IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 12 December 2023, accepted 23 December 2023, date of publication 28 December 2023,
date of current version 25 January 2024.

Digital Object Identifier 10.1109/ACCESS.2023.3347725

RESEARCH ARTICLE

Trinity: In-Database Near-Data Machine Learning Acceleration Platform for Advanced Data Analytics

JI-HOON KIM[1], (Graduate Student Member, IEEE), SEUNGHEE HAN[1], (Graduate Student Member, IEEE), KWANGHYUN PARK[2], SOO-YOUNG JI[3], AND JOO-YOUNG KIM[1], (Senior Member, IEEE)

[1]Korea Advanced Institute of Science and Technology (KAIST), Daejeon 34141, South Korea
[2]Department of Computer Science, Yonsei University, Seoul 03722, South Korea
[3]Samsung Electronics, Suwon 16677, South Korea

Corresponding author: Joo-Young Kim (jooyoung1203@kaist.ac.kr)

This work was supported in part by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2020-0-01847) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation); and in part by Samsung Electronics Co., Ltd (IO201210-07991-01).

ABSTRACT The ability to perform machine learning (ML) tasks in a database management system (DBMS) is a new paradigm for conventional database systems as it enables advanced data analytics on top of well-established capabilities of DBMSs. However, the integration of ML in DBMSs introduces new challenges in traditional CPU-based systems because of its higher computational demands and bigger data bandwidth requirements. To address this, hardware acceleration has become even more important in database systems, and the computational storage device (CSD) placing an accelerator near storage is considered as an effective solution due to its high processing power with no extra data movement cost. In this paper, we propose Trinity, an end-to-end database system that enables in-database, in-storage platform that accelerates advanced analytics queries invoking trained ML models along with complex data operations. By designing a full stack from DBMS's internal software components to hardware accelerator, Trinity enables in-database ML pipelines on the CSD. On the software side, we extend the internals of conventional DBMSs to utilize the accelerator in the SmartSSD. Our extended analyzer evaluates the compatibility of the current query with our hardware accelerator and compresses compatible queries into a 24-byte numeric format for efficient hardware processing. Furthermore, the predictor is extended to integrate our performance cost models to always offload queries into the optimal hardware backend. The proposed SmartSSD cost model mathematically models our hardware, including host operations, data transfers, FPGA kernel execution time, and the CPU cost model uses polynomial regression ML models to predict complex CPU latency. On the hardware side, we introduce the in-database processing accelerator (i-DPA), a custom FPGA-based accelerator. i-DPA includes database page decoder to fully exploit the bandwidth benefit of near-storage processing. It also employs dynamic tuple binding to enhance the overall parallelism and hardware utilization. i-DPA;s architecture having heterogeneous computing units with a reconfigurable on-chip interconnect also allows seamless data streaming, enabling task-level pipeline across different computing units. Finally, our evaluation shows that Trinity improves the end-to-end performance of analytics queries by 15.21× on average and up to 57.18× compared to the conventional CPU-based DBMS platform. We also show that the Trinity's performance can linearly scale up with multiple SmartSSDs, achieving nearly up to 200× speedup over the baseline with four SmartSSDs.

INDEX TERMS Computational storage device, database, data analytics, end-to-end system, hardware accelerator, machine learning, near-data processing, SmartSSD.

The associate editor coordinating the review of this manuscript and approving it for publication was Fabian Khateb.

I. INTRODUCTION

As the volume of data increases exponentially in the era of big data [29], systems for scalable and efficient data analytics play a critical role in a wide variety of domains such as scientific explorations, finance, governance, health care, and web analytics [45]. For this reason, database management system (DBMS), an essential and fundamental platform for large-scale data management, has begun to evolve for analytics workloads [1], [2], [3], [4], [5]. In this new paradigm, we observe that three important yet independent technology trends have been emerged, as illustrated in Figure 1.

First, data analytics extends beyond simple relational query-based analysis, moving into advanced analytics using machine learning (ML), referred to as ML-driven advanced analytics. Traditional analytics queries employ relational algebra such as filter, aggregate, and join in identifying patterns and getting insights from data (e.g., finding min/max/average values from a joined table). However, as the size of data and the data complexity have increased, analyzing data using only relational queries has reached its limitation [60]. On top of many algorithms developed for data analytics, ML finally comes into play in various data analytics applications such as classification [43], regression [52], recognition [27], and prediction [44]. In line with this trend, major enterprise DBMSs, including Microsoft SQL Server [6], Google;s BigQuery [7], and Amazon Redshift [8], try to integrate ML services inside. This integration allows them to leverage ML algorithms without sacrificing their own benefits such as transparent scalability, fine-grained access control, security, and high-availability [9]. Various open-source ML libraries for DBMS like Apache MADlib [10] and SparkML [54] are also emerging to widely support in-DBMS ML inference and training.

Second, hardware acceleration is another big trend in data analytics. Especially, accelerating database operations (e.g., sorting, joins, aggregates, etc.) with specialized hardware such as Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs) [11] has been explored in many recent works. For example, Casper et al. [30] proposed the hardware design for accelerating the various database operations such as selection, merge join, and sorting. He et al. [36] also tried to utilize a GPU as a database engine by modifying relational operations into tensor computations. In addition, major datacenter operators including Microsoft [31], Google [38], and Amazon [12] have also utilized their own specialized hardware for running enterprise-level data analytics. Notably, the introduction of hardware accelerator in data analytics has become more important as analytical workloads have become more complex along with the integration on ML in DBMSs. This is because conventional CPU-based systems are often suffering from a huge performance degradation due to the heavy computational demands of ML runtimes.

Lastly, near-data or in-storage processing has become a major computer system architecture for accelerating data-intensive applications including data analytics. State-of-the-art system architectures that integrate specialized hardware can be grouped into two major categories from the perspective of data movement [28]: out-of-the-wire and bump-in-the-wire architecture (see Figure 2). The out-of-the-wire architecture is the conventional accelerator model, in which the host reads the data and sends them to the specialized hardware for computation offloading. This is the general system architecture that current GPU and FPGA has. On the other hand, the bump-in-the-wire architecture, which is also known as near-data or in-storage processing architecture, places the specialized hardware between the host and storage so that it can perform in-line processing on the read data. Between these two models, as the volume of data increases, the out-of-the-wire architecture suffers from the time and energy overhead required for additional data movement between the host and specialized hardware. As a result, data-intensive applications such as recommendation system [46], [62], nearest neighbor search [41], [42], and data analytics [47], [53], [58] have been widely accelerated using the bump-in-the-wire architecture.

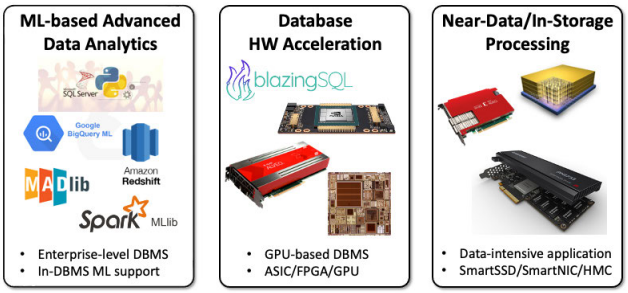


FIGURE 1. Three research cornerstones in data analytics.

These three technology trends are mostly studied in isolation, but several works have been proposed in the intersection of two. A few works (e.g., AQUOMAN [65], Mondrian [34]) have tried to accelerate the data analytics using specialized hardware with bump-in-the-wire architecture, but they only focus on the conventional relational query operations and have not considered in-DBMS ML operations. Some other works (e.g., Gorgon [61], DAnA [50]) have tried to accelerate the ML-based advanced data analytics with specialized hardware accelerator, but they chose the conventional out-of-the-wire architecture in integrating their specialized hardware to the system. Although they briefly mention that the proposed accelerator can be applied to the near-data processing, they do not describe any details on how to use it in DBMS.

To the best of our knowledge, none of the previous works successfully integrate the three cornerstones at once. However, the unification of these three areas will enable a faster and more efficient advanced data analytics in DBMS, breaking the performance limitation of existing database systems. Therefore, we propose Trinity, a system that integrates all these three areas harmonically. Trinity is

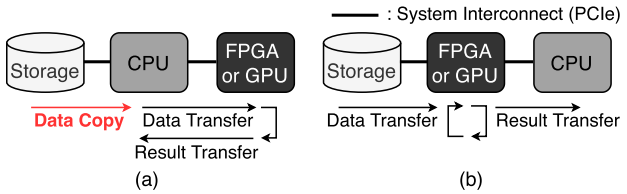


FIGURE 2. Two major models in the integration of specialized hardware: (a) Out-of-the-wire architecture (b) Bump-in-the-wire architecture.

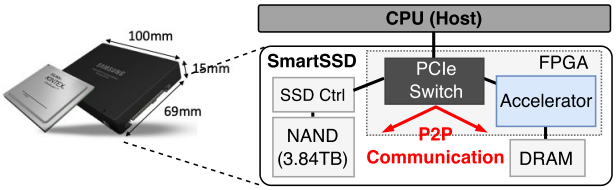


FIGURE 3. System architecture of SmartSSD.

an *in-database, in-storage* platform that implements a deep integration of ML inference pipelines (MADlib [10]) into the state-of-the-art open-source DBMS (PostgreSQL [13]) for advanced data analytics. In particular, we use the SmartSSD [14] that integrates a solid-state drive (SSD) and a FPGA in the U.2 form factor as a new hardware backend of the DBMS to enable near-data acceleration. By placing an in-database ML accelerator directly on the storage device, analytics queries can be processed where data reside. In addition, Trinity can also free up the system resources such as CPU, main memory, and bus bandwidth, which can be potentially used for other tasks running on the host. It is noteworthy that Trinity is the first end-to-end in-DBMS and in-storage platform that not only utilizes the bump-in-the-wire architecture for advanced data analytics but also integrates a software stack for generating optimal query plan and allowing to utilize the new hardware platform (i.e., SmartSSD) in conventional DBMS.

The main contributions of our work are as follows.

- We build the end-to-end in-database, in-storage acceleration platform called Trinity for the first time that can speed up both database and ML pipelines for complex analytical query processing. To this end, we modify PostgreSQL;s internal subsystems and harness SmartSSD for our target in-storage acceleration platform.
- We extend conventional query analyzer and optimizer to address the challenges associated with physical separation and variability in best performing hardware when employing an additional hardware backend. Our enhanced query analyzer includes the capability to convert the extracted query information into a compressed data format that is suitable for our hardware accelerator. Furthermore, our novel query predictor can dynamically select the hardware backend that optimizes the overall system performance. It makes Trinity to consistently execute queries with the best performing hardware within the proposed CPU-SmartSSD system.
- We develop the performance models for both CPU and SmartSSD to estimate the processing time for a given query to decide whether to offload it to the hardware accelerator or not. These models are seamlessly integrated in our extended query predictor. The SmartSSD cost model provides a mathematical representation of our hardware platform, and the CPU cost model leverages polynomial regression ML models to predict complex CPU latency with novel fine-tuning methodology.
- We propose a custom hardware accelerator, in-Database Processing Accelerator (i-DPA) that can accelerate the in-database ML queries with three key ideas. First, database page decoder is integrated in i-DPA to fully leverage the bandwidth benefit of in-storage processing. With database page decoder, i-DPA can also accelerate the decoding process with parallel compute units. Second, dynamic tuple binding is introduced in i-DPA to maintain always high hardware utilization across various query conditions. It can increase the tuple level parallelism up to 8 times. Lastly, streaming-wise pipelined processing are utilized in i-DPA. It allows i-DPA to simultaneously process multiple tuples in parallel with its heterogeneous computing units. Notably, our i-DPA can utilize all three different levels of parallelism existing in data analytics: 1) page-level parallelism, 2) tuple-level parallelism, and 3) task-level parallelism.
- We evaluate the end-to-end performance of Trinity for advanced analytics queries that invoke a wide variety of ML models (i.e., linear/logistic regression, SVM, tree, and MLP) with different datasets. We compare the performance against a conventional CPU- and GPU-based DBMS and provide in-depth analysis on our platform.
- We also increase the database to the terabyte domain and scale up the Trinity to have multiple SmartSSDs. Trinity shows a linear performance gain with the number of devices.

