SaS: SSD as SQL Database System

Jong-Hyeok Park†*, Soyee Choi†**, Gihwan Oh*, Sang-Won Lee*
*Sungkyunkwan University Suwon, Korea
**Samsung Electronics Co. Hwasung, Korea
{akindo19,wurikiji,swlee}@skku.edu,soyee.choi@samsung.com

ABSTRACT

Every database engine runs on top of an operating system in the host, strictly separated with the storage. This more-than-half-century-old IHDE (In-Host-Database-Engine) architecture, however, reveals its limitations when run on fast flash memory SSDs. In particular, the IO stacks incur significant run-time overhead and also hinder vertical optimizations between database engines and SSDs. In this paper, we envisage a new database architecture, called SaS (SSD as SQL database engine), where a full-blown SQL database engine runs inside SSD, tightly integrated with SSD architecture without intervening kernel stacks. As IO stacks are removed, SaS is free from their run-time overhead and further can explore numerous vertical optimizations between database engine and SSD. SaS evolves SSD from dummy block device to database server with SQL as its primary interface. The benefit of SaS will be more outstanding in the data centers where the distance between database engine and the storage is ever widening because of virtualization, storage disaggregation, and open software stacks. The advent of computational SSDs with more compute resource will enable SaS to be more viable and attractive database architecture.

PVLDB Reference Format:
Jong-Hyeok Park, Soyee Choi, Gihwan Oh, Sang-Won Lee. SaS: SSD as SQL Database System. PVLDB, 14(9): 1481 - 1488, 2021.
doi:10.14778/3461535.3461538

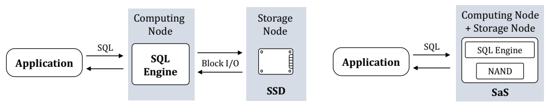
1 INTRODUCTION

All storage systems will eventually evolve to be database systems. - Jim Gray

During the last decade, flash memory solid state drives (SSDs in short) have been relentlessly replacing hard disks as the main storage media because of several advantages such as fast latency, high IOPS/$, and low power consumption. In particular, in warehouse-scale data centers, mainly because of I/O rates many orders of magnitude higher than disks, SSDs are increasingly displacing disk drives as the repository of choice for databases in Web service [9]. In addition, flash storage is 20x cheaper than DRAM in terms of capacity and the bandwidth gap between DRAM and flash is about to disappear [25]. Therefore, flash storage is projected to erode some of the key benefits of in-memory systems [1].

† Both authors contributed equally to the paper
** Work done while in Sungkyunkwan University

Proceedings of the VLDB Endowment, Vol. 14, No. 9 ISSN 2150-8097.
doi:10.14778/3461535.3461538



(a) IHDE Architecture (b) SaS Architecture

Figure 1: Database architectures: IHDE vs. SaS

Meanwhile, with modern computer architecture, the storage has been strictly separated from the host, and they interact with each other with the block I/O interface. The dichotomy of host and storage has forced database engines to segregate their data processing logic from storage; thus, a database engine has to bring data from the storage to the host, read and/or update data, and then write updated data back to the storage for the durability.

This conventional *in-host database engine* (IHDE in short) architecture becomes undesirable when low-latency SSDs are used as storage mainly for three reasons. First, the legacy I/O stack overhead makes fast SSDs perform suboptimally [62]. A storage device is connected to a host through a host bus adapter (HBA) in a computer system (Figure 1.(a)). Between the segregated direct-attached storage and host are located the I/O software stacks. While the I/O stacks contribute to the latency of data access and the energy consumption on hard disks very marginally (*i.e.*, less than 1 percent for both metrics), the same software accounts for 70 percent of the latency and 87.7 percent of the energy on faster SSDs [62], thus deterring SSDs to realize their full performance potentials. Even with numerous I/O stack optimizations [38] or direct access schemes from user-space to storage [12, 29, 37], some overheads such as system calls and memory copy will remain inevitable. Second, the intervening I/O stack will hinder the ample opportunities for vertical optimization between database engines and SSDs. Recently, several novel interfaces for flash storage have been proposed, which allow to achieve the transactional atomicity in databases [35, 48] and manage the file system consistency without resorting to redundant journaling [32, 45]. To be practical, however, those interfaces have to be implanted across all the existing OS and file systems. But, according to our experience [32, 35, 45, 48], it is a non-trivial task to achieve vertical optimizations across the I/O stack, even in a single OS/file system. Third, the physical segregation of host and storage is neither economical nor architecturally elegant. It costs more because of the separate pricing to host and storage, requires more space to accommodate both components, consumes more power, and necessitates the storage interface.

