HiEngine: How to Architect a Cloud-Native Memory-Optimized Database Engine

Yunus Ma, Siphrey Xie, Henry Zhong, Leon Lee, King Lv
Cloud Database Innovation Lab of Cloud BU, Huawei Research Center
hiengine@huaweicloud.com

ABSTRACT

Fast database engines have become an essential building block in many systems and applications. Yet most of them are designed based on on-premise solutions and do not directly work in the cloud. Existing cloud-native database systems are mostly disk resident databases that follow a storage-centric design and exploit the potential of modern cloud infrastructure, such as manycore processors, large main memory and persistent memory. However, in-memory databases are infrequent and untapped.

This paper presents HiEngine, Huawei's cloud-native memory-optimized in-memory database engine that endows hierarchical database architecture and fills this gap. HiEngine simultaneously (1) leverages the cloud infrastructure with reliable storage services on the compute-side (in addition to the storage tier) for fast persistence and reliability, (2) achieves main-memory database engines' high performance, and (3) retains backward compatibility with existing cloud-native database systems. HiEngine is integrated with Huawei GaussDB(for MySQL), it brings the benefits of main-memory database engines to the cloud and co-exists with disk-based engines. Compared to conventional systems, HiEngine outperforms prior storage-centric solutions by up to 7.5× and provides comparable performance to on-premise memory-optimized database engines.

CCS CONCEPTS

• **Information systems → DBMS engine architectures**.

KEYWORDS

Cloud-Native, Memory-Optimized, ARM, Log is Database

ACM Reference Format:
Yunus Ma, Siphrey Xie, Henry Zhong, Leon Lee, King Lv. 2022. HiEngine: How to Architect a Cloud-Native Memory-Optimized Database Engine. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3526043

1 INTRODUCTION

Future data processing workloads and systems will be cloud-centric. Many customers are migrating or have already switched to the cloud, for better reliability, security, and lower personnel and hardware cost. To cope with this trend, a number of *cloud-native* database systems have been developed, such as Amazon Aurora [46], Alibaba PolarDB [2], Huawei GaussDB(for MySQL) (Taurus) [13] and Microsoft Hyperscale (Socrates) [5]. These systems are typically built on top of on-premise open-source (e.g., MySQL) or commercial products (e.g., SQL Server) to allow easy migration and adoption. By offloading much functionality to the storage layer and heavily optimizing components such as the storage engine, they exhibit magnitudes faster and/or cheaper than their on-premise counterparts [46]. However, these systems are mostly disk resident databases that follow a storage-centric design. They have a buffer pools that in-memory databases do not, which directly affects how to architect the engine and exploit hardware performance for in-memory databases in cloud platform. Full disaggregated in-memory databases can reduce the access latency but they behave expensive storage cost [42, 50].

In particular, the availability of multicore CPUs and large main memory has led to a plethora of memory-optimized database engines for online transaction processing (OLTP) in both academia and industry [7, 14, 15, 18, 23–25, 30, 33, 43, 45]. Contrary to conventional engines that optimize for storage accesses, memory-optimized engines adopt memory-efficient data structures and optimizations for multi-socket multicore architectures to leverage high parallelism in processors; I/O happens quietly in the background without interrupting forward processing. This allows memory-optimized engines to support up to millions of transactions per second. Many of today's businesses and applications depend on them, ranging from retail, fraud detection, to telecom and scenarios that were dominated by conventional disk-based systems. As customers that require high OLTP performance migrate their applications to the cloud, it becomes necessary for vendors to offer cloud-native, memory-optimized OLTP solutions.

Existing memory-optimized database engines are mostly monolithic solutions designed for on-premise environments. At a first glance, it may seem trivial to directly deploy an on-premise engine in the cloud, by processing transactions using compute nodes and persisting data and log records in some reliable storage service, such as Amazon S3 or Azure Storage, which transparently replicate data under the hood. However, modern cloud environments exhibit a few idiosyncrasies that make this approach far from the ideal.

First, storage services are typically hosted in separate nodes and so have to be accessed via the network; persisting data directly there on the critical path can add intolerable latency to individual transactions. Especially, this model does not work well in Huawei Cloud where inter-layer networking latency (between

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526043

compute and storage nodes) is much higher than that of intra-layer networking (from one compute node to another, or from one storage node to another). Although throughput can remain elevated with techniques such as pipelined and group commit [22] adopted by existing cloud-native systems [13, 46], this largely defeats the goal of achieving low-latency transactions, a main purpose of using a memory-optimized database engine. In terms of reliability and availability, the direct deployment approach would again waste much space by storing redundant copies as the storage service already replicates data, similar to the cases that were made against deploying directly a traditional database system [2, 5, 13, 46] in the cloud. Finally, the directly deployment approach is also ill-suited in today's cloud which is becoming more heterogeneous as non-x86 (notably ARM[1]) processors make inroads into data centers. Most memory-optimized database systems were designed around the x86 architecture to leverages its fast inter-core communication and total store ordering. It was unclear how well or whether they would work in ARM-based environments. These challenges and issues call for a new design that retains main-memory performance, yet fully adapts to the cloud architecture.

We present HiEngine, Huawei's cloud-native, memory-optimized in-memory database engine built from scratch to tackle these challenges. A key design in HiEngine is (re)introducing persistent states to the compute layer, to enable fast persistence and low-latency transactions directly in the compute layer without going through the inter-layer network on the critical path. Instead, transactions are considered committed as soon as their log records are persisted and replicated in the compute layer. HiEngine uses a highly parallel design to store log in persistent memory on the compute side. The log is batched and flushed periodically to the storage layer in background to achieve high availability and reliability. This is enabled by SRSS [13], Huawei's unified, shared reliable storage service that replicates data transparently with strong consistency. In the compute layer, SRSS leverages persistent memory such as Intel Optane DCPMM [12] or battery-backed DRAM [1, 47] to store and replicate the log tail in three compute nodes. The replication is also done in the storage layer by SRSS, but in the background without affecting transaction latency. This way, HiEngine retains main-memory performance in the cloud, while continuing to leverage what reliable storage services offers in terms of reliability and high availability.

HiEngine also uniquely optimizes for ARM-based manycore processors, which have been making inroads to cloud data centers as a cost-effective and energy-efficient option. The main differentiating feature is that these ARM processors offer even higher core counts their popular x86 counterparts and exhibit worse NUMA effect in multi-socket settings. HiEngine proposes a few optimizations to improve transaction throughput for ARM-based manycore processors, in particular timestamp allocation and memory management. Although there has been numerous work on main-memory database engines that leverage modern hardware, most of them are piecewise solutions and/or focused on particular components (e.g., concurrency control or garbage collection). In this paper, by building HiEngine we share our experience and highlight the important design decisions that were rarely discussed in the research literature for practitioners to bring and integrate ideas together to a realize a system that is production-ready in a cloud environment.

It is also desirable to maintain backward compatibility with storage-centric engines. Traditional cloud-native systems are gaining momentum in the market, with established ecosystem based on previous on-premise offerings. For example, Aurora, GaussDB(for MySQL) and PolarDB all offer MySQL-compatible interfaces to allow applications to migrate easily to the cloud; Microsoft Socrates retains the familiar SQL Server T-SQL interfaces for applications. It is unlikely that all applications and workloads would migrate to main-memory engines: some may not even need the high performance provided by these engines. Similar to the adoption of main-memory engines in on-premise environments, we believe that the new main-memory engine needs to co-exist with existing disk-based engines in the existing ecosystem. In later sections, we describe our approach for two database engines (HiEngine and InnoDB) to co-exist in GaussDB(for MySQL) [13], Huawei's cloud-native, MySQL-compatible database systems offering in production.

Finally, HiEngine introduces a few new techniques in core database engine design to bridge the gap between academic prototypes and real products in the cloud. HiEngine adopts the concept of indirection arrays [24] that ease data access processes, but were limited by issues such as fixed static array sizes. We extend this idea by partitioned indirection arrays to support dynamically growing and shrinking indirection arrays, while maintaining the benefits brought by them. HiEngine also adds support for other notable features such as index persistence and partial memory support, both of which were often ignored by prior research prototypes.

Next, we begin in Section 2 with recent hardware trends in modern cloud infrastructure and Huawei SRSS, which is a critical building block that enables the making of HiEngine. Section 3 first gives an overview of HiEngine architecture and design principles. We then describe the detailed design of HiEngine in Sections 4–5, followed by evaluation results in Section 6. We cover related work in Section 7 and conclude in Section 8.

2 MODERN CLOUD INFRASTRUCTURE

This section gives an overview of recent cloud infrastructure (hardware adoption and architecture) trends at Huawei Cloud which HiEngine is built around.

