD, 2017.

[21] N. Dayan, M. Athanassoulis, S. Idreos. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. TODS'18.

[22] N. Dayan, P. Bonnet, and S. Idreos. GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices. ACM SIGMOD, 2016.

[23] N. Dayan and S. Idreos. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. ACM SIGMOD, 2018.

[24] N. Dayan and S. Idreos. The log-structured merge-bush & the wacky continuum. ACM SIGMOD, 2019.

[25] DBLP. Computer Science Bibliography. *https://dblp.uni-trier.de*, 2019.

[26] J. Dittrich and A. Jindal. Towards a One Size Fits All Database Architecture. CIDR, 2011.

[27] Facebook. RocksDB. *https://github.com/facebook/rocksdb*.

[28] R. C. Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker. Aurum: A data discovery system. IEEE ICDE, 2018.

[29] M. J. Franklin. *Caching and Memory Management in Client-Server Database Systems*. PhD thesis, University of Wisconsin-Madison, 1993.

[30] Google. LevelDB. *https://github.com/google/leveldb/*.

[31] G. Graefe. Modern B-Tree Techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.

[32] G. Graefe, F. Halim, S. Idreos, H. Kuno, S. Manegold. Concurrency control for adaptive indexing. PVLDB'12.

[33] R. A. Hankins and J. M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. VLDB, 2003.

[34] P. Hawkins, A. Aiken, K. Fisher, M. C. Rinard, and M. Sagiv. Concurrent data representation synthesis. PLDI, 2011.

[35] P. Hawkins, A. Aiken, K. Fisher, M. C. Rinard, and M. Sagiv. Data Representation Synthesis. PLDI, 2011.

[36] M. Heimel, M. Kiefer, and V. Markl. Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation. ACM SIGMOD, 2015.

[37] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. VLDB, 1995.

[38] B. Hentschel, M. S. Kester, and S. Idreos. Column Sketches: A Scan Accelerator for Rapid and Robust Predicate Evaluation. ACM SIGMOD, 2018.

[39] S. Idreos. Big Data Exploration. In *Big Data Computing*. Taylor and Francis, 2013.

[40] 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. CIDR, 2019.

[41] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. CIDR, 2007.

[42] S. Idreos, M. L. Kersten, S. Manegold. Self-organizing Tuple Reconstruction in Column-Stores. ACM SIGMOD'09.

[43] S. Idreos and T. Kraska. From auto-tuning one size fits all to self-designed and learned data-intensive systems. ACM SIGMOD, 2019.

[44] S. Idreos, L. M. Maas, and M. S. Kester. Evolutionary Data Systems. *CoRR*, abs/1706.0, 2017.

[45] S. Idreos, O. Papaemmanouil, and S. Chaudhuri. Overview of Data Exploration Techniques. ACM SIGMOD, 2015.

[46] S. Idreos, K. Zoumpatianos, M. Athanassoulis, N. Dayan, B. Hentschel, M. S. Kester, D. Guo, L. M. Maas, W. Qin, A. Wasay, and Y. Sun. The Periodic Table of Data Structures. *IEEE Data Engineering Bulletin*, 41(3):64–75, 2018.

[47] S. Idreos, K. Zoumpatianos, B. Hentschel, M. S. Kester, and D. Guo. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. ACM SIGMOD, 2018.

[48] S. Idreos, K. Zoumpatianos, B. Hentschel, M. S. Kester, and D. Guo. The Internals of The Data Calculator. *CoRR*, abs/1808.02066, 2018.

[49] Y. E. Ioannidis and E. Wong. Query Optimization by Simulated Annealing. ACM SIGMOD, 1987.

[50] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental Organization for Data Recording and Warehousing. VLDB, 1997.

[51] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. BetrFS: A Right-optimized Write-optimized File System. FAST, 2015.

[52] C. Jermaine, E. Omiecinski, and W. G. Yee. The Partitioned Exponential File for Database Storage Management. *The VLDB Journal*, 16(4):417–437, 2007.

[53] O. Kennedy and L. Ziarek. Just-In-Time Data Structures. CIDR, 2015.

[54] M. L. Kersten, S. Idreos, S. Manegold, and E. Liarou. The Researcher's Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds. PVLDB, 2011.

[55] M. L. Kersten and L. Sidirourgos. A database system with amnesia. CIDR, 2017.

[56] M. S. Kester, M. Athanassoulis, and S. Idreos. Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe? ACM SIGMOD, 2017.

[57] M. Kornacker. High-Performance Extensible Indexing. VLDB, 1999.

[58] M. Kornacker, C. Mohan, and J. M. Hellerstein. Concurrency and Recovery in Generalized Search Trees. ACM SIGMOD, 1997.

[59] M. Kornacker, M. A. Shah, J. M. Hellerstein. amdb: An Access Method Debugging Tool. ACM SIGMOD'98.

[60] M. Kornacker, M. A. Shah, and J. M. Hellerstein. Amdb: A Design Tool for Access Methods. *IEEE Data Engineering Bulletin*, 26(2):3–11, 2003.

[61] D. Kossman. Systems Research - Fueling Future Disruptions. In *Keynote talk at the Microsoft Research Faculty Summit*, Redmond, WA, USA, aug 2018.

[62] T. Kraska, M. Alizadeh, A. Beutel, E. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. Sagedb: A learned database system. CIDR, 2019.

[63] T. Kraska, A. Beutel, E. H. Chi, J. Dean, N. Polyzotis. The Case for Learned Index Structures. ACM SIGMOD'18.

[64] A. Lakshman and P. Malik. Cassandra - A Decentralized Structured Storage System. ACM SIGOPS, 2010.

[65] T. J. Lehman and M. J. Carey. A Study of Index Structures for Main Memory Database Management Systems. VLDB, 1986.

[66] Y. Li, B. He, J. Yang, Q. Luo, K. Yi, and R. J. Yang. Tree Indexing on Solid State Drives. PVLDB, 2010.

[67] W. Litwin and D. B. Lomet. The Bounded Disorder Access Method. IEEE ICDE, 1986.

[68] Z. Liu and S. Idreos. Main Memory Adaptive Denormalization. ACM SIGMOD, 2016.

[69] C. Loncaric, E. Torlak, and M. D. Ernst. Fast Synthesis of Fast Collections. PLDI, 2016.

[70] S. Manegold, P. A. Boncz, and M. L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. VLDB, 2002.

[71] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-value Storage. EuroSys, 2012.

[72] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 2004.

[73] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[74] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, G. S. Taylor. Magic: A VLSI Layout System. DAC'84.

[75] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson. The TileDB Array Data Storage Manager. PVLDB, 2016.

[76] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. USENIX ATC, 2016.

[77] E. Petraki, S. Idreos, and S. Manegold. Holistic Indexing in Main-memory Column-stores. ACM SIGMOD, 2015.

[78] H. Pirk, E. Petraki, S. Idreos, S. Manegold, and M. L. Kersten. Database cracking: fancy scan, not poor man's sort! In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, pages 1–8, 2014.

[79] E. Schonberg, J. T. Schwartz, and M. Sharir. Automatic data structure selection in setl. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 197–210, 1979.

[80] E. Schonberg, J. T. Schwartz, and M. Sharir. An automatic technique for selection of data representations in setl programs. *ACM Trans. Program. Lang. Syst.*, 3(2):126–143, apr 1981.

[81] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The Uncracked Pieces in Database Cracking. PVLDB, 2013.

[82] SciDB. SciDB-Py. *http://scidb-py.readthedocs.io/en/stable/*, 2016.

[83] A. Seering, P. Cudré-Mauroux, S. Madden, and M. Stonebraker. Efficient versioning for scientific array databases. IEEE ICDE, 2012.

[84] O. Shacham, M. T. Vechev, and E. Yahav. Chameleon: Adaptive Selection of Collections. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 408–418, 2009.

[85] D. D. Sleator and R. E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, 1985.

[86] Y. Smaragdakis and D. S. Batory. DiSTiL: A Transformation Library for Data Structures. USENIX DSL, 1997.

[87] M. J. Steindorfer and J. J. Vinju. Towards a Software Product Line of Trie-Based Collections. ACM SIGPLAN GPCE, 2016.

[88] M. Stonebraker, A. Ailamaki, J. Kepner, and A. S. Szalay. The future of scientific data bases. IEEE ICDE, 2012.

[89] M. Stonebraker, J. Duggan, L. Battle, and O. Papaemmanouil. Scidb DBMS research at M.I.T. *IEEE Data Eng. Bull.*, 36(4):21–30, 2013.

[90] R. E. Tarjan. Complexity of Combinatorial Algorithms. *SIAM Review*, 20(3):457–491, 1978.

[91] T. J. Teorey, K. S. Das. Application of an Analytical Model to Evaluate Storage Structures. ACM SIGMOD'76.

[92] A. Wasay, M. Athanassoulis, and S. Idreos. Queriosity: Automated Data Exploration. IEEE International Congress on Big Data, 2015.

[93] A. Wasay, Y. Liao, and S. Idreos. Rapid training of very large ensembles of diverse neural networks. *CoRR*, abs/1809.04270, 2018.

[94] A. Wasay, X. Wei, N. Dayan, and S. Idreos. Data Canopy: Accelerating Exploratory Statistical Analysis. ACM SIGMOD, 2017.

[95] L. Wilkinson. *The Grammar of Graphics*. Springer-Verlag, 2005.

