Designing a persistent-memory-native storage engine for SQL database systems

Shohei Matsuura
Database Department, Service Platform Division, Technology Group
Yahoo Japan Corporation
Tokyo, Japan
shmatsuu@yahoo-corp.jp

***Abstract*—We illustrate the design of our in-house storage engine for SQL database systems. The storage engine is designed to be persistent-memory native, meaning that database and transaction log files are placed on persistent memory and accessed with byte granularity from the storage engine. In addition, it is aimed to be practical in industry and highly performant with the use of persistent memory. In this paper, we discuss five essential requirements for such a storage engine to be practical in industry and how they are met in our in-house storage engine. Furthermore, we highlight two important design features, namely, (1) the pre-fault feature and (2) the parallel-logging feature, that have been incorporated to our in-house storage engine, to improve its performance. By meeting the five essential requirements and incorporating the two design features to our in-house storage engine, we implement a persistent-memory-native storage engine for SQL database systems, in-house, that satisfies industry requirements and that is highly performant on write workload on persistent memory.**

Keywords—Persistent Memory, Non-volatile Memory, Database Systems, SQL Databases, RDBMS, Storage Engine

I. INTRODUCTION

Recent introduction of byte-addressable persistent memory such as Intel® Optane™ DC Persistent Memory has elicited database researchers to revise the traditional architecture of SQL database systems to the one that is more suitable for persistent memory. As described in [12], the traditional architecture of SQL database systems assumes that the durable storage of a database is a non-volatile block device such as HDDs or SSDs and that there is a large performance gap in I/Os between such a non-volatile block device and the volatile DRAM. To hide this performance gap in I/Os between these two devices, traditional database systems employ techniques such as writing data to DRAM temporarily and persisting them to a non-volatile block device asynchronously at a checkpoint using write-ahead logging (WAL) algorithm.

Since the introduction of byte-addressable persistent memory, database researchers have begun actively exploring the use of byte-addressable persistent memory as the durable storage of a database to improve the database performance and proposing new database architectures that are more suitable for persistent memory. For example, the studies [7] and [8] explore three designs for a persistent-memory-native storage engine of SQL database systems. In industry, Oracle Corporation's SQL database hardware appliance, Oracle Exadata X8M-2, is now equipped with persistent memory to accelerate transaction log writing [6] [14]. Another example is Microsoft SQL Server 2019. It implements a feature called "Hybrid Buffer Pool", that directly reads data on persistent memory without loading it to DRAM buffer in read operations and that persistent memory serves as an extension of DRAM buffer [13].

With the use of persistent memory as the durable storage for database systems, the performance gap between the durable storage and DRAM becomes much smaller than ever, giving database systems an opportunity to improve their performance. However, the existing research in academia on persistent-memory-native storage engines does not discuss nor present requirements in detail for such storage engines to be practical in industry, that is, for example, it does not consider how to deal with the storage expansion as the data volume grows. Even in industry, the use of persistent memory is now limited to a small part of database operations such as logging or data buffering. Instead, in our research, we envision to expand the use of persistent memory further and to design a persistent-memory-native storage engine for SQL database systems that can satisfy industry requirements. In this paper, we discuss essential requirements for a practical persistent-memory-native storage engine from an industry point of view and illustrate the architecture of our in-house storage engine satisfying them. In addition, we present two important features in our in-house storage engine, (1) the pre-fault feature and (2) the parallel-logging feature, to improve the storage engine performance on persistent memory.

The contributions of this paper are: (1) stating requirements for a persistent-memory-native storage engine for SQL databases to be practical in industry, (2) an illustration of the in-house persistent-memory-native storage engine, designed at Yahoo Japan, to meet the requirements, (3) highlighting two important features and their effects on persistent memory to improve the performance of the in-house storage engine.

The rest of this paper is organized as follows. Section II is the preliminaries. Section III discusses the requirements for a practical persistent-memory-native storage engine from an industry point of view. Section IV and V discuss the design of our in-house storage engine to meet the requirements and the pre-fault feature and the parallel-logging feature. Section VI presents the evaluation results of our in-house storage engine, and Section VII concludes the paper by summarizing our work and the future work.