Further, the IHDE inefficiency is not confined to a single computer node with direct attached storage. Its problem will be further exacerbated in modern data centers which opt for virtualization and disaggregation [76, 78]. The virtualization layer prolongs the I/O stack between database engine and storage. Numerous open software stacks in cloud environment will further widen the gap between two layers [61]. In addition, as the storage is disaggregated from the compute for the elastic computation and their independent scaling, the network latency between the decoupled compute and storage will worsen the data movement overhead of IHDE. Thus, it is compelling to re-architect database engines to minimize data movement in warehouse-scale data centers [1, 68].

Meanwhile, some SSD vendors are striving to enhance their storage devices with more computing power and networking capability [13, 17]. In addition, an OpenSSD board [15] is already equipped with ARM server-grade cortex A53 CPU, FPGA, and 100G Mellanox Ethernet. With networking enabled, the OpenSSD SSD can play the role of stand-alone computer. Note that the A53 CPU is powerful enough to run server-class database applications [57]. In summary, SSDs are evolving from mere storage devices to *super-computers.*

Because it takes time and power to move data from one location to another, all data suffers from data gravity [22]. In particular, in many modern data centers with resources disaggregated, moving data between storage and processors takes more time and as much power as the processing itself. For this reason, in terms of data gravity, it has long been recognized that it is better to pull computation close to the large data than to bring data to the computation [71]. However, the idea of offloading the computation to the storage has been considered impractical mainly for cost reasons [22].

As explained so far, enabling technologies as well as driving forces are mature enough to trigger the paradigm shift in database computing from host-centric to SSD-centric. In particular, the trend toward resource-rich SSDs is a strong enabling force for the *Darwinian evolution* to new SSD-based database architecture. Thus, it is high noon to depart from the 50 years-old IHDE architecture. In this paper, as a true realization of Jim Gray's vision that "*all storage systems evolve to be database systems*" [22], we envision SaS (SSD as SQL database system), as illustrated in Figure 1.(b). Its essential idea is "to ship SQL processing to data, not data to the SQL engine". The SaS approach is in a consistent vein with a recent research direction in database community which offloads some computes to where the data resides (*e.g.*, storage, memory) or along the path the data moves [1, 8, 11, 24, 28, 41, 43, 75], but is more radical in that it offloads a full database engine to SSD.

2 SAS: SSD AS SQL ENGINE

In SaS, a complete database engine (*e.g.*, SQLite) in its entirety is pushed down to SSD and runs tightly integrated with SSD architecture, and SQL is provided as primary interface. With SaS, an SSD plays not only the conventional role of storage but also the role of SQL database engine. Hence, applications in other computing nodes (*e.g.*, Apache Memcached) can, without any intervening in-host SQL engine (Figure 1.(a)), directly interact with the SSD itself using SQL over the network (Figure 1.(b)).

Advantages By embedding a complete database engine into an SSD, SaS *evolves* the SSD from a dummy block device to a DB server, which in turn allows eliminating numerous IHDE-induced technical and economical inefficiencies. First, by offloading a complete database engine into an SSD and tightly integrating both layers, SaS will avoid the IO stack overhead and also realize unprecedented vertical optimizations between SQLite engine and FTL

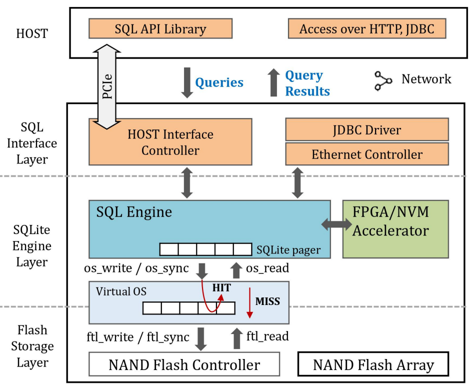


Figure 2: SaS: Overall Architecture

layer. Also, since one physical SaS device can play the role of both computing and storage nodes, thus replacing the segregated host and storage, SaS is an elegant and economical architecture. In particular, SaS can be a *building block for scale-out architecture*, where each SSD has its own computing and networking capabilities [63]. Further, both SSD vendors and customers benefit from SaS. For SSD vendors, the concept of SaS will add significant value to their SSDs by turning simple storage devices into database computing nodes. For customers, SaS is an attractive solution in terms of costs, manageability, and power consumption.