II. BACKGROUND

A. IN-DB ML INFERENCE ACCELERATION

The focus of this paper is to design efficient software and hardware architecture for in-database ML inference. MADlib [37] is one of the state-of-the-art open-source solutions for in-database ML by embedding UDAs/UDFs into the database systems such as PostgreSQL [13] and Greenplum [15]. MLlib [54] is Apache Spark's open-source-based machine learning library, consisting of universal ML algorithms and utilities. Not only these open-source-based DBMSs but also major DBMS vendors also provide their own ML function packages [6], [7], [8], [16]. For instance, Oracle [16] supports in-DBMS ML inference using its PL/SQL packages, and Google's Big Query [7] extends its form of SQL syntax to support ML packages. In our work, we utilize two popular open-source software, MADlib and PostgreSQL, to build a tightly integrated software-hardware system.

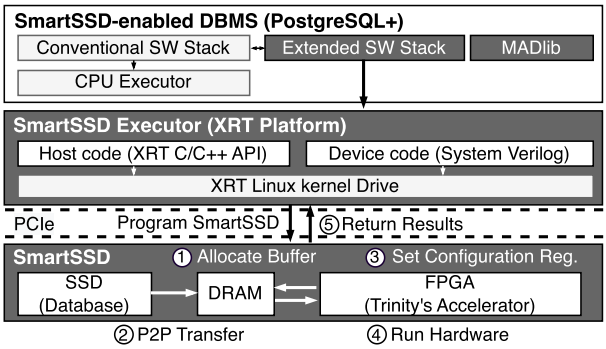


FIGURE 4. System architecture of Trinity.

B. OUT-OF-THE-WIRE VS BUMP-IN-THE-WIRE ARCHITECTURE

The out-of-the-wire architecture is an easy plug-in model of the hardware accelerator to the CPU-based system. As the main controller of the system, the CPU reads data from the storage and relay it to the accelerator. This additional data copy deteriorate the system performance. Intel HARP [55] and most of the conventional accelerators including GPU and FPGA fall in this architecture. In the bump-in-the-wire architecture, the hardware accelerator reads the data and process it directly without any intervention from the CPU. Samsung SmartSSD [14], IBM Netezza [17], AWS F1 [12], and Maxeler [30] adopt this architecture.

C. SMARTSSD: COMPUTATIONAL STORAGE DEVICE

Among many examples of the bump-in-the-wire architecture, SmartSSD is an actively used devices both in academia and industry [41], [42], [47], [53], [58]. As shown in Figure 3, it integrates a Xilinx UltraScale+ FPGA with 4GB DRAM and 3.84TB NAND Flash arrays in the U.2 form factor for data processing in the storage device. The FPGA chip contains 1.14 million logic cells, 1968 DSP slices, and 34.6 Mbits on-chip SRAM. The SmartSSD device is connected to the host CPU through the PCIe Gen3 × 4 interface, which gives a theoretical maximum bandwidth of 4 GB/s. One of the main features of the SmartSSD platform is that there is a PCIe switch presented in the FPGA to provide three-way paths: between the CPU and FPGA, between the CPU and SSD, and between the FPGA and SSD. Especially, the internal peer-to-peer (P2P) communication between the FPGA and SSD enables in-storage processing with removing unnecessary data copy to the host. The 4GB DRAM attached to the FPGA can work as a buffer memory when data moves between FPGA and CPU or between FPGA and SSD because the FPGA does not have enough memory on the chip. It provides a speed of 2400Mbps.

III. SYSTEM OVERVIEW

Figure 4 shows the high-level system architecture of Trinity, which is a complete full-stack system for in-database ML inference based on computational storage device (i.e., SmartSSD [14]). To provide seamless integration of the SmartSSD backend, the software stack of Trinity consists of an extended version of the PostgreSQL database system, dubbed PostgreSQL+, and Xilinx FPGA runtime (XRT [18]).

PostgreSQL is a widely used, advanced, open-source object-relational database system known for its extensibility, allowing users to plug in their custom logic within the DBMS. Our PostgreSQL+ modifies the PostgreSQL;s internal subsystems, such as the query analyzer, optimizer, and executor to enable the use of SmartSSD backend within the DBMS. It also integrates MADlib [10], an open-source library that supports scalable in-database ML operations for both structured and unstructured data. When PostgreSQL+ analyzes and converts host queries for SmartSSD backend utilization, the XRT platform takes on the crucial role of configuring the hardware and delivering the commands to the hardware for actual query processing. Specifically, the XRT platform is a software interface that facilitates communications between the host CPU and SmartSSD. It involves a host code that includes commands for device programming, buffer allocation, data transfer, and kernel execution, and a device code that describes the custom hardware accelerator. Based on the XRT application programming interfaces (APIs), the XRT Linux kernel driver executes the requested commands. It manages memory allocation and maps the kernel virtual address to the physical address. The XRT Linux kernel driver also sets the configuration registers of the hardware accelerator and controls the execution flow of FPGA. As the main hardware platform of Trinity, SmartSSD is responsible for storing the whole database in its high-capacity SSD. In addition, its internal accelerator can perform processing directly on the database without sending it to the host.

Overall, the XRT platform executes the following process to run the hardware accelerator on FPGA. First, it allocates buffers on the FPGA's local DRAM for the input/model tables and output data (①). Afterward, the input and model tables stored in a database of SSD are directly transferred to the FPGA DRAM through the P2P communication [19] (②). Once the data are ready, the XRT platform sends the address of each allocated buffer and the meta-data received from PostgreSQL+ to the configuration registers of the hardware accelerator (③). Finally, it sends a run signal to operate the hardware accelerator (④). The hardware accelerator then starts to process the query by referring to the configuration registers. When the kernel execution is done, the XRT platform brings the final results from the FPGA's DRAM to the host (⑤).

IV. SOFTWARE STACK FOR TRINITY

DBMS is responsible for checking the syntax of the current query string and generating an optimal query plan based on its cost estimates. In the conventional DBMS, the CPU is the only platform for query processing, so the CPU has to perform the above preprocesses and execute queries as well. However, as the query involves lots of complex operations, like ML, the time spent by the executor can be excessively

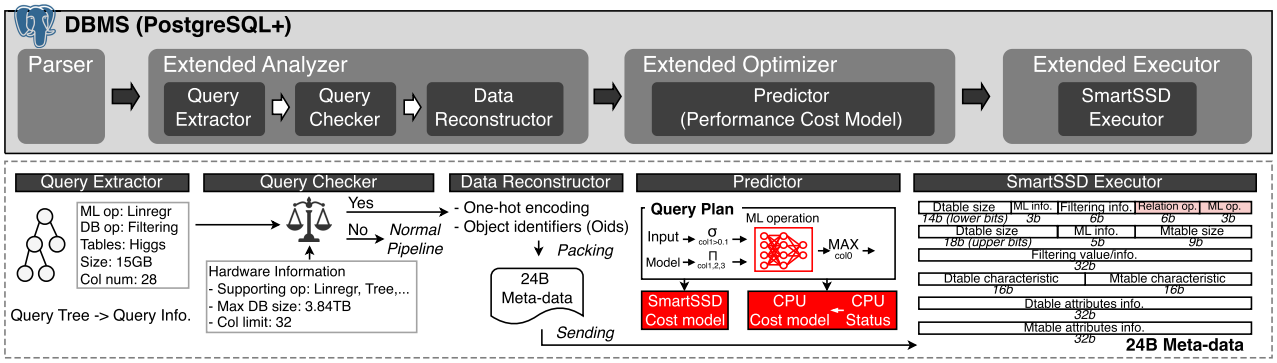


FIGURE 5. Execution pipeline of PostgreSQL+.

long. Introducing a new hardware is a promising solution to solve this problem, but utilizing the new hardware backend in DBMS causes new challenges in overall system design.

Physical Separation: Hardware accelerators are usually connected to the CPU via a PCIe system interconnect as external devices. It means that the hardware accelerator cannot directly know the current query information being processed on the CPU side. For the hardware to know the query information, DBMS should be able to transfer the current query information into the hardware. Moreover, since the custom hardware accelerator has its own interface to communicate with the host, the extracted query information should be able to be converted into the data format of hardware.

Selective Query Offload: Since the hardware accelerator has limited memory and compute resources, the hardware accelerator has limitation in supporting all existing queries of DBMS. Therefore, DBMS should be able to determine whether queries can be supported in hardware or not based on the hardware and query information. In addition, queries that cannot be processed in hardware should be able to be handled via conventional CPU executor.

Different Best Performing Hardware: Even query can be executed in hardware accelerator, using a hardware accelerator does not guarantee the best system performance for entire queries. Depending on the query characteristics like database size and model complexity, the optimal hardware backend can be varied. Especially, since both data and model characteristics are determined by the query presented at the runtime, DBMS should make a runtime offloading decision for better system performance based on the cost modeling. Moreover, with the introduction of a new hardware backend, the cost model should be able to predict the objective metrics like latency to effectively compare the performance of two different hardware options.

Our Approach: Trinity utilizes the SmartSSD as a new hardware backend for accelerating advanced data analytics query processing. Considering the above system-level challenges, we propose PostgreSQL+, an extended version of PostgreSQL that can run the SmartSSD executor [1] along with the CPU executor. To this end, the analyzer and optimizer of PostgreSQL have been modified.

[1] In here, the executor means the hardware itself that processes the query. SmartSSD executor, SmartSSD, and SmarSSD backend are identical.

TABLE 1. Parameters used in the SmartSSD cost model.

Parameter	Description
$T_{smartssd}$	Total execution time of SmartSSD
T_{host}	Host execution time of SmartSSD
$T_{transfer}$	Data transfer time of SmartSSD
T_{kernel}	FPGA kernel execution time of SmartSSD
T_{buf}	CreateBuffer API execution time in host
T_{others}	Other APIs, except CreateBuffer API, execution time in host
T_{s2f}	SSD to FPGA data transfer time
T_{f2h}	FPGA to host data transfer time
N_{page}	Number of pages to process
N_{core}	Number of cores instantiated in user FPGA
R_{user}	FPGA resources available for user (LUT, FF, RAM, DSP)
R_{core}	FPGA resources that a core occupies (LUT, FF, RAM, DSP)
C_{kernel}	Cycle count for processing single page with target workload
$C_{frequency}$	FPGA clock frequency
S_{db}	Total processing database size
R_{buf}	Effective memory allocation rate
R_{s2f}	Effective data transfer rate between SSD and FPGA
R_{f2}	Effective data transfer rate between FPGA and host

A. OVERALL EXECUTION PIPELINE

Figure 5 shows the overall execution pipeline of PostgreSQL+. When the PostgreSQL+ receives the query from the user, the extended analyzer first determines whether the hardware accelerator can process the current query or not. To this end, query extractor extracts the query information from the query tree and query checker compares this extracted information with internal hardware information to identify whether the current query can be processed in the hardware accelerator or not. The hardware information contains the corner parameters that our hardware accelerator can cover such as the maximum database size, maximum number of attributes, and supported operations. If the current query does not fit in this hardware restriction, the analyzer marks the current query to be processed with the existing CPU executor. Since PostgreSQL+ maintains backward compatibility to PostgreSQL, normal CPU execution pipeline is also available. As queries that cannot be supported by the hardware accelerator skips the subsequent offloading processes, they show almost identical performance with conventional PostgreSQL execution. On the other hand, if the query checker determines the current query can be processed in our hardware accelerator, data reconstructor

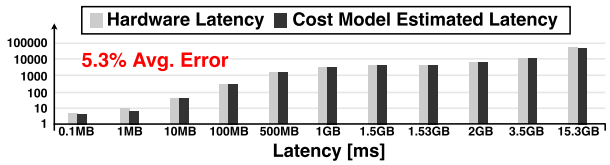


FIGURE 6. SmartSSD cost model accuracy with different database sizes.

converts the extracted query information (e.g., operation type, table name, selected attributes) to the numeric data format of the hardware accelerator using one-hot encoding and object identifiers. These converted data are finally packed in six 4-byte meta-data (i.e., 24-byte). After that, the predictor of extended optimizer determines the optimal hardware backend between the CPU and SmartSSD executor by estimating the execution time of each hardware. To this end, it includes the performance cost model for each executor. By comparing the estimated latency of two cost models, predictor makes the query to be processed via hardware that exhibits faster processing speeds. If the execution time of SmartSSD is faster than that of CPU, the query is offloaded to the hardware accelerator by transferring the generated meta-data to the SmartSSD.