978-1-6654-2375-5/21/$31.00 ©2021 IEEE

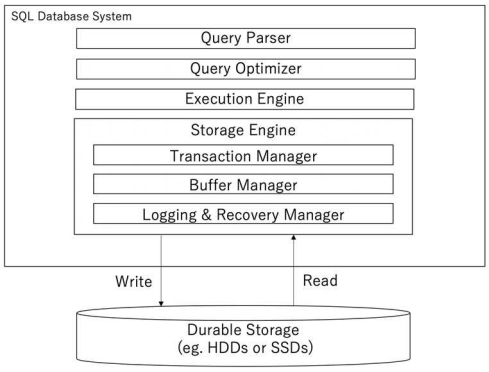

Fig. 1 Traditional Architecture of SQL Database Systems

II. PRELIMINARIES

In this section, we briefly describe the traditional architecture of SQL database systems and existing research to adapt database systems to persistent memory.

A. Traditional Architecture of SQL Database Systems

The traditional architecture of SQL database systems, described in [4] and [12], is depicted in Fig. 1. Although we can describe the architecture in more detail, we only highlight the major components that are relevant to our discussion. A traditional SQL database system consists of the following major components below:

- Query Parser: This component parses users' queries and transform them into a form that can be passed to the next component in query processing, Query Optimizer, for optimizing the queries.
- Query Optimizer: This is the component to optimize users' queries. For example, based on the presence of an index, it determines to use an index for data retrieval, rather than fully scanning the entire data and filtering out unnecessary records to form the desired result set.
- Execution Engine: This is the component to execute users' queries, following the direction of the query optimizer. It requests the storage engine to perform I/Os to the durable storage.
- Storage Engine: This component issues I/Os to the durable storage at a request of the execution engine. The component is also equipped with a transaction manager and a logging & recovery manager, to provide users with transaction support. In addition, it is equipped with a buffer manager that uses DRAM as the buffer to hide the I/O performance difference between the durable storage and DRAM.

The use of persistent memory as the durable storage for a database makes the I/O performance between the durable storage and DRAM closer than before, giving SQL database systems that adapt it a good opportunity to improve their performance. There is active research going on both in academia and industry to revise the traditional architecture of SQL database systems to a more optimal one for persistent memory. Since the storage engine is the component that can get direct benefits from it, the current research in the area mainly focuses on re-architecting the storage engine component to the one more suitable for persistent memory. In the next section, we highlight some of such research both in academia and industry.

B. Existing Research to Adapt Persistent Memory

In this section, we introduce some of existing research to adapt persistent memory in SQL database systems.

The research [7] and [8] study three designs for a persistent-memory-native storage engine. One design is NVM-InP Engine, where data are synchronously persisted on persistent memory and old data are overwritten with the new data at the same location, upon an update operation. Second design is NVM-CoW Engine. This design also persists data on persistent memory synchronously but uses the copy-on-write mechanism in update. The third design is NVM-log Engine that employs LSM-tree as the storage model. In this design, the SSTable is placed on persistent memory and a transaction log record contains only a pointer to a record on the SSTable on persistent memory to avoid data duplication. These studies compare the performance of the three storage engines, on an emulator, using several benchmark workloads, but they lack to discuss how to make them practical in industry. For example, they do not discuss how to deal with the storage expansion as the data volume grows and how to handle the case when the transaction log files become full, but they need to continue transaction processing. These two cases always happen in the real world, especially in the internet industry, because it keeps recording users' transactions 24 hours/365 days online and the data volume and the transaction log files increase monotonically.

In industry, Oracle Exadata X8M-2 uses persistent memory to accelerate transaction log writing [6] [14]. It consists of database servers that run a database service and storage servers that provide the database service with storage. When a database server commits a transaction, it writes transaction logs to persistent memory on the storage servers with RDMA-writes. Upon a successful completion of the RDMA-writes, the transaction successfully completes. The logs on persistent memory are then periodically flushed to a slower durable storage, SSDs or HDDs. With this architecture, Oracle reports that it has achieved up to 8x faster log writes than before [6].

Microsoft SQL Server 2019 is another example to adapt persistent memory. It implements a feature called "Hybrid Buffer Pool" [13] and expands the capacity of the regular DRAM buffer. Data placed on persistent memory are memory-mapped and directly read from there without them being copied to the DRAM buffer. This architecture reduces the number of memory copies in query processing, hence improving the database performance. It also contributes to reducing the amount of DRAM required for the regular buffer to meet a performance goal.