2.1 Hardware Trends and Challenges

Modern cloud computing infrastructure decouples compute and storage elements to achieve high scalability: Conceptually, one set of nodes are dedicated to compute-intensive tasks ("compute" nodes or instances), and another set of nodes are dedicated to storage ("storage" nodes or instances). Most servers in cloud data centers are x86-based and software is thus designed around the x86 architecture which features stronger but relatively fewer cores per chip (e.g., a couple of dozens).

Represented by products such as Amazon Graviton [3], Ampere Altra [4] and Huawei Kunpeng [19], ARM-based manycore processors are making inroads into cloud data centers for better price

[1]For example, both Huawei Cloud [19] and Amazon Web Services [3] have deployed ARM processors in their cloud offerings.

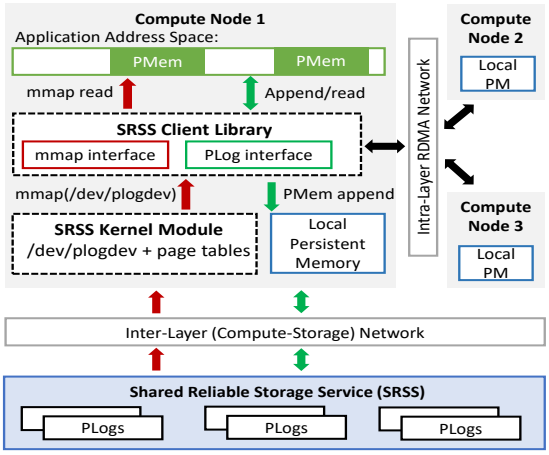


Figure 1: Memory-semantic and compute-side persistent storage support in SRSS. Data is replicated three ways in the compute and storage layers. A specialized `mmap` kernel driver supports consistent read from storage layer and local or remote persistent memory via low-latency storage network.

per performance, customizability and energy profiles. Compared to their x86 counterparts, these ARM processors typically feature many more cores. For example, Huawei Kunpeng 920 and Ampere Altra offer up to 64 and 80 cores. But typically these cores offer weaker compute power and when integrated into a multi-socket server, exhibit more severe NUMA effect. The high parallelism and NUMA effect necessitate further optimizations and tuning of database systems to extract the full potential of highly parallel ARM processors; we describe related optimizations in HiEngine in later sections.

In the decoupled architecture, compute nodes typically features large main memory (enough for keeping at least the working set, if not the whole database), but limited and ephemeral local storage that will not get persisted across instance instantiations. Storage nodes are the only permanent home of data and focus more on providing high capacity and high bandwidth. Data is typically replicated multiple times under the hood by the reliable storage service running in the storage layer for high reliability and availability guarantees. Applications running on the compute nodes must go through the network to reach the storage tier to access data. Reducing such network roundtrips has been a major focus in prior cloud-native systems, as it often dominates the total runtime [2, 5, 13, 46].

Typically, storage and compute nodes are maintained in separate pods, incurring higher cross-layer network communication cost between the compute and storage layers. Although fast networks (e.g., 40-200Gbps Ethernet and Infiniband) with remote direct memory access (RDMA) capabilities are becoming a norm in data centers for communications within and across compute/storage layers (i.e., intra-layer communication), there is still a noticeable gap between the compute and storage layers. This poses non-trivial challenges for reducing end-to-end commit latency, especially for main-memory database systems. For example, Huawei Cloud already leverages RDMA within its storage service to provide fast and strong consistency guarantees, but the inter-layer network latency (between the compute and storage layers) is about 3–5× longer than intra-layer latency on the compute side.

Meanwhile, persistent memory products such as Intel Optane DCPMM [12] and flash-backed NVDIMMs [1, 47] are blurring the boundary between memory and storage with byte-addressability and persistence on the memory bus. They present both opportunities and challenges to the decoupled architecture. On the one hand, the application would require byte-addressability to realize the full potential of persistent memory, making it desirable to equip persistent memory on compute nodes, instead of hiding them behind the networked I/O interface in the storage tier. Main-memory OLTP engines can potentially benefit from this setup to achieve high throughput and reduce transaction commit latency in the cloud. On the other hand, deploying persistent memory on the compute side naturally goes against the stateless nature of compute nodes, make them another permanent home of data. We describe in the next section how SRSS, a core component of Huawei Cloud that bridges this gap and provides the necessary persistence primitives for HiEngine in the cloud.

2.2 SRSS: Huawei's Reliable Storage Service

Now we introduce the shared reliable storage service that is currently in production in Huawei Cloud. Detailed discussions of them are out of the scope of this paper; we focus on the aspects that would impact how we architect HiEngine.

SRSS is Huawei's next-generation distributed storage service built on top of modern SSDs using RDMA over fast data center networks to offer replication, strong consistency guarantees and high performance across availability zones and data centers. It is designed as a log-structured, append-only store that replicates data 3-ways under the hood. The basic unit of data storage is a persistent log (PLog), which is a contiguous, fixed-size chunk (64MB or larger) in SSD. SRSS provides applications typical interfaces that one can find in log-structured storage: create/open/close PLog, append and read; in-place update is disallowed by its nature [13]. Applications communicate through the SRSS client, which is a thin layer deployed on compute nodes to provide the aforementioned interfaces, and communicate with the storage backend. SRSS uses a pool of SSDs and storage nodes to provide reliability with strong consistency. Upon receiving write (append) requests, the storage backend synchronously replicates data to three storage nodes in parallel, and returns to client signaling success only when data is persisted across all three nodes. In case of failures during the (replicated) write operation, SRSS "seals" (i.e., permanently closes) a PLog which would disallow any further writes to it; the application then needs to retry its write request while the storage backend switches to use another node. Read can be served by any replica, as determined by SRSS's routing layer and partitioning scheme.

2.3 Compute-Side Persistence in SRSS

With persistent memory (PM)[2] and low-latency RDMA network, SRSS extends persistence to the compute layer with the aforementioned PLog abstraction to provide persistence, high availability and strong consistency in the compute layer. In addition to the `open`, `close`, `append` and `read` interfaces, on the compute side, SRSS uniquely provides memory semantics via memory map (`mmap`) [21]

[2]Not limited by particular PM technologies, which can be products such as Intel Optane DCPMM or NVDIMMs based on DRAM and flash memory with supercapacitors.

that mimic the counterparts in a single node system. This allows a compute node to persist locally and remotely *within* the compute layer, without (more expensive) roundtrips between the storage layer on the critical path.

This is a crucial feature that enables the making of HiEngine. Figure 1 describes the high-level idea; for clarity we omit DRAM and local scratch SSD storage in the figure. Each compute node is equipped with persistent memory that can be read directly after it is `mmap`'ed to the application's address space. This is facilitated by a SRSS kernel module that exposes a character device (`/dev/plogdev`) for the user space. Internally, it implements the relevant file operations interfaces (e.g., the `mmap` interface that can be issued against `/dev/plogdev`), handles page faults (from local/remote persistent memory or the storage layer), and maintains virtual-physical address mappings. SRSS customizes the `mmap` interface to support both local and remote accesses from other compute nodes. Applications can also `mmap` data in the storage layer, by specifying corresponding `mmap` flags. Writes to compute-side storage is replicated three ways by SRSS client to other compute nodes via high-speed RDMA network in the compute layer. Meanwhile, the application can choose to store data in the storage tier, which also provides strong consistency, but also offers such ability across availability zones.

Note that SRSS offers log-structured storage, so writes have to be append operations via traditional PLog interfaces (green arrows in the figure); in-place updates are not supported. This applies to both the storage and memory semantics. The `mmap`-based accesses (red arrows in the figure) are therefore limited to read-only operations. The application must write using the append-only interfaces to append data in local/remote persistent memory and/or the storage layer. Similar mechanisms can be used to support compute-side persistence using SSDs; we leave it as future work.

3 HiEngine ARCHITECTURE

In this section, we give an overview of HiEngine's architecture and design principles. We expand on details in Sections 4–5.

3.1 Overview

HiEngine uses compute-side persistence to achieve main-memory performance and support larger-than-memory databases in the cloud. This is realized by a three-layer physical architecture shown in Figure 2, which consists of (1) a SQL engine, (2) a storage engine and (3) a storage backend (SRSS). The SQL and storage engines are deployed in the compute layer, while the storage backend is maintained in a separate set of nodes. The SQL engine accepts user requests and is responsible for routing record access operations to the storage engine. It supports both interpreted and compiled execution models for better performance. The storage backend runs in the storage layer and is maintained by SRSS to provide a shared, reliable and log-structured storage service. Similar to Socrates, HiEngine has a three-tier logical architecture: compute layer, logging layer in compute-side and remote shared storage. HiEngine stores data permanently in both the compute and storage, the logging layer provides low-latency data access and fast instance restore.