[96] H. Xing, S. Floratos, S. Blanas, S. Byna, Prabhat, K. Wu, and P. Brown. ArrayBridge: Interweaving Declarative Array Processing in SciDB with Imperative HDF5-Based Programs. IEEE ICDE, 2018.

[97] S. B. Yao. An Attribute Based Model for Database Access Cost Analysis. TODS, 1977.

[98] S. B. Yao and D. DeJong. Evaluation of Database Access Paths. ACM SIGMOD, 1978.

[99] S. B. Yao and A. G. Merten. Selection of File Organization Using an Analytic Model. VLDB, 1975.

[100] M. Zhou. *Generalizing Database Access Methods*. PhD thesis, University of Waterloo, 1999.

[101] K. Zoumpatianos, S. Idreos, T. Palpanas. Indexing for interactive exploration of big data series. ACM SIGMOD'14.

A Human-in-the-loop Perspective on AutoML: Milestones and the Road Ahead

Doris Jung-Lin Lee[†*], Stephen Macke[‡*], Doris Xin[†*], Angela Lee[‡], Silu Huang[‡], Aditya Parameswaran[†]
{dorislee,dorx,adityagp}@berkeley.edu | {smacke,alee107,shuang86}@illinois.edu
[†]University of California, Berkeley | [‡]University of Illinois, Urbana-Champaign | [*]Equal Contribution

1 Introduction

Machine learning (ML) has gained widespread adoption in a variety of real-world problem domains, ranging from business, to healthcare, to agriculture. However, the development of effective ML solutions requires highly-specialized experts well-versed in both statistics and programming. This high barrier-of-entry stems from the current process of crafting a customized ML solution, which often involves numerous manual iterative changes to the ML workflow, guided by knowledge or intuition of how those changes impact eventual performance. This cumbersome process is a major pain point for machine learning practitioners [4, 53] and has motivated our prior work on Helix, a declarative ML framework [52] targeted at supporting efficient iteration.

To make ML more accessible and effortless, there has been recent interest in AutoML systems, both in industry [2, 1, 21] and in academia [15, 37], that automatically search over a predefined space of ML models for some high-level goal, such as prediction of a target variable. For certain tasks, these systems have been shown to generate models with comparable or better performance than those generated by human ML experts in the same time [35, 26]. However, our preliminary study of ML workflows on OpenML [48] (an online platform for experimenting with and sharing ML workflows and results) shows that AutoML is not widely adopted in practice—accounting for fewer than 2% of all users and workflows. While this may be due to a lack of awareness of these tools, we believe that this sparse usage stems from a more fundamental issue: *a lack of usability*.

Our main observation is that the fully-automated setting that current AutoML systems operate on may not be a one-size-fits-all solution for many users and problem domains. Recent work echoes our sentiment that AutoML's complete automation over model choices may be inadequate in certain problem contexts [18, 50]. The lack of human control and interpretability is particularly problematic when the user's domain knowledge may influence the choice of workflow [18], in high-stakes decision-making scenarios where trust and transparency are essential [50], and in exploratory situations where the problem is not well-defined [11]. This trade-off between control and automation has been a century-long debate in HCI [23, 22, 44, 5], with modern reincarnations arising in conversational agents, interactive visual analytics, and autonomous driving. A common interaction paradigm to reconcile these two approaches is a *mixed-initiative* approach, where "intelligent services and users...collaborate efficiently to achieve the user's goals" [23].

Along the footsteps of these seminal papers, here, we outline our vision for a Mixed-Initiative machine Learning Environment (MILE), by rethinking the role that automation and human supervision play across the ML development lifecycle. MILE enables a better user experience, and benefits from system optimizations that both leverage human input and are tailored to the fact that MILE interacts with a human in the loop. For example, our earlier work HELIX [52] leveraged the fact that workflow development happens iteratively, to intelligently

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

materialize and reuse intermediate data products to speed up subsequent iterations. Similarly, as discussed later in this paper, leveraging user domain knowledge has the potential to drastically narrow down the exhaustive search space typically employed by existing AutoML systems.

By considering the trade-offs between system acceleration and user control, we organize our paper based on three increasing levels of autonomy—user-driven, cruise-control, and autopilot—drawing an analogy with driving. The different levels of autonomy characterize the degree of user control and specification of problem requirements versus the amount of system automation in identifying desired workflows (as illustrated in Figure 1). Starting from the manual, user-driven setting (§2), we describe system challenges in enabling users to rapidly try out different workflow variants, including techniques that speed up execution time to achieve interactive responses, and those that improve debugging and understanding of different workflow versions. Next, in the cruise-control setting (§3), the system works alongside users collaboratively in search of a workflow that fits the user's needs, by letting users declaratively specifying problem requirements, and identifying desired workflows via a dialog with the user. Finally, in the fully-autonomous, autopilot setting (§4), we outline several techniques that would improve and accelerate the search through different ML design decisions. At a high level, these techniques hinge on *accelerated search* via AutoML-aware work-sharing optimizations and *more intelligent search* via knowledge captured from user-driven ML workflows. Therefore, a holistic system for varying levels of autonomy is crucial.

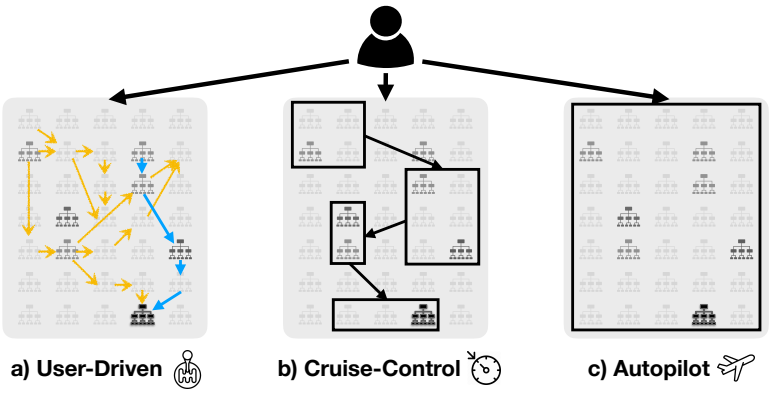

Figure 1: Three levels of autonomy in the ML lifecycle. The gray box represents the space of all possible workflows for a given task. Darker workflows have better performance, and the darkest workflow at the bottom is the desired workflow. a) User-driven: the user has to specify the next workflow to explore in every iteration; a novice user (yellow arrows) might take more iterations than an expert (blue arrows) to reach a "good" workflow. b) Cruise-control: the user steers the system towards a set of changes (a black box) to be explored automatically by the system. c) Autopilot: the user specifies only the dataset and the ML objective for the machine to automatically find the optimal workflow.

Our goal for characterizing the design space of such systems into the three representative levels is to bridge the knowledge gap between novice and expert users and thereby democratize the process of ML development to a wider range of end-users. For example, to build an image classifier, an expert user might want to explicitly choose which model and preprocessing techniques to use, but leaving the manual search of the hyperparameter settings to the system, whereas a novice might opt for the fully-autonomous setting to search for any optimal workflow. By addressing the research challenges in each level of autonomy, we can envision an intelligent, adaptive, multi-tiered system that dynamically adjusts to the appropriate balance between usability and customizability depending on the amount of information present in the user input. In addition, by supporting different levels of autonomy in a single system, the system can synthesize knowledge from expert users (such as knowing that a convolutional neural network would be most suitable for building an image classifier) to help the non-experts.

Across the different levels of autonomy, we encounter research challenges from the fields of ML, databases (DB), and HCI. The ML challenges include meta-learning techniques for intelligently traversing the search space of models. The database challenges include optimization of time and resources required for this search process leveraging common DB optimizations such as pipelining, materialization, and reuse. The HCI challenges include designing abstractions that make it easy to communicate high-level problem requirements to the system, as well as providing interpretable system outputs and suggestions. The challenges from these three fields are not isolated

but instead impact each other. For example, developing DB optimizations that speed up execution time also leads to a more responsive and interactive user experience. Creating more usable debugging tools also improves model transparency and interpretability. For these reasons, in order to design a holistic solution, it is crucial to work at the intersection of these fields to tackle the challenges across the different levels of autonomy.

In our envisioned Mixed-Initiative machine Learning Environment (MILE), users work collaboratively with machines in search for an optimal workflow that accomplishes the specified problem goals. As illustrated in Figure 2, MILE is reminiscent of a relational database management system (DBMS). At the top, different application units and users can communicate with the system via a declarative ML intent query language called MILEAGE (described in §3). Users do not have to use this query language directly; instead, we envision interactive interfaces that can generate such queries automatically based on user interactions (analogous to form-based interfaces generating SQL), as in our Helix IDE [51] or TensorBoard [36]. Given a MILEAGE query, the optimizer represents and traverses through the search space defined by the MILEAGE query. The output of the optimizer is a workflow, akin to a query execution plan, where the workflow's overall performance is modeled as a cost function. Similar to how operators can be reordered and pushed down to achieve the same result while optimizing for performance in an DBMS, there may be certain ML workflow choices that have equivalent results, but leads to better performance. The obtained workflow is provided to an execution engine that performs the actions specified by the workflow, leading to a set of ML results (which can be labels for classification or a table of prediction values). Finally, the ML results are communicated back to the users for debugging and analysis.