B. PERFORMANCE COST MODEL

The performance cost model for each executor predicts the execution time of target hardware considering data, query and system characteristics. Since the performance cost model should be adaptable to various systems while maintaining high accuracy for general usage of PostgreSQL+, we try not to overfit the models into specific configurations. Therefore, SmartSSD cost model has architectural parameters in mathematical formula, and we present online fine-tuning method for CPU cost model.

1) SMARTSSD COST MODEL

The total execution time of the SmartSSD can be expressed as sum of the host operation time, data transfer time, and FPGA kernel time (1). Table 1 lists the parameters used in the SmartSSD cost model.

$$T_{smartssd} = T_{host} + T_{transfer} + T_{kernel} \tag{1}$$

T_{host} denotes the time spent to run XRT APIs in the host for the communication between the host and the hardware kernel. Among many function calls, *clCreatebuffer* that allocates a desired size of buffer in the FPGA's DRAM takes most of the host time. $T_{transfer}$ represents the time spent for actual data movement across the host, SSD, and FPGA. It can be described as the sum of the time for fetching the selected tables from the SSD to FPGA and the time for transferring the final result from the FPGA to host. These two parameters can be further represented as in (2).

$$T_{host} = T_{buf} + T_{others}, \quad T_{transfer} = T_{s2f} + T_{f2h} \tag{2}$$

The time for allocating buffer and transferring data is proportional to the target database size. In order to reflect this, T_{buf} has an effective memory allocation rate R_{buf}, which represents the time required to allocate a unit memory size. Similarly, T_{s2f} and T_{f2h} have effective data transfer rates as sub-parameters. These variable rate parameters are pre-measured in our system and we observe that they are constant when the database size is over 0.5GB but linearly decreases under it. Once the rate parameters are determined, the memory allocation and data transfer time are given as in (3).

$$\begin{aligned} T_{buf} &= S_{db} \times R_{buf} \\ T_{s2f} &= S_{db} \times R_{s2f}, \ T_{f2h} = S_{db} \times R_{f2h} \end{aligned} \tag{3}$$

The last term, T_{kernel}, represents the time spent on executing the FPGA kernel. It can be represented by multiplying the average number of pages processed by a core and the average single page execution time of a core, as expressed in (4). The former can be calculated by dividing the total number of pages by the number of cores. The total number of pages is calculated as the database size divided by the page size, 8KB. Note that the number of cores is determined by the available user resources in the FPGA. If the target device has a bigger FPGA, the number of cores will be higher. The latter means the time spent on processing a given workload for a single page data. In order to calculate this, the cycle count required to process a single page is multiplied by the FPGA's clock period. The cycle count is extracted from the designed hardware accelerator and is pre-stored in the cost model.

$$\begin{aligned} T_{kernel} &= (N_{page}/N_{core}) \cdot (C_{kernel}/C_{frequency}) \\ N_{page} &= S_{db}/8KB, \quad N_{core} = Floor(R_{user}/R_{core}) \end{aligned} \tag{4}$$

Figure 6 shows the accuracy of the proposed SmartSSD cost model. We measure the execution time of SmartSSD when the database size varies from 0.1MB to 15.3GB and compare them with the predicted values by the model. As a result, the cost model shows only 5.3% average error rate for the experiment. It achieves a high accuracy because it predicts the FPGA kernel operation time and considers both the host and data transfer time for SmartSSD execution. In addition, since the cost model reflects the variable rate parameters according to the database size, reliable results can be obtained.

2) CPU COST MODEL

Unlike the SmartSSD's performance cost model, the CPU's cost model is difficult to represent with a few linear equations as it is a complex system with influences such as caching and IO operations. In addition, the query processing time in CPU shows different characteristics according to the complexity of the query. Therefore, we triage the queries into a few groups based on the complexity and use a polynomial regression ML model in each group to predict the CPU latency. Figure 7-(a) shows CPU cost model generation pipeline.

❶ *Initial stage*: At the initial stage, PostgreSQL+ executes the queries on the CPU side to collect experimental measurements. For each query, CPU cost model first stores the executed query information (i.e., the number of tuple,

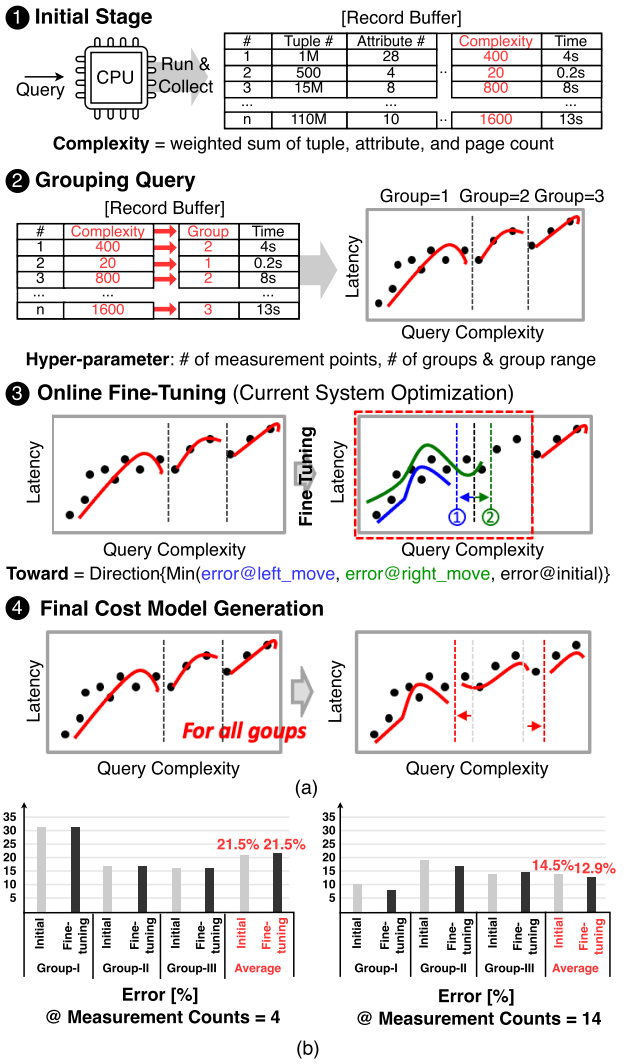


FIGURE 7. (a) CPU cost model generation pipeline (b) CPU cost model accuracy.

attribute, page, and execution time) in record buffer of PostgreSQL+ and estimates the complexity of the query by calculating the weighted sum of its tuple, attribute, and page count.

❷ *Grouping query stage*: Based on this query complexity, the recorded queries are divided into the cost group that includes the current complexity cost value. Then, if the CPU cost model gathers all the user-defined number of measurements for each cost group, it finally generates third-order polynomial curves for each group. Figure 7-(2) shows the example of our grouping query process when the cost model is set to have three cost groups whose range are 0-100, 100-1K, and the rest. We also assume that the minimum measurement points for each group is three. When the recorded measurement points of each cost group exceed the three, CPU cost model starts to generate the polynomial graph for each group, and finally it has three distinct performance model. The number of measurement points, total group counts and cost range of each group are hyper-parameters of CPU cost model that users can define in initial stage. As the total number of groups and measurement points increases, the accuracy of the entire cost model becomes high, while the time required for cost model generation increases. By making this trade-off relationship adjustable, our CPU cost model can generate the optimal cost model considering the current user's situation.

❸ *Online fine-tuning stage*: Since the initial cost ranges are determined by the hyper-parameters set by users, the value may or may not work well with current CPU system. To handle this issue, we propose an online fine-tuning method that adjusts the cost ranges to minimize the total amount of errors between the polynomial models and the measurements. The fine-tuning starts from the first two cost groups. As represented in the red box of Figure 7-(3), It first selects the closest point to the borderline from each group, i.e., the largest from the first group and the smallest from the second group and moves the borderline either to the left to let the second group include the largest of the first group (blue: ①) or to the right to let the first group include the smallest of the second group (green: ②). For each scenario, the CPU cost model regenerates the polynomial curves and recalculates the errors. Based on the errors, it decides to move the borderline toward where the total error decreases. Once it decides the direction, it repeats the above borderline moving process until the total error does not decrease anymore.

❹ *Final stage*: The cost model applies this range adjustment to all borderlines in ascending order. Then, the CPU cost model can generate the final cost models for current CPU system.

Figure 7-(b) shows the changes in accuracy by the proposed fine-tuning method when the number of measurements in each cost group is set to 4 and 14. We initially set PostgreSQL+ to have three cost groups whose range are 0-3K, 3K-300K, and the rest. The experiment is conducted on the Intel Xeon Gold 6226R CPU. When the polynomial model of each cost range is generated by only four measurements, the cost model shows 21.58% error on average and there was no improvement with the fine-tuning due to the small number of measurements. On the other hand, the model's average error is measured 14.54% when the number of measurements is 14. With the online fine-tuning method, the error is further reduced to 12.97%, which is 1.12× improved. Although the online fine-tuning requires additional time in range adjustment and polynomial fitting, it does not affect the overall performance because it happens only once during the system bring-up time. It is worthwhile because the accurate performance modeling is the key in PostgreSQL+'s runtime offloading decision.

3) OFFLOADING ACCURACY

Based on the estimated times from our cost models, the predictor finally determines whether it offloads the query to the SmartSSD executor or not. Due to our sophisticated performance models, the predictor successfully offloads

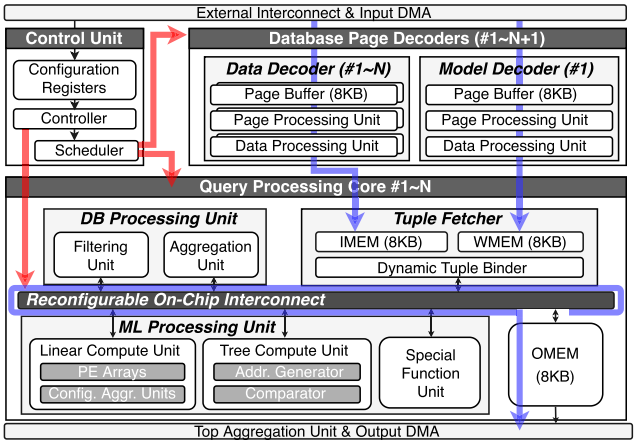


FIGURE 8. Overall i-DPA microarchitecture (red and blue line represent the control and data path, respectively).

169 out of 176 micro-benchmarks queries to the optimal hardware backend, yielding 96% accuracy.

V. HARDWARE FOR TRINITY

The hardware accelerator implemented in the FPGA of SmartSSD processes the queries offloaded by PostgreSQL+. For taking advantage of in-storage acceleration and enabling efficient in-DBMS ML inference acceleration, following observations are considered in hardware design.

