Optimizing In-memory Database Engine for AI-powered On-line Decision Augmentation Using Persistent Memory

Cheng Chen[*] [†]
Jun Yang[*]
Mian Lu
Taize Wang
Zhao Zheng
Yuqiang Chen
Wenyuan Dai
4Paradigm Inc.
[chencheng,yangjun01,lumian,wangtaize,
zhengzhao,chenyuqiang,daiwenyuan]
@4paradigm.com

Bingsheng He
Weng-Fai Wong
National University of Singapore.
[hebs,wongwf]@comp.nus.edu.sg

Guoan Wu
Yuping Zhao
Andy Rudoff
Intel Corporation
[dennis.wu,yuping.zhao,andy.rudoff]
@intel.com

ABSTRACT

On-line decision augmentation (OLDA) has been considered as a promising paradigm for real-time decision making powered by Artificial Intelligence (AI). OLDA has been widely used in many applications such as real-time fraud detection, personalized recommendation, etc. On-line inference puts real-time features extracted from multiple time windows through a pre-trained model to evaluate new data to support decision making. Feature extraction is usually the most time-consuming operation in many OLDA data pipelines. In this work, we started by studying how existing in-memory databases can be leveraged to efficiently support such real-time feature extractions. However, we found that existing in-memory databases cost hundreds or even thousands of milliseconds. This is unacceptable for OLDA applications with strict real-time constraints. We therefore propose **FEDB** (**F**eature **E**ngineering **D**ata**b**ase), a distributed in-memory database system designed to efficiently support on-line *feature extraction*. Our experimental results show that FEDB can be one to two orders of magnitude faster than the state-of-the-art in-memory databases on real-time feature extraction. Furthermore, we explore the use of the Intel Optane DC Persistent Memory Module (PMEM) to make FEDB more cost-effective. When comparing the proposed PMEM-optimized persistent skiplist to the FEDB using DRAM+SSD, PMEM-based FEDB can shorten the tail latency up to 19.7%, reduce the recovery time up to 99.7%, and save up to 58.4% total cost of a real OLDA pipeline.

PVLDB Reference Format:
Cheng Chen, Jun Yang, Mian Lu, Taize Wang, Zhao Zheng, Yuqiang Chen, Wenyuan Dai, Bingsheng He, Weng-Fai Wong, Guoan Wu, Yuping Zhao, and Andy Rudoff. Optimizing In-memory Database Engine for AI-powered On-line Decision Augmentation Using Persistent Memory. PVLDB, 14(5): 799 - 812, 2021.
doi:10.14778/3446095.3446102

[*]Both authors contributed equally to the paper
[†]C. Chen, is also with National University of Singapore.

Proceedings of the VLDB Endowment, Vol. 14, No. 5 ISSN 2150-8097.
doi:10.14778/3446095.3446102

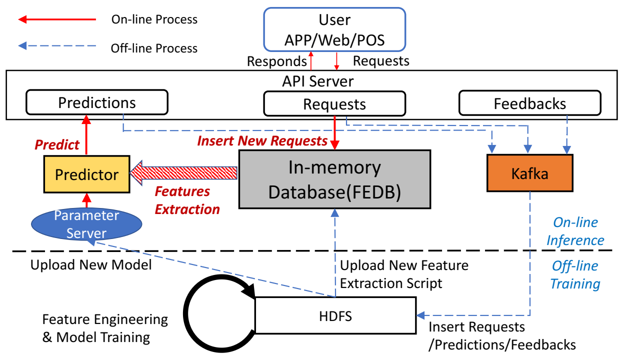


Figure 1: Workflow of AI-powered OLDA Systems

1 INTRODUCTION

Over the past decade, artificial intelligence (AI) has increasingly attracted a significant amount of attention from both academia and industry. As computational power increases, AI makes substantial progress in integrating into the daily work routines of enterprises to improve productivity. In various fields such as financial services, healthcare, retail and transportation, **on-line decision augmentation (OLDA)** powered by AI has become one of the fastest-growing and promising paradigms to enable timely decision making. According to the latest report from Gartner [59], the market value of OLDA will reach $2.2 billion by 2025 and account for 44% of AI market value by 2030. Therefore, this paper focuses on improving the performance and cost-effectiveness of OLDA pipelines.

We have studied many OLDA pipelines in the real world, and the designs are similar. Fig. 1 shows a representative OLDA workflow from 4Paradigm, an OLDA solution provider in China. Similar workflows can be found in other OLDA systems. A typical OLDA system consists of two separated sub-systems, (1) the off-line training system that derives a trained model for a specific problem using historical data and (2) the on-line inference system which interacts with the front-end (e.g. software/mobile app/website) to process prediction/scoring requests. The result of the on-line inference is calculated through the trained model by feeding the extracted features as its input. The response time of these requests is the key performance metric of the entire OLDA system as it directly affects the user experience. Thus, many OLDA pipelines have rigid time constraints on such real-time feature extraction operations.

Most of the real-time features are required to be computed over multiple time windows. To achieve low-latency feature extraction, existing in-memory databases can ideally be leveraged for these operations. Specifically, with in-memory databases, the **on-line feature extraction** process involves a new record insertion followed by an enormous number of concurrent analytical queries related to the new record. However, our studies showed that the latency of extracting real-time features grows proportionally to the number of time windows in two state-of-the-art in-memory databases. As a result, if we increase the number of time windows to get better prediction accuracy, the response time will be unacceptable for those OLDA systems that have strict timing constraints.

In this paper, we propose FEDB (**F**eature **E**ngineering **D**ata**b**ase), a distributed in-memory database system that specifically designed to efficiently support on-line feature extraction in OLDA systems with its LLVM-based [67] query execution engine and in-memory structure based on a double-layered skiplist.

In order to further reduce the total ownership cost of FEDB, we propose taking advantage of the recently announced persistent memory product named Intel® Optane™ DC Persistent Memory Module (PMEM). PMEM has a lower per-GB cost, higher density and is non-volatile compared with DRAM. Previous studies [8, 15, 16, 34, 37, 40, 48, 62, 69] have shown that making in-memory data structures persistent in PMEM without compromising data consistency is a non-trivial task. Specifically, we extend FEDB with a PMEM-optimized persistent skiplist.

Our contributions are summarized as follows:

- We describe and analyze the entire data processing workflow of OLDA using an example from the fraud detection system of 4Paradigm. We summarize the characteristics and challenges on efficiently supporting the OLDA applications.
- We propose FEDB for more efficient real-time feature extraction. We also introduce FEQL (Feature Extraction Query Language), a SQL-like language to facilitate feature extraction in FEDB. The FEQL engine utilizes the LLVM compiler framework for efficient query compilation and execution.
- We propose a PMEM-optimized persistent skiplist and integrate it into the storage engine of FEDB. Our proposal not only improves the utilization of PMEM, but also enables faster recovery.
- Under the real-world fraud detection workloads, our experimental results show that, FEDB can be up to two orders of magnitude faster than the state-of-the-art commercial database systems. Furthermore, PMEM-based FEDB can shorten up to 19.7% of the tail latency, reduce up to 99.7% of the recovery time, and save up to 58.4% total cost of the fraud detection system compared to FEDB using DRAM+SSD.

The rest of the paper is organized as follows. Sec. 2 introduces the background of OLDA, on-line feature extraction and PMEM. Sec. 3 states the motivation of this study. Sec. 4 proposes the design and implementation of FEDB, followed by the details of the PMEM-optimized design in Sec. 5. The experimental results are shown in Sec. 6 and Sec. 7 concludes this paper.

2 BACKGROUND & RELATED WORK

2.1 On-line Decision Augmentation (OLDA)

Over the years, OLDA has attracted significant interest from both industry and academia. According to the report from International Data Corporation [25], 4Paradigm is ranked first in China for market share amongst all machine learning platforms. It has provided OLDA solutions and helped 8, 000+ customers in 12, 000+ scenarios to complete AI transformation during the past 5 years.

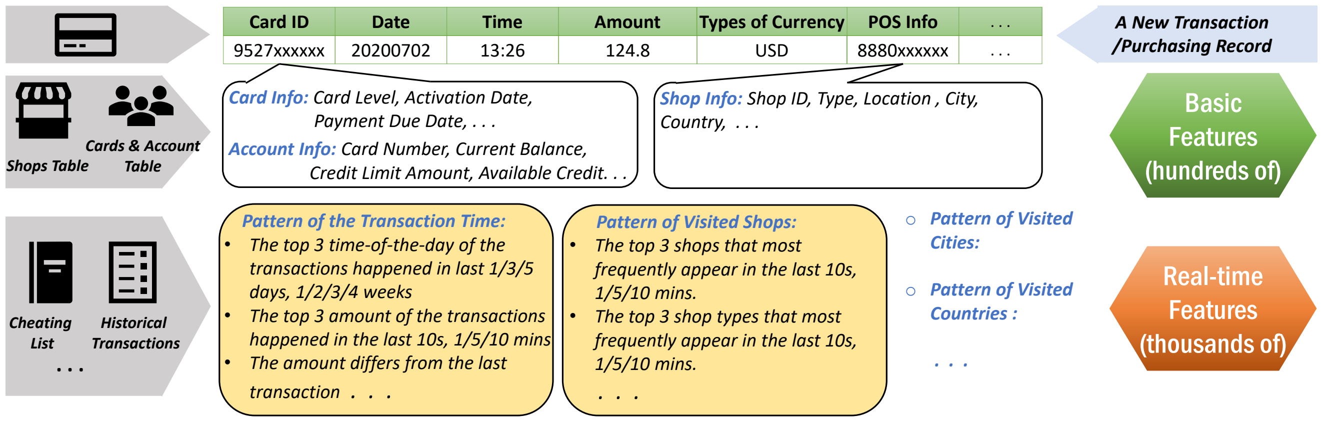
We will dive deeper into the OLDA pipelines with the fraud detection system from 4Paradigm that is in active deployment as an example. In the deployment, the AI-powered solution is able to detect fraud instantly with high accuracy, and it has already helped thousands of millions of users from a top bank in China. As shown in Fig. 1, when a user swipes his/her credit card through a point-of-sale (POS) device, the credit card transaction is sent to an API server requesting for a prediction on whether it is a fraud. Upon receiving the request, the OLDA system first inserts the new transaction into an in-memory database then immediately extracts features based on the user's and shop's profile, historical transactions, etc. These features are fed into a pre-trained machine learning model that scores and predicts the probability of the current transaction being a fraud. At a later stage, the final prediction result is checked against the user's behavior (such as whether the user reports the trade as a fraud, etc) and used as feedback for off-line training to further tune the model.