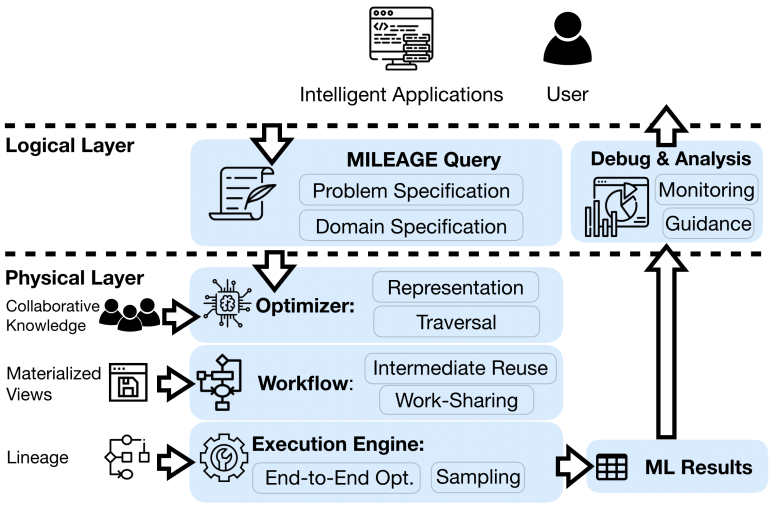

Figure 2: MILE System Overview

2 User-Driven

In this setting, the user makes all the modeling decisions, including the initial workflow and iterative changes to the workflow to evaluate in subsequent iterations. The role of the system is to help users rapidly explore the specified changes and to track the results across iterations. This setting affords the users full control of the modeling process, which may be desirable for several reasons. For example, the application has stringent security, privacy, or regulatory requirements; or the dataset is extremely large, limiting the number of experiments that can be run. However, since MILE provides no input on modeling decisions, this setting is more suited for expert ML users who demand full control of the modeling process. Even though expert ML users dictate the modeling iterations, there are many challenges associated with focusing their attention on modeling and maximizing their productivity. Below, we discuss concrete challenges in accelerating execution to increase interactivity and in helping users interpret and debug ML outputs.

2.1 Interactivity

In each iteration, users rely on the results from the current and past models to make decisions about what to try next, such as a new set of features, a different model hyperparameter value, or a new model type. Shortening the execution time to obtain the model metrics and predictions would greatly improve interactivity and can be

accomplished through several different system optimizations.

Materialization and reuse. From our preliminary study of ML workflows from OpenML, we observed that from one iteration to the next, users tend to reuse a large portion of the workflow. About 80% of the iterations on OpenML reuse over half of the operators from the previous iteration. Incremental changes, while desirable for controlled experiments with quantifiable impact, result in a great deal of redundant computation that can be materialized and reused in subsequent iterations to speed up iterative execution.

While materialization and reuse techniques in databases are well-studied, applying them to ML workflows presents new challenges. First, naïve solutions such as materializing all operator results can be wasteful and detrimental to performance, and reuse is not always the optimal policy. In some cases, recomputing from inputs can be more efficient than loading previous results. Interdependency of materialization and reuse decisions in a workflow DAG complicates the storage and runtime tradeoff. For example, materializing all of the descendants of an operator O precludes the need for materializing O, but O needs to be loaded in a subsequent iteration if any of the descendants are modified. Additionally, users can make arbitrary changes in each iteration. Making effective materialization decisions hinges upon the ability to anticipate iterative changes.

HELIX investigates some of these challenges and provides heuristics for solving the problem. The materialization problem is proven to be NP-hard [52]. As a next step, building a predictive model of what users may do next may help prioritize what to materialize, given enough training data on how developers iterate on ML workflows. OpenML is a great source for gathering such training data, as it records complete workflow traces. However, a solution based purely on historical data may not respond adequately to the idiosyncrasies of a specific user—the system must also be able to adapt quickly to the behaviors of a new user. To this end, reinforcement learning (RL) can be used to explore complex materialization strategies, as others have done for query optimization [30] and view maintenance [34].

End-to-end optimization. While materialization and reuse optimize across iterations, end-to-end optimization focuses on the entire workflow in a single iteration. An important area that merits more research efforts is the joint optimization of the data-preprocessing component, (primarily relational operators) and the ML component (primarily linear algebra–or LA–operators) of the workflow. Relational operators deal in columns and rows, while LA operators deal in vectors and matrices. Although there is an intuitive correspondence between the columns and rows of a table and the columns and rows of a matrix, systems aimed at integrating ML into databases usually do so via a special vector data type for performance reasons (see [31] for a survey). Chen et. al. propose a formal framework to unify LA and relational operators in the context of factorized ML, in which ML operators are decomposed and pushed through joins [12]. Their framework can be extended to support a wider range of cross-relational and LA optimizations. For example, Sommer et. al. observe that sparsity arises from many sources in ML and can be exploited to optimize the execution plan for LA operators [45]. If we were able to connect each vector position to the corresponding columns between LA and relational operators, we can leverage sparsity to optimize the relational portion of the workflow as well, e.g., automatically dropping, at the earliest opportunity, columns that correspond to zero-weight features in the ML model. Extending the framework in Chen [12] to support the sparsity optimization requires tracking additional column-wise provenance.

An orthogonal direction is to use approximation computing techniques such as sampling and quantization to speed up end-to-end execution for rapid feedback to users regarding the quality of the current workflow. For sampling, one needs to ensure that the sample is representative of the underlying dataset and that the same sample is used throughout the workflow, ideally without modifying existing operators. Quantization pertains to using imprecise weights represented by fewer bits to obtain model performance comparable with the full-precision version [25].

2.2 Interpretability and Debuggability

Another important aspect of helping users make modeling decisions is assisting in the analysis of the *artifacts* involved in the modeling process, including input data, intermediate results of operators within the workflow,

models, and outputs [17]. Deciding on effective iterative changes requires a thorough understanding of the behavior of the current model. To make sense of existing models, users might need to examine a number of artifacts and the relationships between them. For example, to debug a bad prediction, the user might look at the model and the most relevant training examples that led to the prediction. This process requires tracking a combination of coarse-grained and fine-grained lineage across related artifacts.

Coarse-grained. Artifacts in this category include workflow versions and the metadata associated with each version, such as model hyperparameters and performance metrics. A number of systems have been developed to facilitate the recording of iterative workflow versions and metadata [49, 54, 17]. These systems enable tracking via user-specified logging statements injected into their workflow programs. The goal is to be minimally intrusive and lightweight—the system does not need to provide special implementations for each existing operator since the user specifies the metrics of interest explicitly. However, a more automatic solution to log metadata can leverage a combination of data lineage tracking and program analysis. Schelter et. al. propose a declarative language for users to specify logging at the operator and workflow level instead of by individual metrics [41]. We envision taking this one step further and completely automating the tracking of metadata, by injecting logging statements inside the compiler via program analysis. With the ever-growing body of ML frameworks, it is not scalable to implement solutions specific to each framework. Instead, we should focus on common model representations, such as PMML[1] and ONNX[2], that are easily portable across frameworks, akin to how LLVM handles many different languages with a unified intermediate representation (IR) for analysis. The model IRs have unified representations of operator parameters and input/output types, as well as mechanisms for users to annotate operators, which can be leveraged to specify custom logging requirements.

Fine-grained. As mentioned previously, fine-grained data lineage is helpful for diagnosing model issues, e.g., tracing back to the input data that led to a particular bad prediction. Supporting fine-grained data lineage in data-intensive, ad-hoc analysis is challenging for several reasons: 1) the amount of possible lineage data to track is often combinatorial with respect to the raw data; 2) the workloads are unpredictable; and 3) query results need to be served in real-time. The common technique to address 3) is to leverage results precomputed offline, but 2) makes it difficult to predict what precomputation would be beneficial and 1) makes it infeasible to precompute all possible results in the absence of predictability. We identify three promising directions that can effectively address all three confounding challenges.

During debugging, users create a great deal of *ephemeral knowledge* that could potentially help with future debugging but is currently wasted. For example, a user runs a query to find outlier values for a given column. The next time someone else needs to debug an application using data from the same column, the outliers could potentially help explain anomalous behaviors, or the system could recommend outliers as a diagnostic for a different column for related issues. Aghajanyan et. al. [3] propose a system for capturing insights from these exploratory workloads to assist with future debugging. Doing so not only reduces redundant computation but also makes future workloads more regular by guiding user exploration with predefined operations. Consequently, the system can selectively store a small subset of fine-grained lineage most likely to accelerate user-interactivity during debugging in a cost-effective manner. Research in systematizing ad-hoc workloads into knowledge that assists with future debugging is still nascent and warrants further investigation.

Even with selective storage, the amount of fine-grained lineage data is still potentially huge and requires carefully designed storage schemes. In HELIX, we have begun to explore storing lineage in the materialization of workflow intermediates. The idea is that while it is prohibitive to store the fine-grained input-output relationship for every single operator in the workflow, we can selectively store only the output of expensive operators and replay the cheap operators on top of materialized intermediates to recover the full lineage. For serving fine-grained lineage in real-time, SMOKE [40] is an in-memory database for serving fine-grained lineage for relations operators at interactive speed, using heuristics to solve the problem of materialization and replay explored in

[1] http://dmg.org/pmml/v4-3/GeneralStructure.html

[2] https://onnx.ai/

HELIX. Whether SMOKE or the techniques within can be generalized for ML operators and to dataset that do not fit in memory posits interesting research challenges.

3 Cruise-Control

Unfortunately, the fully user-driven setting is the modus operandi for ML application development supported by the majority of existing tools, irrespective of user expertise. In this section and the next, we explore system designs to help change the landscape of ML tooling, making it more accessible to a wider range of users.