To extend the results of these existing research and the use of persistent memory further, at Yahoo Japan, we conduct research to design a storage engine for SQL database systems that natively uses persistent memory and that can be used to host our internet services.

In the next section, we discuss the requirements we have set in designing such a storage engine and what makes a storage engine practical in industry.

III. Requirements for a Practical Persistent-memory-native Storage Engine

Yahoo Japan is an internet service company in Japan with more than 52 million login users monthly and delivers more than 100 internet services from e-commerce to online news service to its users. Thousands of SQL database instances are running in Yahoo Japan's data centers for these internet services. To adapt our SQL database systems to the new hardware, persistent memory, and to improve the systems' performance, we design a practical persistent-memory-native storage engine in-house. In designing such a storage engine, we consider how to meet the current requirements for our SQL database systems to deliver reliable internet services and database operations with our new storage engine. Based on this consideration, we derive the following five requirements for a practical persistent-memory-native storage engine:

- Requirement #1: Scale with Data Volume

The storage engine must scale with data volume. The total volume of data has been growing each year at Yahoo Japan, as it expands its business. In addition, our internet services operate 24 hours/365 days online and keep recording users' transactions in our SQL database systems, increasing the data volume. In order to satisfy this business requirement, it must scale with the data volume growth.

- Requirement #2: Transaction

Many of our internet services require transaction support at the database layer, because they require consistent views to users while running multiple queries to a database simultaneously and committed transactions are never lost. To fulfill this requirement with the new storage engine, it needs to support transaction as well. It must be able to respond to simultaneously running queries with consistent views, persist committed transactions, and recover a database to a consistent state after a failure.

- Requirement #3: Continuous Operation

Our internet services operate 24 hours/365 days. To support such non-stop internet services, database systems must be able to perform necessary operations such as transaction logging without stopping them. To fulfill this requirement in the new storage engine, it must be capable of performing operations such as transaction logging in any case without stopping database instances.

- Requirement #4: Performance

Database performance is critical in internet services, as it directly impacts the application performance and the user experience. For a better internet service experience, database systems must be performant as much as possible. Since the new storage engine is aimed to be persistent-memory native and to improve the performance, it must be performant and get benefits from the use of the new device.

- Requirement #5: MySQL Compatibility

Yahoo Japan is one of the largest MySQL operators in Japan. As a result, a large number of our internet services run on it. To benefit these services with the new hardware technology, persistent memory, and for their smooth transition to the new storage engine, we require the new storage engine to be MySQL compatible as much as possible.

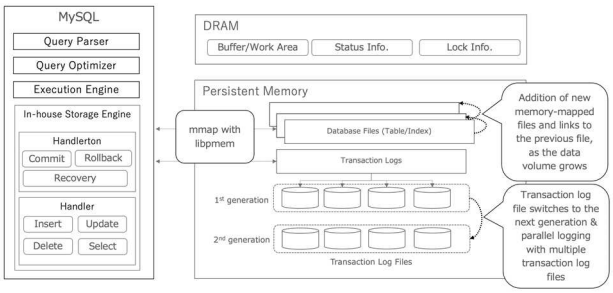

Fig. 2 Design of the In-house Persistent-memory-native Storage Engine

IV. Design of a Practical In-house Persistent-memory-native Storage Engine

In this section, we present our design of a practical persistent-memory-native storage engine. The design is shown in Fig. 2. To be persistent-memory native, we place both the database files and the transaction log files on persistent memory formatted in FS-DAX mode [16] and use the memory as their durable storage. The storage engine accesses these as memory-mapped files in byte granularity. To access these files as memory-mapped files, we utilize the low-level library, libpmem, in PMDK (Persistent Memory Development Kit) [15] and flush CPU cache to ensure the durability of data as needed. In this new storage engine, we also use DRAM. On DRAM, we secure the area "Buffer/Work Area" that is used as work area for sorting and distinction of data when processing queries. On DRAM, we also secure the area to store status information such as the number of active connections and the number of transactions that have been executed, and the information about locks that active transactions acquire and wait for.