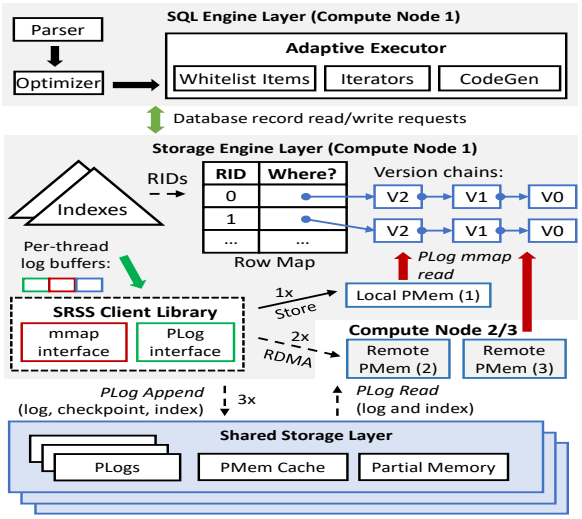


Figure 2: HiEngine architecture. Data (i.e., the log) is permanently stored and replicated in both the compute and storage layers leveraging SRSS compute-side persistence.

Transactions commit directly after persisting log records in three compute nodes synchronously using SRSS compute-side persistence introduced in Section 2.3. In the background, the log is de-staged to the storage tier for and compacted periodically to reclaim storage space and for archival purposes. Under the hood, SRSS replicates data across compute nodes and storage nodes; read requests can be satisfied via `mmap` locally, and cross node boundaries when necessary (e.g., accessing data that has been archived or de-staged to the storage layer).

Additional read-only replicas (compute-side) can be spawn on demand by loading data from the storage tier or the compute tier, depending on application need: for example, applications that do not require high freshness may simply start a new instance using data loaded from the storage tier to reduce cost and pressure on user and compute-layer network. HiEngine's compute-side logging persistence is also particularly useful for customers who prefer to reboot a crashed/failed instance rather than completely loading a brand new instance (for various reasons): it provides both fast recovery and high availability and reliability by leveraging a reliable storage service (SRSS). It is worth emphasizing that the system cannot tolerate availability zone (AZ) failure while the logging layer server is co-located with a compute layer server on the same physical machine. Deploying logging layer to a separate remote memory servers that provide failure tolerance of cross-AZ can introduce high access latency from traversing network. Our current design assumes that the logging layer server is co-located with a compute layer, and we are exploring other solutions.

To better understand the design of HiEngine, Table 1 presents the logical architecture and key features of various main in-memory and disk resident database engines from industry and academia. HiEngine is a cloud native industrial OLTP in-memory database, while Socrate, Taurus, and Aurora are disk resident databases especially with buffer pools components. NAM-DB [50] is a disaggregated in-memory database prototype with two-tier architecture, and it has expensive data access cost across network. HiEngine

Table 1: Logical Architecture Comparison for Popular Database Engines

Systems	Design Principle	Log is Database	Disaggregated Architecture	Main Location
Aurora [46]	Storage-centric	Yes	Compute Layer + Shared Storage Layer	SSD/HDD
Taurus [13]	Storage-centric	Yes	Compute Layer + Shared Storage Layer	SSD/HDD
PolarDB [2]	Storage-centric	No	Compute Layer + Shared Storage Layer	SSD/HDD
Socrates [5]	Storage-centric	Yes	Compute Layer + Logging Layer + Shared Storage Layer	SSD/HDD
HiEngine	Memory-centric	Yes	Compute Layer + Logging Layer + Shared Storage Layer	DRAM/NVM
ERMIA [24]	Memory-centric	Yes	Not Disaggregated	DRAM
Hekaton [14]	Memory-centric	No	Not Disaggregated	DRAM/SSD
NAM-DB [50]	Memory-centric	No	Compute Layer + Shared Storage Layer (Memory)	DRAM
FaRM [42]	Memory-centric	No	Compute Layer + Shared Storage Layer (Memory)	DRAM/NVM

achieves microsecond-level latency on NVM through `mmap` for logging and data storage, while Socrate and others have millisecond-level latency on SSD. Socrate and HiEngine have the similar compute-storage architecture with three-layer in which a computing-side logging layer provides fast instance restore.

This architecture paves ways for extending HiEngine to support distributed version of HiEngine that is a shared-nothing cloud native distributed in-memory database, its main features including: 1) providing high scalability based on data sharding, 2) reducing distributed transaction commit latency based on local persistent logging storage, 3) implementing the distributed SQL engine, 4) and a high-precision global clock improves the performance of timestamp allocation. In this paper we focus on the design and implementation of HiEngine in a single-master setup.

3.2 Log-Centric MVCC Storage Engine

HiEngine is built around the idea of "the log is the database" and integrates several key techniques to achieve high performance and leverage the underlying reliable storage service (SRSS) to achieve high reliability. Changes are persisted only in the log, without extra copies as the "real" database found in conventional designs. Prior work has identified such log-centric designs to be suitable for cloud-native systems Aurora [46] and Taurus [13]; we reach a similar conclusion for main-memory engines: it eases replication, reduces network traffic and leverages well the asymmetric sequential/random access storage speeds. Conceptually, HiEngine maintains a replicated log and facilitates data access through a level of indirection [24, 41] that maps record IDs to record locations in a "row ID map." The row ID map is inspired by similar data structures used by ERMIA [24] and Bw-tree [31], and is a core data structure in HiEngine to support efficient recovery, checkpointing, multi-version concurrency control (MVCC) and larger-than-memory databases. While prior work like ERMIA has laid out the high-level idea, it still leaves a gap for actual production use and misses important features (e.g., extensibility and support for big tables), requiring a revisit and re-design. In this paper we show how HiEngine designs and optimizes the "row ID map" for memory-optimized database.

3.3 SQL Engine with Full-Stack CodeGen

To realize the full potential of main-memory databases, it is necessary to move from traditional interpreted query execution to compiled execution [14, 36]. In in-memory databases, the high performance of transaction engines is severely underperformed when the native SQL engine is involved, which is related to the efficiency of interpretation execution and networking latency.

HiEngine's SQL engine implements full-stack code generation for efficient query execution: Compared to prior work and other systems, HiEngine not only generates code for data manipulation language (DML) operations (e.g., data accesses such as record read, update and insert), but also data definition language (DDL) operations, such as table creation and schema changes, to further reduce overheads. The main tradeoff is that DDLs operations may take longer to finish, however, we find its benefits outweigh the drawback in practice.

3.4 Deployment

As a memory-optimized database engine, HiEngine complements existing storage-centric database engines running in most cloud-native database systems. It can be deployed as a separate engine inside a host system, sitting "vertically" side-by-side with another storage engine, or as a transparent ACID cache in front of another storage engine in the host (horizontal deployment). Figure 3 shows the two deployment modes in GaussDB(for MySQL) [13], Huawei's MySQL-compatible cloud-native database offering. Under both modes, applications communicate with the database system in existing MySQL-compatible interfaces, maintaining backward compatibility.

Our main focus in this paper is the vertical integration approach shown by Figure 3(left). In this mode, HiEngine co-exists with the existing InnoDB engine in GaussDB(for MySQL). HiEngine leverages the multi-engine support in MySQL/GaussDB(for MySQL) to register HiEngine as another database engine. MySQL/GaussDB(for MySQL)'s SQL layer provides a common set of components, such as the SQL parser, networking layer, and thread pool. To use HiEngine, the user only needs to declare a table to be stored by HiEngine in the schema, using the `WITH ENGINE=HiEngine` parameter in `CREATE TABLE` statements. Queries (compiled or interpreted) will then be routed to the corresponding storage engine for execution. Our current implementation limits transactional accesses to a single engine, i.e., a query/transaction cannot access more than one engine. Supporting cross-engine transactions is on-going work.

3.5 Life of a Transaction

Now we walk through on a high level how transactions are processed in by HiEngine architecture, before introducing the details

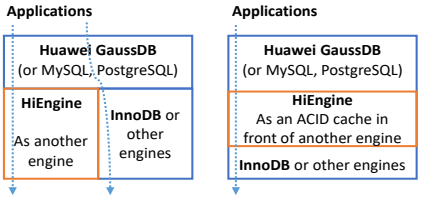


Figure 3: HiEngine can be deployed as a separate database engine (left) or as an ACID cache in front of a conventional engine such as InnoDB in Huawei GaussDB(for MySQL). Applications communicate with the system through MySQL/-PostgreSQL compatible interfaces.

of each component. Applications talk to HiEngine using a MySQL-compatible interface and may open a session to work on a transaction. Each session is handled by a worker thread that is part of a thread pool to avoid context switching overheads. HiEngine binds transaction worker threads to physical cores and leverages asynchronous I/O for persistence. Queries in the transaction can be executed using interpretation or compilation and once they pass through the networking, parser and optimizer layers, they reach HiEngine kernel in the form of both reads and writes.