In the cruise-control setting, the user specifies their problem requirements and any domain-specific knowledge to MILE. MILE then automatically searches through the space of potential workflow options and recommends an acceptable workflow that meets the specification, via a dialog with the user. Since the technology for recommending acceptable/accurate models overlaps heavily with that the model search capabilities in the autonomous setting (§4), here, we focus our discussion on the challenges associated with designing the appropriate logical representation of the model space that the end-user interacts with. This logical representation abstracts away the underlying details of how the search and inference are performed, so that changes or additions of new search strategies and models would not affect end-user experience. As in Figure 2 and described in this section, the logical representation consists of two components to facilitate a dialog between user and system (akin to a dashboard). From user to system, we first describe a language that enables users to express their ML 'intent'. Then, going from system to user, we discuss interfaces that communicate system outputs to the user.

3.1 Declarative Specification of ML Intent

Unlike traditional programming where the problem solution is deterministic, since ML is data-dependent, even an expert will only have a vague impression of what their optimal workflow would look like, based on their desired properties for the eventual model outputs. However, users often have some preferences, constraints, requirements, and knowledge regarding the problem context (collectively referred to as *intent*) constraining the space of potential model solutions. We will first describe two characteristics of ML intents (ambiguity and multi-granularity) that presents research challenges in operationalization. We illustrate these characteristics via a hypothetical example of a user developing a cancer diagnostic model based on clinical data. Next, we describe our proposed solution strategy in developing a declarative language, MILEAGE, that enable users to specify their ML intent. While prior work has proposed declarative specification of ML tasks [29] and feature engineering processes as UDFs [6], these endeavors have been focused on a specific aspect of the ML lifecycle. We argue for a holistic view that enables users to express a wide range of high-level, nuanced ML intents to the system.

Ambiguous Intents. Our first challenge is that ML intents can often be *ambiguous*—in other words, high-level, qualitative problem requirements are ambiguous and often do not translate exactly to a low-level workflow change (e.g., hyperparameter setting, preprocessing choice). For instance, in the cancer diagnostic example, the ML developer might indicate that the desired model should be interpretable to physicians and patients alike. In addition, since records are transcribed by human clinicians, the model must be robust to noisy and potentially erroneous data inputs, as well as missing values. Another reason why ML intents can be ambiguous is that problem requirements often stem from 'fuzzy' domain-specific knowledge. For example, an oncologist may indicate that because lung cancer is more fatal (higher mortality rates) than other types of cancer, false negatives should be penalized more heavily in the lung cancer diagnosis model. There are many potential approaches to operationalize these and other similar problem requirements, including modifying regularization, developing a performance metric beyond traditional classification accuracy, or choosing a model that is robust to class imbalance. The challenge therefore lies in understanding how can we can map these ambiguous high-level problem requirements to suggest some combination of workflow changes.

Multi-Granularity Intents. Another challenge in designing an ML intent language is that user requests are *multi-granularity*, encompassing a variety of different input types at different levels of granularity. At the highest level, a user can specify problem requirements, goals, or domain knowledge; at an intermediary level, users can refine or prune out portions of the search space (ranges of parameter options), at the lowest level, users can fix specific preprocessing procedures or hyperparameter values (similar to what one would do in the user-driven setting). The multi-granularity nature of ML intent stems from users with different expertise levels. For example, a clinician might only be able to specify the desired task goal (e.g., predict attribute 'mortality rate'), whereas an ML expert might have more specific model requirements (e.g., use a random forest with 10-50 trees, max tree depth of 4-8, with entropy as split criteria). The research challenge lies in developing a specification framework that can adapt and jointly perform inference based on signals from different granularities, as well as appropriate interfaces to elicit different input requirements from users. Both of these challenges demand a more holistic understanding of the interdependencies and impact of different ML design choices, and how they affect the resulting workflow characteristics.

MILEAGE **Improvements.** Our proposed solution is to develop MILEAGE, a declarative specification language that allows users to communicate their ML intent to the system. MILEAGE needs to be able to interpret two different types of specifications, requests regarding problem details and requests involving domain knowledge. Existing AutoML systems often require some form of problem specification [2, 1, 19], but do not account for domain specification. Domain specification consists of domain-specific knowledge that influences workflow decisions, such as the knowledge about mortality rates of different types of cancer and the presence of noisy and missing values in the data collection process. While problem specification is a required component of the query, domain specification is optional information that is helpful for improving the model. Together, the MILEAGE query consisting of domain and problem specification causes the system to search through potential workflow options, with the system returning a ranked list of optimal workflows. Each of these workflows may be conveniently specified through something declarative like the Helix-DSL [52], or be compiled into imperative scripts (such as TensorFlow and Scikit-Learn).

Further drawing from the analogy with SQL for DBMS, we outline several desired properties in the language design for such a system. Starting from the top of the stack in Figure 2, the declarative language should act as a logical representation that supports physical independence with respect to how the search is actually done under-the-hood. Since ML research and practice is fast-paced and highly-evolving, the logical representation established by the declarative language ensures that if we have new representations, models, knowledge sources, or search strategies, the underlying changes can be completely hidden away from end-users. The declarative language also serves as a common, unifying exchange format across different end-user applications. Apart from the logical representation of the task definition in the problem specification, there are additional language components for specifying views and constraints. *View definitions* specify what intermediate output from the workflow can be materialized and reused in the later part of the workflow. Views can be defined explicitly by the user or by an intelligent application (such as Helix [52], DeepDive [43]) to create and materialize views. The intelligent application keeps track of what has been materialized and notifies the optimizer to reuse any views that are already materialized whenever appropriate. *Constraints* are parts of the problem specification that limits certain portions of the solution search space, in order to ensure the consistency and validity of the resulting workflow. These constraints may be in the form of performance requirements, some measure of model robustness or fairness (e.g., checking that training data is not biased towards a particular racial group), or specifying the latency budget allocated to search. Both the view and constraint specification are optional declarations as part of the language and accounted for inside the optimizer.

3.2 Communicating System Suggestions to Users

While the declarative query language enables users to communicate their intents to the system, there is also research challenges in the reverse direction, in communicating system suggestions to users. In this section, we

briefly outline several important unsolved research problems related to how the system communicates suggestions to: 1) guide users towards better workflow decisions (guidance) and 2) correct and prevent users from overlooking potential model errors (correction).

Guidance Towards Optimal Workflow Decisions. Given a declarative specification of ML intent, the system automatically traverses through the search space and returns suggestions regarding the workflow choices. Existing human-in-the-loop AutoML systems [50, 11] feature visualizations for model comparison, with users able to assign a computational budget to the AutoML search, specify the search space and model choices, or modify the problem specification. It remains an open research question as to what types of workflow recommendations and end-user interaction for model-steering would be most useful to users. What is the appropriate granularity of feedback to provide to the user that would be useful in guiding them towards an optimal workflow? Should we be suggesting modifications to an existing workflow, offering users to chose between multiple workflows, or recommending the entire workflow? Moreover, what aspects of the workflow development phase require the most amount of guidance and assistance? For example, while most AutoML systems have focused on model selection and hyperparameter search, it may possible that users actually need more assistance with feature engineering or data preprocessing. One of our goals in studying workflows on OpenML is to understand where existing ML developers struggle the most in the development lifecycle (spending most amount of time or with minimal performance gains). These observations will guide our work to address existing pain-points in the ML development lifecycle. A far-reaching research question is to examine whether workflow recommendations from a well-designed guidance system have the potential to educate novice users about best practices in model development. These best practices can include knowledge about what model to pick over another in certain scenarios or preprocessing techniques when the input data exhibits certain characteristics. The recommendation system acts as a personal coach that could teach the user practical ML skills while they are performing an ML task.

Corrective Suggestions via Proactive Monitoring. Many recent examples pervasive in the media have highlighted how errors from production ML system can have detrimental and unintended consequences, from systematic biases in a recidivism predictions[3] to racist and vulgar responses in a public chatbot[4]. To this end, several systems have been proposed to validate and monitor the quality of production ML workflows [10, 42, 13]. These systems monitor the model and raise a red-flag when the result from the deployed ML pipeline is potentially erroneous (e.g., when the input data distribution changes drastically). If done properly, the proactive monitoring capabilities of MILE may the potential for enhancing user's trust in the final ML model that is developed, improve the overall production model quality, and reduce the cost of model maintenance.

4 Autopilot

Depending on the user's level of expertise, they may wish to maintain some degree of control (Figure 1b), steering MILE between portions of the search space, or they may wish to delegate the entire search process to MILE (Figure 1c), letting it decide how and where to search. Doing this correctly would be the "holy grail" of AutoML: a general, hands-free technology capable of producing an ML workflow with adequate performance, within a given time or resource budget.