In-DBMS ML Query Characteristic: We first analyze the in-database ML queries supported by the MADlib to define the operations that the hardware accelerator should support. Although MADlib supports various ML models such as linear/logistic regression, k-nearest neighbors, and tree methods, we find that there are two primary operations: linear and tree-based operations. By covering these two ML operation types, the hardware accelerator can handle four out of six model categories in the supervised learning section of MADlib. Moreover, among various relational operations (i.e., sorting, join, filtering and aggregation), we confirm that filtering and aggregation operations are appropriate to support in the SmartSSD. Given that the SmartSSD has limited DRAM capacity and same in/out bandwidth, the other operations like sorting, join, and group-by are not suitable because they require a larger output bandwidth or a large DRAM capacity to store partial results during the computation. Based on these observations, our custom hardware accelerator is finally designed to support linear and tree-based ML operations with filter- and aggregate-based relational operations.

3-levels of Parallelism: ML-based advanced data analytic has huge computations with large input data table, suggesting the need for a parallelization. We confirm that there exist three different levels of parallelism that hardware can utilize in query processing. First, since the database stores the data table with the unit size of the page (8KB), the multiple pages can be processed in parallel (page-level parallelism). In addition, as the single page includes multiple tuple data and there are no dependency between the input data in

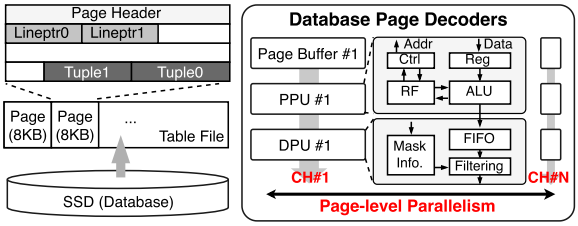


FIGURE 9. Example of page layout and block diagram of database page decoder.

ML inference, multiple tuple data can also be processed simultaneously (tuple-level parallelism). Lastly, task-level parallelism is possible if input data are pipelined across the computing units. The proposed hardware accelerator utilizes all these three different levels of parallelism to process the query faster.

Pre-Determined Page Format: We confirm that all the pages in DBMS have its own pre-determined layout format. The host CPU performs this page decoding in the conventional CPU-based DBMS platform for query processing, but since the database pages stored in the storage device need to be directly processed on the FPGA in our SmartSSD, the hardware should be able to decode them without the help of CPU to keep bandwidth benefit of in-storage processing.

A. IN-DATABASE PROCESSING ACCELERATOR

Based on these, we present the in-Database Processing Accelerator (i-DPA) that can accelerate the in-database ML queries with three key ideas. First, i-DPA employs database page decoder to fully leverage the bandwidth benefit of in-storage processing and accelerate the decoding process with parallel compute units. Second, i-DPA utilizes dynamic tuple binding to enhance the number of tuples that run in parallel and maintain the high hardware utilization for various query conditions. Lastly, i-DPA processes the tuple data with streaming-wise pipelined operation. It allows i-DPA to simultaneously process multiple tuples in parallel with its heterogeneous computing units. Figure 8 shows the overall architecture of i-DPA. It consists of control unit, $N+1$ database page decoders, and N query processing cores. i-DPA processes input tuple data in a streaming fashion across the heterogeneous computing units and the final results are transferred back to the host through the output direct memory access (DMA) module of the top aggregation unit. In the following section, we explain the details of microarchitecture.

B. MICROARCHITECTURE

1) CONTROL UNIT

The control unit controls the overall execution of i-DPA. It consists of configurations registers, controller and scheduler. When i-DPA receives start signal from the host, the configuration register is first set to the 24-byte metadata (see Figure 5) received from the host. Then, the controller decodes each operation and conditional field to determine the hardware configuration. Especially, the routing

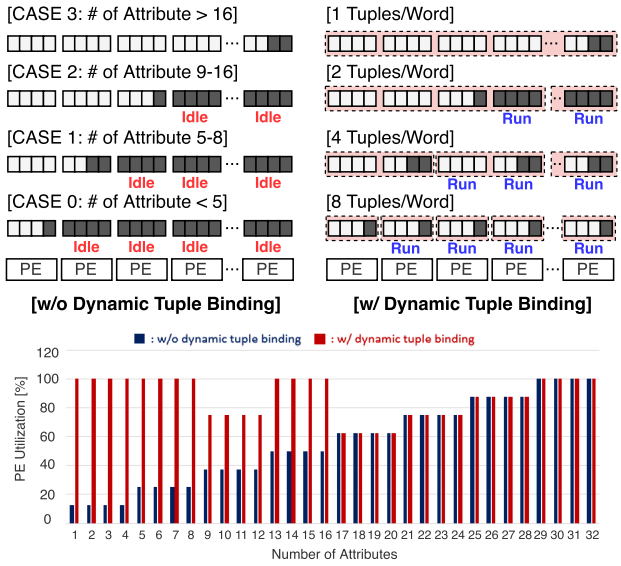


FIGURE 10. Dynamic tuple binding.

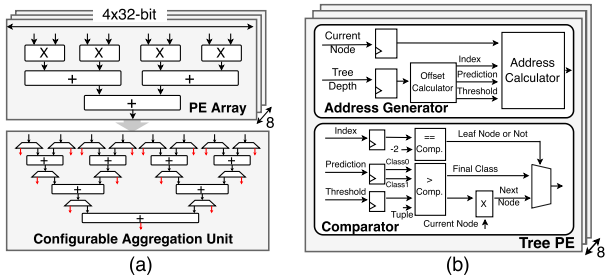


FIGURE 11. ML compute units: (a) Linear compute unit (b) Tree compute unit.

of reconfigurable on-chip interconnect is configured based on the relational and ML operation code field. The controller figures out the query's entire operation flow by decoding 9 bits of operation code and selects all computing units to process the current query. Based on this, it routes the selected computing units in a right order by changing the connection of switches in the on-chip interconnect. It allows i-DPA to support various queries having different execution flow. The other hardware configurations like the number of iteration that core to run and specific conditional values of several computing units are also determined by the controller. With these hardware configurations, the scheduler selects the proper finite state machine to process the current query and controls overall execution of i-DPA. Considering that instructions for data analytics can be complex and large due to the various computations and dataflows of query processing, our 24-byte meta-data with hardware-side interpreting solution can significantly reduce the time overhead for generating and transferring the instructions.

2) DATABASE PAGE DECODER

The database page decoder loads database pages to i-DPA's page buffers and extracts the tuple data from them by decoding the DBMS-formatted page. This tuple extraction offloading gives Trinity the bandwidth benefit of in-storage processing and the opportunity to accelerate the page decoding. Figure 9 shows the PostgreSQL;s page layout and the detailed block diagram of the proposed database page decoder. The page layout consists of three main blocks: page header, line pointer array, and tuple array. The page header contains general page information, and the line pointer array contains the offset address and the length of each tuple data. The tuple array is a set of actual data and it also has a structured tuple header having tuple information and the offset to the start of raw tuple data. The page processing unit (PPU) of database page decoder extracts raw tuple data with two decoding stages: 1) page header processing and 2) tuple header processing. To this end, it includes the register files (RF) to store temporal extracted data and arithmetic units (ALU) to calculate address and offset of the tuple data. Once the 8KB of page data is loaded through the input DMA module of PPU, it starts to extract the start pointer of the free space from the page header to know the number of line pointers in the page. Then, PPU brings the line pointer sequentially from the page buffer and calculates the address and size information of each tuple data using internal arithmetic units. Lastly, PPU consecutively brings the tuple data and extracts the raw tuple data by calculating the offset of the raw tuple data based on the information of the tuple header. This decoding process is pipelined until all raw tuple data are extracted from the page. However, as an input query does not always need all the attributes of tuple data, the data processing unit (DPU) is responsible for reconstructing the tuple data to contain only the necessary attributes for actual processing. It receives the raw tuple data from the PPU via FIFO and filters only the valid attributes based on the bit-vector of the attribute mask received from the host. The selected attributes are packed into the final tuple data and stored either in the input or weight memory of the processing core. The i-DPA has total $N + 1$ decoders. Among them, N decoders are allocated for the input data and one is used for the model weight data. Therefore, the multiple input pages can be decoded simultaneously through separated decode channel. Moreover, since the following N query processing cores process the input data from multiple pages in parallel while sharing the weight data, N pages can be processed simultaneously in i-DPA (page-level parallelism).

3) QUERY PROCESSING CORE

The query processing cores are the main computational blocks that execute the requested computations of the offloaded queries. It consists of two big ML and database processing units. The database processing unit includes filtering and aggregation unit, and the ML processing unit includes linear and tree compute unit with special function unit. These compute units are fully-connected by the reconfigurable on-chip interconnect that enables flexible data streaming among different compute units.

4) TUPLE FETCHER

The tuple fetcher has a 8KB of input and weight memory to store the input and model data. It prefetches multiple tuple data from each memory and binds them in a single word to increase the parallelism of the tuple processing. The i-DPA processes the tuple data based on the unit of a 1024-bit word, in which each tuple element is a 32-bit floating point number. However, the size of the entire tuple can be varied as each query can select a different number of attributes for processing. In this condition, if the query processing core can process only a fixed number of tuples without considering the tuple size, the compute utilization would decrease especially when the number of attributes is small. To solve this issue, the dynamic tuple binder dynamically adjusts a tuple packing density of a current 1024-bit word based on the number of attributes that each tuple has. Figure 10 shows the four cases of dynamic tuple binding. Based on the number of attributes, the 1024-bit word can be divided into multiple sub-words having different bit-width (CASE 0-3). Each sub-word includes a tuple, and the sub-words are processed in parallel. For this setting, dynamic tuple binder fetches a 1024-bit word from each of the input and weight memory, and distributes them into the sub-words according to the number of attributes. If the number of attributes is less than 5 (CASE 0), it aligns the read data into 8 tuples with zero padding. Likewise, if the number of attributes is 5-8 (CASE 1), 9-16 (CASE 2), and 17-32 (CASE 3), it aligns the data into 4 tuples, 2 tuples, and 1 tuple, respectively. Through the dynamic tuple binding, i-DPA can supply different numbers of tuples to the underlying compute units. As a result, dynamic tuple binding increases i-DPA's tuple-level parallelism up to 8× when the tuple size becomes smaller and increases its average hardware utilization from 56.25% to 87.5% when i-DPA processes various queries having different number of attributes.

5) LINEAR COMPUTE UNIT

The linear compute unit performs dot products between the input and weight tuple. As shown in Figure 11-(a), it consists of eight processing element (PE) arrays and a configurable aggregation unit, in which a single PE array includes four PEs and an adder tree. Each PE can multiply two 32-bit floating-point tuple elements, and the adder tree accumulates all the multiplication results of the array. The configurable aggregation unit consists of multiplexer array and multiple adders. It configures the proper aggregation links based on the number of tuples that are processed in parallel and performs additional aggregation across the PE arrays.

6) TREE COMPUTE UNIT

As shown in Figure 11-(b), the tree compute unit is composed of eight tree PEs, in which each tree PE comprises an address generator and comparator. For tree traversing, the address generator calculates the memory address to access current node's data information (i.e., index, threshold, prediction) and load them from weight buffer. The comparator compares the input data value against the current node's threshold value and decides the next node to traverse based on this result. However, if the comparator determines that the current node is a leaf node, it returns the final classification result and finishes the process. The tree compute unit repeats the above process for all input tuples.

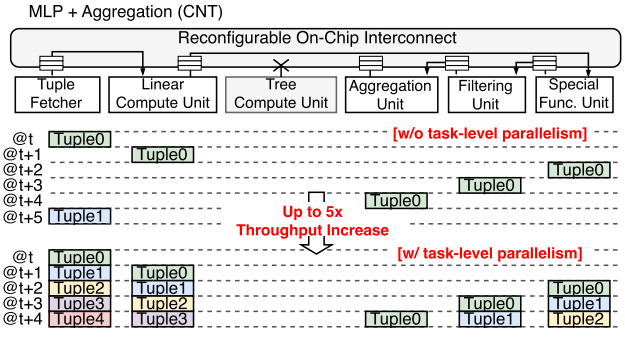


FIGURE 12. i-DPA's task-level parallelism.