Target Applications Taking into account the limited resource capabilities of CPU and DRAM in today's SSDs, we do not expect in the foreseeable future that SaS can compete with the existing IHDEs at the enterprise-class applications. However we believe that SaS can be an attractive, cost-effective database computing and storage substrate for numerous emerging applications such as edge computing, IoT, and smart city [46, 50, 69], whose workloads can be characterized as data-intensive, intermittent, bursty, and unpredictable. In such environments where computing power tends to lack and power-efficiency, cost-effectiveness and manageability are important metrics, SaS could be "a natural selection" over the IHDE architecture with separate host and storage. In addition, SaS could be deployed in the cloud environment as *serverless solutions* for small-scale blog databases with intermittent connections since SaS can start at no latency and can scale to zero for unpredictable connections and disconnections [2, 3].

2.1 Architecture and Design Goals

The vision of SaS cannot be realized by simply porting a database engine to run inside SSD controllers. Its realization brings at least three challenges: limited computing power, limited OS functionalities, and SQL interface support on top of SSD with block interface. At the same time, the SaS approach provides opportunities such as vertical optimization and hardware-based acceleration.

With these challenges and opportunities in mind, we set three design goals of SaS, explained below. Figure 2 shows the strawman SaS architecture where its three layers of SQL interface, SQLite engine, and flash storage are illustrated using the Cosmos Open SSD [49].

SQL Interface from SSD A prerequisite for an SSD to evolve into an SQL database system is to support the SQL interface. As shown in Figure 1, SaS supports SQL as its primary interface over the PCIe storage protocol as well as the network (*i.e.*, JDBC). In particular, it is quite novel but challenging to support the tuple-oriented SQL interface over the block-oriented storage interface.

Vertical IO Optimization As discussed above, the IHDE approach is suboptimal due to the legacy IO stack overhead and limited vertical optimization. The second design goal of SaS is to eliminate IO stack overhead and explore various vertical IO optimizations between SQLite engine and flash storage layer. This goal is achievable by integrating the database engine with FTL: the engine bypasses the kernel IO stacks and directly interacts with FTL. Fortunately, as detailed later, the integration of SQLite engine with the FTL code in Cosmos board can be accomplished with minimal changes in both modules because SQLite, like other DBMS engines, has its virtual OS layer for portability across different OSes [60].

Hardware-assisted Acceleration Another design goal of SaS is to accelerate its performance by exploiting hardware resources which are expected to be available from computational SSDs [14]. This goal is well-aligned with research themes outlined in the Seattle report [1], including heterogeneous computation, NVM, and HW-SW co-design. As recently demonstrated [39, 43], FPGA specialization can power-efficiently accelerate a few key operations in SQL engines such as selection, group-by, and even index probing. We expect that the tight architectural integration of CPU and FPGA in the Cosmos+ board [40] enables a more efficient and flexible offloading than the existing approaches. In addition, with the help of NVM (Non-Volatile Memory) directly accessible from the CPU of the Cosmos board, SaS will be able to realize near zero-latency durability, thus boosting the performance of OLTP workloads [34, 53].

2.2 Related Work

In-Storage Processing. The idea of offloading computation down to the storage and thus minimizing the data transfer between host and storage in SaS is not new or novel. Since the late 1970s, numerous in-storage processing (ISP in short) schemes have been intermittently proposed in the names of database machine, intelligent, and active disk [10, 36, 55]. The ISP approach has been recently revisited in the context of SSDs [18, 23, 30, 65]. Though, the ISP approach has not been so successful in the commercial market, except for Oracle's Exadata [70]. To our best knowledge, all the existing ISPs have *partially offloaded* specific data-intensive operations such as selection, join, and aggregation to the storage while the main body of the database engine resides in the host and thus still has to interact with the storage device. In contrast, SaS fully offloads an SQL engine to the storage so that it can completely bypass the IO stack overhead and freely explore vertical optimizations.

The idea of ISP has recently revived in cloud databases running on data centers where compute and storage nodes are decoupled for elastic computing. To minimize the increased access latency to the storage in disaggregated data centers, database engines need to be rearchitected [1]. For instance, Amazon AuroraDB [68] pushes redo log processing to the storage node, instead of writing each dirty page in its entirety. Unlike AuroraDB, SaS is running a full database engine inside the storage itself. In this respect, SaS can be said to take the "anti-disaggregation" approach.