We described some of the challenges associated with exposing these capabilities to the user in the previous section; in this section, we focus on the system side. The major difficulty associated with driverless AutoML is that the design space of possible ML workflows suffers from a combinatorial curse of dimensionality. Therefore, the challenge is: *how do we make the AutoML search process over this design space as efficient as possible?*

There exists an enormous breadth of work from the AutoML community on how to best automate hyperparameter search [8, 7, 19], how to warm-start hyperparameter search [14, 16, 20], how to learn how to select good

[3] https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing

[4] https://en.wikipedia.org/wiki/Tay_(bot)

workflow *configurations*[5] [9, 47], how to trade-off model performance and search time by modeling the time budget into multicriterion optimization problems, and even how to learn the structure of neural architectures [38, 55] and other ML workflows. We foresee that MILE, which caters to multiple levels of expertise, can gather additional data on effective ML workflows from expert users (beyond the traces available in public metadata repositories such as OpenML [48]), or "meta-features", that can then be leveraged by existing meta-learning techniques. Beyond these existing techniques for *smarter search*, we envision that MILE will apply ideas from the DB community for *faster search*. We propose two concrete realizations of this vision: First, MILE can work in tandem with AutoML search strategies that operate on progressively larger samples of training data in order to evaluate the "goodness" of particular configurations. In many cases, intermediates computed on smaller training data samples can be reused for larger samples. These optimizations are inspired by prior work for employing smart caching and reuse in large-scale [46] and iterative [52, 43] ML workflows for future use. Second, for AutoML, we often know an entire up-front *set* of ML workflow configurations, unlike the iterative development setting explored in Helix [52] where the configurations are revealed one-by-one. As such, MILE can identify and exploit opportunities for *workload sharing*, whereby multiple workflows are combined, thereby making better use of locality and data parallelism. We describe these two directions next.

Progressive Intermediate Reuse. Some modern, *multi-fidelity* Hyperparameter Optimization (HPO) techniques approximately evaluate the generalization ability of ML workflow configurations by training on samples of the training data; see [24, 28, 39] for examples. If a configuration C is promising on a subset S of the training data, it might then be evaluated on a larger subset S' of training data. How can we use the work already done for C on S to speed up training on S'?

We propose that any processing composed of *associative operations* can be reused when increasing the training data size. To give a concrete example, consider PCA, which computes principal components via a singular value decomposition on the feature covariance matrix. To compute the principal components for the set of feature values associated with S', we first need a corresponding covariance matrix for S'. If we only have the covariance matrix for S without any additional information, it is not enough to help us compute S' — we must start from scratch and perform a full pass over the training data. However, if we cache counts, sums, and pairwise dot products for features in S, we can update these cached quantities with *only the data in* $S' \setminus S$, thanks to the associativity of $(+)$, after which we can proceed with the SVD as normal.

The major research challenge is to develop an optimizer that automatically identifies computations composed of associative operations. The output of such operations can be cached between runs on successively larger subsets of the training data, leading to potential speedups.

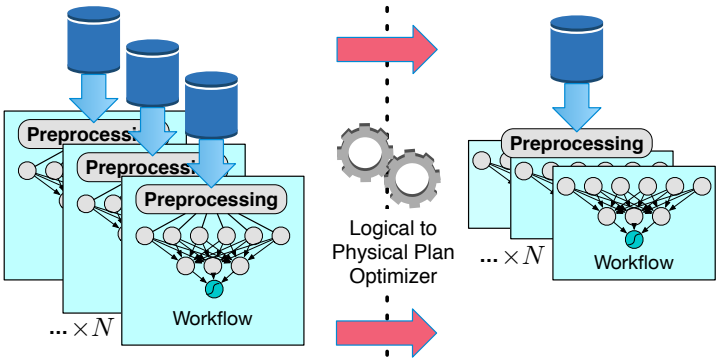

Figure 3: Compiling a logical specification of multiple workflow configurations to a physical work-sharing plan.

Work-Sharing Optimizations. AutoML search strategies typically must select from a number of ML workflow configurations. The space of configurations to evaluate, however, is typically high-dimensional. Blackbox HPO methods like grid search and random search [7] must perform large numbers of workflow evaluations, and even multi-fidelity methods like successive halving [27], Hyperband [33] and ABC [24] that leverage approximation to speed up workflow evaluation must still try out a large number of hyperparameter configurations.

Our key observation is that, at least in some cases, the configurations share a large amount of identical computation. This, in turn, can be exploited to reduce I/O and memory latency by *using the same training batches to train multiple models simultaneously*. Although

[5] An ML workflow configuration is comprised of the hyperparameter and other settings determining the workflow's behavior.

any single model will train more slowly than if it were to receive dedicated hardware resources, an ML workflow in which N models are trained concurrently will be faster than separate workflows for which each model is trained in series. Furthermore, if the training batches are preprocessed on-the-fly, the preprocessing need only be done once for concurrent training, compared to N times for serial training.

Though this technique should generalize to many kinds of ML workflows, we envision that it will be especially fruitful for training multiple neural network configurations simultaneously. As GPU and TPU accelerators increase in speed, memory capacity, and memory bandwidth, it is increasingly challenging for CPU cores to handle ETL tasks (reading from disk, parsing, shuffling, batching) so as to maximize accelerator utilization. Giving these accelerators more work is one way to alleviate this bottleneck.

This observation thus motivates the research direction of compiling a *logical* specification of a set of ML workflow configuration evaluations into a *physical* representation optimized for locality and data parallelism. Our multi-configuration physical planner is illustrated abstractly in Figure 3.

Although the kinds of work-sharing optimizations described have the potential to accelerate search through hyperparameter configurations, we foresee some difficulties along the road. First of all, MILE will need to facilitate work-sharing without requiring separate, bespoke implementations of ML models specialized for work-sharing. Secondly, we foresee that it will be nontrivial to make these work-sharing strategies operate with existing strategies to avoid overfitting to a fixed validation set. For example, one strategy [32] uses a separate shuffling of the training and validation splits for each workflow configuration to avoid overfitting to a static validation set. Employing such a strategy in concert with work-sharing optimizations will require careful maintenance of additional provenance information during training, so that some configuration C knows to selectively ignore the examples that appear in other configurations' training splits but also appear in C's validation split.

5 Conclusion: Going the Extra MILE

Present-day ML is challenging: not everybody can get mileage out of it. While AutoML is a step in the right direction, there are many real-world settings that require fine-grained human supervision. We propose MILE, an environment where humans and machines together drive the search for desired ML solutions. We identified three settings for MILE representing different levels of system automation over the design space of ML workflows—user-driven, cruise-control, and autopilot. The hope is that regardless of your desired setting, MILE gets you there faster. By catering to users with different levels of expertise, we hope to pool their collaborative experience in improving search heuristics. We also explore research opportunities in accelerating execution by applying traditional database techniques, such as materialization, lineage-tracking, and work-sharing. We hope that our MILE vision serves as a roadmap for researchers to address the valuable opportunities that stem from humans-in-the-loop of the machine learning lifecycle.

References

[1] Automated ML algorithm selection & tuning - Azure Machine Learning service. `https://docs.microsoft.com/en-us/azure/machine-learning/service/concept-automated-ml`.

[2] AutoML: Automatic Machine Learning. `http://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html`.

[3] S. Aghajanyan, R. Batoukov, and J. Zhang. Signal Fabric—An AI-assisted Platform for Knowledge Discovery in Dynamic System. Santa Clara, CA, 2019. USENIX Association.

[4] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann. Software engineering for machine learning: A case study. IEEE Computer Society, May 2019.

[5] S. Amershi et al. Guidelines for Human-AI Interaction. *CHI 2019*, pages 13–26.

[6] M. Anderson et al. Brainwash: A Data System for Feature Engineering. *CIDR*, 2013.

[7] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[8] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[9] P. B. Brazdil, C. Soares, and J. P. Da Costa. Ranking learning algorithms: Using ibl and meta-learning on accuracy and time results. *Machine Learning*, 50(3):251–277, 2003.

[10] E. Breck, N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich. Data Validation For Machine Learning. *Sysml*, 2019.

[11] D. Cashman, S. R. Humayoun, F. Heimerl, K. Park, S. Das, J. Thompson, B. Saket, A. Mosca, J. Stasko, A. Endert, M. Gleicher, and R. Chang. Visual Analytics for Automated Model Discovery. *arXiv:1809.10782*, 2018.

[12] L. Chen, A. Kumar, J. Naughton, and J. M. Patel. Towards linear algebra over normalized data. *Proceedings of the VLDB Endowment*, 10(11):1214–1225, 2017.

[13] Y. Chung, T. Kraska, N. Polyzotis, and S. E. Whang. Slice Finder: Automated Data Slicing for Model Interpretability. *SysML*, pages 1–13, 2018.

[14] M. Feurer et al. Using meta-learning to initialize bayesian optimization of hyperparameters. In *Proceedings of the 2014 International Conference on Meta-learning and Algorithm Selection-Volume 1201*, pages 3–10. Citeseer, 2014.

[15] M. Feurer et al. Efficient and robust automated machine learning. In *NeurIPS'15*, 2015.

[16] N. Fusi, R. Sheth, and M. Elibol. Probabilistic matrix factorization for automated machine learning. In *Advances in Neural Information Processing Systems*, pages 3348–3357, 2018.

[17] R. Garcia, V. Sreekanti, N. Yadwadkar, D. Crankshaw, J. E. Gonzalez, and J. M. Hellerstein. Context: The missing piece in the machine learning lifecycle. In *KDD CMI Workshop*, volume 114, 2018.

[18] Y. Gil et al. Towards human-guided machine learning. In *IUI '19*, pages 614–624, New York, NY, USA, 2019. ACM.

[19] D. Golovin et al. Google vizier: A service for black-box optimization. In *SIGKDD'17*, pages 1487–1495. ACM, 2017.

[20] T. A. Gomes, R. B. Prudêncio, C. Soares, A. L. Rossi, and A. Carvalho. Combining meta-learning and search techniques to select parameters for support vector machines. *Neurocomputing*, 75(1):3–13, 2012.

[21] Google. AutoML Tables. `https://cloud.google.com/automl-tables/`.