In the entire OLDA processing cycle, the timeliness of *on-line inference* is one of the most important factors for the system performance, which is also the main focus of this paper. The on-line inference process consists of several steps including the on-line feature extraction, scoring and prediction. All these steps have to be completed within tens of milliseconds after users swipe their credit card. In the entire process, we found that the on-line feature extraction is the most time-consuming part and a major bottleneck in reducing the response time of the OLDA applications [70].

2.2 Online Feature Extraction

Feature extraction can be recognized as obtaining the "hidden information" in the raw data that is the key to achieve better accuracy of the trained model. For example, detecting fraud is highly related to recent behaviors, and real-time features make a great impact on improving the prediction accuracy. As shown in Fig. 2, the extraction of real-time features has the following characteristics.

Firstly, most of the real-time features have to be computed using the information from the newly generated transaction record, thus, these features can not be pre-extracted until the new record is received. For instance, the feature that records the difference in



Card ID | Date | Time | Amount | Types of Currency | POS Info | . . .
9527xxxxxx | 20200702 | 13:26 | 124.8 | USD | 8880xxxxxx | . . .
A New Transaction /Purchasing Record
Shops Table
Cards & Account Table
Card Info: Card Level, Activation Date, Payment Due Date, . . .
Account Info: Card Number, Current Balance, Credit Limit Amount, Available Credit. . .
Shop Info: Shop ID, Type, Location , City, Country, . . .
Basic Features (hundreds of)
Cheating List
Historical Transactions
. . .
Pattern of the Transaction Time:
The top 3 time-of-the-day of the transactions happened in last 1/3/5 days, 1/2/3/4 weeks
The top 3 amount of the transactions happened in the last 10s, 1/5/10 mins
The amount differs from the last transaction . . .
Pattern of Visited Shops:
The top 3 shops that most frequently appear in the last 10s, 1/5/10 mins.
The top 3 shop types that most frequently appear in the last 10s, 1/5/10 mins.
. . .
Pattern of Visited Cities:
Pattern of Visited Countries :
. . .
Real-time Features (thousands of)

Figure 2: An Illustration of On-line Feature Extraction of Fraud Detection System in 4Paradigm

the amount between the last and new transaction can only be calculated after getting the new record.

Secondly, most real-time features are time-related and required to be computed over multiple time windows. For example, based on the shops or time-of-the-day the transactions most frequently happened, a user's habit can be inferred. To fully describe a user's recent (e.g., 1/5/30/60 minutes), medium-term (e.g., 1/3/7 days) and long-term (e.g., weeks or months) purchasing habits, we need to look into multiple time windows. Based on our experience, the more time windows are involved in the feature extraction process, the more accurate the prediction will be.

Lastly, OLDA systems extract a large number of real-time features to achieve high prediction accuracy. In practice, we have extracted thousands of the real-time features for each prediction in our real-world fraud detection system, which dominates the overall response time. From the database point of view, the real-time feature extraction can be represented as inserting a new record, followed by a large number of queries in previous time windows.

2.3 Database Support for OLDA Workload

To support OLDA applications efficiently, we have revisited several relevant database systems. Relational in-memory databases that support OLTP, OLAP, or HTAP workload are potential candidates to efficiently support the feature insertions and queries in previous time windows. The OLTP databases widely use a row-based storage engine focusing on optimizing transactional insertion performance [23, 35, 45], and the OLAP databases apply a column-based storage engine that is more suitable for read-only analysis workload [6, 56–58, 71, 72]. OLDA workloads differ from OLTP/OLAP workloads. OLDA is similar to HTAP in that both reads and writes are involved [5, 18, 42, 43, 54]. However, OLDA considers the combined "insert + queries" operations as a unique pattern of "one insertion followed by aggregation queries on a large number of columns over multiple time windows". The latter queries always require the prior insertion to compute their results. Time-series databases (TSDB), such as the TimescaleDB in PostgreSQL [26], InfluxDB [27], etc, were originally designed and optimized for applications such as distributed system monitoring and the Internet of things. The workload optimized by TSDB is also different from the OLDA workload. In TSDB workloads, 95% - 99% of operations are write operations [2]. OLDA workloads, on the other hand, have a large number of read-only queries. There are a number of recent works aimed at better supporting AI workloads [3, 12, 22, 36, 38, 41, 50, 52, 53, 55, 61]. However, to the best of our knowledge, there are no database design dedicated to the OLDA workload.

2.4 Intel Optane DC Persistent Memory

Table 1: Data Size of Our Fraud Detection System

# of Users	# of Cards	Historical Trans.	Mem. Usage
0.7 billion	1 billion	>1 billion	>3 terabytes

As shown in Table 1, the in-memory data of our real-world fraud detection occupied more than 3 terabyte (TB) of memory. We can only store the transactions of the last 3 months due to the DRAM capacity limitation. When using the database replica during the deployment, it further doubles/triples the memory usage and significantly increased the hardware cost.

The gap between the huge memory demand and the physical limitation of DRAM capacity in a single server significantly increased the overall cost of the OLDA solution. Compared with DRAM, the *non-volatile random access memory* (NVRAM) provides much larger capacity [34], and it gives us a way to address the above issue. NVRAM is a set of persistent memory technologies that provide byte addressable random access and the capability to persist data even after power failure. Intel Optane DC Persistent Memory Module (PMEM) [28] is the first commercial product of the Non-volatile memory technology now available in the market. PMEM provides similar performance yet at a lower price per GB [1]. This technology gives us a new option to tackle "memory-hungry" applications. As shown in Fig. 3, PMEM can work in two modes: *Memory Mode* and *App Direct Mode*.

Memory Mode: PMEM is working as the system memory while DRAM works as a direct-mapped cache. The operating system can

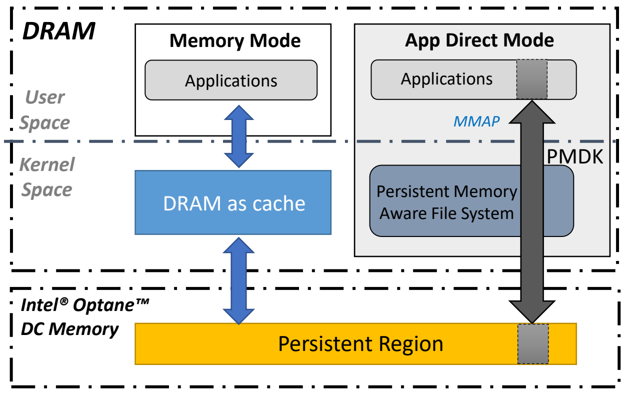


Figure 3: Intel Optane DC Persistent Memory Module

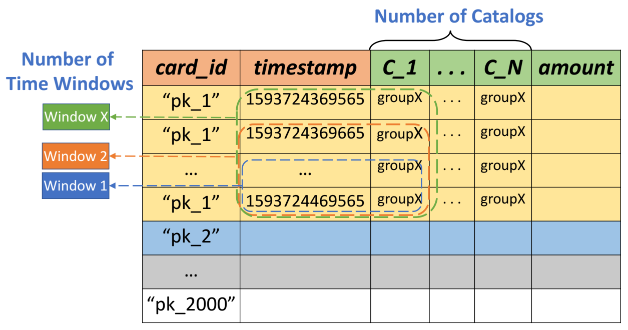
directly use the entire PMEM as a 'large' memory. Note that although PMEM is non-volatile, there is no persistency guarantee for in-memory data in this mode. However, existing in-memory applications will work as before without modification.

App Direct Mode: This mode provides a way to directly access PMEM. The operating system recognizes PMEM as a persistent memory device. When PMEM is mounted with a PMEM-aware file system (such as Ext4/XFS-DAX and NOVA [68]), a user application can map a file into the user memory space. By using POSIX API mmap, the applications can directly access PMEM without the overhead of kernel system calls.

A series of works have been done to explore the integration of PMEM into the system design [4, 7, 9, 11, 13, 19, 20, 44, 47]. Due to the read/write performance gap between DRAM and PMEM, a number of works [8, 10, 37, 48, 51, 60, 63, 64] have been proposed to reduce the performance penalty of writes when applying PMEM. Encouraged by those studies, we studied the PMEM-based data structure to reduce the total ownership cost of OLDA applications.