HiEngine implements multi-versioning and snapshot isolation[3] to support a wide variety of workloads. The way it works is similar to most other multi-version systems. Each transaction is accompanies by two timestamps, a begin timestamp and a commit timestamp. Both are logical timestamps that represent the relative order between a pair of transactions' begin and commit times, respectively. The commit timestamps are also referred to as commit sequence numbers (CSNs) throughout the paper. Upon a transaction starts, it reads the counter to obtain a begin timestamp, and upon commit, it atomically increments the counter by one using the atomic `fetch-add` instruction to obtain a CSN, which determines the dependency order of the transaction among others. Versions in HiEngine are stamped with the creator's CSN so that a transaction is only able to read the latest version that was created before its begin timestamp. After accessing records, the transaction either commits by forcing its log records through SRSS's append interface or abort by discarding all the modifications it made during forward processing.

4 STORAGE AND INDEXING

HiEngine takes a departure from traditional page-based data storage and organizes data records in the form of logs without the "page" concept. The log in HiEngine supports data accesses and versioning, in addition to its original purpose, crash recovery. Tables then become logical constructs which are collections of data records residing in the log, without extra copies of the "real" database. A record in turn is represented by one or multiple versions chained together following new-to-old order by a logical timestamp [49]. Transactions then choose the proper versions to access based on their own begin timestamps (i.e., the latest version that was generated before the transaction started). To update a record, a new version (log record) of it must be appended to the log with its in-memory representation prepended to the record's version chain. Stale versions in memory and storage are garbage collected once they are no longer needed using epoch-based reclamation.

In the rest of this section we describe in detail (1) how HiEngine facilitates data access in a log-centric architecture using indirection arrays, (2) how HiEngine organizes persistent data using SRSS PLog abstractions, and (3) new optimizations enabled by indirection and practical considerations needed in a production system.

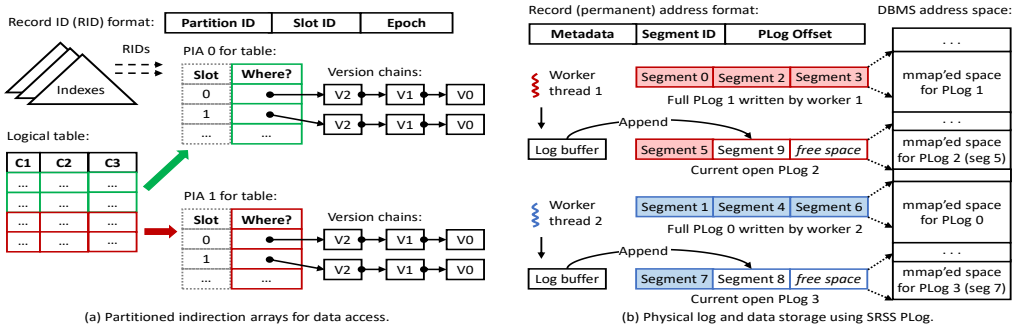
4.1 Partitioned Indirection Arrays

The key data structure in HiEngine that realizes the idea of "the log is the database" is *indirection arrays* similar to those used by ERMIA [24] and BwTree [31]. An indirection array maps record IDs (RIDs) to record addresses (an offset into the logical log or a virtual memory pointer if the record resides in memory). Each table is represented by an indirection array which conceptually is a sequential array where each element stores the address of the record's latest version which further points to an older version, and so on. To access a record, a transaction thread may first traverse the index to retrieve the RID, and then use it as the index into the indirection array to locate the actual record in memory or storage. Transactions install new versions by issuing a `compare-and-swap` instruction on the target record's indirection array entry (more details later).

Indirection arrays bridge the gap between the log and "the database." Their benefits are multi-fold, especially in a cloud environment. Both recovery and replication become as simple as setting up the indirection arrays with record addresses, without having to re-instantiate physical versions by loading data possibly from the storage tier over the network. Checkpointing also becomes lightweight as one only needs to persist the indirection arrays, instead of real data. Unlike in traditional disk-based systems, in HiEngine an RID not only uniquely identifies a record but also never changes throughout the lifetime of a record. Some systems map secondary index keys to primary keys or virtual memory pointers to records, requiring changing all secondary indexes in case the primary key changes, or any update is applied to the record if the system is multi-versioned. With indirection arrays, typically no index changes are needed when a record is updated; the only exception is when the changed field constitutes an index key, in which case then the corresponding (primary and/or secondary) secondary index (key) must be updated. Since indexes do not store "real" record data, update operations will not involve changing the internal structure of any index; this further saves CPU cycles and potential contention on the index structure. Indirection arrays also provide an easy way to support numerical, auto-increment indexes where the RIDs are implicitly keys and can be used to directly query a table.

At a first glance, it may seem trivial to adopt this approach. In practice, however, a more careful design is necessary in a production-grade system: (1) indirection arrays should be able to grow and shrink dynamically, and (2) accessing indirection arrays should be as fast as possible, preferably as simple as accessing a static array; this makes a tree-based or hash-based data structure less desirable. Prior prototypes mostly side-stepped these problems with pre-allocated arrays [24] that would waste main memory, and/or impose a low maximum table size; neither is desirable in a production system.

[3] Most users have adapted their applications for lower isolation levels (e.g., snapshot isolation or repeatable read) to guarantee correct behaviors [40]. Support for "true" serializability (i.e., forbidding anomalies such as write skew [9] and read-only anomaly [16]) in HiEngine is an on-going effort.



(a) Partitioned indirection arrays for data access.

(b) Physical log and data storage using SRSS PLog.

Figure 4: HiEngine's partitioned indirection array to retain constant access time and support dynamic and large table sizes. Each table is logically partitioned and represented using one or multiple fixed-size indirection arrays that map record IDs to record locations (in memory, local or remote storage).

We resolve these issues with partitioned indirection arrays (PIA) that logically and horizontally partition a table. Instead of using one indirection array, we use multiple fixed-size arrays per table. This allows us to create or delete arrays on demand without having to provision a huge array or be limited by the size of a single array. Meanwhile, PIAs can still be accessed in a single step without complex hashing or tree traversals. As shown in Figure 4(a), each PIA entry is an 8-byte word, consisting of a (1) 16-bit partition ID, (2) 32-bit slot ID and (3) an epoch number used for garbage collection (described later). To locate a record, the transaction takes the highest 16 bits of the target RID to locate the PIA, and then uses the 32-bit slot ID to directly access the PIA entry. Creating a partition simply means reserving a contiguous range of 2^{32} *virtual* addresses in the database engine process address space (since there are 32 slot bits per PIA entry); the physical pages backing up this array are later allocated on demand by the OS. As a result, a table may contain up to 281 billion records stored in up to 65,536 partitions.

Each PIA entry stores the address of the record's most recent version, which can be a (1) virtual memory pointer or (2) permanent address whose format is shown in Figure 4(b). In the former case, the virtual address could refer to a chunk of dynamically allocated memory in the compute node (e.g., as a result of a recent update), or a pointer to a `mmap`'ed region brought up by SRSS's memory semantic support (e.g., as a result of reading from a recent checkpoint stored in a compute or storage node). Each version in turn stores the address of the next version, and so on.

4.2 Reliable and Scalable Redo-Only Logging

Now we describe how logs are physically organized and managed using SRSS PLogs maintained in the compute and storage layers. The log in HiEngine plays the dual-role of supporting crash recovery and data accesses. As described in previous sections, HiEngine works with software and hardware infrastructure in the cloud. Specifically, the storage layer (SRSS) in Huawei Cloud provides a unified PLog abstraction to organize data in the storage and compute tiers. Each PLog has a maximum size of 4GB, yet the database log is an append-only abstraction that represents an infinite stream of data. Our major goal here is therefore to architect a database write-ahead logging approach on top of PLogs.