To scale with data volume, for meeting the requirement #1, we design our storage engine to create a new memory-mapped file on persistent memory of a specified size by a parameter in the configuration file, when it tries to insert a new record, but no more space is available for the new record in the existing memory-mapped file. After it creates a new memory-mapped file, it inserts the new record to the newly created file and links it to the previous file as a list. In this manner, as more data is stored with this storage engine, a linked list of database files is created. When it is directed for a full table scan by the optimizer and the executor, it traverses the link of the memory-mapped database files until the end to scan the entire table.

To support transaction, the requirement #2, we employ well-proven techniques. For atomic operation and recovery operation to a consistent state, we implement transaction logging and ARIES described in [12]. For committed transactions, we roll-forward to reflect them to the database, and for uncommitted transaction, we undo them from the database, to recover the database to a consistent state. To provide a consistent view to each of concurrently running queries, we employ the snapshot isolation described in [18]. For atomic operation and recovery

Table I. Evaluation Environment

CPU	Intel Xeon Gold 6230R 2.1GHz x 2 (Total 104 Cores)
DRAM	DDR4-192GB
Persistent Memory	Intel® Optane™ DC Persistent Memory
SSDs	SATA SSD 1.92TB (OS Boot, Load Data)
OS	CentOS 7.8
DBMS	MySQL 8.0.19 with the In-house Storage Engine

operation, transaction logs are placed on persistent memory and memory-mapped to the storage engine to make the engine persistent-memory native.

Also, to support continuous operations of database systems, the requirement #3, we place transaction log files on persistent memory that have a generation. Transaction logs are always appended to a transaction log file and monotonically increases in size. So, if a transaction log file becomes full at some point, the storage engine makes that transaction log file inactive and switches an active transaction log file to a new one, a new generation of a transaction log file, online, to continue operations. This is the generation switch of a transaction log file online. By switching an active transaction log file to the next generation online, the storage engine keeps operating without stopping transactions even in the case that a transaction log file becomes full. If the last generation of a transaction log file is used up, we then go back to the first generation and write a new transaction log there.

In Fig. 2, one might notice that there are multiple transaction log files in one generation. This is to write transaction logs in parallel to improve the logging performance of the storage engine and to address the requirement #4. We explain the details on this point in Section V.

To satisfy the requirement #5, the compatibility with MySQL, we design and implement our storage engine as a MySQL's storage engine. MySQL has a pluggable storage engine architecture, that users can choose a storage engine, e.g. InnoDB, MyISAM, CSV, etc, based on their needs. As we focus on the storage engine, we do not modify the other components of MySQL including the query parser, the optimizer, and the execution engine, in our research. Operations such as insert, update, delete, and select to an SQL table are implemented by implementing the abstract class "handler" that specifies APIs of table operations in MySQL. Transaction operations such as commit, rollback, and recovery are implemented by implementing the abstract class "handlerton" in MySQL.

V. Pre-fault Feature & Parallel-Logging Feature

In this separate section, we highlight the following two features that are incorporated to our in-house storage engine to benefit from the performance of persistent memory.

A. Pre-fault Feature

The first feature is the pre-fault feature. It is a feature to eagerly cause page-fault against memory-mapped files on persistent memory before the storage engine accesses them for query processing. It is known that page-fault causes the

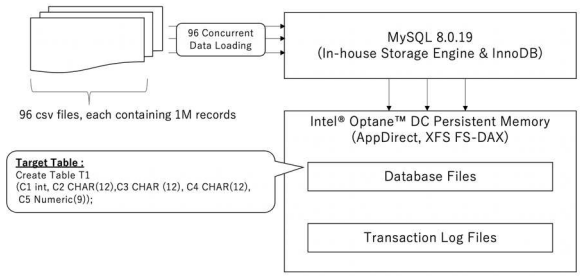

Fig. 3 Evaluation Workload

significant performance overhead when accessing memory-mapped files on persistent memory [11]. To eliminate this performance overhead in our in-house storage engine and to benefit from the use of persistent memory in performance, we design a feature to initiate page-fault in advance before the storage engine accesses to a new region of a memory-mapped file. For this purpose, we implement, in our storage engine, dedicated threads, separate from the storage engine's main threads, and we call this feature as the pre-fault feature. These dedicated threads start running before the storage engine starts accepting table operations upon a start of MySQL server and start scanning the database files and the transaction log files on persistent memory that are memory-mapped to the storage engine. When the storage expansion happens, they start scanning the newly created file before the storage engine's main threads access it. With this design, we can eliminate the occurrence of page-fault and the performance decrease of our storage engine due to it. The effect of this feature is studied in Section VI.