3 MOTIVATION

3.1 Motivations with Existing In-Memory Databases


Number of Catalogs
Number of Time Windows
Window X
Window 2
Window 1

card_id	timestamp	C_1	. . .	C_N	amount
"pk_1"	1593724369565	groupX	. . .	groupX	
"pk_1"	1593724369665	groupX	. . .	groupX	
...	...	groupX	. . .	groupX	
"pk_1"	1593724469565	groupX	. . .	groupX	
"pk_2"					
...					
"pk_2000"					

Figure 4: Table Schema Example

As discussed in Sec 2.3, online feature extraction in OLDA systems can be done using existing in-memory database systems. In practice, real-time feature extraction can be done via TopN queries on different columns over multiple time windows. However, we found that their performance cannot meet the strict timing constraints of real-world OLDA applications such as fraud detection. We perform a case study with the OLDA pipeline mentioned in Fig. 2. A sample table schema is shown as Fig. 4. The table can be wide, consisting of (1) *card_id*, (2) *timestamp*, (3) *amount* and (4) *catalog[1..N]*. Those columns are designed for the TopN queries. For example, *shop_info* can be transformed into multiple columns like *catalog1*(*C_1*) having a value from the set {supermarket/canteen...} to describe the types of shops, *catalog2*(*C_2*) having a value from the set {morning/afternoon/night...} to describe what time-of-the-day the transactions happened, etc. In addition to the primary key being the combination of (*card_id*, *timestamp*), extra indexes for each column that are required to perform the TopN queries are also maintained in the form of a combined index with *timestamp* so as to obtain the best performance.

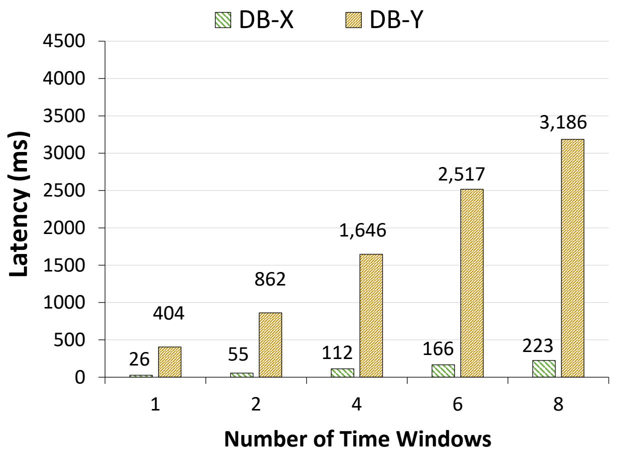


Figure 5: Performance of Extracting Real-time Features over Varied Number of Time Windows

Given the *card_id* of 'D', a current timestamp 'CUR_TS' and a time window 'W', we can execute a SQL query like "SELECT GROUP_CONCAT(catalogX) FROM (SELECT catalogX FROM table WHERE card_id = D and CUR_TS < W) GROUP BY catalogX ORDER BY count (*) DESC LIMIT 3" to find the Top3 most frequent values of *catalogX* in the time window W. To extract the real-time features in M time windows from a table with N catalogs, the above SQL needs to be issued $N \times M$ times. We measured the duration of executing these queries concurrently in two popular in-memory database systems denoted as DB-X and DB-Y. Both of them are state-of-the-art commercial in-memory DBMS that are well-known to support HTAP workloads.

Fig. 5 shows the average execution time for extracting the real-time features in the table above with 32 catalogs. As the number of time windows increases, the execution time of both systems increases proportionally. According to our experience in OLDA workloads, the more time windows are used to extract the real-time features, the more accurate the prediction will be. In practice, our fraud detection system requires at least 10 time windows (some catalogs may need more) to achieve an acceptable prediction accuracy with a requirement of sub 100ms on response time. However, both DB-X and DB-Y cannot meet the latency requirement with the number of time windows larger than 3. This indicates that existing database systems are not well-optimized for concurrent TopN queries on different columns in multiple time windows for real-time feature extraction in OLDA pipelines.

3.2 Motivations for PMEM

With the large main memory footprint, OLDA workloads have the following challenges.

(1) Hardware cost becomes a challenge when using this solution for OLDA applications. Due to the rapid data growth as we observed in many applications, OLDA applications will require more than tens of terabytes memory. DRAM capacity per DIMM, as well as the number of DIMM slots on the mainboard are physically limited. The demand for huge memory would therefore translate to higher cost either in the form of an expensive customized machine, or a large number of regular machines organized as a cluster. In both cases, system development and maintenance will be increased accordingly.
(2) Due to the volatility of DRAM, most in-memory databases still use persistent secondary storage devices (HDDs/SSDs) as persistent/backup storage. Normally, data is periodically synced to the secondary storage device to minimize the negative performance impact as the write performance of these devices is orders of magnitude slower than DRAM. However, under insertion-heavy workloads, tail latency can be too high that the strict timing requirement of OLDA applications cannot be met.
(3) Long recovery time affects the system availability. Note that one hour of downtime per year decreases the system availability under 99.99% [21]. When recovering from failure, the system has to reload all the data from the slow persistent storage device. As the size of the data stored in DRAM is huge, the recovery can be very time-consuming (can last a few hours or even more).

PMEM provides us a more cost-efficient option to deploy applications such as our FEDB for OLDA applications. Although directly using PMEM in *memory mode* is the easiest way to address the first challenge without modifying existing applications, the second and the third challenges remain unsolved as the non-volatility of PMEM is not utilized. Hence, using PMEM in its *app direct mode* to store and process all data is a better option. Previous works [8, 14, 15, 37, 48, 62, 69] have shown that persistent memory programming [29] is non-trivial and error-prone. Therefore, we developed a PMEM-based in-memory data structure for FEDB to enable in-PMEM data consistency and recoverability without using the traditional logs/snapshots persistence techniques.

4 FEDB

In this paper, we propose FEDB, an in-memory database engine specifically designed to support online feature extraction.

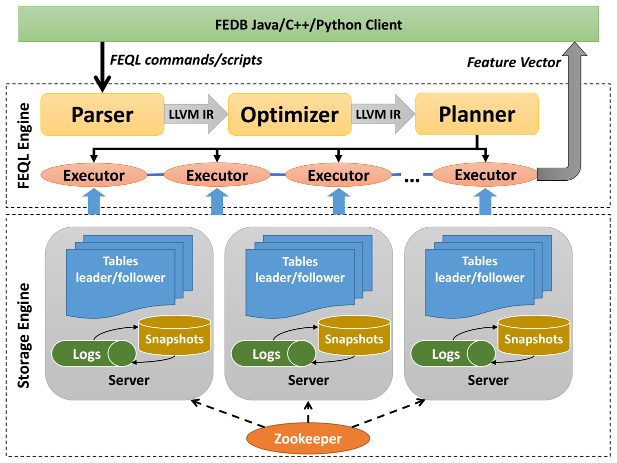


Figure 6: Architectural Overview of FEDB on DRAM+SSD

4.1 Design overview

FEDB is responsible for feature extraction in the workflow of AI-powered OLDA systems. To facilitate and optimize feature extraction, we have designed a SQL-like language called **FEQL** (Feature Extraction Query Language) which can be used to define all the features that need to be extracted from the original structured (table-formatted) data. As shown in Fig. 6, FEDB consists of two major components, **FEQL Engine** which utilizes Low Level Virtual Machine (LLVM) [67] to parse, optimize and execute the given FEQL, and **Storage Engine** which stores the actual table-formatted data with timestamps. Given the execution plan, the executors in **FEQL Engine** interact with the underlying storage engine to retrieve the actual data and compute feature as the output.

FEQL is similar to the standard SQL in many ways. FEQL introduces some new syntaxes to define and execute typical feature extraction queries more efficiently. For instance, one of the key features of FEQL is the ability to define multiple time windows in a single query so that those TopN queries in multiple time windows are no longer executed in each time window individually. We will discuss this in more details in Sec. 4.2.

The underlying storage engine of FEDB adopts a distributed leader/follower architecture to guarantee high availability (HA) of data using tools like Zookeeper [24]. Note that Fig. 6 shows FEDB with DRAM+SSD, which adopts a traditional persistence model (logs and snapshots) to make data recoverable after system reboots. We adopt a DRAM-based double-layered skiplist as the index to make TopN queries in multiple time windows efficient. Furthermore, to address the drawbacks of using DRAM mentioned in the previous section, we adopt PMEM and design a PMEM-based persistent skiplist to replace the DRAM-based one in FEDB so that data persistence can be achieved without using logs/snapshots on SSDs. We will discuss the details of the storage engine in Sec. 4.3.

4.2 FEQL Engine

FEQL Engine of FEDB adopts the state-of-the-art query compilation and execution techniques that utilizes the LLVM compiler [67] to transform the query into assembler code. FEQL engine supports not only the standard SQL-like commands and syntaxes like create/drop tables, insert/delete records, projection, group by, etc., but also allows User Defined Function (UDF) and User Defined Aggregate Function (UDAF) in a similar way as MemSQL [43].