The classic approach to write-ahead logging is centralized logging where all the threads and transactions share a single log buffer, even if they do not have conflicts, creating a major bottleneck in manycore systems. HiEngine adopts a variant of distributed logging [24, 45, 48] with commit pipelining [22]. Instead of using a global log buffer, transactions accumulate log records in private buffers until commit time. We maintain two threads per CPU core, one as the transaction worker, the other as the I/O thread that is responsible for flushing logs asynchronously via the PLog append interface. Upon commit, the worker thread hands over the transaction buffer to a queue maintained by the I/O thread, which will then issue an asynchronous I/O call to flush the buffer into the underlying PLog. Once the log buffer is handed over to the I/O thread, the worker thread is free to handle another transaction. For transactions whose log records are successfully persisted, the I/O thread will notify the client for successful commit. Log records generated by loser transactions are discarded directly, without being flushed to the persistent log. Therefore, the log in HiEngine is redo-only and contains only committed data.

Physically, the log is organized in segments which in turn are mapped to PLogs. Figure 4(b) describes the overall design. Segments are allocated and deallocated dynamically by worker threads on demand. Each thread maintains an open PLog that can accommodate multiple segments and an open segment that accepts log records upon transaction commit. When a segment becomes full a new one is open, which may cause a new PLog to be created if the current open PLog becomes full (over 4GB). Each segment in HiEngine has a fixed size (e.g., 128MB) and is identified by a PLog ID and an offset into the PLog that stores the segment. A segment may not cross PLog boundaries, but a transaction is allowed to cross segment boundaries, i.e., the transaction may generate log records that end up in more than one segment. Log records are written to segments/PLogs using SRSS's append interface which in turn replicates the log three ways on the compute and storage tiers (Section 2.3). However, once the data is successfully replicated and persisted in the compute side, the transaction is considered to be committed and the result is returned to the client.

As an append-only system, in HiEngine data records can be only updated using append writes. Read accesses to data in compute-side PM or the storage tier is done by SRSS `mmap`. As shown in Figure 4(b), full PLogs which contain multiple segments can be `mmap`'ed using SRSS into the address space of the engine. Individual segments can also be `mmap`'ed to provide direct access. Newly generated versions first reside in the main memory of the compute

node, which may later be evicted to make room for new updates and reads. Leveraging the high capacity of PM, evictions are expected to be rare and the vast majority of accesses are done in memory or PM without network I/O on the critical path.

Similar to several other memory-optimized systems [14, 24, 45], an update operation in HiEngine generates a new version that contains the complete content of a record. This allows easy implementation of multi-versioning without the need to reconstruct versions on the fly by applying redo log records which may involve expensive storage reads over the network (as done by some cloud-native systems such as GaussDB(for MySQL). The drawback is larger footprint; we allow a limited number of deltas to be generated to balance reconstruction cost and space consumption. Based on the segmentation design, as shown in Figure 4(b) top, each persistent record version (i.e., log record) is addressed by an 8-byte word, in which the high 16 bits identify a segment, and the low 32 bits are used as the offset into the PLog. The remaining 16 bits are used as metadata during runtime, giving a maximum of 65,536 segments.

Segment size can be adjusted dynamically according to profiling results and user data size to limit the number of PLogs in the system. Our current implementation uses one PLog per segment and sets segment size to 128MB. This allows HiEngine to support maximum database size of 8TB. The remaining practical issue is to locate the corresponding PLog that is backing a segment. PLogs are identified by 24-byte identifiers in SRSS, posting significant overhead and would not fit in the 16-bit reserved for segment IDs. We maintain a mapping between PLog IDs (which are used by SRSS to identify PLogs) and segment IDs. The mapping itself is stored in a designated PLog whose ID is stored in a well-known location (such as management nodes) for the system to get bootstrapped.

4.3 Dataless Checkpoint and Parallel Recovery

To accelerate recovery, HiEngine regularly runs incremental and full checkpoints in the background without blocking forward processing. With indirection arrays, checkpointing becomes very lightweight and only needs to persist indirection arrays without making copies of in-memory data, or writing record data to storage nodes. The checkpointing thread starts by selecting the current durable CSN as the checkpoint CSN, and then going through all entries of each PIA. Recall that indirection array entries store record addresses, which can be a storage address (which consists of a segment ID, PLog offset and metadata) or a memory pointer. For entries that contain a storage address, it is written directly to the checkpoint PLog. Otherwise, we follow the version chain to reach the version that is visible to the checkpoint CSN (i.e., smaller than it) and store the permanent storage address of that version to the checkpoint PLog. This way, the checkpointing process always obtains a consistent view of the database. The generated checkpoint image (which essentially is a snapshot of the PIAs) is stored locally in compute nodes (and in the storage tier asynchronously) for recovery later. We describe index persistence later. Index checkpoints for instant recovery were proposed in our another work [28].

Many databases [6, 20, 38] follow and optimize the ARIES protocol [35], in which recovery mechanism is designed for disk resident and page-oriented databases. CTR [6] is a typical work built by ARIES systems and MVCC, which enables fixed-time recovery by making use of different data versions rather than using the original WAL log. However, HiEngine provides the features of "log is data" and tuple-oriented data storage, its recovery is lightweight and only restores the PIA instead of full versioned data, thanks to the use of indirection arrays. Upon recovery, the checkpoint image is `mmap`'ed via SRSS for re-constructing the PIAs, i.e., creating PIA structures in memory and filling in their entries with addresses recorded in the checkpoint image. Log records generated after the checkpoint CSN are replayed to bring the system back to the latest state. Note that when PIAs are being reconstructed in both phases (reading the checkpoint image and replaying logs), no actual record structures or data are created or loaded from the storage or compute nodes. Recovery is finished once the PIAs are setup; later accesses will bring data versions into main memory on demand via `mmap`'ed read.

At a high level, the replay phase can be parallelized by having multiple replay threads scan the logs (one per thread) sequentially and if logging operation is an insert or update, we update the corresponding PIA entry to the permanent address of the log record (i.e., the record version). If the log record represents a delete, we clear the corresponding PIA entry, however, the original epoch part of the record is preserved. Recall that HiEngine maintains multiple log streams (one per thread) and uses redo-only logging. Thus, log records for the same data records may be scattered in multiple logs; a blind parallelization approach that scans and applies log records in parallel would not recover the database correctly. To solve this problem, in HiEngine a replay thread would overwrite the PIA entry with a new record address only if the entry points to an older record version (done by compare the records' creation time represented by their CSNs). The overwrite process is also done using a `CAS` instruction to properly coordinate multiple replay threads working on the same record.

4.4 Garbage Collection and Log Compaction

HiEngine adopts epoch-based reclamation [17] to recycle stale record versions in memory (e.g., as a result of an update). Each thread maintains a bag of stale versions which can be removed from version chains once no transaction is going to ever need them. Each worker thread in HiEngine maintains a *readCSN* which represents the current snapshot of the database as "seen" by the worker thread, and is refreshed whenever a transaction concludes (by reading the central CSN counter, described in Section 3.5). The minimum read CSN among all worker threads then would be the low watermark: all the record versions before the watermark can be recycled. The actual recycling process is interspersed between transactions (i.e., done by worker threads upon transaction commits and aborts) [8] and can also be scheduled in the background. Note that the above garbage collection process applies to record versions generated in main memory only, for example as a result of recent updates. With more updates continuously being applied, some of them may be evicted (i.e., dropped from main memory) and a subsequent access would need to access it through SRSS `mmap`. Stale versions are garbage collected as part of the log compaction process which we describe next.

As records are appended to log segments, versions of the same record may appear in different log segments (PLogs), and logically adjacent records in a table may appear in different parts of a log segment or even different PLogs. This makes most record accesses unclustered with little locality and is a common problem of append-only systems which trade off locality for fast append performance. HiEngine compacts log segments in permanent storage (compute and storage tiers) periodically to (at least partially) restore locality by rearranging data. This is done in two flavors (full or incremental) to reduce overhead. To start the compaction process, a compaction thread (or a set of them) takes a snapshot of the current lowest read LSN among all threads and use it to determine which log records should be discarded; this read LSN is also stored locally as the "compaction LSN" for partial compaction. A full compaction takes out all the versions and records of a table to cluster data in new storage space (i.e., PLogs and segments) so that adjacent records occupy a contiguous chunk of storage space to recover locality. The process runs on compute nodes and resembles the checkpointing process described in Section 4.3 and keeps adding new records to new PLogs (log segments) and updating the permanent addresses appeared in PIAs. Once finished, the new PLogs are `mmap`ed to server new accesses, and the compacted segments are discarded once compacted and no thread is still using them via `mmap`. Partial compaction works in the same way, except that only it rewrites the new changes in a specified period time in the past (specified by the current CSN and previous compaction CSN).