[22] J. Heer. Agency plus automation: Designing artificial intelligence into interactive systems. *Proceedings of the National Academy of Sciences*, 116(6):1844–1850, 2019.

[23] E. Horvitz. Principles of mixed-initiative user interfaces. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 159–166, 1999.

[24] S. Huang, C. Wang, B. Ding, and S. Chaudhuri. Efficient identification of approximate best configuration of training in large datasets. In *AAAI/IAAI (to appear)*, 2019.

[25] I. Hubara et al. Quantized neural networks: Training neural networks with low precision weights and activations. *JMLR*, 18(1):6869–6898, 2017.

[26] F. Hutter, L. Kotthoff, and J. Vanschoren, editors. *Automatic Machine Learning: Methods, Systems, Challenges*. Springer, 2018. In press, available at http://automl.org/book.

[27] K. Jamieson and A. Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pages 240–248, 2016.

[28] R. Kohavi and G. H. John. Automatic parameter selection by minimizing estimated error. In *Machine Learning Proceedings 1995*, pages 304–312. Elsevier, 1995.

[29] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. MLbase: A Distributed Machine-learning System. In *CIDR*, volume 1, pages 2–9, 2013.

[30] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.

[31] A. Kumar, M. Boehm, and J. Yang. Data management in machine learning: Challenges, techniques, and systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1717–1722. ACM, 2017.

[32] J.-C. Lévesque. Bayesian hyperparameter optimization: overfitting, ensembles and conditional spaces. 2018.

[33] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018.

[34] X. Liang, A. J. Elmore, and S. Krishnan. Opportunistic view materialization with deep reinforcement learning. *arXiv preprint arXiv:1903.01363*, 2019.

[35] Y. Lu. An End-to-End AutoML Solution for Tabular Data at KaggleDays, May 2019.

[36] D. Mané et al. Tensorboard: Tensorflow's visualization toolkit, 2015. `https://www.tensorflow.org/tensorboard`.

[37] R. S. Olson and J. H. Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In Hutter et al. [26], pages 163–173. In press, available at http://automl.org/book.

[38] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean. Efficient neural architecture search via parameter sharing. In *International Conference on Machine Learning*, pages 4092–4101, 2018.

[39] F. Provost, D. Jensen, and T. Oates. Efficient progressive sampling. In *KDD'99*, pages 23–32. ACM, 1999.

[40] F. Psallidas and E. Wu. Smoke: Fine-grained lineage at interactive speed. *Proceedings of the VLDB Endowment*, 11(6):719–732, 2018.

[41] S. Schelter, J.-H. Boese, J. Kirschnick, T. Klein, and S. Seufert. Automatically tracking metadata and provenance of machine learning experiments. In *Machine Learning Systems workshop at NIPS*, 2017.

[42] S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Biessmann, and A. Grafberger. Automating large-scale data quality verification. *Proceedings of the VLDB Endowment*, 11(12):1781–1794, 2018.

[43] J. Shin, S. Wu, F. Wang, C. De Sa, C. Zhang, and C. Ré. Incremental knowledge base construction using deepdive. *Proc. VLDB Endow.*, 8(11):1310–1321, July 2015.

[44] B. Shneiderman and P. Maes. Direct manipulation vs. interface agents. *Interactions*, 4(6):42–61, 1997.

[45] J. Sommer et al. Mnc: Structure-exploiting sparsity estimation for matrix expressions. In *SIGMOD'19*. ACM, 2019.

[46] E. Sparks. *End-to-End Large Scale Machine Learning with KeystoneML*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2016.

[47] L. Todorovski, H. Blockeel, and S. Dzeroski. Ranking with predictive clustering trees. In *European Conference on Machine Learning*, pages 444–455. Springer, 2002.

[48] J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo. Openml: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

[49] M. Vartak et al. Modeldb: a system for machine learning model management. In *HILDA'16*, page 14. ACM, 2016.

[50] Q. Wang et al. Atmseer: Increasing transparency and controllability in automated machine learning. In *CHI'19*, pages 681:1–681:12, New York, NY, USA, 2019. ACM.

[51] D. Xin, L. Ma, J. Liu, S. Macke, S. Song, and A. Parameswaran. Helix: accelerating human-in-the-loop machine learning. *Proceedings of the VLDB Endowment*, 11(12):1958–1961, 2018.

[52] D. Xin, S. Macke, L. Ma, J. Liu, S. Song, and A. Parameswaran. Helix: Holistic optimization for accelerating iterative machine learning. *Proceedings of the VLDB Endowment*, 12(4):446–460, 2018.

[53] Q. Yang, J. Suh, N.-C. Chen, and G. Ramos. Grounding interactive machine learning tool design in how non-experts actually build models. In *Proceedings of the 2018 on Designing Interactive Systems Conference 2018*, pages 573–584. ACM, 2018.

[54] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, et al. Accelerating the machine learning lifecycle with mlflow. *Data Engineering*, page 39, 2018.

[55] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv:1611.01578*, 2016.

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 need 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, GPU, 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-monitoring, self-diagnosis, self-healing, self-assembling, and self-security capabilities. On the other hand, we enable databases to provide AI capabilities using declarative languages in order to lower the barrier of using AI.

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

1 INTRODUCTION

Databases have played a very important role in many applications and been widely deployed in many fields. Over the past fifty years, databases have undergone three main revolutions.

The first generation is stand-alone databases, which address the problems of data storage, data management and query processing [**?**]. 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 [**?**]. 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/users and diversified computing power.

(1) Traditional database design is still based on empirical methodologies and specifications, and require heavy 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 DBAs to tune the knobs to adapt to different scenarios. Recently the database committee attempts to utilize machine learning techniques [**?**, **?**, **?**] 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 accurate estimation. Recently deep learning based techniques [**?**, **?**] are proposed to estimate the cost and cardinality which also achieve better results. Moreover, learning-based optimizers [**?**, **?**], learning-based index recommendation [**?**], learning-based automatic view generation [**?**] provide alternative optimization opportunities for database design. Third, traditional databases are designed by database architects based on their experiences. Recently some learning-based self-designed techniques are proposed, e.g., learned indexes [**?**] and learned NoSQL database design [**?**]. Thus we can utilize AI techniques to enhance databases and make databases more intelligent [**?**, **?**].

(2) Traditional databases focus on *relational model* and provide 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 a new database system that can integrate multiple models (e.g., relational model, graph model, tensor model) to support diversified applications (e.g., relational data analysis, graph computing and machine learning). Moreover, we can embed AI algorithms into databases, design in-database machine learning frameworks, 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, NVM, RDMA. 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 particular, on one hand, XuanYuan integrates AI techniques into databases to provide self-configuring, self-optimizing, self-monitoring, self-diagnosis, self-healing, self-security and self-assembling 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. Moreover, 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 statistics collection, online database state monitoring, 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 assemble the components to form a database in order to achieve the best performance for different scenarios. This is similar to AlphaGO, which can explore

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

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

Table 3: Five levels of AI-native database

Level	Feature	Description	Example
1	AI-advised	Plug-in AI engine	○ Workload Management (e.g., workload scheduling) ○ SQL Optimization (e.g., SQL rewriter, index/view advisor) ○ Database Monitor (e.g., knob tuner, system statistics) ○ Database Security (e.g., autonomous auditing/masking)
2	AI-assisted	Built-in AI engine	• Self-configuring (e.g., online knob tuning) • Self-optimizing (e.g., SQL optimization, data storage) • Self-healing (e.g., fault recovery, live migration) • Self-diagnosis (e.g., hardware/software error) • Self-monitoring (e.g., monitor workload/system state) • Self-security (e.g., tractable, encryption, anti-tamper)
3	AI-enhanced	Hybrid DB&AI engine	○ Learning-based Database Component • Learning-based rewriter • Learning-based cost estimator • Learning-based optimizer • Learning-based executor • Learning-based storage engine • Learning-based index ○ Declarative AI (UDF; view; model-free; problem-free)
4	AI-assembled	Heterogeneous processing	○ Self-assembling ○ Support new hardware (e.g., ARM, GPU, NPU)
5	AI-designed	The life cycle is AI-based	Design, coding, evaluation, monitor, and maintenance

more optimization spaces than humans. The fifth is AI-designed database, which integrates AI into the life cycle of database design, development, 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

We present the design of AI-native databases and Figure 1 shows the architecture. Next we discuss the five levels of AI-native databases as shown in Table 3.

2.1 Level 1: AI-Advised Database

The first level, AI-advised databases, provides offline optimization of the database through automatic suggestions [**?**, **?**, **?**]. The plugged-in AI engine is loosely coupled with databases. Limited by available resources, the AI engine mainly provides auxiliary tools from four aspects.