B. Parallel-logging Feature

In addition to the pre-fault feature, we also implement a feature to write transaction logs in parallel in the storage engine to improve the performance of our storage engine. In transaction systems, it is widely known that the transaction logging is a major source of performance bottleneck. This is, in one aspect, because transaction logs must be totally ordered to secure the consistency of databases, requiring the serialization of concurrently running transactions. However, recently, database researchers have proposed an algorithm to increase the parallelism in transaction log writing, while securing the total orders of transactions [17]. In our storage engine, we employ this state-of-the-art parallel logging algorithm for writing transaction logs. For this purpose, we form a generation of transaction log files with multiple transaction log files. When concurrently running transactions request to persist their transaction logs simultaneously, we order each of these transactions with a centralized counter and use a hash function to distribute each transaction to write their transaction logs to different transaction log files in parallel. In this manner, we can increase the degree of parallelism and benefit from the higher I/O bandwidth of persistent memory in transaction log writing.

VI. Evaluation & Discussion

In this section, we evaluate the performance of our in-house persistent-memory-native storage engine, compared to the performance of InnoDB running on persistent memory. InnoDB is a widely used storage engine in MySQL and it supports

Table II. Loading Time with and without Pre-fault Feature

Storage Engine	In-house Storage Engine with Pre-fault Feature	In-house Storage Engine without Pre-fault Feature
Loading Time	1	5.86

transactions. In addition, we discuss the effects of the two features, (1) the pre-fault feature and (2) the parallel-logging feature, in our storage engine and how they contribute to improving the performance of our storage engine on persistent memory.

Based on the design described in the previous two sections, we have implemented our in-house storage engine for MySQL 8.0.19 and evaluated its performance on the hardware environment described in Table I. In the evaluation, we configure Intel® Optane™ DC Persistent Memory in AppDirect mode with interleaving and format it in XFS FS-DAX mode, on which we place the database files and the transaction log files. And, to determine the effects of the pre-fault feature, we implement two operation modes in our storage engine, one with the feature and the other without the feature.

A. *Evaluation Workload*

In order to determine the effectiveness of the pre-fault feature and the parallel-logging feature, we choose a write-only workload, data loading, in our evaluation. It is because a good workload to determine their effectiveness. In data loading, the storage engine always accesses a new region of memory-mapped database files to append new data, that always incurs page-fault if no care is taken. By comparing the data loading time of the storage engine with the pre-fault feature and without the feature in data loading, we can evaluate the effectiveness of the pre-fault feature. In addition, since in data loading a large amount of transaction log records are generated and written to the transaction log files, it is also a good workload to see the effect of the parallel-logging feature.

The details of the evaluation workload are shown in Fig. 3. Since we use a server with 104 cores in our evaluation, we run the data loading at the 96 concurrency, the closest power of 2 that this server can run threads simultaneously, to utilize nearly all the cores, splitting the original data into 96 separate csv files. Each csv file contains one million records in it, and each thread is assigned to a dedicated csv file to load the data. To avoid any additional overheads in our evaluation, we do not define any indexes or constraints to the target table. The target table has a very simple structure with just 5 columns. We then compare the loading performance of our in-house storage engine and InnoDB on persistent memory in MySQL 8.0.19.

B. *Effect of Pre-fault Feature*

First, we discuss the effect of the pre-fault feature. Table II shows the data loading time of our in-house storage engine with the pre-fault feature and without the feature. It is shown with a relative number, making the loading time of the storage engine with the feature as 1. As we can see from Table II, the storage engine with the pre-fault feature has a faster loading time and improves the data loading performance more than 5x, compared to the storage engine without it. Fig. 4 shows the CPU utilizations and the output of perf during the data loading in both cases. As we can see, if we don't implement the feature, most of the CPU time is spent by the kernel to handle page-fault, and only a small portion of CPU time is consumed by the userspace, causing a performance degradation of the storage engine during the data loading. With pre-fault, it is not consumed by the kernel to handle page-fault, but it is mostly consumed by the userspace functions such as "row_to_heap" function of the storage engine. From this result, we can see the effectiveness of the pre-fault feature and its importance when designing a persistent-memory-native storage engine for SQL database systems.