4.5 Append-Only and Partial-Memory Index

Indexes play important roles in delivering high performance in memory-optimized systems. In the cloud, indexes in HiEngine faces two main challenges that were largely left out by prior research prototypes of memory-optimized systems: (1) persistence and (2) partial memory (spill-out) support of the index structures themselves. Most classic memory-optimized engines assume indexes are rebuilt upon recovery and therefore do not persist the indexes themselves during forward processing [24, 45], which can be unrealistic and time-consuming as indexes can get large as the database grows [34]. Unlike traditional disk-based B-trees and hash tables, most index structures were also designed for main-memory environments without support for partially spilling to storage, which is necessary in a production system handling various workloads. It is also important to reduce I/O during spill-outs in a cloud environment as I/Os may have to go through the network.

Indexes in HiEngine use an append-only, partial-memory design that is similar to log-structure merge (LSM) trees [37] to solve these problems. The LSM-like design allows us to leverage fast append-only writes offered by SRSS. Currently we use concurrent adaptive radix trees (ART) [29] as the baseline index structure, however, our approach is also applicable to other index structures, such as B-trees. Each index consists of (1) one main in-memory component that accepts both read (search, scan) and write (insert, update) requests, and (2) a list of read-only components that have been written out to storage via SRSS in local PM or the storage nodes; the list is sorted in reverse tree creation order. When free memory is under a predefined threshold, the in-memory index will be serialized and saved to storage, joining the list of read-only components. At the same time, a new empty in-memory component is created to accommodate new addition/update/deletion requests. Once the index is serialized to storage, it is marked as read-only and can be `mmap`'ed via SRSS for future accesses. Similar to LSM trees, an index operation requires checking potentially multiple index components, starting from the latest (in-memory component) to older read-only components. We maintain a background thread to garbage collect and merge read-only components to limit the number of read-only components.

The key for this approach to work well is to limit the impact on performance of garbage collection and merge operations. Indexes in HiEngine only store key–RID mappings as a result of its indirection-based append-only storage design, instead of real data like some LSM-based systems do. This greatly reduces the overhead of merging and compacting tree components as no record data movement is involved in the process (compacting the versioned records is a separate process as described previously). Since the in-memory component is only written out (serialized) to storage under memory pressure, it is desirable to limit the amount of memory needed for serialization. We do so by limiting the serialization and merge processes to use a constant amount of memory. We describe the merge process first, as HiEngine treats serialization as a special merge case of merging the in-memory component with an empty index. To merge two indexes, HiEngine recursively checks each corresponding level of nodes in both trees. For each tree level l, there are three cases regarding the nodes in both trees at level l:

(1) Internal nodes for both trees. In this case, we place the node with a shorter prefix to the left, and the other to right, load prefix because prefix length maybe longer than maximum prefix storage. If these two nodes have different prefixes, a new node is created and the common part of prefix is set. If there's no common part in their prefixes, then the new node is set without a prefix. Then the left and right nodes are inserted into this new node with their own prefixes. Otherwise, the two nodes have exactly the same prefix, their children are merged into a new node.
(2) Internal nodes in one tree, and leaf nodes in the other. The trick is also in handling prefixes: we need to compare the internal node's prefix with the leaf node's key byte by byte. If the same part is shorter than the internal node's prefix, then the prefixes of the two nodes have diverged and a new node should be created with the remaining key bytes inserted into this new node. If the same part of the prefix is the same or longer than the internal node's prefix length, then the leaf node can be simply merged into the internal node.
(3) Leaf nodes for both trees. In this case we find a proper container node with the same prefix of these two leaf nodes.

Each child of the two nodes is then extracted to further repeat the above merge process recursively. The max depth of recursion is the height of the tree. For each level, we temporarily create a node on the stack to save the merged node. The merged/serialized node is then written out in batches via SRSS's append interface to the underlying storage, which can then be `mmap`'ed to provide index accesses. As a result, the amount of extra heap-allocated memory

needed can be a constant and is only determined by SRSS I/O size: SRSS receives a heap-allocated in-memory buffer to initiate an I/O operation. In the extreme case, HiEngine only needs to reserve a single page for SRSS I/O.

5 TRANSACTION MANAGEMENT

We now discuss how transactions are managed and scheduled.

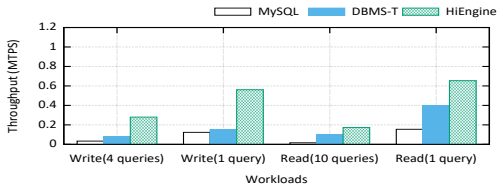
5.1 MVCC Protocol

As mentioned in Section 3.5, HiEngine uses multi-version concurrency control (MVCC) and uses logical timestamps to control version visibility. The protocol is similar to that of many prior systems [14, 24, 33]; here we describe the basic ideas, followed by the unconventional aspects of it in HiEngine. To support MVCC, in each version we embed a few necessary fields: 1) Commit sequence number that denotes the logical creation time of the version; 2) A pair of timestamps (`tmin` and `tmax`) that determine the life span of the version; 3) Address (a virtual memory pointer or permanent address) of the next older version.

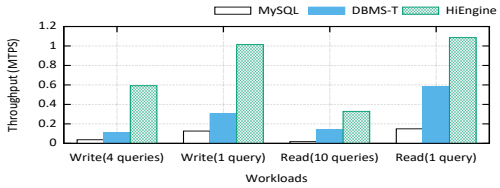
A transaction is assigned a begin timestamp (read CSN) and a globally unique transaction ID (TID) when it begins. Version visibility is done by comparing the transaction's begin timestamp with available versions of the target record: the latest version that was created before the transaction began is visible. To install a new version (update a record), the transaction issues a `compare-and-swap` instruction to update the record's PIA entry to point to the new version in memory. In the new version, the TID is written in the `tmin` field to indicate the version is not committed yet, so other reader transactions can skip it. Meanwhile, the old version's `tmax` field is and updated to contain the updater transaction's TID. Each transaction also maintains a read set and a write set. At commit time, the transaction will first acquire a CSN using an atomic `fetch-add` instruction on the global CSN counter, and replace the TIDs in `tmin` and `tmax` fields from TIDs to CSN; after this point, the versions become "committed" and can be used by other transactions. For aborted transactions, all the newly installed versions will be recycled and unlinked from the version chains with the PIA entry restored to contain the original value.

5.2 Early Commit

HiEngine's concurrency control protocol ensures that log records are persisted sequentially by using an optimistic lock and a `map` of transaction status. The lock-based approach is widely used in many database engines [14, 24, 38, 42, 45]. To improve performance, HiEngine supports early release of row locks, which can result in out-of-order persistence of logs since transactions can get the uncommitted records that are visible for them. Register-and-report is a general approach for management of commit dependencies: T1 registers its dependency with T2 and T2 informs T1 when it has committed or aborted [27]. HiEngine also embraces this ideas in engineering implementations to manage transaction dependencies. HiEngine also provides conflict detection and wait timeout processing for transactions.



(a) Performance of inlined queries.



(b) Performance of stored procedures.

Figure 5: Performance of read and write

5.3 Logical Clock and Global Clock

HiEngine uses two kinds of timestamp grant mechanism: logical clock and global clock based on Huawei hardware clock. We implement logical clock by setting a global centralized atomic variable shared by the distributed nodes, which is pushed forward (FAA) by transactions over one-sided RDMA. MVCC protocols in HiEngine works with a local CSN counter that serves as a standalone mode of our logical approach. However, under a distributed database, an important issue in logical clock is the latency of fetching timestamp across RDMA network become a bottleneck when the number of nodes increases. The performance is limited by the max NIC ability with 1.5 million PPS(Packet Per Second), and average latency reaches 40us (3 nodes) and becomes more expensive as the node concurrency increases.

Recent database systems also benefit significantly from microsecond-level time-uncertainty bound for each node(denoted as ε) [11, 32, 42]. HiEngine guarantees the same level of timestamp grant latency based on our clock synchronization. It has ε of 10µs for atomic clocks, or 20us without atomic clock. It performs two times faster than the logical clock and breaks the limitation of NIC. So we draw an experimental conclusion that centralized logical clock is not the best choice in distributed databases, the high-precision global clock can effectively provide low latency of timestamp grant and high scalability for performance.

6 EXPERIMENTAL EVALUATION

This section presents an experimental evaluation of HiEngine using microbenchmarks and standard TPC-C [44] benchmarks.

6.1 Experimental Setup

We focus on the vertical integration approach described in Section 3.4 where HiEngine is integrated with Huawei GaussDB(for MySQL) as an individual engine.

Queries are redirected to the proper engine (InnoDB or HiEngine) depending on in which engine the table is created. Both engines share the same SQL layer (networking, SQL parser and optimizer)

except that HiEngine employs code generation to remove overhead in the SQL parser and optimizer.