To better support real-time feature extraction, FEQL engine introduces some new syntaxes and the corresponding optimizations. Specifically, FEQL introduces a new TIME_WINDOW function and allows multiple time windows to be defined in a single command. Note that this is different from the standard WINDOW function in SQL, which defines a window of rows with a given *length* around the current row to perform calculation/aggregation across the set of data in the window [39]. For example, considering the table schema in Sec. 3.1, to find the Top3 values of catalog1 and catalog2, in the last 1 day/1 week for *card_id* D at the current time, we can use a single FEQL query as follows:

```
SELECT TopN_Frequency(catalog1, 3) OVER w1,
    TopN_Frequency(catalog1, 3) OVER w2,
    TopN_Frequency(catalog2, 3) OVER w1,
    TopN_Frequency(catalog2, 3) OVER w2
    FROM table
    TIME_WINDOW w1 as
        (PARTITION BY table.card_id TIME_RANGE
        BETWEEN 1d PRECEDING AND CURRENT TIME),
    TIME_WINDOW w2 as
        (PARTITION BY table.card_id TIME_RANGE
        BETWEEN 1w PRECEDING AND CURRENT TIME)
    WHERE table.card_id=D;
```

To optimize the performance of typical feature extraction queries that involve multiple time windows, FEQL adopts an optimization technique called *time window reuse*. Note that, in an FEQL query, the largest time window always covers all the smaller ones. Therefore, instead of processing each window one by one, FEQL only scans the data once for the largest time window and calculates the query result for all the time windows at the same time. Our experimental results showed that such an optimization significantly reduces the feature extraction time for real-time features with a large number of time windows to explore. In addition, FEQL is also capable of processing different columns (e.g. catalogX) in parallel.

4.3 Storage Engine

In this subsection, we present the storage engine of FEDB using DRAM+SSD. We will present the PMEM-optimized storage engine in Section 5.

4.3.1 ***DRAM-based Double-layered Skiplist.*** The in-memory storage engine is mainly designed to optimize the performance of two tasks, (1) storing timestamped table-formatted data and (2) retrieving data with given key and time range. We used a *double-layered skiplist* as the core data structure to store all the data in memory. As a well-known data structure for its simplicity and intrinsic lock-free between read and write, the skiplist has been widely integrated in many in-memory database systems such as Redis [17] and MemSQL. The table-formatted data is defined by a *schema* indicating the name and type of all the columns in a similar way to that in a relational database. In addition, the schema also defines which column(s) is/are timestamps and which column(s) is/are used for time-windowed queries. More specifically, for each

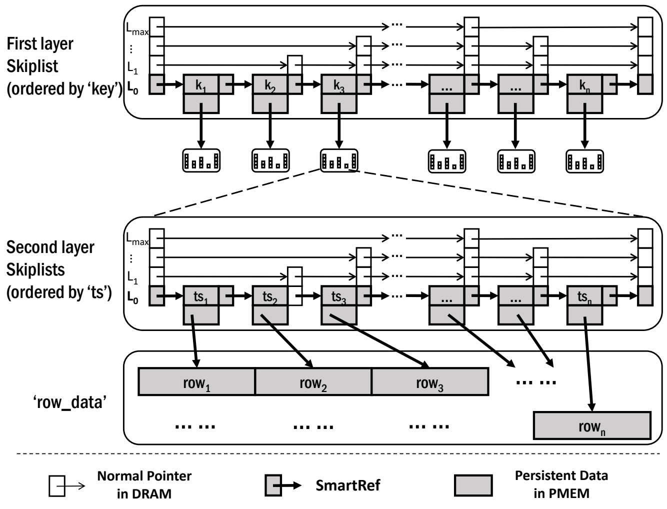


Figure 7: In-memory Double-layered (Persistent) Skiplist

column the time-windowed queries have to be performed on, a double-layered skiplist is constructed. The first layer uses the value of the column as the key and the pointer to a second-layer skiplist as the value. The second-layer skiplist stores the timestamps for each value of the column as the key and the pointer to the row data as the value. Thus, the time-windowed scan on a specific column can be efficiently performed by searching the first-layer skiplist for the specified 'key', followed by scanning the second-layer skiplist for the given time window to locate the target 'row_data'.

Fig. 7 (ignore the shading part of the figure at this moment) illustrates an example of storing a table with one column named ***key*** as the index column and another one named ***ts*** as the timestamp column. Note that if multiple index columns are defined in the same table, only the double-layered skiplists are constructed while the row_data is reused through the pointer in the second-layer skiplist. This design of shared row_data is very important to the memory space utilization as the number of columns that the time-windowed queries are performed on can be very large in OLDA application such as fraud detection.

4.3.2 ***SSD-based Persistence Model.*** FEDB adopts a *persistence model* to make the data recoverable upon system restart by using *logs* and *snapshots*, which can be found in almost all in-memory databases such as Redis [17] and SAP HANA [18]. However, the additional log writes have to be synced to the persistent storage, which is quite slow on traditional HDDs and flash SSDs compared to memory operations. Syncing logs more frequently can result in a worse performance but less frequently may cause more data loss when the system crashes. As FEDB is not required to support transactions with full ACID properties, it allows users to configure the time interval between two consecutive log syncing operations to balance the performance and acceptable range of data loss. In deployment, FEDB uses high-end SSDs to store the logs.

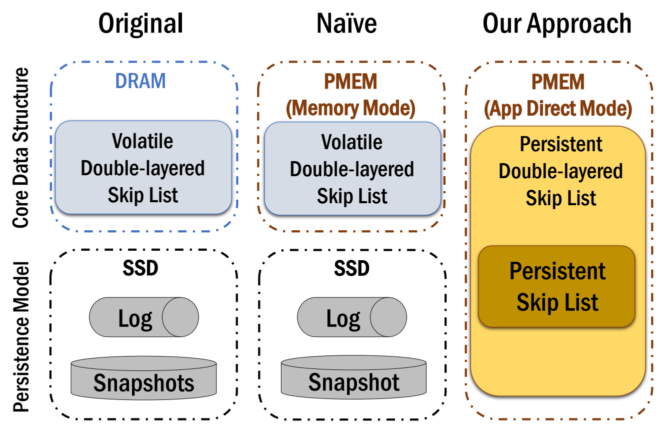


Figure 8: Different Ways of Using PMEM in FEDB

5 PMEM-OPTIMIZED FEDB

We start by using PMEM with minimal modification in the code of FEDB. Compared to DRAM+SSD-based storage engine (the leftmost subfigure in Fig. 8), using PMEM in *Memory Mode* as the working memory for the core data structure (the middle subfigure in Fig. 8) does not involve any code modification. Instead, a tool provided by Intel (ipmctl [32]) is used to configure the PMEM to make the operating system treat it as main memory. However, this straightforward method still needs to maintain logs/snapshots on SSD and does not utilize the non-volatility of PMEM. Using PMEM in *App Direct Mode* and utilizing its non-volatility to create a new storage engine makes it possible to guarantee data recoverability without logs and snapshots. To achieve that, we implemented a PMEM-based persistent skiplist, and integrate it into the double-layered skiplist (the rightmost subfigure in Fig. 8). Our experimental results showed that , in comparison with the DRAM+SSD approach, this approach not only eliminates the negative performance impact of syncing logs, but also achieves instant recovery upon system restart.

5.1 PMEM-based Persistent Skiplist

Implementing a persistent skiplist for PMEM is non-trivial as it requires additional logic to deal with the space management of persistent memory and in-memory data consistency upon system failure. The two main challenges in implementing a persistent data structure are as follows:

- **Atomic persistent memory allocation/release** becomes essential for the space management of persistent memory. For example, the memory leak and dangling pointer problem may happen when system failure occur between allocating the persistent memory space of an object and storing its address persistently.
- **Data consistency upon system failure** becomes problematic as most of the basic operations that modify in-memory data structures cannot be done in a single CPU instruction. As a result, when a system failure occurs, interrupting the execution at any point in time may corrupt the entire data structure. Therefore, extra effort must be made to guarantee data consistency. Moreover, the memory store instruction cannot guarantee that the data can be persisted in PMEM unless special instructions - (CLFLUSHOPT or CLWB followed by SFENSE, denoted as FLUSH) are executed. Note that FLUSHing a memory object larger than 8 bytes is not atomic [31].

We use PMDK and libpmemobj-cpp [33] from Intel to address the first challenge by calling provided APIs to allocate/free persistent memory *atomically*. Internally, PMDK keeps track of the addresses of all the objects being allocated/freed in the persistent memory so that it can undo/redo the interrupted operations during recovery. It is still costly but saves a large amount of engineering work to make pointer-based data structure persistent in PMEM. In our implementation of persistent skiplist, we manually manage a 'free list' for the nodes of persistent skiplist to remove the allocation/release operations from the critical path of data processing. As a result, we can benefit from the easy implementation using PMDK without compromising the performance much.

There are two existing approaches to tackle the second challenge. The first is to implement the traditional logging/copy-on-write (COW) algorithms within the data processing logic of the user application itself. This approach poses the implementation complexity to users, and also it can be error-prone. The second approach is to use the transaction support in PMDK that can guarantee the atomicity of multiple memory operations, which uses COW internally. Although both the logging/COW approaches are able to guarantee the correctness, they are not efficient as both require the inherent and extra write overhead. Therefore, our implementation of persistent skiplist adopts neither of them. Inspired by the existing data structures for persistent memory [8, 15, 37, 48, 69], we redesign the procedure of insertion, deletion and search in the persistent skiplist to use only 8-byte atomic writes followed by a FLUSH. By doing so, the data consistency can be maintained upon system failure without using expensive logging/COW.

CompareAndSwap(CAS) [66] technology is the key to maintain non-blocking execution in a single-writer-multi-reader scenario for the persistent skiplist. However, if it operates in persistent memory, we need to handle the possible inconsistency caused by a write-after-read dependency where one thread persistently writes new data computed/derived from the result of reading some data that might not be persisted [65]. This can be solved by a *flush-on-read* method: any read operations on such data must be preceded by a FLUSH.