Workload Management. AI-based models can be used to control the workload from three aspects. First, AI-based models can benefit workload modeling. Directly modeling a workload with independent features (e.g., tables, columns, predicates) may lead to great information loss, such as the reference correlations among different tables. So instead we use an encoder-decoder model to learn an abstract representation of user workloads, which can reflect the correlation among the basic features. Second, AI-based models can be used for workload scheduling. Considering hybrid OLAP and OLTP workloads, AI-based models can estimate the required resources (e.g., CPU, RAM, DISK) and running time of each query. Then AI-based models can prioritize the workload, assign a high

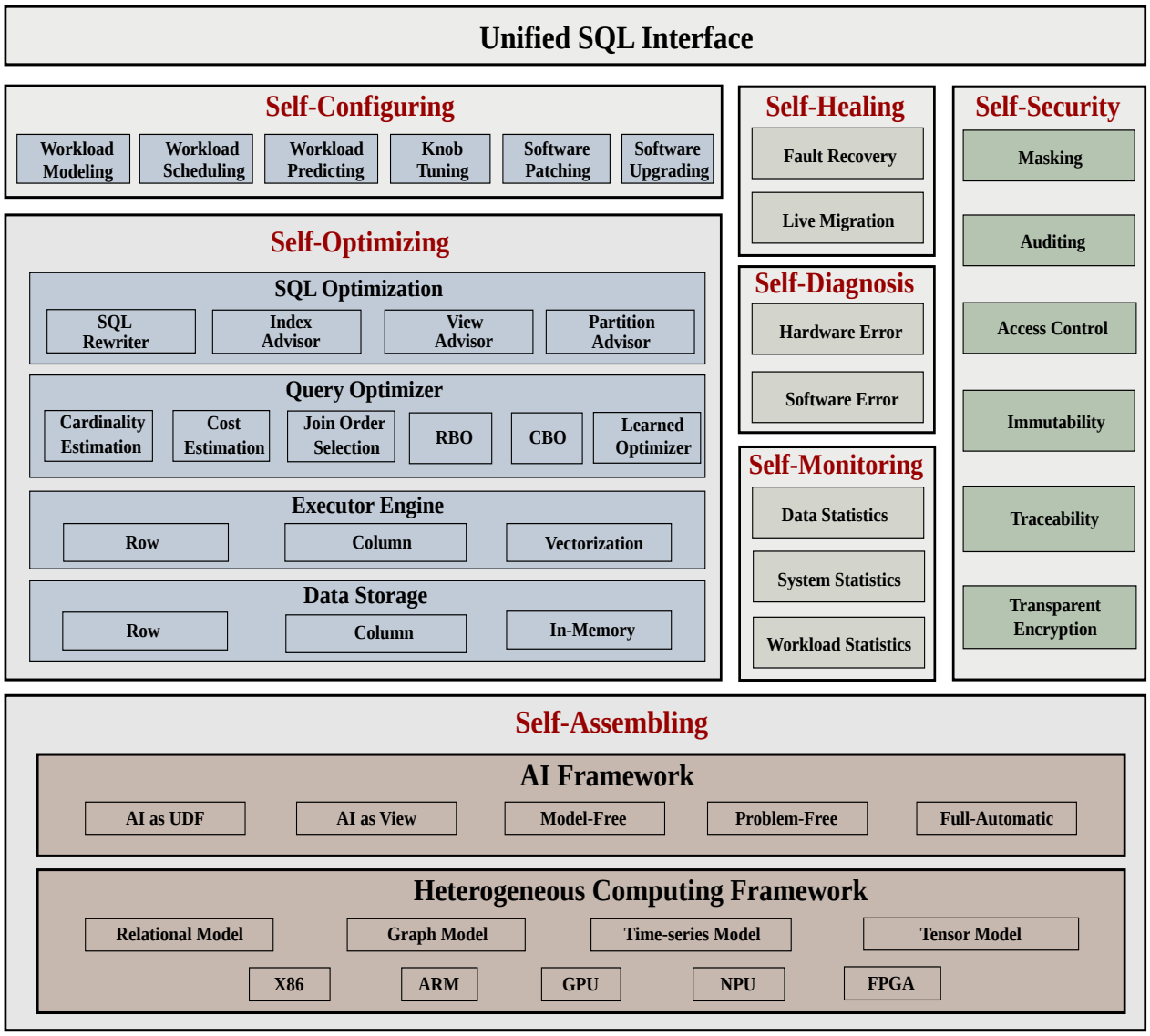

Figure 1: The architecture of AI-Native databases

priority for the mission-critical queries, and defer the execution of queries that require heavy resources. Moreover, resource scheduling depends heavily on some parameters related to resource control, e.g., maximum concurrent IO throughput. But these parameters are static and need to be manually configured. Reinforcement learning can be used to learn relations between database (physical/logical) states, workload and database resources, and provides a reasonable and robust scheduling mechanism. Third, AI-based models can be used for workload prediction. We can predict the possible workloads in order to adapt to future workloads. Traditional workload prediction methods rely on database experts based on statistical data, which can not guarantee high accuracy. Instead, machine learning methods for workload forecasting [**?**] have better adaptability to different workloads.

SQL Optimization. It optimizes SQL queries from the following aspects. First, *SQL rewriter* can rewrite the poor SQL queries issued by ordinary users into well-formulated queries in order to improve the performance. Traditional methods rely on DBAs, which are impractical for heavy workloads. So AI-based methods can provide a rewriting tool to learn the principles of SQL writing (e.g., avoiding full table scanning, selecting indexed columns for joins) and optimize the SQL structure. Second, *index advisor* can be optimized by AI-based methods.

Database indexes are very important to improve the efficiency of SQL queries on complex data sets [**?**]. However, the traditional methods build the indexes based on DBAs' experiences, which cannot scale to thousands of tables. AI-based models can learn the benefit and cost to build an index given a query workload, and then automatically recommend the index based on the learned information. Third, *view advisor* can be optimized by AI-based models. Given a set of queries, we can first extract the equivalent sub-queries, select the sub-queries with high frequency, and learn the benefit and cost of building views on the sub-queries. Then materialized views can be automatically recommended by AI-based models. Fourth, *partition advisor* can be optimized by AI-based models. Traditionally the partitions are generated based on some primary keys or manually specified keys by DBAs. However, these methods may not find the best partition strategy. Instead, the AI-based models can learn the possible distribution of the partitions, estimate the benefits for different workloads, and recommend high-quality partitions.

Database Monitor. It monitors the database states, tunes database configuration and avoids database fail-overs. First, for *data statistics*, it automatically monitors the access frequency on table columns, data updates, and data distribution among different shards. Second, for *system statistics*, it automatically monitors the status of the database system (e.g., the number of batch requests per second, the number of user connections, network transmission efficiency). Then it analyzes those indicators using machine learning algorithms, tunes system parameters and provides early warnings for abnormal events. Third, for *workload statistics*, it monitors the performance of user workload and profiles on how workload varies. It can also predict future workloads.

DB Security. It combines security and cryptography technology with AI techniques to enhance the security of databases. First, *autonomous masking* aims to hide privacy data such as ID number. Autonomous masking judiciously selects which columns to be masked based on historical data and user-access patterns. Second, *autonomous auditing* optimizes the auditing from two aspects: data pre-processing and dynamic analysis. Traditional auditing often requires auditors to obtain a large number of business data, which is hard to obtain. Autonomous auditing not only saves manpower cost, but also helps auditors to make better decisions by providing useful information from massive data. Third, *autonomous access control* can automatically detect system vulnerabilities. Existing autonomous detecting methods are mainly to retrieve through security scanning, but cannot detect unknown security vulnerabilities. AI-based models can discover security vulnerabilities [**?**], which not only detect the most known vulnerabilities in a vulnerability database, but also predict and evaluate potential vulnerabilities.

2.2 Level 2: AI-Assisted Database

The second level, AI-assisted databases, integrates an AI engine into the database kernel for run-time optimization. AI components (e.g., tuning model, workload scheduling, view advisor) can be merged into the corresponding database components [**?**]. In this way, AI capabilities are integrated into the working procedure of the database. For example, if embedding the tuning model into the query optimizer, we can first conduct query tuning for each query (e.g., tuning user-level parameters to better adapt to the query features), and then normally generate and execute the query plan. The advantage of AI-assisted databases is that 1) it can provide more fine-grained optimization; 2) it can reduce more overhead by embedding an AI engine into the kernel.

Moreover, built-in AI engine can provide self-configuring, self-optimizing, self-monitoring, self-healing, self-diagnosis, and self-security services.

Self-configuring. Database can automatically tune their own configuration to adapt to changes in workload and environment. First, the workload can be self-configured. Database can execute queries in different granularities in parallel, each of which has various requirement of system resources and performance. Database can configure the workload based on the workload features, which are obtained by conducting workload modeling, scheduling and predicting. Second, databases include several configuration mechanisms, such as knob tuning, software patching, software upgrading and etc. For example, all databases have hundreds of tunable knobs, which are vital to nearly every aspect of database maintenance, such as performance, availability, robustness, etc. However,