6.1.1 Hardware Environments. We use two kinds of environments, one with a single server, the other with a cloud deployment of separate replicated storage and compute nodes. The single-server setup allows us to observe HiEngine's raw engine performance and the latter reveals its end-to-end performance behavior when deployed in a cloud environment. The cloud environment is an ARM-based cluster that implements the aforementioned three-copy architecture. Both the storage and compute nodes are deployed with SRSS to provide the append and mmap abilities. Since the cluster is ARM-based and is yet to provide support for scalable persistent memory products such as Intel Optane DCPMM, we use DRAM in compute nodes to deploy SRSS client-side persistence; this effectively simulates the behavior of NVDIMMs.

For single-node experiments, we use both the ARM and x86-based platforms. The ARM platform is a Huawei TaiShan 200 Server with two 64-core Huawei Kunpeng 920-6426 processors (128 cores in total) clocked at 2.6GHz and 1TB of DRAM. The x86 platform is a dual-socket server with two 24-core Intel Xeon processors clocked at 2.6GHz (48 physical cores and 96 hyperthreads in total) and 512GB of DRAM.

6.1.2 Baseline Systems. We compare HiEngine with three other industrial-strength systems: DBMS-T (Huawei GaussDB(for MySQL) and without HiEngine integration), vanilla MySQL and DBMS-M (storage engine of openGauss). Both DBMS-T and MySQL use InnoDB as their storage engine. However, DBMS-T introduces improvements in the SQL layer and offloads certain functionality to the storage tier to better leverage cloud infrastructure, whereas vanilla MySQL suffers from inefficiencies such as duplicated data storage [13]; we present its numbers as a baseline for reference. DBMS-M is a recent commercial memory-optimized engine, which adopts optimistic concurrency control and also features query compilation but is not tailor made for a cloud-native environment. For fair comparison, we deploy DBMS-M only to persist data in the compute tier so that network I/O will not dominate runtime.

6.1.3 Workloads. HiEngine uses Sysbench and standard TPC-C benchmarks.

Sysbench, for all the systems under comparison, we run OLTP microbenchmarks using sysbench [26], which is a commonly used tool for benchmarking database systems in the MySQL ecosystem [13, 46]. The microbenchmarks are modeled after YCSB [10] and features short transactions involving pure reads/writes and mixed read/write workloads.

TPC-C, we also use TPC-C [44] for more realistic workloads to understand the end-to-end performance behavior of each system. It involves a mix of five concurrent transactions of different types and contains nine tables that simulate an online order processing application [44]. The initial data size of loading is 100MB per warehouse. We implement all the five standard mix transactions with Neworder 45%, Payment %43, Ordstat 4%, Delivery 4%, StockLevel 4%.

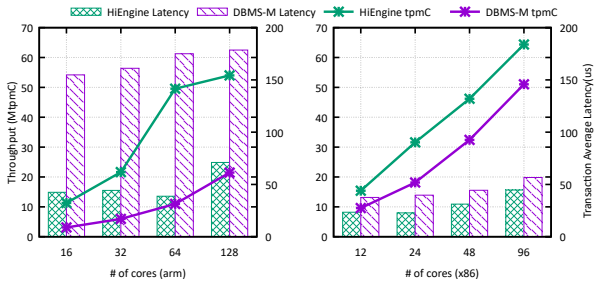


Figure 6: Overall performance on ARM and X86 platform

6.2 End-to-End Performance

We first discuss the performance of HiEngine and others using the cloud setup with compute-side persistence and replicated storage. Figure 5(a) shows the performance of each system running a read-only and a write-only microbenchmark when query requests are interpreted. HiEngine outperforms 3.6× and 7.5×-8.4× for write-only workloads compared to DBMS-T and vanilla MySQL, respectively. For pure read workloads, HiEngine exhibits 1.6× higher performance than that of DBMS-T and 4.2×-10.8× than that of MySQL.

Figure 5(b) shows the database performance with storage procedure when query requests are compiled. The 100% write performance of HiEngine is 3X-5X that of original DBMS-T and 8X-16X that of native MySQL. The throughput of simple transactions up to 1 million TPS. It is nearly 1 times as large as prepare+execute. The read-only performance of HiEngine is 2X-3X that of DBMS-T and 7X-19X that of native MySQL. We also analysis the performance impact of two platforms (X86 vs.ARM) under modified sysbench read only and write only workloads with 1 query, HiEngine achieves 5% higher throughout on ARM-based single node cluster.

6.3 Scalability

Figure 6 depicts the overall performance of HiEngine as well as DBMS-M. As demonstrated in Figure 6, HiEngine significantly outperforms DBMS-M on both platform, it outperforms DBMS-M by 2x on average (up to 4.5X) on ARM machine and achieves an average speedup of 30% against DBMS-M on X86. However, when the number of CPU cores is greater than 64, HiEngine behaves poor scalability of multi-NUMA/multi-Socket than DBMS-M on the ARM platform due to the introduction of cross-socket remote accesses. Further experimental evaluation and optimization are conducted later.

Figure 7 describes performance impact from workload partition and memory allocation policy. When the workload has partition characteristics in both engine, the proportion of across-NUMA remote accesses decreases by 26% and the tpmC performance increases by 20%. In this experiment, HiEngine storage engine outperforms DBMS-M by 60% whatever the experimental combination is. However, as described in Figure 7, because of the design of transactional thread-local row cache in DBMS-M, it introduces less cross-NUMA remote access than HiEngine in the combination of workload partition and local NUMA policy.

We quantify the effects of performance degradation on multi-socket NUMA machine depending on how threads and their memory are placed. Figure 7 indicates the impact of cross-socket access under several combination of memory node and CPU node

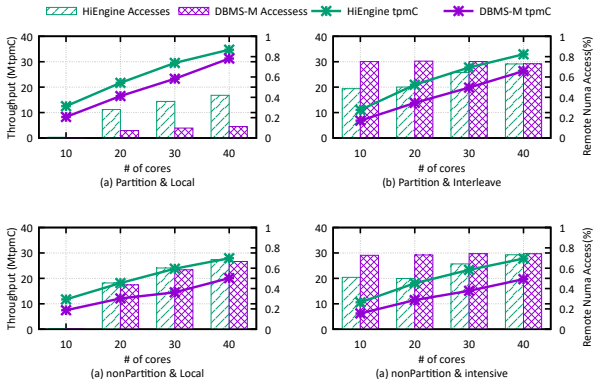


Figure 7: Performance impact of workload partition and memory allocation policy

in HiEngine. We conduct this experiment on ARM-2sockets-4dies machine, 32 CPU cores and threads per die. 2 CPU dies and 2 memory nodes are used in every experimental case. In addition, the warehouse number equals threads as well as cores, and using partitioned workload. The case1 shows the optimal performance without remote access involves. However, case2 illustrates the opposite condition to case1, which results in 69% remote accesses. According to the TPCC model and experimental results, the performance decreases approximately by 5% for every 10% increase in cross-socket remote access. This becomes a performance challenge and ongoing work on multi-socket ARM platform.

6.4 RTO with Parallel Recovery

Recovery Time Objective (RTO) is the duration of downtime that a business can tolerate and the database can be able to recover the data [39]. Figure 8 illustrates the RTO performance improvement of our parallel recovery algorithm, which uses 40 cores and 40 warehouses. Running 27 seconds with 40 worker threads will produce about 100GB of log, and the RTO is about 244s. The longer the run time, the longer the RTO. The optimized RTO performance was improved by 10X. It also explains the need for frequent checkpoints. Detailed discussions about comprehensively RTO evaluation are presented in our another upcoming work [28].

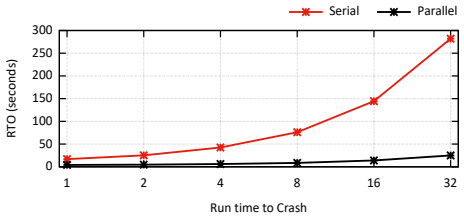


Figure 8: Performance speedup from parallel recovery

7 RELATED WORK

Our work is related to memory-optimized database engines and cloud-native systems.There have been numerous proposals [14, 23–25, 30, 33, 45] of memory-optimized database engines on modern hardware. MVCC has been a popular choice among in these systems, including academic prototypes and industrial-strength products. Hekaton [14] uses the Bw-tree [31] and an optimistic MVCC protocol with support for pessimistic transactions for concurrency control. FOEDUS [25] extends the main-memory engine design from Silo [45] to further scale up on machines with hundreds of cores and persistent memory. Hyper [23] supports hybrid OLTP and OLAP with code generation and MVCC. MOT [7] is a memory optimized in-memory production engine and integrated with Huawei openGauss. Compared to prior systems, HiEngine leverages ART [29] trees, append-only storage with indirection arrays [24] to provide MVCC capabilities, and uses the reliable storage services in the cloud to utilize PM with high availability and reliability.