To efficiently implement *flush-on-read* to make the CAS work correctly on persistent memory, we leverage the vacant lower four bits in a normal 64-bit pointer on a 64-bit machine [65]. We further improve it by embedding both *'deleted'* and *'dirty'* bits into the special pointer, denoted as SmartRef. SmartRef is essentially a 64-bit unsigned integer (same size as the normal pointer), which uses the lowest bit as 'dirty' bit and the second lowest one as 'deleted' bit. PersistRead() takes SmartRef as input. If it is marked dirty, it will be FLUSHed with 'dirty' bit cleared.

Algorithm 1: RebuildUpperLevel

```
Input: head
Output: success
for i=1:maxHeight-1 do
    cur[i] ← head;
end
cur0 ← head;
while cur0 ≠ tail do
    next0 ← PersistRead(cur0.next[0]);
    for i=1:cur0.height-1 do
        cur[i].next[i] ← next0;
        cur[i] ← next0;
    end
    cur0 ← next0;
end
for i=1:maxHeight-1 do
    cur[i] ← tail;
end
return TRUE;
```

Algorithm 2: Insert

```
Input: K, V
Output: success
found ← find(K, preds[], succs[]);
if (!found) then
    new_node ← GetFreeNodeAndInitial(V);
    new_node.level ← RandomHeight();
    for i=0:level-1 do
        AtomicStore(new_node.next[i],
          SmartRef(succs[i], FALSE, FALSE));
    end
    for i=0:level-1 do
        while TRUE do
            pred ← preds[i];
            succ ← succs[i];
            expected ← SmartRef(succ, FALSE, FALSE);
            dirty ← (i == 0?TRUE : FALSE);
            if CAS(pred.next[i], expected,
              SmartRef(new_node, FALSE, dirty)) then
                break;
            end
            if i=0 then
                PersistRead(pred.next[i]);
            end
        end
    end
    RemoveFromFreeList(new_node);
    return TRUE;
end
return FALSE;
```

Given the fact that writes in PMEM is slower than that in DRAM, we further optimize our PMEM-based persistent skiplist by reducing the number of writes on PMEM. As shown in Fig. 7, only the next pointers (i.e. SmartRef) on level 0 and the actual data are stored persistently in PMEM, while those on the remaining levels are using normal pointers in DRAM. We also developed a recovery procedure to rebuild them upon system failure, which scans the skiplist via SmartRef on level 0 and relink the pointer on the upper level according to the height of each node. The concept of rebuilding-on-recovery has been widely adopted in other data structures for persistent memory [69]. Algorithm 1 shows the pseudo code of rebuilding the next pointers on upper levels upon recovery. The steps involved are as follows: (1) initialize the iterator-like structure cur[1..max_height-1], which represents the node the iterator of each level (except level 0) points to (line 1-3); (2) traverse the skiplist through the next pointer on level 0 (cur0.next[0] where cur0 represents which node the iterator of level 0 points to), for every cur0, re-link the next pointers of all the levels between 1 and the height of cur0 to point to cur0.next[0] (line 4-12); (3) and on reaching the tail in level 0, link the next pointer of the last node on each level to the tail (line 13-15).

The procedure of search is similar to the standard DRAM-based skiplist except (1) the execution of PersistRead() before entering the target node and (2) ignoring the nodes marked as 'toDelete'. The removal procedure starts with marking the target node as 'toDelete' persistently, then utilizes CAS to update the next pointers (SmartRef on level 0) of the preceding nodes from the top level down to level 0.

The insertion starts with searching for the position to insert the new node with the provided key-value pair. After finding the preceding and succeding node, the next pointers of the new node on each level are linked to the succeeding node, then the next pointers of the preceding node on each level are updated to point to the new node. Algorithm 2 shows the pseudo code of inserting a key-value pair, <K, V>, into the persistent skiplist. The steps involved are: (1) locate the position to insert a new node with K, and obtain the preceding/succeeding nodes at all levels through find() (line 1); (2) call GetFreeNodeAndInitial() to get a free node (pre-allocated via the make_persistent_atomic() API) from the free list and initialize it with V, and the next pointers pointing to the succeeding nodes (line 3-7); (3) starting from level 0 to top level, update the next pointer of the preceding nodes at all levels to the new node (line 8-21); (4) remove the new node from the free list (line 22). Note that when CAS is done, only the SmartRef on level 0 in the preceding node is marked as 'dirty'.

6 EXPERIMENTAL RESULTS

In this section, we first present the performance of FEDB with DRAM+SSD against two commercial in-memory database systems named as DB-X and DB-Y under real-time feature extraction workloads. Then we evaluate the different ways of using PMEM in FEDB under both micro benchmarks and a real-world fraud detection workload from 4Paradigm.

Table 2: DBMS Used in This Paper

Name	In-memory Storage Engine	Log Storage
D-FEDB	Volatile skiplist in DRAM	NVMe SSD
PM-FEDB	Volatile skiplist in PMEM (Memory Mode)	NVMe SSD
PA-FEDB	Persistent skiplist, All levels persist in PMEM (App Direct Mode)	N.A.
PO-FEDB	Persistent skiplist, Only level 0 persisted in PMEM (App Direct Mode)	N.A.
DB-X	Commercial in-memory database in DRAM	NVMe SSD
DB-Y	Commercial in-memory database in DRAM	NVMe SSD

6.1 Experiment Setup

For each database server, we have 28 Cores 2.7 GHz Xeon Platinum 8280L (2 sockets, 1.75MB/28MB/38.5MB for L1/L2/L3 caches of each socket, respectively), 384 GB (12×32GB) DRAM and 1.5 TB (12×128GB) PMEM, and we use 750 GB Intel Optane SSD DC P4800X as storage. The OS is CentOS-7 with kernel 5.1.9-1.el7. We use another dedicated server connected with 100GbE network to run the in-house benchmark tool based on JMH (Java Microbenchmark Harness) [46] which communicates with different databases through their JAVA client. The benchmark tool can either generate synthetic feature extraction queries or replay the traces from real-world feature extraction workloads in our fraud detection system. All results shown are the average of 10 runs for each experiment.

All the configurations of the databases used in this section are listed in Table 2. The server of the D-FEDB, DB-X, DB-Y keeps the same hardware configuration mentioned above except that those PMEM DIMMs are replaced by 32GB DRAM DIMMs (total 768 GB DRAM per server). D-FEDB and PM-FEDB are the same in software implementation but different in the memory used. PM-FEDB uses PMEM in *Memory Mode* as the working memory while D-FEDB uses DRAM. D-FEDB, PM-FEDB, DB-X and DB-Y use the default log and snapshot persistence model that is commonly found in in-memory databases. Both log and snapshot files are stored in the NVMe SSD with ext4 file system. PA-FEDB and PO-FEDB use PMEM in *App Direct Mode* to store the data persistently without log and snapshot. As shown in Fig. 7, PO-FEDB only FLUSHs the level 0 in the persistent skiplist, while PA-FEDB FLUSHs all the layers. The persistent skiplist is implemented using PMDK-1.7, libpmemobj-cpp-1.8 and memkind-1.9.

Experiment outline: We first compare the performance of the D-FEDB with the commercial in-memory databases DB-X and DB-Y under simulated OLDA workloads (Sec. 6.2). Then we use D-FEDB as a baseline to compare the performance of the storage engine of different PMEM-based FEDB variants under synthetic workloads such as put-only and scan-after-put (Sec. 6.3.1). Lastly, we use the real-world fraud detection workload from 4Paradigm discussed in Sec. 2.1 to compare the impact of using different FEDB variants in terms of latency, recovery time and total cost (Sec. 6.3.2).

6.2 Comparison with Existing Database

In this section, we evaluate the performance of D-FEDB against DB-X and DB-Y. We report the average latency normalized to the fastest latency. The table schema is the same as shown in Fig. 4. The table consists of *card_id* (varchar(20)), *timestamps* (bigint), *C_[1..N]* (varchar(20)) and *amount*(double). *C_1* to *C_N* stand for catalog 1 to catalog N, which are the columns ready for TopN queries. The combination of (card_id, timestamps) is the primary key. To speed up TopN queries, we create extra indexes for all the combinations of (catalogX, timestamps). At the data preparation stage, we first insert 2000 card_id from pk_1 to pk_2000, and for each card_id, we insert 1000 records with a 100 ms interval. The value of the catalog column is generated from the set {“group1”,“group2”,...,“group10”} randomly. When starting the test, we first insert a new record with a random card_id as well as a series of random values for *C_1* to *C_N* columns. Then we execute the same TopN feature extraction workload over multiple time windows (described in Sec. 3.1). We consider both the insertion and a series of the TopN queries as an entire transaction extraction procedure, and report its latency.