7) SPECIAL FUNCTION UNIT

The special function compute unit handles the sub-operations required for ML inference. It contains various floating-point arithmetic units that can support addition, subtraction, multiplication, and exponential operation. For the logistic regression, it performs exponential operation on the dot product results to compute the final possibility and for multi-layer perceptron, it performs input normalization and activation functions.

8) DB COMPUTE UNIT

The relational compute unit comprises filtering and aggregation unit. The parameters like threshold value and filtering/aggregate column are received from the configuration registers. For the filtering operations, the filtering unit determines whether it processes the current input tuple or not based on the filtering value and column information. If it decides not to process the current tuple, it deactivates the entire unit using the valid signal. The aggregation unit performs various aggregate operations such as count, max, min, avg, and sum. It first extracts the selected column data and performs aggregate operation on all the incoming tuples, while storing the intermediate results in the internal registers.

9) TASK-LEVEL PARALLELISM

These compute units can process input tuple data independently and each compute unit is designed to process the nest tuple right after the current tuple. For seamless data streaming, each compute unit is connected with reconfigurable on-chip interconnect via first-in-first-out (FIFO) and employs double buffering to prevent stalls. As shown in Figure 12, it makes different tuple data be processed in parallel across the different computing units (i.e., task-level pipelining). Consequently, the i-DPA can achieve maximum 5× higher throughput than conventional tuple-by-tuple processing.

[Apache MADlib In-DBMS ML Queries]

Q1: Linregr
```
SELECT madlib.linregr_predict(
    [model.c1, ..., model.cn],
    [input.a1, ..., input.an])
FROM input, model;
```

Q2: Linregr + Filtering
```
SELECT madlib.linregr_predict(
    [model.c1, ..., model.cn],
    [input.a1, ..., input.an])
FROM input, model
WHERE input.c1 > 0.5;
```

Q3: Linregr + Aggregation
```
SELECT AGG (linregr)
FROM (
  SELECT madlib.linregr_predict(
    [model.c1, ..., model.cn],
    [input.a1, ..., input.an])
  FROM input, model) AS linregr
```

Q4: Linregr + Filtering + Agg
```
SELECT AGG (linregr)
FROM (
  SELECT madlib.linregr_predict(
    [model.c1, ..., model.cn],
    [input.a1, ..., input.an])
  WHERE input.c1 > 0.5
  FROM input, model) AS linregr
```

Q5: Logregr
```
SELECT madlib.logregr_predict(
    [model.c1, ..., model.cn],
    [input.a1, ..., input.an])
FROM input, model;
```

Q6: Logregr + Filtering
```
SELECT madlib.logregr_predict(
    [model.c1, ..., model.cn],
    [input.a1, ..., input.an])
FROM input, model
WHERE input.c1 > 0.5;
```

Q7: Logregr + Aggregation
```
SELECT AGG (logregr)
FROM (
  SELECT madlib.logregr_predict(
    [model.c1, ..., model.cn],
    [input.a1, ..., input.an])
  FROM input, model) AS logregr
```

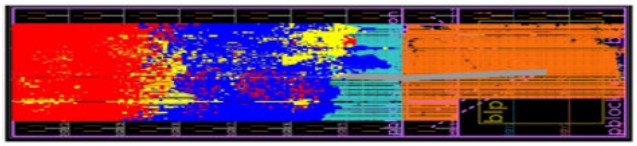
Q8: Logregr + Filtering + Agg
```
SELECT AGG (logregr)
FROM (
  SELECT madlib.logregr_predict(
    [model.c1, ..., model.cn],
    [input.a1, ..., input.an])
  WHERE input.c1 > 0.5
  FROM input, model) AS logregr
```

Q9: SVM
```
SELECT madlib.svm_predict(
    model,
    input,
    id,
    output);
```

Q10: MLP
```
SELECT madlib.mlp_predict(
    model,
    input,
    id,
    output);
```

Q11: Tree
```
SELECT madlib.tree_predict(
    model,
    input,
    output);
```

FIGURE 13. In-DBMS ML queries to evaluate Trinity.



	Specifications				
FPGA	Xilinx Kintex UltraScale+				
Frequency	170MHz				
	Resource Utilization				
	LUT	FF	BRAM	URAM	DSP
Interface	128665	183077	311.5	8	9
Core #0	102672	92142	15	32	342
Core #1	110092	92437	15	32	342
Others	10740	9757	22	0	10
Utilization	67.4%	36.1%	46.9%	56.3%	35.7%

FIGURE 14. Implementation result of Trinity's i-DPA.

VI. EXPERIMENTAL EVALUATION

In this section, we evaluate the benefits of Trinity on analytics queries that invoke ML inference over real-world datasets. First, we compare Trinity with the baseline CPU-based PostgreSQL (Section VI-A). Then, throughout a set of micro-experiments, we show the effect of Trinity's near-data processing (Section VI-B) and give detailed analysis of Trinity in various aspects (e.g., dataset complexity, data operations, ML models) in Section VI-C along with above terabyte database size (Section VI-D), and multiple SmartSSDs (Section VI-E). We also compare Trinity against the GPU-based system (Section VI-F).

System Setup: We use two in-database ML platforms for evaluation: the baseline (CPU-based) DBMS and Trinity. We use PostgreSQL v12.6 [20] for the baseline DBMS and integrate MADlib v1.17.0 [10] in both platforms. The CPU-based platform runs on the server with two Intel Xeon Gold 6226R CPUs (2.9GHz, 32 threads), 192GB DIMM, and 1TB SSD. The Trinity system runs on the computational storage server that has two Intel Xeon Silver 4210 CPUs (2.2GHz, 10 threads), 156GB DIMM, and SmartSSD.

Datasets: We chose four datasets for evaluation, which have different numbers of attributes: Higgs [21] (28 attributes), Algerian Forest [22] (13 attributes), Wilt [23] (5 attributes), and Haberman's survival [24] (3 attributes). All the datasets are publicly available from the UCI machine learning repository [25]. For a fair comparison, we replicate each of the datasets to have the same tuple numbers: 0.5k, 11M, 25M, or 110M. The database sizes vary from 32KB to 15.3GB.

Queries: Figure 13 lists the 11 queries we used in evaluation. All of these queries are real in-database ML queries supported by the Apache MADlib, covering a diverse range of ML algorithms: linear regression (Linregr), logistic regression (Logregr), support vector machine (SVM), multi-layer perceptron (MLP), and tree model (Tree). For the mixed queries having both ML and relational operations, we also include the queries having filtering and aggregation operation on top of the linear and logistic regression models. The linear/logistic regression and SVM workload require the same number of model parameters as the attribute number of input tuple. In the case of MLP workload, the default model configuration is three fully-connected layers, where the hidden layer has the same size as the input layer (i.e., Higgs: 28-28-2, Algerian forest: 13-13-2, Wilt: 5-5-2, Haberman: 3-3-2). The default tree depth is set to four, with a total of 15 tree nodes.

FPGA Resource Utilization: Figure 14 shows the FPGA resource utilization for the Trinity's i-DPA implementation on the SmartSSD. As the proposed architecture allows multiple query processing cores for higher throughput, we successfully integrated two cores with the given FPGA resource. The current design is bounded by the LUT logic resource with more than 65% utilization at 170MHz operating frequency.

A. END-TO-END TRINITY EVALUATION

In this section, we evaluate Trinity in comparison with the baseline CPU-based DBMS to verify its effectiveness. We measure the query processing time of 11 in-DBMS ML queries on the four datasets when the number of tuples varies among 0.5k, 11M, 25M, and 110M.

1) END-TO-END SYSTEM PERFORMANCE

Figure 15 shows the end-to-end query processing times of the CPU-based DBMS and Trinity. The measured latency

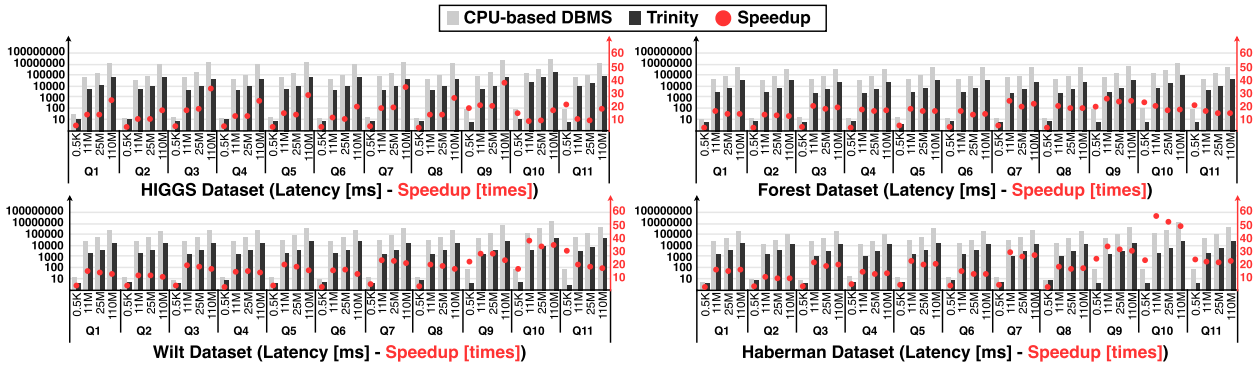


FIGURE 15. End-to-end query processing times and speedups of Trinity over the baseline.

in y-axis is represented in log scale. The Trinity's query processing time includes the time spent on the host side by the full stack of PostgreSQL+. We confirm that the host time is relatively constant throughout the experiments to 2.96ms on average. It accounts for 53.7% of the total time at the smallest tuple number (i.e., 0.5K), but occupies less than 0.1% in other cases with more than 10M tuples. Except when the tuple number is too small, the overall latency is proportional to the number of tuples.

The CPU-based DBMS shows a similar trend with Trinity. It shows faster processing speed as the dataset becomes smaller and simpler with less attributes. It takes a longer time to process a larger database with more tuples and more complex ML models. One big difference is that the CPU-based DBMS shows performance degradation when the dataset size becomes large due to the caching inefficiency, whereas Trinity shows a linear performance scaling to the database size. In the CPU-based DBMS, it takes 20.8× more latency in processing 110M tuples than the case of 11M tuples on the Higgs dataset. It is 2.08× slower than the ideal scaling from the 11M tuple size. In addition, we confirm that the query with aggregate operations takes a longer time in the CPU-based DBMS compared to the query having only ML operations. Unlike in Trinity, the CPU-based DBMS does not have parallel processing units for each computation. Therefore, the increase in computation by aggregate operations negatively affects the CPU's query processing time. We also observe that the CPU-based DBMS takes longer time in Q9, Q10, and Q11 than the other queries, especially when the database size is small. This is because the initial processing time for activating these queries on the MADlib (e.g., transforming to the executable queries) dominates the overall latency.

2) COMPARISON

Figure 15 also shows the speedup of Trinity compared to the CPU-based DBMS using red dots. The speedup varies from 0.85× to 57.18× and shows 15.21× faster query processing on average than the CPU-based DBMS. It is noteworthy that the Trinity shows better results over the CPU-based system even in the small size of database except only one case (i.e., Q8 in Haberman with 0.5k tuples).

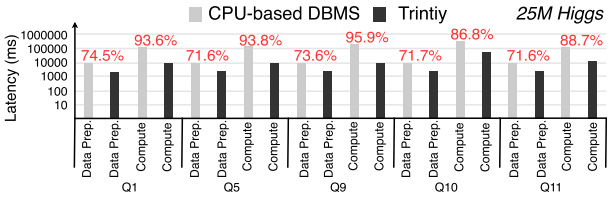


FIGURE 16. End-to-end time breakdown results.