C. *Effect of Parallel-logging Feature*

Next, we discuss the effect of the parallel-logging feature. This is the feature to write transaction logs in parallel to improve the performance of the storage engine. For the evaluation of this feature, we use the same data loading workload in Fig. 3 and evaluate how the loading time changes with respect to the number of parallel log write. In this evaluation, we use the storage engine with both the pre-fault feature and the parallel-logging feature.

The evaluation result is shown in Table III. In the table, it compares the loading time with a different degree of parallel log

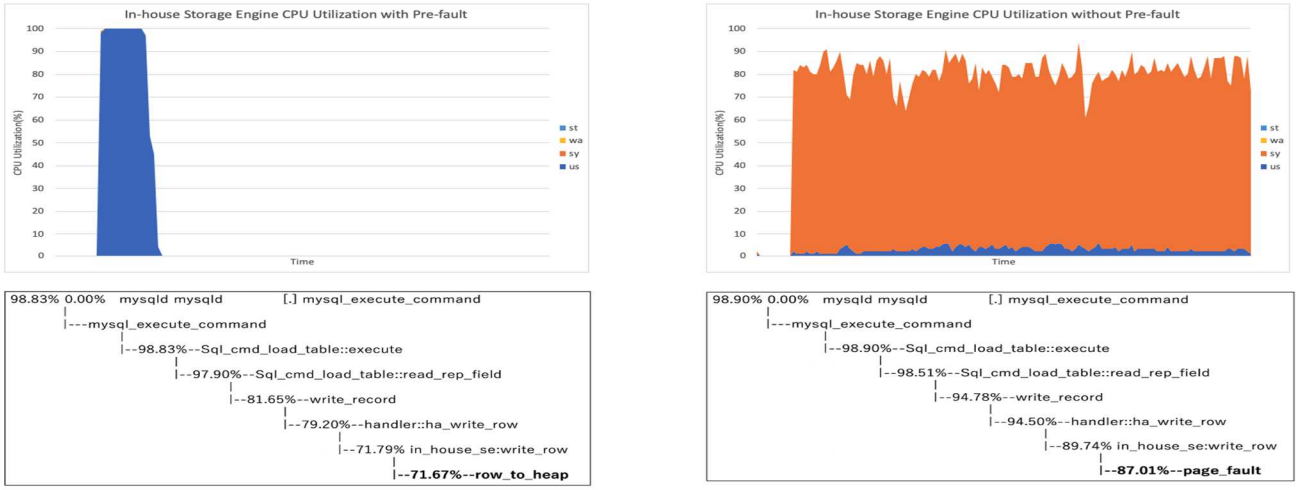

Fig. 4 CPU Utilizations and Perf Outputs of the In-house Storage Engine with and without Pre-fault Feature

Table III. Loading Time with the Number of Parallel Log Write

Number of Parallel Log Write	1	2	4	8	10	12	16
Loading Time	1.31	1.06	1.00	1.17	1.18	1.05	1.28

write, making the loading time with a 4 parallel log write as 1. As we can see, with the parallel-logging feature, it can improve the loading performance more than 30%. However, increasing the degree of parallelism too much also causes a performance degradation. It may be that this is due to the contention on the centralized counter to totally order transaction logs or due to I/O bandwidth saturation of persistent memory, but we need to further investigate the cause in our future research.

D. *Performance Improvement by the In-house Storage Engine*

Finally, we discuss the performance improvement achieved by the in-house persistent-memory-native storage engine. It is compared to InnoDB operating on persistent memory, where its database files and transaction log files are placed on persistent memory, in data loading. Our in-house storage engine runs with the pre-fault feature and with the parallel-logging feature with a 4 parallel log write.

The result is shown in Table IV. The loading time is shown with a relative number, making the loading time of our storage engine as 1. As we can observe from this table, our in-house storage engine exhibits more than 50x faster data loading time than InnoDB on persistent memory, achieving a significant performance improvement with our engine.