We performed two experiments. For the first experiment, we fix the number of catalogs to 2 (catalog1 and catalog2 only) and execute TopN queries over different number of time window. As show in Fig. 9, D-FEDB consistently outperforms both DB-X and DB-Y when the number of time windows increases from 1 to 8. In particular, the latency (1 time window) of DB-X and DB-Y is 3.25× and 8.79× that of the D-FEDB’s latency, respectively. The latency gap further increases to 4.74×, 6.94×, 8.73× (for DB-X) and 10.83×, 18.76×, 17.63× (for DB-Y) when the number of time window is increased to 2, 4 and 8, respectively. There are two factors that improved the performance of FEDB. First, the double-layered skiplist storage engine is optimized for the queries over a series of time windows. More importantly, the FEQL Engine only have to scan the data once over the largest time window and calculates the query result for all the time windows. That is the reason why

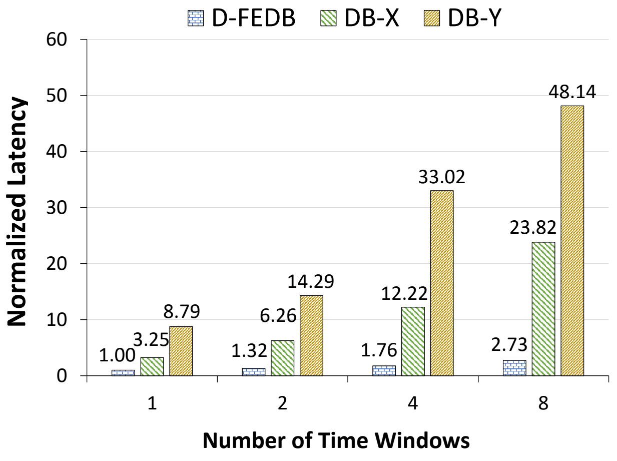


Figure 9: Latency Comparison of On-line Feature Extraction over Varied Number of Time Windows

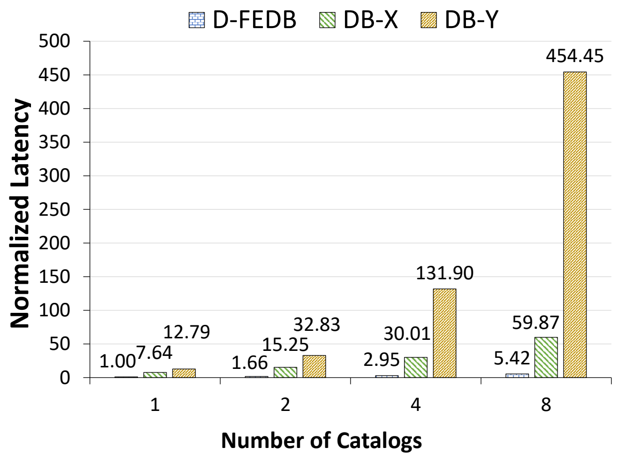


Figure 10: Latency Comparison of On-line Feature Extraction over Different Kinds of Keys

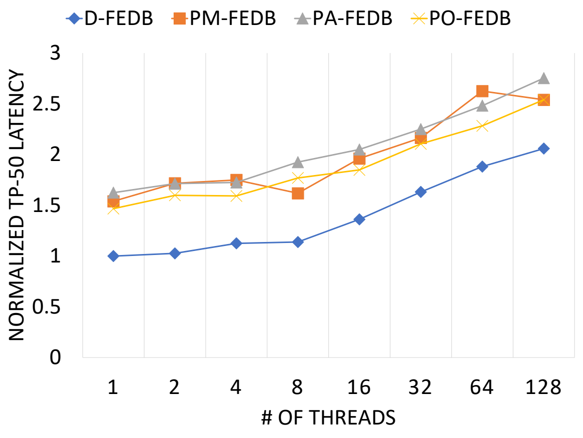
the more time windows are involved, the bigger the performance advantage D-FEDB gains.

In the second experiment, we fix the number of time windows to 10, and change the number of catalogs from 1 to 8. As described in Sec. 4.3, FEDB creates different double-layered skiplist for each index. Thus the TopN queries of different catalogX can execute in parallel through different double-layered skiplist indexes. As shown in Fig. 10, DB-X has 7.64×–11.05× higher latency compared with D-FEDB, and DB-Y performs worse with 12.79× – 83.85× longer latency than D-FEDB, respectively. On the other hand, the latency of D-FEDB only increases 66%, 195% and 442% when the number of the catalogs increases to 2, 4 and 8, respectively.

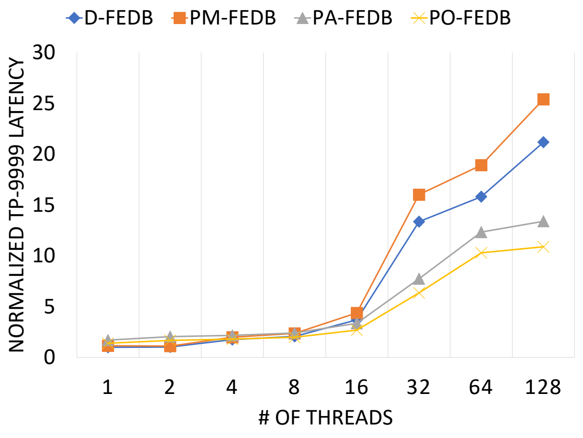
6.3 PMEM Optimized FEDB

In this section, we evaluate the effectiveness of using PMEM in FEDB. We use D-FEDB as a baseline to evaluate the performance of the different variants of the PMEM-based FEDB cluster. Since latency is the main performance metric in OLDA pipelines, we present the 50th/99th/9999th percentile latency [49] (denoted as TP-50/TP-99/TP-9999), normalized to the fastest latency.

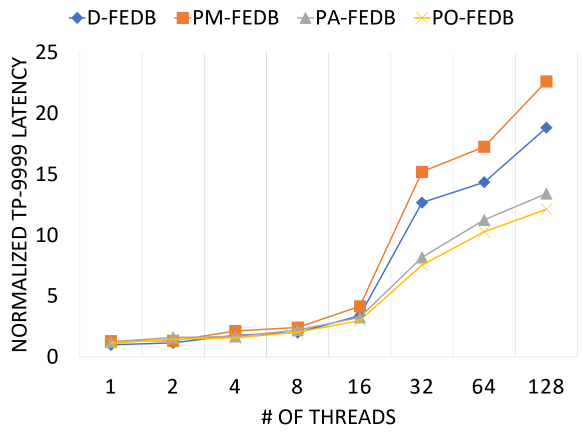
6.3.1 ***Micro-benchmark.*** To focus on the effect obtained from optimizing of the storage engine, we use direct operation commands instead of the FEQL interface to compare the performance of four FEDB variants summarized in Table 2. For each request for on-line inference in the OLDA systsem, FEDB performs a new record insertion followed by a large number of concurrent scans in the previous time windows. When the OLDA operations are pass to the storage engine level, it is converted into a series of put and scan operations. The latency of a put operation is important because the scan operations have to wait until freshly generated data is put into the database. Therefore, we first use a micro benchmark to measure the latency of the put operation. For a fair comparison, we preloaded about 400GB of data into the database before the measurement. The schema we use has a 8-byte key, a 100-byte



(a) TP-50 Latency of Put.



(b) TP-9999 Latency of Put.



(c) TP-9999 Latency of Put+Scan.

Figure 11: Performance Comparison of Different FEDB's Variants under Micro-benchmark Workloads

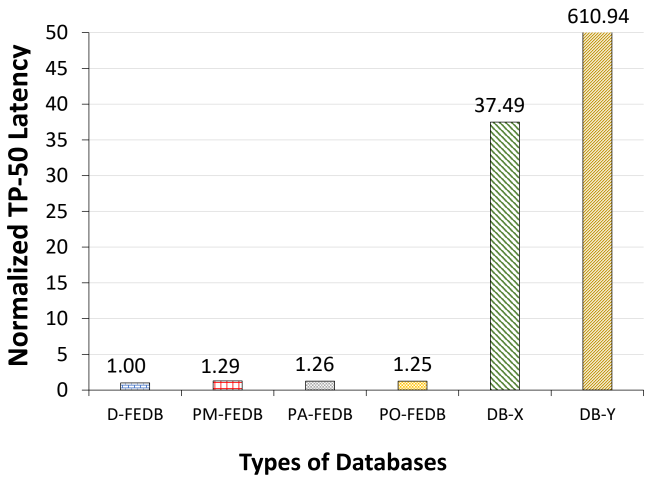


Figure 12: Latency under Real-world Workloads On Different DBMS

value and a 8-byte time stamp (ts). By default, every table has eight partitions and no replication.

Fig. 11a shows the TP-50 latency of put operations with varied number of threads. As expected, due to the performance advantage of DRAM, D-FEDB outperforms the other three configurations. In most cases, PM-FEDB performs the worst and is unstable because (1) the preloaded table has occupied all the DRAM cache for PMEM in *memory mode* and the put operation may trigger the data to be flushed from DRAM cache to PMEM frequently, and (2) it still needs to sync logs to the SSD periodically.

On the other hand, the TP-9999 latency of put in both PA-FEDB and PO-FEDB outperforms the other two configurations when the number of threads is larger than 8 as shown in Fig. 11b. Specifically, compared to D-FEDB and PM-FEDB, PA-FEDB can reduce the TP-9999 latency by up to 36.8% and 47.3%, respectively. In addition, since the data is only persisted in the bottom level, PO-FEDB can further reduce the TP-9999 latency by up to 48.6% and 57.1%, respectively. Meanwhile, PM-FEDB with the worst performance indicates that when persistency is required, utilizing PMEM in App Direct Mode is a more efficient choice.

To mimic the access pattern of AI-powered OLDA applications, we also measure the latency of a combined operation that consists of a put followed by a scan. As shown in Fig. 11c, although D-FEDB has the best scan performance due to the advantage DRAM has over PMEM in read performance, PO-FEDB still performs the best if we consider the combined operation especially when the number of threads exceeds 16. The TP-9999 latency can be reduced by as much as up to 35.6% compared to D-FEDB.