High-level Storage Interfaces SaS provides the SQL language as its interface. Likewise, there have been other innovative approaches which support high-level semantic interface beyond the dummy block interface, such as OSD (Object Storage Device) [20] and KV-SSD (Key-Value SSD) [54]. In particular, KV-SSD is recently proposed to support the simple put/get interface of key-value stores as the new abstraction of SSDs. We believe that, compared to object or key-value interfaces, the SQL interface is the most appropriate abstraction for offloading database management functionality to computational SSDs. First of all, SQL is the lingua franca for data access, and thus, the existing SQL-based application can easily migrate to SaS. Second, SaS carries out query processing inside SSD so that the data movement between query processor and flash memory chip would be quite less than that in OSD and KV-SSD approaches.

IO Stack Optimizations SaS is different from the existing IO stack optimization schemes [12, 29, 37, 38]: while the latter aims at minimizing IO stack overhead, SaS will eliminate the overhead since no data need to move between host, and storage and the IO stack between DB engine and FTL is also removed. In addition, SaS can further reduce IO operations by vertically integrating the DB engine with FTL, which is orthogonal to any IO stack optimization.

NVM-based Databases As NVM (non-volatile memory) becomes commercially available (*e.g.*, Intel's DC Optane Persistent Memory), numerous NVM-based database techniques have been proposed to leverage its characteristics such as low latency and byte-addressability for better performance [6, 66, 67]. Since NVM is, like DRAM, directly accessible by CPU, NVM-based database systems can achieve persistence and scalability while eliminating the IO stacks. However, since the role of NVM is still unclear for database workloads due to its price [26], we expect that flash will be used as the primary storage for databases. Hence, we pursue SaS as the opposite extreme of NVM-based databases.

3 SAS-ZERO: CURRENT STATUS

To demonstrate that SaS could be a competitive execution substrate for next-generation database engines and also to identify its opportunities as well as limitations, we have prototyped a simple version of SaS (hereafter, *SaS-Zero*) on the real Cosmos+ OpenSSD [40]. The board employs a controller based on Dual Core ARM Cortex-A9 on top of Xilin Zynq-7000 board with 256KB SRAM, 1GB DDR3DRAM, and 32GB MLC Nand flash memory. In SaS-Zero, SQLite [58] is chosen as the database engine primarily because of its small codebase, tiny memory footprint, and minimal OS dependency. The source code of SQLite 3.29 amalgamation version written in C was compiled as the firmware for the Cosmos+ board and then offloaded down to the designated NAND flash memory area in the SSD. In addition, the bootloader of the SSD was modified to load the SQLite runtime binary to the designated DRAM area upon booting. The SQLite heap memory including the page cache are also located in Cosmos+ board DRAM, together with the FTL mapping information. Thus, as depicted in Figure 2, the SQLite engine itself, not FTL, is responsible for interfacing with host applications.

3.1 SQL Interface Layer

SaS-Zero currently supports the SQL language through the storage interface and the Ethernet network interface.

First, when directly attached to the host, SaS supports SQL commands by two vendor specific commands, *sas_query* (0x81) and *sas_result* (0x82). To be specific, by leveraging the *nvme-io-passthru* command, which allows to submit arbitrary IO commands [16, 47], host-side applications can send SQL statements and receive their results to/from SaS via newly specified sas-commands. The format of the commands is almost same as the existing NVMe read/write command. However, if *sas_query* command is issued, SaS will regard the data from the host to the firmware buffer through DMA as an SQL statement, then execute it on the SQLite inside SaS, and save the query's result, including even error message at the buffer. Meanwhile, if *sas_result* command is issued, SaS will pass the query result from the firmware buffer to the host buffer via DMA.

Second, when accessed over the network via JDBC-like interface, the network layer in SaS-Zero is implemented by porting the network module to handle packet processing, user-level network protocol processing (TCP), non-blocking, and event-driven scheduling. To be specific, SaS-Zero implements using the lwip raw API to execute callback functions, which are invoked by the lwip core function when the corresponding event occurs [27]. This user-level TCP stack was taken because the OS service is unavailable with the Cosmos+ board. Bypassing the OS kernel while handling SQL statements through the network controller, SaS-Zero can avoid the interference of the networking stack scheduler and also the context switch overhead, achieving higher throughput and lower latency.