VII. Conclusion & Future Work

In this paper, we describe our in-house persistent-memory-native storage engine that uses persistent memory as the durable storage of a database and transaction log files. It is designed to be practical in industry and performant on persistent memory. Our evaluation shows that, in write workload, it is more than 50x performant than InnoDB, on persistent memory. Also, the evaluation shows the contributions of the pre-fault and the parallel-logging features in the storage engine in improving its performance.

At the end, we discuss our future work. In this research, we implement a feature to expand database files on persistent memory, as the data volume grows. Although our design works well, as long as the data fits on persistent memory, we need to go further to handle the case the data volume exceeds the capacity of persistent memory. In this case, we employ another technique, data-tiering, by combining persistent memory and non-volatile block devices with a larger capacity, to handle more data. Another feature that we plan to design is the high-availability feature. With transaction logs, we can recover a database to a consistent state after a failure. Additionally, to ensure our internet services do not stop even in the case of a data center failure, we implement the high-availability feature in our storage engine.

Table IV. Loading Time of the In-house Storage Engine and InnoDB

Storage Engine	In-house Storage Engine with Pre-fault & Parallel-logging Features	InnoDB on Persistent Memory
Loading Time	1	58.29

References

[1] A. Renen, L. Vogel, V. Leis, T. Neumann, A. Kemper, "Persistent Memory I/O Primitives," DaMON '19: Proceedings of the 15th International Workshop on Data Management on New Hardware, 2019, pp. 1-7.

[2] A. Rudoff, Persistent Memory Programming, ; login:; Vol 42. No. 2, 2017.

[3] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwarz, "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," ACM Transactions on Database Systems, Vol. 17, No. 1, 1992, pp. 94-162.

[4] H. Garcia-Molina, J. Ullman, J. WIDOM, *Database System Implementation*, Prentice Hall, NJ, USA, 2000.

[5] Intel® Optane™ DC Persistent Memory DC Persistent Memory, Intel Corporation. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

[6] J. Shi, Exadata with Persistent Memory: An Epic Journey. SNIA Persistent Memory Summit 2020. https://www.snia.org/sites/default/files/PM-Summit/2020/presentations/11_PMEM_Jia_Shi_final_PM_Summit_2020_v2.pdf

[7] J. Arulraj, "The Design and Implementation of a Non-Volatile Memory Database Management System," Ph.D thesis, Carnegie Mellon University, 2018.

[8] J. Arulraj, A. Pavlo, *Non-Volatile Memory Database Management Systems*, Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2019.

[9] J. Arulraj, A. Pavlo, S. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," SIGMOD '15: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015, pp. 707-722.

[10] J. Arulraj, M. Perron, A. Pavlo, "Write-behind logging," Proceedings of VLDB Endorsement, Vol. 10, No. 4, 2016, pp. 337-348.

[11] J. Choi, J. Kim, H. Han, "Efficient memory mapped file I/O for in-memory file systems," USENIX HotStorage '17, 2017.

[12] J. Gray, A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, CA, USA, 1992.

[13] Microsoft SQL Server 2019 Hybrid Buffer Pool, Microsoft Corporation. https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/hybrid-buffer-pool?view=sql-server-ver15&viewFallbackFrom=sqlallproducts-allversions

[14] Oracle Exadata Database Machine, Oracle Corporation. https://www.oracle.com/engineered-systems/exadata/

[15] PMDK Persistent Memory Development Kit https://pmem.io/pmdk/

[16] S. Scargall, *Programming Persistent Memory: A Comprehensive Guide for Developers,* Apress Open, 2020.

[17] T. Tanabe, H. Kawashima, O. Tatebe, "Integration of parallel write ahead logging and Cicada concurrency control method," 2018 IEEE International Conference on Smart Computing, 2018.

[18] Y. Wu, J. Arulraj, J.Lin, R. Xian, A. Pavlo, "An empirical evaluation of in-memory multi-version concurrency control," Proceedings of the VLDB Endorsement, Vol. 10, No.7, 2017, pp. 781-792.

[19] Y. Wu, K. Park, R. Sen, B. Kroth, J. Do, "Lessons learned from the early performance evaluation of Intel optane DC persistent memory in DBMS," DaMON '20: Proceedings of the 16th International Workshop on Data Management on New Hardware, 2020, pp. 1-3.

TRADEMARKS AND REGISTERED TRADEMARKS