6.3.2 *Real-world Application Workload.*

We use a trace from the fraud detection application (mentioned in Sec. 2.1) to benchmark our FEDB variants. FEDB applies three replicas and the total size of in-memory data is around 10 TB. We deploy two FEDB clusters: DRAM cluster and PMEM cluster. The hardware configuration of both cluster is the same as mentioned in Sec. 6.1. For a fair comparison, we reserve 128 GB DRAM in each server in both clusters for the operating system, etc. We consider the total latency of one put operation followed by multiple time-windowed scans.

There are some interesting observations observed from the real-world workload. First, data access can be temporally out of balance. The frequency of fraud detection requests increases significantly before or during some public holidays as well as during the lunch/dinner time. Second, a large number of detection requests reach FEDB simultaneously, and it generates a series of bursts that may cause a longer waiting time. Most of the transactions with longer latency are issued during such “peak hour”. Third, more than 90% of the operations are read, while more than 80% of them are time-windowed access in this trace.

We test the fraud detection workload to evaluate on-line feature extraction latency. As shown in Fig. 12, D-FEDB yields the shortest latency, with both the PM-FEDB and the PA-FEDB achieving comparable performance. All variants of the FEDB are much faster than DB-X and DB-Y by approximately 30× and 600×, respectively. To quantify the effectiveness of using PMEM to optimize the tail latency, we focus on evaluating the performance of the different variants of FEDB. Fig. 13 shows the normalized TP-50/TP-90/TP-9999 latency comparison among four FEDB variants. In particular, TP-9999, which represents the worst-case response time, is 8.24× longer than TP-99 and 15.33× longer than TP-50 on D-FEDB, respectively. For TP-50 and TP-99, D-FEDB shows the best performance similar to those in micro-benchmark. PO-FEDB and PA-FEDB are 25-29% slower than D-FEDB in terms of TP-50 and TP-99, but they both outperform D-FEDB in TP-9999 latency. Specifically, PO-FEDB shows the shortest TP-9999 latency which is 19.7% faster than D-FEDB. Due to the additional FLUSH on the upper layer pointers in the persistent skiplist, PA-FEDB performs slightly worse than PO-FEDB but is still 17.2% faster than D-FEDB.

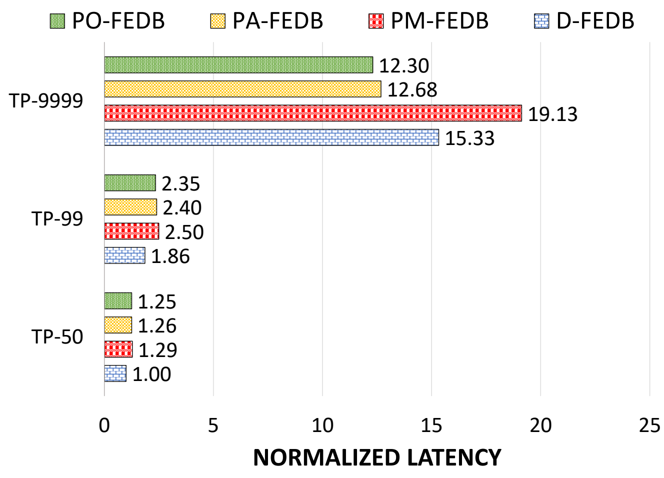


Figure 13: Tail Latency of Different FEDB’s Variants under Real-world Workloads

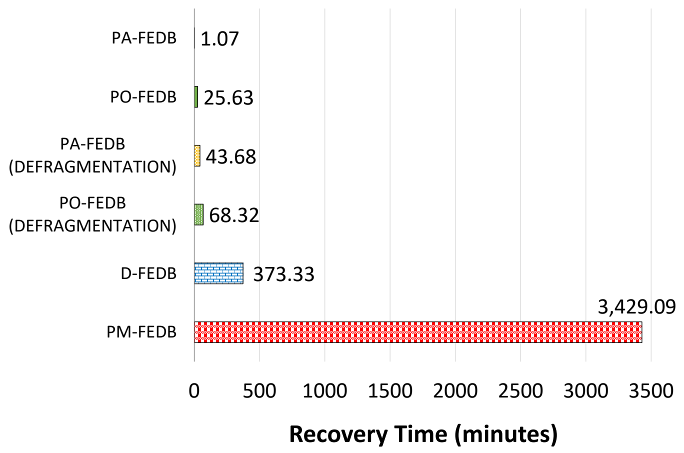


Figure 14: Recovery Time under Real-world Workloads

The optimization of TP-9999 significantly enhances the user experience, especially during the peak hours. On the other hand, PM-FEDB has the worst performance, which has 24.8-34.4% longer latency compared with D-FEDB in all kinds of latency. This suggests that using PMEM in a naïve way does not benefit performance.

Another benefit of using PMEM in *App Direct Mode* in FEDB is the much shorter recovery time upon system failure. Fig. 14 shows the recovery times of the six configurations. We added an option to execute PMEM space defragmentation after failure for PA-FEDB and PO-FEDB. D-FEDB and PM-FEDB have much longer recovery time compared with PA-FEDB and PO-FEDB because they have to load all the data from the snapshot on the SSD and replay the log. PM-FEDB requires 9.18× longer recovery time compared with D-FEDB. Since each PMEM server in the cluster has significantly more memory, PM-FEDB has to load more data from the SSD. This longer recovery time is also due to the inferior write performance of PMEM compared to DRAM. PA-FEDB is able to save 99.7% recovery time because it persists all the data in PMEM. As mentioned in Sec. 5.1, PO-FEDB needs to rebuild the pointer on the upper level of each node by scanning all the nodes through the pointers on level 0 of the skiplist. Thus, the recovery time of PO-FEDB is longer than that of PA-FEDB. Although PMEM space defragmentation in PO-FEDB and PA-FEDB is time consuming, the recovery time of both PO-FEDB (with defragmentation) and PA-FEDB (with defragmentation) can still be reduced by 81.7% and 88.3% compared to D-FEDB, respectively.

We also compare the overall cost between the original FEDB and the PMEM-optimized solution. As shown in Table 3, after excluding the DRAM space reserved for the operating system, etc, our PMEM server has 1.5 TB of PMEM and 256 GB of DRAM to store all of FEDB's data including both in PMEM and in DRAM (such as the upper layer of the skiplist). On the other hand, the DRAM server has 640 GB DRAM after reserving the same DRAM space as PMEM server. When deployed for fraud detection using DRAM server, the OLDA system will need sixteen DRAM servers to store all the tens of TB of in-memory data. The number of servers decreases to six if using PMEM servers. Both the lower dollar per GB of PMEM as well as the cost decreasing due to involving fewer servers (such as less power cost per year, etc) bring the cost reduction of PMEM server solution. PMEM server solution brings an overall cost that is 58.4% lower than the DRAM solution. Similar cost-saving result has been published previously [30].

Table 3: Cost Comparison: DRAM vs PMEM

	DRAM Server	PMEM Server
Memory Capacity	768 GB	1920 GB
DRAM	24×32GB	12×32GB
DRAM reserved for OS, etc	128GB	128GB
PMEM	N.A.	12×128GB
# of Servers Needed for the real-world trace	16	6
Normalized Cost	1	41.6%

7 CONCLUSIONS AND FUTURE WORK

In this paper, we have described and analyzed the entire workflow of AI-based OLDA applications using the example of a fraud detection pipeline from 4Paradigm. We have identified real-time feature extraction as the key performance bottleneck in OLDA applications. This prompted us to propose FEDB, a distributed in-memory database system designed to efficiently support real-time feature extraction in OLDA pipelines. FEDB can be one to two orders of magnitude faster than state-of-the-art in-memory databases under the real-world fraud detection workloads. To further reduce the total ownership cost of FEDB, we propose taking advantage of Intel's Optane DC Persistent Memory Module (PMEM), and extended FEDB with a PMEM-optimized persistent skiplist. Our experimental results show that PMEM-based FEDB can shorten the tail latency up to 19.7%, reduce the recovery time up to 99.7%, and save up to 58.4% of the total cost of OLDA system compared to FEDB using DRAM+SSD.

We are currently working on preparing a utility library to allow users to use our persistent skiplist[1]. Meanwhile, we are also working on integrating SparkSQL with FEDB[2]. This will allow SparkSQL users to use FEDB to seamlessly accelerate their AI-powered applications.

REFERENCES

[1] Paul Alcorn. 2019. Intel Optane DIMM Pricing. https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html. Last accessed on 02-July-2020.
[2] Alibabacloud. 2019. *Key Concepts and Features of Time Series Databases*. https://www.alibabacloud.com/blog/key-concepts-and-features-of-time-series-databases_594734 Last accessed on 02-July-2020.
[3] Salem Alqahtani and Murat Demirbas. 2019. Performance Analysis and Comparison of Distributed Machine Learning Systems. *arXiv:1909.02061* (2019).
[4] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, et al. 2017. SAP HANA adoption of non-volatile memory. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1754–1765.
[5] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. 2017. The case for heterogeneous HTAP. In *8th Biennial Conference on Innovative Data Systems Research*.

[1] https://github.com/4paradigm/pmemstore
[2] https://github.com/4paradigm/SparkSQLWithFeDB

[6] Jason Arnold, Boris Glavic, and Ioan Raicu. 2019. A High-Performance Distributed Relational Database System for Scalable OLAP Processing. In *2019 IEEE Int. Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 738–748.