these configurations require to be tuned 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. First, we can design a learning-based query optimizer in cost/cardinality estimation, join order selection and data structures. 1) AI techniques can optimize cost/cardinality estimation, which is vital to query plan selection. However, database mainly estimates cardinality based on raw statistics (e.g., number of distinct values, histograms) and is poor in estimating the resulting row number of each query operator (e.g., hash join, aggregate, filter), by using histograms. AI-based methods, e.g., Tree-LSTM, can learn data distribution in depth and provide more accurate cost/cardinality estimation. 2) AI techniques can optimize join order selection. Different join schemes have a great impact on query performance and finding the best plan is an NP-hard problem. With static algorithms (e.g., dynamic programming, heuristics algorithm), the performance of join order selection in databases is limited by the quality 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. 3) Data partitions, indexes, and views can also be online recommended by built-in AI models. Second, we can utilize learning-based models to optimize executor engine and data storage. For executor engine, we consider two aspects. 1) We provide hybrid query executing methods such as row-based and column-based. Here AI can serve different data applications (e.g., row-based for OLTP, column-based for OLAP). 2) We provide tensor processing engine to execute AI models. To enhance AI techniques to optimize database components, databases can execute AI models natively with a vectorization engine.
Self-monitoring. Database can automatically monitor database states (e.g., read/write blocks, concurrency state, working transactions) and detect operation rules, e.g., root cause analysis rules. It can monitor database states (e.g., data consistency, DB health) in the life-cycle. The monitored information can be used by self-configuring, self-optimizing, self-diagnosis, and self-healing. Note that it may have some overhead to monitor the database states and we need to minimize the overhead for monitoring database states.
Self-diagnosis. Self-diagnosis includes a set of strategies to diagnose and correct abnormal conditions in databases, which are mostly caused by errors in hardware (e.g., I/O error, CPU error) and software (e.g., bugs, exceptions). Self-diagnosis helps to guarantee services even if some database nodes work unexpectedly. For example, in case of data-access error, memory overflow or violation of some integrity restrictions, database can automatically detect the root causes using the monitored states and thus cancel the corresponding transactions in time.
Self-healing. Databases can automatically detect and recover from database problems (e.g., poor performance, hardware/software fail-overs). First, it isolates different sessions or users and avoids affecting others users when encountering errors. Second, it adopts AI-advised database tools to reduce the recovery time and saves humans from the failure-recovery loop. Third, it can kill some abnormal queries that take too many resources.
Self-security. Self-security includes several features. First, the data in the life-cycle (including storage, memory and CPU) are always encrypted, which should be unreadable for the third party. Second, the data access records should be tractable in order to get the access history of the data. Third, the data should be tamper-proofed in order to prevent malicious modification of data. Fourth, AI-based models can be used to automatically learn the attacking rules and prevent the unauthorized access and attack patterns. Fifth, it can automatically detect sensitive data using AI-based models.

2.3 Level 3: AI-Enhanced Database

The third level, AI-enhanced database, not only uses AI techniques to improve the database design but also provides in-database AI capabilities.

2.3.1 Learning-based Database Components

Most of database core components are designed by humans based on their experiences, e.g., optimizer, cost estimation, index. However, we find that many components can be designed by AI-based models. First, the traditional indexes, e.g., B-tree, R-tree, can be designed by AI-based models. For example, learned indexes are verified that they can reduce the index size and improve the query performance [**?**]. Second, the cost/cardinality estimation can be optimized by deep learning, because the empirical methods cannot capture the correlations among different tables/columns while deep learning can capture more information using deep neural networks. Third, the join order selection problem is an NP-hard problem and traditional heuristics methods cannot find the best plan; while the deep reinforcement learning techniques can learn more information and get better plan. Fourth, query optimizer replies on cost/cardinality estimation, indexes, join order selection, etc, and an end-to-end learning based optimizer is also promising.

Thus many database components can be enhanced by AI-based methods, which can provide alternative strategies beyond traditional empirical techniques.

2.3.2 In-Database AI Capabilities

Although AI can address many real-world problems, there is no widely deployed AI systems that can be used in many different fields, because AI is hard to be used by ordinary users. Thus we can borrow database techniques to lower the barrier of using AI. First, SQL is easy to be used and widely accepted, and we can also extend SQL to support AI. Second, we can utilize database optimization techniques to accelerate AI algorithms, e.g., indexing, incremental computing, and sharing computations.

We categorize the techniques of supporting AI capabilities in database in five levels.

AI Models As UDFs. We embed AI frameworks (e.g., MADlib, TensorFlow, Scikit-learn) in database and provide user-defined functions (UDFs) or stored procedures (SPs) for each algorithm. Then users can call UDFs or SPs from databases to use AI algorithms.

AI Models As Views. If a user wants to use AI algorithms (e.g., random forests) in the first level, the user requires to first train the model and then use the trained model. In the second level, we can take an AI algorithm as a view, which is shared by multiple users. If an algorithm is used by a user, we can materialize the model and other users can directly use the model. The model can also be updated by incrementally training.

Model-free AI. In the first and second levels, the user must specify the concert algorithms (e.g., k-means for clustering). Actually, users may only know which problems should be addressed, e.g., clustering or classification, but do not know which algorithms should be used. In this way, database can automatically recommend the algorithms that best fit the user scenarios.

Problem-free AI. The users even cannot specify the problems that require to be addressed, e.g., classification and clustering. Given the database, the problem-free AI can automatically find which problems can be addressed by AI algorithms and recommend suitable AI algorithms.

Full-automatic. The system can automatically discover AI opportunities, including discovering the problems, the models, the algorithms, relevant data, and training methods.

2.3.3 Hybrid AI and DB Engine

The above methods still use two engines: AI engine and DB engine. It calls for a hybrid AI and DB engine that provides both AI and DB functionalities. Then given a query, the SQL parser parses the SQL query and produces a general-purpose query plan. Based on the operators in the plan, it decides whether it utilizes relational data models or utilizes AI models. For relational data models, the query plan is sent to the database executor; otherwise, it is sent to the AI executor. Moreover, it also calls for new models to support both relational algebra and tensor models, and in this way, we can utilize a unified model to support both AI and database.

2.4 Level 4: AI-Assembled Database

The fourth level, AI-assembled database, not only automatically assembles database components to generate a database that can best fit a given scenario, but also schedules the tasks to diversified hardware.

Self-assembling. First, each database component has multiple options. For example, optimizers include cost-based model, rule-based model, and learned-based model. We first take each component variant as a service. Then we select the best components based on users' requirements. Note that different variants of the same component should adopt the same standard interface such that the components can be assembled.

Thus we propose a self-assembling module. For different scenarios, 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 the query level. One possible method is to use reinforcement learning (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 [**?**] can be used. Compared with other RL algorithms with discrete outputs, DDQN eliminates the problem of overestimation (deviate greatly from the optimal solution) by choosing decoupling actions and calculating the target Q value.

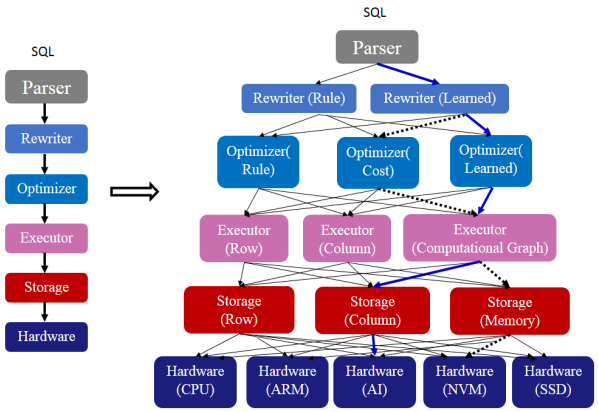

Figure 2: AI-Assembled Database

However, the RL-based routing algorithm has two problems. Firstly, Q network will not generate scores 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 an 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.

Generative Adversarial Network (GAN) [**?**] can better solve those problems of end-to-end path selection. We use G network to generate path vectors based on the 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 [**?**]. Firstly, G network is used as an agent in the 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 the traditional RL algorithm, each action is scored by D network to guide the generation of the whole path sequence.

Heterogeneous Computing. We need to make full use of diversified computing power. Note that DB and AI usually require different computing power and hardware. For DB, traditional optimizer processes queries with CPU. While AI requires new AI chips to support parallel processing (e.g., GPU and NPU) and self-scheduling. And now many applications need to use both DB and AI techniques, especially in large data analysis scenarios. Thus we need to support multiple models, e.g., relation model, graph model, stream model and tensor model. We can automatically select which models should be used. We also need to be able to switch computing powers based on the models. For example, for the tuning module in the optimizer, when training the tuning model, we

fetch training data into memory with AI chips and conduct backward propagation (training the neural network) with NPU. For the join and selection operations, we may still use the traditional hardware. We also need to study whether a model (e.g., relational model) can be transformed into other models (e.g., tensor model). The ultimate objective is to fully unleash the power of diversified computing, including x86, ARM, GPU, NPU. We aim to continuously push our AI strategy forward and foster a complete computing ecosystem.

2.5 Level 5: AI-Designed Database

The fifth level, AI-designed databases, is fully designed by AI, including design, coding, evaluation, monitoring and maintenance. In this way, AI techniques are integrated into the whole database life cycle so as to achieve the best performance for both DB and AI.

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. Thus these human-designed databases may not be optimal because they may fall into a local optimum. It calls for automatically designing a database using AI techniques, which can adapt to different scenarios.

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 databases. 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. It calls for effective methods to schedule the tasks.

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 think that hybrid DB and AI online analytic processing on multi-model data should be the next generation OLAP, i.e., *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. Third, it calls for an effective tool to automatically evaluate a feature or a component (even a database).

3.4 AI4DB

There are several challenges that embed AI capabilities 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 a large number of 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 a better choice for cost estimation while reinforcement learning may be a better 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 adapt to different database 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 AI algorithms. For example, self-driving vehicles require 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 rainy day, but not many redundant examples. Thus we can index the samples and features for effective training.

Discover AI Models. Ordinary users may only know their requirements, e.g., using a classification algorithm to address a problem, but do not know which AI algorithms should be used. Thus it is important to automatically discover AI algorithms. Moreover, it is also challenging to reuse the well-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 a tiny database embedded in small devices. There are several challenges in designing such a tiny database. The first is database security to protect the data. 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-monitoring, self-healing, self-diagnosis, self-security, and self-assembling, 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, and AI-designed. We also discussed the research challenges and provided opportunities in designing an AI-native database.

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
10662 Los Vaqueros Circle
Los Alamitos, CA 90720-1314

Non-profit Org.
U.S. Postage
PAID
Los Alamitos, CA
Permit 1398