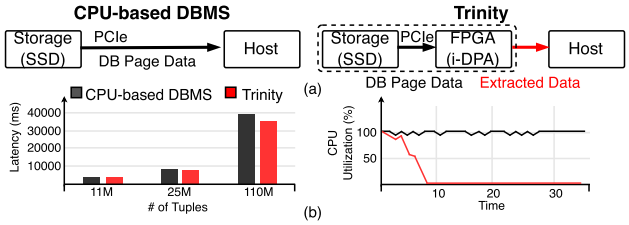


FIGURE 17. (a) System diagram of CPU-based DBMS and Trinity (b) Query processing time and host CPU utilization for performing basic scan operation.

Even though the CPU-based DBMS works well when the dataset becomes smaller and simpler, Trinity still outperforms by using abundant parallelism. We also check the time breakdown results for a detailed analysis. The latency of both systems can be divided into data preparation time for database movement with page decoding and compute time for processing given workloads. Trinity requires additional host time to operate SmartSSD, but it only occupies less than 10% of the entire processing time. As shown in Figure 16 (y-axis: log scale), the data preparation time and the compute time are largely reduced by 72.7% and 91.8% on average, respectively. As Trinity can reduce the data to the host by 92.8% in an MLP workload through near-data processing and decode multiple page simultaneously, the data preparation time can be significantly reduced. Similarly, the compute time can be significantly reduced due to the abundant parallelism of the i-DPA.

B. NEAR-DATA PROCESSING

Unlike the CPU-based DBMS, the Trinity's in-storage accelerator performs direct page decoding and other database operations on the data so that it can send only the extracted raw or processed data to the host CPU (Figure 17-(a)).

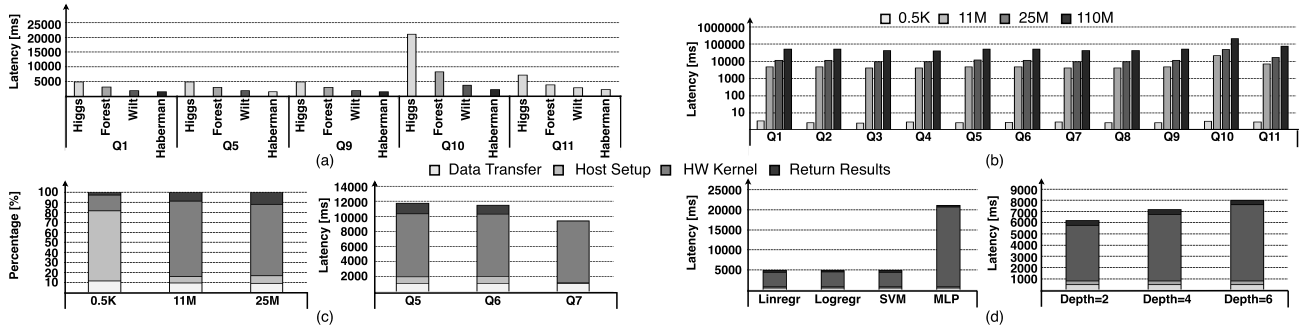


FIGURE 18. Micro-benchmarks results: (a) Query processing time for different datasets. (b) Latency variance according to the number of tuples. (c) Time breakdown when the number of tuples changes and when the relational operation is included in query. (d) Latency variance according to the model complexity.

It can both reduce the total amount of data to be sent to the host as well as save the host's computation resource. To verify these advantages, we run the full database table scan which reads all the raw data from the database. Figure 17-(b) shows each system;s query processing time (left) and the host CPU utilization (right). We use the Higgs dataset with varying tuple number from 11M to 110M. As a result, Trinity shows a 5.4%, 7.8%, and 9.7% less latency than those of the CPU-based DBMS when the number of tuples is 11M, 25M, and 110M, respectively. This is mainly because of the two reasons: 1) i-DPA's hardware decoding is faster than the CPU's and 2) the raw data size is decreased by 14% compared to the original table size so does the data transfer time. Considering that the data table scan achieves little data reduction compared to other operations such as ML and aggregation, 14% data reduction is a minimum gain that Trinity can achieve from near-data processing. In case of aggregation, Trinity can reduce the data traffic by more than 95%. Moreover, given that the scan operation is necessary for entire DBMS operations, this result implies that Trinity can achieve better performance even in unsupported operations such as sorting and join by utilizing SmartSSD as a page decoder in entire CPU-SmartSSD system. Another significant benefit is that the CPU usage of Trinity is significantly low due to the effect of offloading the decoding. Throughout the scanning 11M tuple data, the CPU-based DBMS shows an average of 98% CPU utilization while Trinity shows an average of 16.5%. We also observe that Trinity's CPU utilization converges to almost zero once it finishes SmartSSD's device setup. These freed CPU resources can be potentially used for other activities.

C. MICRO-BENCHMARKS

In this section, we analyze the measured results in terms of dataset complexity, data scalability, data operation, and model complexity.

1) DATASET COMPLEXITY

We check the query processing time of each ML model for the datasets having different attribute numbers. Figure 18 (a) shows the measurement results of the four datasets when the number of tuples is 11M. We do not depict the results from the other sizes as they have the same tendency. In this experiment, Trinity shows faster query processing time when the dataset becomes simpler with fewer attributes. This is because the dynamic tuple binding increases parallelism of the i-DPA and the accelerator can access fewer pages to process the same number of tuples when the dataset has a smaller number of attributes.

2) DATA SCALABILITY

Figure 18 (b) shows the query processing time of 11 queries when the number of tuples changes from 0.5k to 110M. We depict only the results of the Higgs dataset as a representative. In this graph, we confirm that the latency increases almost linearly when the number of tuples increases from 11M to 110M, but it does not scale in that way for small tuple sizes such as 0.5k. This is mainly because of the host overhead of using SmartSSD. As shown in Figure 18 (c)-(left), the dominant operation changes depending on tuple numbers. Since the host setup time is relatively constant regardless of the tuple number, the host operation dominates the entire latency when the tuple size is small, but the kernel execution time becomes dominant as the tuple size gets large. This variation makes Trinity to have linear relationship in latency only when the tuple size is big enough (i.e., > 0.1M).

3) DATA OPERATIONS

Figure 18 (c)-(right) depicts another time breakdown result showing how relational operations affect the overall performance. We plot the results of three queries that have the same ML operation but with different relational operations (Q5: nothing, Q6: filter, Q7: aggregate) for the Higgs dataset with 25M tuples. Trinity shows faster speed when the query includes the relational operations because the total output size to be transferred to the FPGA's DRAM and the host is reduced by discarding the input tuples or accumulating partial sums within the hardware. Moreover, as the small output size requires a small buffer size on the FPGA;s DRAM, the host setup time is also decreased.

4) ML MODEL COMPLEXITY

We also find out that the execution time has a dependency on model complexity. Figure 18 (d) depicts the latency

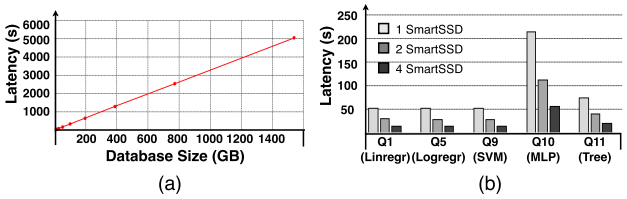


FIGURE 19. (a) Terabyte measurement results (b) Scaled-up performance.

results for various linear models and tree models (depth = 2, 4, 6) for the Higgs dataset with 11M tuples. Based on the results, we observe that the execution times of linear/logistic regression, and SVM are similar within the same dataset, as their computational loads are similar. However, if the computational load is significantly large, like the MLP model, the query processing time becomes long as well. We also confirm that the overall latency marginally increases as the tree depth becomes deeper.

D. BEYOND TERABYTE DATABASE

Trinity can handle datasets up to 3.84TB with a single SmartSSD, which is the capacity of SmartSSD device. Although we already show that the end-to-end query time with Trinity increases linearly with the dataset size, we try to confirm this result with expanding the dataset size up to the terabyte level. Figure 19-(a) shows the experimental results of the linear regression workload when the dataset size increases up to 1.5TB. Eventually, Trinity takes 5047.59s to process the 1.5TB database, and it still shows the linear increase in latency as the database size becomes above terabytes. Based on this result, we can estimate that Trinity will take 12893s to process the maximum 3.84TB of database. Considering the CPU-based DBMS tends to be much slower for a large database, e.g., hundreds of gigabytes, due to its inefficient memory sub-systems, Trinity will easily outperform it in the domain of terabyte databases.

E. TRINITY WITH MULTIPLE SMARTSSDS

As the database size increases, enterprise-level DBMSs optimize the database performance on a single node and scale out to a distributed storage system to satisfy a tight service-level-agreement. To this end, current CPU-based DBMS tries to use multiple threads to improve the single node performance, but the gain is often marginal. In addition, this excessive burden on the CPU hinders the scaling out to multiple nodes as the CPU is required to control network and storage devices. Trinity, on the other hand, is a better solution for performance scale-up in a single node and further scale-out. It can easily extend to have multiple SmartSSDs with low CPU maintenance due to its U.2 form factor. Figure 19-(b) shows the latency of the five different ML models in the Higgs dataset with 110M tuples when the number of SmartSSDs varies from 1 to 4. In order to utilize multiple SmartSSDs, we uniformly distribute database in each devices. As the number of devices becomes double and quadruple, the latency reduces by 1.85× and 3.66× on

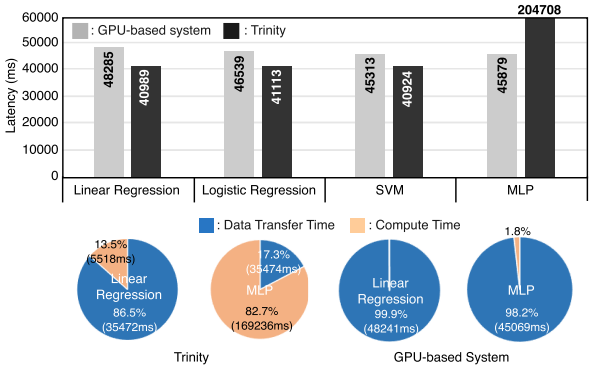


FIGURE 20. Performance comparison against GPU-based system with latency breakdown results.

average, respectively. This shows that Trinity achieves almost linear performance improvement in processing the same query on multiple devices. Comparing this result with the previous evaluation results, Trinity with four SmartSSDs achieves nearly 200× speed up over the CPU-based DBMS.

F. COMPARISON AGAINST GPU SYSTEM

Lastly, we compare the Trinity against the GPU-based system. Although MADlib has started to support GPU using the internal deep learning package since MADlib v1.20.0, we observe that it is difficult to use for the baseline of the GPU system due to the lack of optimization. When we process the MLP query with 0.75M Higgs dataset using the deep learning package of MADlib, it shows 2.18× slower speed than the conventional CPU-based system. Therefore, in this experiment, we measure the database transfer time from SSD to the host and add the time required for GPU processing. We use TITAN RTX GPU for measuring the GPU processing time. Figure 20 shows the latency of two different systems when we perform linear regression, logistic regression, and MLP workload with 110M Higgs dataset. As shown in the graph, Trinity shows the higher performance than the GPU-based system in the models that require low computational amount like linear regression, logistic regression and SVM. It shows 1.14× faster processing speed on average than the GPU-based system. However, GPU shows 4.4× higher processing speed than the Trinity for the MLP model that requires high computation amount. For the detailed analysis, we also represent the latency breakdown of two different systems for the representative linear regression and MLP model. In terms of data transfer time from storage to hardware accelerator, Trinity shows 1.3× faster speed than the GPU-based system on average due to the benefit of SmartSSD's near-data processing. However, GPU shows a much faster kernel execution speed (almost 150×) than the Trinity. This is because GPU has abundant hardware resources having over 100× more significant computing units and 42× higher DRAM bandwidth. However, SmartSSD's lack of hardware resources can be compensated through the better scalability of SmartSSD. SmartSSD is easier to scale up than GPU due to its U.2 storage form factor

(i.e., 48 SmartSSDs vs 4 GPUs in a 2U server). When we project this 12× higher scalability in previous measurement results, Trinity can achieve 14.13× and 2.69× faster query processing speed in both ML models than the GPU-based system.