The cloud presents a very different architecture that separates compute and storage, requiring a departure from the traditional monolithic design and leading to the rise of cloud-native systems. The main theme is co-design with the cloud infrastructure, i.e., stateless compute nodes and reliable storage services, by offloading appropriate functionality to the storage tier. Amazon Aurora [46], Microsoft Socrates [5], Huawei GaussDB(for MySQL) [13] and Alibaba PolarDB [2] are representative products in this area. They are designed to leverage the storage tier's high reliability and high availability provided and reduce network traffic. Most of them are also log-centric, treating the log as the database, but are still storage-centric and are based on prior on-premise solutions (MySQL/Postgres and SQL Server). In most cases, compute nodes in these systems are also stateless, allowing a user to bring up a database instance on demand by reading the data from a globally available storage service. HiEngine takes a similar approach but judiciously uses compute-side persistence to achieve low latency memory-optimized transactions, but still leverages the reliable storage services with reliability and availability guarantees.

8 CONCLUSION

Modern cloud infrastructure has been evolving quickly to feature modern hardware such as manycore processors, large main memory and fast RDMA networks. Existing cloud-native systems were mostly storage-centric designs that leave the potential of such hardware largely untapped. Meanwhile, the cloud presents a unique set of challenges for database engines, especially memory-optimized ones due to the separation of compute and storage elements. To fill this gap, we propose HiEngine, a cloud-native memory-optimized database engine that is designed to fully leverage such hardware in a cloud environment. HiEngine brings the benefits of memory-optimized database engines to the cloud, leveraging log-centric storage and compute-side persistence supported by Huawei's reliable storage services. HiEngine is integrated into GaussDB(for MySQL) that is Huawei's next-generation cloud-native distributed database. It can be deployed as an individual engine or an ACID cache in front of another engine. Evaluation results show that HiEngine is able to provide up to 7× better performance compared to prior systems.

ACKNOWLEDGMENTS

The authors gratefully thank the anonymous reviewers for their constructive comments and suggestions. We also thank professor Tianzheng Wang for his helpful discussion and review during paper preparation.

REFERENCES

[1] AgigaTech. 2017. AgigaTech Non-Volatile RAM. (2017). http://www.agigatech.com/nvram.php.

[2] Alibaba Cloud. 2018. PolarDB: Deep Dive on Alibaba Cloud's Next-Generation Database. (2018). https://www.alibabacloud.com/blog/deep-dive-on-alibaba-clouds-next-generation-database_578138

[3] Amazon Web Services. 2020. AWS Graviton Processor: Enabling the best price performance in Amazon EC2. https://aws.amazon.com/ec2/graviton.

[4] Ampere Computing. 2020. Ampere Altra Processor. https://amperecomputing.com/altra/.

[5] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. ACM.

[6] Panagiotis Antonopoulos, Peter Byrne, Wayne Chen, Cristian Diaconu, Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Prashanth Purnananda, Adrian-Leonard Radu, Chaitanya Sreenivas Ravella, and Girish Mittur Venkataramanappa. 2019. Constant Time Recovery in Azure SQL Database. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2143–2154. https://doi.org/10.14778/3352063.3352131

[7] Hillel Avni, Alisher Aliev, Oren Amor, Aharon Avitzur, Ilan Bronshtein, Eli Ginot, Shay Goikhman, Eliezer Levy, Idan Levy, Fuyang Lu, Liran Mishali, Yeqin Mo, Nir Pachter, Dima Sivov, Vinoth Veeraraghavan, Vladi Vexler, Lei Wang, and Peng Wang. 2020. Industrial-Strength OLTP Using Main Memory and Many Cores. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3099–3111.

[8] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Scalable Garbage Collection for In-Memory MVCC Systems. *Proc. VLDB Endow.* 13, 2 (Oct. 2019), 128–141.

[9] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2008. Serializable Isolation for Snapshot Databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 729–738.

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 261–264. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett

[12] Rob Crooke and Mark Durcan. 2015. A Revolutionary Breakthrough in Memory Technology. *3D XPoint Launch Keynote* (2015).

[13] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, and Yongjun He. 2020. Taurus Database: How to Be Fast, Available, and Frugal in the Cloud. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1463–1478.

[14] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, New York, NY, USA, 1243.

[15] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Rec.* 40, 4 (Jan. 2012), 45–51.

[16] Alan Fekete, Elizabeth O'Neil, and Patrick O'Neil. 2004. A Read-Only Transaction Anomaly under Snapshot Isolation. *SIGMOD Rec.* 33, 3 (Sept. 2004), 12–14.

[17] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of Memory Reclamation for Lockless Synchronization. *J. Parallel Distrib. Comput.* 67, 12 (Dec. 2007), 1270–1285.

[18] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2020. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. *Proc. VLDB Endow.* 13, 5 (Jan. 2020), 629–642.

[19] Huawei Cloud. 2020. Elastic Cloud Server. https://www.huaweicloud.com/en-us/product/ecs.html.

[20] IBM DB2. 2022. Crash recovery. https://www.ibm.com/docs/en/db2/11.1?topic=recover-crash-recovery.

[21] IEEE and The Open Group. 2016. The Open Group Base Specifications Issue 7, IEEE Std 1003.1. (2016).

[22] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: a scalable approach to logging. *PVLDB* 3, 1 (2010), 681–692.

[23] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. 195–206.

[24] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*. 1675–1687.

[25] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 691–706.

[26] Alexey Kopytov. 2020. sysbench - A modular, cross-platform and multi-threaded benchmark tool. https://github.com/akopytov/sysbench.

[27] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proc. VLDB Endow.* 5, 4 (dec 2011), 298–309. https://doi.org/10.14778/2095686.2095689

[28] Leon Lee, Siphrey Xie, Yunus Ma, and Shimin Chen. 2022. Index Checkpoints for Instant Recovery in In-Memory Database Systems. *Proc. VLDB Endow.* (2022).

[29] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, USA, 38–49.

[30] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. High Performance Transactions in Deuteronomy. In *Conference on Innovative Data Systems Research (CIDR 2015)*.

[31] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-Tree for New Hardware Platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. 302–313.

[32] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. 2020. Sundial: Fault-tolerant Clock Synchronization for Datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1171–1186. https://www.usenix.org/conference/osdi20/presentation/li-yuliang

[33] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 21–35.

[34] Lin Ma, Joy Arulraj, Sam Zhao, Andrew Pavlo, Subramanya R. Dulloor, Michael J. Giardino, Jeff Parkhurst, Jason L. Gardner, Kshitij Doshi, and Stanley Zdonik. 2016. Larger-than-Memory Data Management on Modern Storage Hardware for in-Memory OLTP Database Systems. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (San Francisco, California) *(DaMoN '16)*. Association for Computing Machinery, New York, NY, USA, Article 9, 7 pages.

[35] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (mar 1992), 94–162. https://doi.org/10.1145/128765.128770

[36] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550.

[37] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (June 1996), 351–385.

[38] Oracle. 2020. MySQL. *http://www.mysql.com* (2020).

[39] Oracle. 2022. Oracle Database VLDB and Partitioning Guide. https://docs.oracle.com/database.

[40] Andrew Pavlo. 2017. What Are We Doing With Our Lives? Nobody Cares About Our Concurrency Control Research. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 3.

[41] Mohammad Sadoghi, Kenneth A. Ross, Mustafa Canim, and Bishwaranjan Bhattacharjee. 2013. Making Updates Disk-I/O Friendly Using SSDs. *PVLDB* 6, 11 (2013), 997–1008.

[42] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. 2019. Fast General Distributed Transactions with Opacity. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 433–448. https://doi.org/10.1145/3299869.3300069

[43] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of an Architectural Era: (It's Time for a Complete Rewrite). (2007), 1150–1160.

[44] Transaction Processing Performance Council (TPC). 2010. TPC Benchmark C (OLTP) Standard Specification, revision 5.11. http://www.tpc.org/tpcc.

[45] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*.

18–32.
[46] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. *SIGMOD* (2017), 1041–1052.
[47] Viking Technology. 2017. DDR4 NVDIMM. (2017). http://www.vikingtechnology.com/products/nvdimm/ddr4-nvdimm/.
[48] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging through Emerging Non-Volatile Memory. *Proc. VLDB Endow.* 7, 10 (June 2014), 865–876.
[49] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (March 2017), 781–792.
[50] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.* 10, 6 (feb 2017), 685–696. https://doi.org/10.14778/3055330.3055335