[7] Joy Arulraj. 2019. Data Management on Non-Volatile Memory. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. 1114.

[8] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. BzTree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment* 11, 5 (2018), 553–565.

[9] Joy Arulraj and Andrew Pavlo. 2017. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 1753–1758.

[10] Joy Arulraj and Andrew Pavlo. 2017. How to Build a Non-Volatile Memory Database Management System. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 1753–1758. https://db.cs.cmu.edu/papers/2017/p1753-arulraj.pdf

[11] Joy Arulraj and Andrew Pavlo. 2019. Non-volatile memory database management systems. *Synthesis Lectures on Data Management* 11, 1 (2019), 1–191.

[12] Andrew Chen, Andy Chow, Aaron Davidson, Arjun DCunha, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Clemens Mewald, Siddharth Murching, Tomas Nykodym, et al. 2020. Developments in MLflow: A System to Accelerate the Machine Learning Lifecycle. In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*. 1–4.

[13] Cheng Chen, Qingsong Wei, Weng-Fai Wong, and Chundong Wang. 2019. NV-Journaling: Locality-Aware Journaling Using Byte-Addressable Non-Volatile Memory. *IEEE Trans. Comput.* 69, 2 (2019), 288–299.

[14] Cheng Chen, Jun Yang, Qingsong Wei, Chundong Wang, and Mingdi Xue. 2016. Fine-grained metadata journaling on NVM. In *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–13.

[15] Shimin Chen and Qin Jin. 2015. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797.

[16] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1077–1091.

[17] Salvatore Sanfilippo et. al. 2009. Redis. https://redis.io/. Last accessed on 02-July-2020.

[18] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA database: data management for modern business applications. *ACM Sigmod Record* 40, 4 (2012), 45–51.

[19] Shen Gao, Bingsheng He, and Jianliang Xu. 2015. Real-Time In-Memory Checkpointing for Future Hybrid Memory Systems. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 263–272.

[20] Google. 2019. *In-Memory Database.* https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory Last accessed on 02-July-2020.

[21] The TANEJA Group. 2012. *number of nines availability of systems.* http://tanejagroup.com/files/Compellent_TG_Opinion_5_Nines_Sept_20121.pdf Last accessed on 02-July-2020.

[22] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 1035–1050.

[23] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the 2019 Int. Conference on Management of Data*. 651–665.

[24] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In *USENIX annual technical conference*, Vol. 8. Boston, MA, USA.

[25] IDC. 2019. *IDC marketscape: Manufacturer evaluation of China machine learning development platform 2019.* https://www.idc.com/getdoc.jsp?containerId=CHC45389019 Last accessed on 02-July-2020.

[26] Timescale Incorporated. 2019. *TimescaleDB.* https://github.com/timescale/timescaledb Last accessed on 02-July-2020.

[27] InfluxData. 2019. influxDB. https://www.influxdata.com/. Last accessed on 02-July-2020.

[28] Intel. 2015. Intel® Optane™ DC persistent memory. "https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html. Last accessed on 02-July-2020.

[29] Intel. 2015. Pmem.io. https://pmem.io/libpmemobj-cpp/, Last accessed on 26-January-2020.

[30] Intel. 2019. *The Challenge of Keeping up with data.* https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf) Last accessed on 02-July-2020.

[31] Intel. 2019. *Introduction to programming for persistent memory.* https://github.com/pmemhackathon/2019-11-08/blob/master/PMEM_INTRO.pdf Last accessed on 02-July-2020.

[32] Intel. 2019. Ipmctl. https://github.com/intel/ipmctl. Last accessed on 02-July-2020.

[33] Intel. 2019. libpmemobj. https://github.com/pmem/libpmemobj-cpp/, Last accessed on 02-July-2020.

[34] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[35] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, et al. 2015. Oracle database in-memory: A dual format in-memory database. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 1253–1258.

[36] Berti-Equille Laure, Bonifati Angela, and Milo Tova. 2018. Machine learning to data management: A round trip. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1735–1738.

[37] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. 2017. {WORT}: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 257–270.

[38] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. 2018. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 611–626.

[39] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. 2015. Efficient Processing of Window Functions in Analytical SQL Queries. *Proceedings of the VLDB Endowment* 8, 10 (2015), 1058–1069.

[40] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302* (2020).

[41] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. 2018. Accelerating machine learning inference with probabilistic predicates. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 1493–1508.

[42] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. 2017. BatchDB: Efficient isolated execution of hybrid OLTP + OLAP workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 37–50.

[43] MemSQL. 2013. https://www.memsql.com/, Last accessed on 02-July-2020.

[44] Microsoft. Oct 30, 2018. *Windows Server 2019 with Intel® Optane™ DC persistent memory.* https://techcommunity.microsoft.com/t5/Storage-at-Microsoft/The-new-HCI-industry-record-13-7-million-IOPS-with-Windows/ba-p/428314 Last accessed on 02-July-2020.

[45] MySQL. 1995. https://www.mysql.com/, Last accessed on 02-July-2020.

[46] OpenJDK. 2013. https://openjdk.java.net/projects/code-tools/jmh/, Last accessed on 02-July-2020.

[47] Oracle. Sep 16, 2019. *Oracle Database with Intel Optane DC Persistent Memory.* https://www.oracle.com/corporate/pressrelease/oow19-oracle-intel-partner-optane-exadata-091619.html Last accessed on 02-July-2020.

[48] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, 371–386.

[49] Top percentile. 2019. TP-X. https://support.huaweicloud.com/intl/en-us/productdesc-apm/apm_06_0002.html. Last accessed on 02-July-2020.

[50] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2018. Data lifecycle challenges in production machine learning: a survey. *ACM SIGMOD Record* 47, 2 (2018), 17–28.

[51] Georgios Psaropoulos, Ismail Oukid, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Bridging the latency gap between NVM and DRAM for latency-bound operations. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. ACM.

[52] Alexander Ratner, Dan Alistarh, Gustavo Alonso, David G Andersen, Peter Bailis, Sarah Bird, Nicholas Carlini, Bryan Catanzaro, Eric Chung, Bill Dally, et al. 2019. SysML: The New Frontier of Machine Learning Systems. (2019).

[53] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid training data creation with weak supervision. *Proceedings of the VLDB Endowment* 11, 3, 269.

[54] Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. 2020. Adaptive HTAP through Elastic Resource Scheduling. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 2043–2054.

[55] Theodoras Rekatsinas, Sudeepa Roy, Manasi Vartak, Ce Zhang, and Neoklis Polyzotis. 2019. Opportunities for data management research in the era of horizontal AI/ML. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2323–2323.

[56] Babak Salimi, Corey Cole, Peter Li, Johannes Gehrke, and Dan Suciu. 2018. HypDB: a demonstration of detecting, explaining and resolving bias in OLAP queries. *Proceedings of the VLDB Endowment* 11, 12 (2018), 2062–2065.

[57] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q Ngo, and XuanLong Nguyen. 2019. A layered aggregate engine for analytics workloads.

In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. 1642–1659.

[58] Ji Sun, Zeyuan Shang, Guoliang Li, Dong Deng, and Zhifeng Bao. 2017. Dima: A distributed in-memory similarity-based query processing system. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1925–1928.

[59] Tracy Tsai. 2019. *Competitive Landscape: AI Startups in China.* Technical Report. Stamford, USA.

[60] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. ACM, New York, NY, USA, 1541–1555.

[61] Manasi Vartak, Joana M F. da Trindade, Samuel Madden, and Matei Zaharia. 2018. Mistique: A system to store and query model intermediates for model diagnosis. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 1285–1300.

[62] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory.. In *FAST*, Vol. 11. 61–75.

[63] Chundong Wang, Sudipta Chattopadhyay, and Gunavaran Brihadiswarn. 2019. Crash recoverable ARMv8-oriented B+-tree for byte-addressable persistent memory. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 33–44.

[64] Chundong Wang, Qingsong Wei, Lingkun Wu, Sibo Wang, Cheng Chen, Xiaokui Xiao, Jun Yang, Mingdi Xue, and Yechao Yang. 2018. Persisting RB-tree into NVM in a consistency perspective. *ACM Trans. on Storage (TOS)* 14, 1 (2018), 1–27.

[65] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 461–472.

[66] Wikipedia. 2019. *Compare-and-swap*. https://en.wikipedia.org/wiki/Compare-and-swap Last accessed on 02-July-2020.

[67] Wikipedia. 2019. *LLVM*. https://en.wikipedia.org/wiki/Click-through_rate Last accessed on 02-July-2020.

[68] Jian Xu and Steven Swanson. 2016. {NOVA}: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies ({FAST} 16)*. 323–338.

[69] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: reducing consistency cost for NVM-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. 167–181.

[70] Luo Yuanfei, Wang Mengshuo, Zhou Hao, Yao Quanming, Tu WeiWei, Chen Yuqiang, Yang Qiang, and Dai Wenyuan. 2019. AutoCross: Automatic Feature Crossing for Tabular Data in Real-World Applications. *arXiv preprint arXiv:1904.12857* (2019).

[71] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, et al. 2019. AnalyticDB: real-time OLAP database system at Alibaba cloud. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2059–2070.

[72] Yu Zhang, Shan Wang, Jiaheng Lu, et al. 2018. Fusion OLAP: Fusing the Pros of MOLAP and ROLAP Together for In-memory OLAP. *IEEE Transactions on Knowledge and Data Engineering* 31, 9 (2018), 1722–1735.