VII. DISCUSSION

A. ADVANCED ANALYSIS

1) POWER EVALUATION

To evaluate the Trinity;s energy efficiency, we conducted power measurements on SmartSSD. We confirm that SmartSSD consumes 20.9W power on average for processing our in-DBMS ML queries. This makes Trinity consume almost 2.5× higher power consumption than the conventional CPU system equipped with the SSD having same 3.84TB capacity and external bandwidth. Nevertheless, as Trinity shows 15.21× higher query processing speed on average than the conventional CPU-based system, it can achieve nearly 6.1× higher energy-efficiency compared to the CPU-based system. Furthermore, GPU (i.e., TITAN RTX) shows roughly 14× larger power consumption than the SmartSSD. When we apply this to our GPU evaluation results, Trinity can achieve 7.01× and 1.39× higher energy-efficiency for data-intensive and compute-intensive workloads, respectively, the GPU-based system.

2) COST EVALUATION

For the datacenter enterprises, Total Cost of Ownership (TCO) is also an important metric for system implementation considerations. In this context, we compare the Trinity to traditional CPU-based DBMS in terms of performance-per-dollar. Upon surveying publicly available pricing data for SSDs and SmartSSD, we confirm that SmartSSD is approximately 5.5× more expensive than its SSD counterparts with equivalent storage capacity and bandwidth. Based on this, we incorporate this cost information in our evaluation results. Consequently, Trinity shows 2.77× higher performance-per-dollar ratio than CPU-based DBMS. As Trinity maintains linear performance improvement as the number of SmartSSD increases, this advantage remains consistent even in scaled-up system. In summary, Trinity can outperform the traditional CPU-based DBMS in performance-per-dollar even if the additional costs are required to introduce the new hardware in DBMS.

3) RESOURCE PERFORMANCE TRADE OFF

Trinity employs a SmartSSD as a new hardware platform for processing data analytics query processing. By integrating a new hardware backend, Trinity can reduce host CPU and memory usage with achieving faster query processing speed. For example, CPU utilization decreases from 98% to an average of 16.5%, and the average query processing speed increases 15.21×. Therefore, resource in SmartSSD gives significant impact on overall system performance. The key elements are FPGA resources, DRAM bandwidth, and storage-FPGA internal bandwidth. Increasing FPGA resources and DRAM bandwidth reduces the overall processing time of i-DPA. Increasing FPGA resources enables I-DPA to integrate more query processing cores within a single device, and boosting DRAM bandwidth enhances I-DPA;s speed in accessing page data in DRAM. As each core requires different pages and processes them independently in parallel, almost linear performance improvements can be achieved.In contrast, increasing storage-FPGA internal bandwidth reduces data transfer time of SmartSSD. Greater internal bandwidth minimizes data transfer time between storage and FPGA;s local DRAM, maximizing the benefits of neat-storage processing. However, practical constraints related to strict storage form factor and cost limitations may restrict these enhancements in SmartSSD implementation.

B. TRINTIY'S LIMITATION AND FUTURE WORKS

1) SYSTEM SCALABILITY

Due to the U.2 storage form factor of SmartSSD, it is easy to scale-up to 24 or 48 cards using a commercially available storage server. This scalability advantage is one of the key features of SmartSSD. In the scenario of employing 48 SmartSSDs, the server can leverage 96 query processing cores with 184.32TB storage and 192GB DRAM. In Section VI-E, we confirm that linear performance improvement can be achieved when each device has model data locally, and the substantial input data is evenly distributed across all devices. However, in reality, data can be located in storage with a variety of ways, and it can affect in overall system;s performance. For example, if model data is distributed across devices to maximize storage efficiency and the system requires the data transfer among multiple devices to access required data, the large data transfer costs limit the Trinity;s ideal linear performance improvement. To address this challenge, advanced scheduling methods are required to hide the data movement latency among multiple devices. Furthermore, efficient hardware resource management and workload allocation strategies are required for successful system scale-up. These aspects are significantly important, but are left for future works aimed at constructing an advanced Trinity system.

2) FAULT TOLERANCE AND SECURITY

Fault tolerance and security are important factors for a robust DBMS. However, the introduction of SmartSSD into DBMS for faster query processing speed introduces potential challenges in maintaining DBMS;s high fault tolerance and security. As Trinity inherently supports full backward compatibility of the existing CPU pipeline, it can address the unexpected the hardware;s failure by detouring to CPU pipeline. However, regarding security, the raw page data movement from storage to SmartSSD;s FPGA can create a vulnerability in data security. Addressing these issues are important in real-world system implementation. To enhance fault tolerance, implementing checkpoint mechanisms for storing intermediate processing results can significantly reduce hardware failure overhead. For increased security,

we can directly use encrypted pages in SmartSSD. This approach requires the integration of additional hardware logic for decryption. Additionally, exploring the implementation of homomorphic encryption, which enables data processing while maintaining encryption, stands as a promising strategy to mitigate data security vulnerabilities. These are left for the further works for maturation of the Trinity system.

3) LONG-TERM MAINTENANCE

ML models for data analytics are continuously advancing. MADlib, utilized in Trinity, also consistently deploys the updated version to support the latest ML algorithms. While linear and tree operations dominate the modern ML algorithms for advanced data analytics, there is a need to accommodate new operations as time progresses. This entails two key modifications. First, a new compute unit should be designed and integrated in i-DPA. It requires the expertise in hardware description languages like SystemVerilog. However, I-DPA;s heterogeneous computing unit architectures and separate control and data unit minimizes the complexity of hardware modification. In addition, PostgreSQL+ should be modified to support the new operations. The SmartSSD cost model should include the hardware information for this operation. Although the introduction of SmartSSD in DBMS requires additional modification to support new operations or update the system, the existing designs of i-DPA and PostgreSQL+ minimize this complexity through straightforward software and hardware implementation.

4) OTHERS

Trinity's current primary focus is on demonstrating the benefits of computational storage devices within the ML-based advanced data analytics. As a result, Trinity is still in its early development stage and holds substantial potential for future enhancements, even excluding the above scalability, fault tolerance, security, and long-term maintenance. At present, Trinity's functionality is limited to specific operations. However, real-world analytics queries often have various operations, including sorting, join, matrix-matrix multiplication, and convolutional operations. Expanding trinity's capabilities to accommodate a wider range of ML and database queries is a necessary step to increase the versatility of Trinity. In addition, Trinity currently concentrates on ML inference. Nevertheless, there is a growing interest in conducting ML model training within a database management system for secure advanced data analytics. Given that trinity's CPU-SmartSSD system can effectively handle the increased computation and memory requirements of ML training through near-storage processing, extending Trinity's scope to include ML model training can also be a valuable prospect.

VIII. RELATED WORKS

A. HARDWARE ACCELERATION FOR DATABASE SYSTEMS

Many recent works [26], [30], [32], [34], [35], [36], [40], [47], [49], [57], [59], [63], [64], [65], [66] have tried to accelerate relational database queries with ASIC, FPGA, and GPU. Q100 [64] and Andrea et al. [49] are typical database accelerator composed of ASIC tiles for each database primitive. Many prior works [30], [32], [35], [59], [63] have accelerated several relational operations using various hardware architectures on FPGA. BlazingSQL [26], He et al. [36], and TCUDB [66] similarly use a GPU as a SQL engine with modifying relational operators into tensor computations. However, these previous works suffer from unnecessary data copies to the host due to their out-of-the-wire-architecture. Several works [33], [34], [40], [47], [56], [57], [65] have tried to accelerate the database operations using near-data processing. Mondrian [34] and Tiago et al. [40] use Hybrid Memory Cube (HMC), integrating SIMD units for processing queries. AQUOMAN [65] presents a general analytic query processor that can directly access the NAND flash arrays. Lee et al. [47] and NASCENT [57] also use the SmartSSD for accelerating filtering and sort operations. However, all of above works only focus on the legacy database operations, not including ML operations for advanced analytics. Their dedicated computing units for database operations and general CPU cores cannot handle ML operations and not enough to handle compute intensive ML operations.

B. IN-DATABASE ML ACCELERATION

Several works [28], [39], [50], [61] have tried to accelerate the ML-driven advanced analytics. Zahra et al. [28] and ColumnML [39] try to accelerate certain ML algorithms in DBMS using the FPGA. On the other hand, DAnA [50] and Gorgon [61] propose general hardware architectures that can support various ML algorithms. Although DAnA have assumed the direct page access from FPGA to database buffer pools, it only achieves a 4× speedup over CPU-based DBMS. Gorgon can achieve a high performance by using ASIC's abundant hardware resources, but its ideal performance is limited by up to 80% of its peak performance due to the IO bottleneck. Moreover, these accelerators have no system-level considerations to be integrated in real database systems. To our knowledge, Trinity is the first work that presents not only the in-database ML accelerator but also the full software stack to utilize a new hardware backend in a real DBMS. Additionally, not for a data analytics, but there are several works [48], [51] to accelerate the ML algorithms with near-storage processing. However, their direct NAND flash access system architecture is hard to justify due to the SSD's strict form factor and their array-type hardware architecture are not suitable for in-database ML acceleration because majority of in-database ML queries is traditional ML algorithms not deep learning algorithms like convolutional neural network.

IX. CONCLUSION

In this paper, we present Trinity, an in-database, in-storage platform that integrates ML pipelines into a popular DBMS on top of the CSD device. Starting with the development of an

effective software stack that can dynamically determine the optimal hardware backends, we have developed the hardware accelerator that can directly unpack the storage's data pages and perform analytics operations with parallel computing units. Finally, Trinity improves the end-to-end performance of analytics queries by 15.21× on average and up to 57.18x compared to the conventional CPU-based DBMS platform.

REFERENCES

[1] *Microsoft SQL Server*. [Online]. Available: https://www.microsoft.com/en-us/sql-server

[2] *Amazon Redshift*. [Online]. Available: https://aws.amazon.com/redshift

[3] *Google BigQuery*. [Online]. Available: https://cloud.google.com/bigquery

[4] *IMB Db2 Database*. [Online]. Available: https://www.ibm.com/products/db2-database

[5] *Oracle*. [Online]. Available: https://www.oracle.com/index.html

[6] *Microsoft SQL Machine Learning*. [Online]. Available: https://docs.microsoft.com/en-us/sql/machine-learning/?view=sql-server-ver15

[7] *Google BigQuery Machine Learning*. [Online]. Available: https://cloud.google.com/bigquery-ml/docs

[8] Amazon Redshift Machine Learning. [Online]. Available: https://aws.amazon.com/blogs/big-data/create-train-and-deploy-machine-learning-models-in-amazon-redshift-using-sql-with-amazon-redshift-ml/

[9] *DBMS Benefit*. [Online]. Available: https://learning.shine.com/talenteconomy/ta-technology/advantages-of-database-management-system/

[10] *Apache MADlib*. [Online]. Available: https://madlib.apache.org/

[11] *Field Programmable Gate Array (FPGA)*. [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html

[12] *AWS Amazon EC2 F1 Instances for Educators*. [Online]. Available: https://aws.amazon.com/education/F1-instances-for-educators/

[13] *PostgreSQL DBMS*. [Online]. Available: https://www.postgresql.org/

[14] *SmartSSD*. [Online]. Available: https://samsungsemiconductor-us.com/smartssd/

[15] *Greenplum Database*. [Online]. Available: https://greenplum.org/

[16] *Oracle Machine Learning for SQL*. [Online]. Available: https://www.oracle.com/database/technologies/datawarehouse-bigdata/oml4sql.html

[17] *The Netezza FAST Engines Framework*. [Online]. Available: http://www.monash.com/uploads/netezza-fpga.pdf

[18] *Xilinx XRT*. [Online]. Available: https://xilinx.github.io/XRT/2023.1/html/index.html

[19] *Peer-to-Peer Communication (P2P)*. [Online]. Available: https://xilinx.github.io/XRT/master/html/p2p.html

[20] *PostgreSQL 12.6 Document*. [Online]. Available: https://www.postgresql.org/docs/12/release-12-6.html

[21] *HIGGS Dataset*. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/HIGGS

[22] *Algerian Forest Fires Dataset*. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Algerian+Forest+Fires+Dataset++

[23] *Wilt Dataset*. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/wilt

[24] *Haberman's Survival Dataset*. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/haberman's+survival

[25] *UCI Machine Learning Repository*. [Online]. Available: https://archive.ics.uci.edu/ml/index.php

[26] *Blazing SQL*. [Online]. Available: https://blazingsql.com/

[27] Y. Anzai, *Pattern Recognition and Machine Learning*. Amsterdam, The Netherlands: Elsevier, 2012.

[28] Z. Azad, R. Sen, K. Park, and A. Joshi, "Hardware acceleration for DBMS machine learning scoring: Is it worth the overheads?" in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2021, pp. 243–253.

[29] L. Cai and Y. Zhu, "The challenges of data quality and data quality assessment in the big data era," *Data Sci. J.*, vol. 14, pp. 1–10, May 2015.

[30] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, Feb. 2014, pp. 151–160.

[31] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–13.

[32] R. Chen and V. K. Prasanna, "Accelerating equi-join on a CPU-FPGA heterogeneous platform," in *Proc. IEEE 24th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2016, pp. 212–219.

[33] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart SSDs: Opportunities and challenges," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*. New York, NY, USA: Association for Computing Machinery, Jun. 2013, pp. 1221–1230, doi: 10.1145/2463676.2465295.

[34] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The Mondrian data engine," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 639–651.

[35] Z. F. Eryilmaz, A. Kakaraparthy, J. M. Patel, R. Sen, and K. Park, "FPGA for aggregate processing: The good, the bad, and the ugly," in *Proc. IEEE 37th Int. Conf. Data Eng. (ICDE)*, Apr. 2021, pp. 1044–1055.

[36] D. He, S. Nakandala, D. Banda, R. Sen, K. Saur, K. Park, C. Curino, J. Camacho-Rodríguez, K. Karanasos, and M. Interlandi, "Query processing on tensor computation runtimes," 2022, *arXiv:2203.01877*.

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

[38] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 1–12.

[39] K. Kara, K. Eguro, C. Zhang, and G. Alonso, "ColumnML: Column-store machine learning with on-the-fly data transformation," *Proc. VLDB Endowment*, vol. 12, no. 4, pp. 348–361, Dec. 2018.

[40] T. R. Kepe, E. C. de Almeida, and M. A. Z. Alves, "Database processing-in-memory: An experimental study," *Proc. VLDB Endowment*, vol. 13, no. 3, pp. 334–347, Nov. 2019.

[41] J.-H. Kim, Y.-R. Park, J. Do, S.-Y. Ji, and J.-Y. Kim, "Accelerating large-scale nearest neighbor search with computational storage device," in *Proc. IEEE 29th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2021, p. 254.

[42] J.-H. Kim, Y.-R. Park, J. Do, S.-Y. Ji, and J.-Y. Kim, "Accelerating large-scale graph-based nearest neighbor search on a computational storage platform," *IEEE Trans. Comput.*, vol. 72, no. 1, pp. 278–290, Jan. 2023.

[43] S. B. Kotsiantis, I. Zaharakis, and P. Pintelas, "Supervised machine learning: A review of classification techniques," *Emerg. Artif. Intell. Appl. Comput. Eng.*, vol. 160, pp. 3–24, Oct. 2007.

[44] K. Kourou, T. P. Exarchos, K. P. Exarchos, M. V. Karamouzis, and D. I. Fotiadis, "Machine learning applications in cancer prognosis and prediction," *Comput. Struct. Biotechnol. J.*, vol. 13, pp. 8–17, Jan. 2015.

[45] R. Kune, P. K. Konugurthi, A. Agarwal, R. R. Chillarige, and R. Buyya, "The anatomy of big data computing," *Softw., Pract. Exper.*, vol. 46, no. 1, pp. 79–105, 2016.

[46] Y. Kwon, Y. Lee, and M. Rhu, "TensorDIMM: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 740–753.

[47] J. H. Lee, H. Zhang, V. Lagrange, P. Krishnamoorthy, X. Zhao, and Y. S. Ki, "SmartSSD: FPGA accelerated near-storage data analytics on SSD," *IEEE Comput. Archit. Lett.*, vol. 19, no. 2, pp. 110–113, Jul. 2020.

[48] S. Liang, Y. Wang, Y. Lu, Z. Yang, H. Li, and X. Li, "Cognitive SSD: A deep learning engine for in-storage data retrieval," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*. Renton, WA, USA: USENIX Association, Jul. 2019, pp. 395–410. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/liang

[49] A. Lottarini, J. P. Cerqueira, T. J. Repetti, S. A. Edwards, K. A. Ross, M. Seok, and M. A. Kim, "Master of none acceleration: A comparison of accelerator architectures for analytical query processing," in *Proc. ACM/IEEE 46th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2019, pp. 762–773.

[50] D. Mahajan, J. Kyung Kim, J. Sacks, A. Ardalan, A. Kumar, and H. Esmaeilzadeh, "In-RDBMS hardware acceleration of advanced analytics," 2018, *arXiv:1801.06027*.
[51] V. S. Mailthody, Z. Qureshi, W. Liang, Z. Feng, S. G. de Gonzalo, Y. Li, H. Franke, J. Xiong, J. Huang, and W.-M. Hwu, "DeepStore: In-storage acceleration for intelligent queries," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 224–238.
[52] D. Maulud and A. M. Abdulazeez, "A review on linear regression comprehensive in machine learning," *J. Appl. Sci. Technol. Trends*, vol. 1, no. 4, pp. 140–147, Dec. 2020.
[53] P. Mehra, "Samsung SmartSSD: Accelerating data-rich applications," in *Proc. Flash Memory Summit*, 2019.
[54] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "MLlib: Machine learning in Apache Spark," *J. Mach. Learn. Res.*, vol. 17, pp. 1–7, Jan. 2016. [Online]. Available: http://jmlr.org/papers/v17/15-237.html
[55] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, H. Mitchel, S. Subhaschandra, A. Sheiman, T. Whisonant, and P. Gupta, "A reconfigurable computing system based on a cache-coherent fabric," in *Proc. Int. Conf. Reconfigurable Comput. FPGAs*, Nov. 2011, pp. 80–85.
[56] K. Park, Y.-S. Kee, J. M. Patel, J. Do, C. Park, and D. J. DeWitt, "Query processing on smart SSDs," *IEEE Data Eng. Bull.*, vol. 37, no. 2, pp. 19–26, Jun. 2014.
[57] S. Salamat, A. Haj Aboutalebi, B. Khaleghi, J. H. Lee, Y. S. Ki, and T. Rosing, "NASCENT: Near-storage acceleration of database sort on SmartSSD," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2021, pp. 262–272.
[58] B. Samynathan, K. Chapman, M. Nik, B. Robatmili, S. Mirkhani, and M. Lavasani, "Computational storage for big data analytics," in *Proc. 10th Int. Workshop Accelerating Anal. Data Manage. Syst.*, 2019.
[59] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Brezzo, S. Asaad, and D. E. Dillenberger, "Database analytics: A reconfigurable-computing approach," *IEEE Micro*, vol. 34, no. 1, pp. 19–29, Jan. 2014.
[60] C.-W. Tsai, C.-F. Lai, H.-C. Chao, and A. V. Vasilakos, "Big data analytics: A survey," *J. Big Data*, vol. 2, no. 1, pp. 1–32, Dec. 2015.
[61] M. Vilim, A. Rucker, Y. Zhang, S. Liu, and K. Olukotun, "Gorgon: Accelerating machine learning from relational data," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, May 2020, pp. 309–321.
[62] M. Wilkening, U. Gupta, S. Hsia, C. Trippel, C.-J. Wu, D. Brooks, and G.-Y. Wei, "RecSSD: Near data processing for solid state drive based recommendation inference," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2021, pp. 717–729.
[63] L. Woods, Z. István, and G. Alonso, "Ibex: An intelligent storage engine with support for advanced SQL offloading," *Proc. VLDB Endowment*, vol. 7, no. 11, pp. 963–974, Jul. 2014.
[64] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The architecture and design of a database processing unit," *ACM SIGARCH Comput. Archit. News*, vol. 42, no. 1, pp. 255–268, Apr. 2014.
[65] S. Xu, T. Bourgeat, T. Huang, H. Kim, S. Lee, and A. Arvind, "AQUOMAN: An analytic-query offloading machine," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2020, pp. 386–399.
[66] Y.-C. Hu, Y. Li, and H.-W. Tseng, "TCUDB: Accelerating database with tensor processors," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, Jun. 2022, pp. 1360–1374, doi: 10.1145/3514221.3517869.

JI-HOON KIM (Graduate Student Member, IEEE) received the B.S. degree in electrical engineering from Kyung-Hee University, Suwon, South Korea, in 2017, and the M.S. degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 2019, where he is currently pursuing the Ph.D. degree. His current research interests include AI/ML hardware accelerator (ASIC and FPGA) design, efficient AI/ML system design, energy-efficient processing-in/near-memory architecture, and hardware/software co-design for DNN processing.

SEUNGHEE HAN (Graduate Student Member, IEEE) received the B.S. degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 2022, where he is currently pursuing the M.S. degree. His current research interests include efficient architecture design for AI systems, hardware/software co-design for end-to-end system design, and near-data processing architecture.

KWANGHYUN PARK received the M.Sc. and Ph.D. degrees in computer science from the University of Wisconsin-Madison. He is an Assistant Professor with the Department of Computer Science, Yonsei University, working on database systems and machine learning. His research interests include large-scale data processing, intersection of database systems, and machine learning.

SOO-YOUNG JI is a Senior Engineer with the Memory Business Advanced Solution Team, Samsung Electronics.

JOO-YOUNG KIM (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 2005, 2007, and 2010, respectively. He is currently an Associate Professor with the School of Electrical Engineering, KAIST, where he is also the Director of the AI Semiconductor Systems Research Center. Before joining KAIST, he was a Senior Hardware Engineering Lead with Microsoft Azure, Redmond, WA, USA, working on hardware acceleration for its hyper-scale big data analytics platform named Azure Data Lake. He was also one of the initial members of Catapult Project, Microsoft Research, Redmond, where he deployed a fabric of FPGAs in datacenters to accelerate critical cloud services, such as machine learning, data storage, and networking. His research interests span various aspects of hardware design, including VLSI design, computer architecture, field-programmable gate array (FPGA), domain-specific accelerators, hardware/software co-design, and agile hardware development. He was a recipient of the 2016 IEEE Micro Top Picks Award, the 2014 IEEE Micro Top Picks Award, the 2010 DAC/ISSCC Student Design Contest Award, the 2008 DAC/ISSCC Student Design Contest Award, and the 2006 A-SSCC Student Design Contest Award. He served as an Associate Editor for IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I: REGULAR PAPERS, from 2020 to 2022.