3.2 SQLite and Virtual OS Layer

Like other DBMSs, SQLite relies on several OS functionalities such as file I/O and memory management. Unfortunately, however, the Cosmos+ board does not support any OS as of now. Thus, for all OS primitives on which SQLite is dependent, we have to customize corresponding functions. Here we elaborate on how SaS-Zero makes SQLite engine directly interface with the vanilla FTL in Cosmos+ board with no intervention of OS kernel. We also discuss how technical issues such as heap memory allocation, address translation, and IO completion are handled in SaS-Zero. Also, we illustrate two optimizations enabled by the integration of SQLite and FTL.

VFS4Cosmos When it needs to use any file service to store and access data, SQLite does not directly call the underlying OS but instead invokes its virtual OS interface (*VFS*) layer [60]. The VFS layer then redirects the request to the underlying OS. The indirect abstraction of file service via the *VFS* layer is intended to make SQLite portable across various operating systems. In order to bypass the OS kernel for file operations, we extended the vanilla FTL in the Cosmos+ to support file operations for SQLite and also accordingly modified the VFS layer to call the extended FTL. Thus, upon receiving any file operation (*i.e.*, OS_read, OS_write, or OS_sync) from the upper SQLite layer, the modified VFS, *VFS4Cosmos*, will redirect the operation to the corresponding operation of the extended FTL (*i.e.*, ftl_read, ftl_write, or ftl_sync). With these minimal changes in both VFS and FTL layers, all the upper SQLite layers work without any change, the legacy OS kernel layers are completely bypassed, and two layers of SQLite and FTL are more tightly integrated.

Unified Space Management and Address Translation Since each file system and FTL manages its space and carries out address translation, many approaches have been proposed to unify the redundant functionalities [44, 73]. In this regard, the integration of SQLite and FTL in SaS will bring an opportunity for trimming the redundant space management functionalities at two layers and reducing the address translation overhead. SaS-Zero as of now regards the whole physical space of NAND flash memory as one file. Thus it can support only a single database instance. For this reason, Sas-Zero can skip the offset-to-LBA translation and instead translate the file offset into a logical block address (*i.e.*, LBA) using the simple modular operation.

Memory Allocation The memory management functionality is essential to run SQLite. In addition, the DRAM area with limited capacity has to be shared by the SQLite and the Cosmos+ firmware (*i.e.*, FTL) in SaS-Zero. The Cosmos+ board, however, does not provide any primitive for memory management on which SaS-Zero and SQLite rely. Because of this, for memory management in SaS-Zero, we borrowed the *memsys5* memory allocator in SQLite [59], which allows static allocation of SQLite memory region. The memsys5 allocator is designed for use on embedded systems such as Cosmos+ board. When SQLite is compiled with the option SQLITE_ENABLE_MEMSYS5 enabled, the memsys5 memory allocator will be included in the build. Under the memsys5 memory allocator, a fixed size of DRAM area is statically pre-allocated and a first-fit and buddy-allocator algorithm will, using the region, handle dynamic memory (de-)allocation requests from the SQLite. Thus, while using dynamic memory (de-)allocations, SQLite will never access data out of the bound. In this way, SaS-Zero can prevent SQLite's dynamic memory allocation from encroaching the metadata area of FTL.

Polling-based IO Completion After submitting read, write, or sync IO request to the storage, SQLite has to wait until the IO completes. Thus, SaS-Zero needs an IO completion model. Despite its system overheads of interrupt handling, the interrupt-driven model has been taken by the most OS kernels because the model has worked well for slow rotating harddisks. With SaS-Zero, however, we decided to take the synchronous polling-based IO completion model for several reasons. First, Cosmos+ board does not provide any OS support for interrupt handling. Second, the synchronous completion allows I/O requests to bypass the kernel's heavyweight asynchronous block I/O subsystem. Third, and most importantly, on top of fast flash storage, the synchronous polling is, despite its spin-waiting for the completion, known to outperform the asynchronous interrupt [72]. Lastly, since VFS4Cosmos in SaS now directly communicates with FTL, we believe that the polling mechanism can be further enhanced to optimize the polling time as well as to reduce CPU cycles and thus power consumption for polling.

Reducing Memory Copies For each page write operation in IHDE, the same page content has to be multiply copied along the multiple IO stacks, such as file systems, device drivers, and DMA; from user buffer to kernel buffer, to device buffer (via DMA request), to NAND buffer, eventually to NAND flash chip. The same is true with the page read operation. In contrast, since SaS can minimize the layers between applications and NAND flash memory chips and data is processed near the storage, it will drastically reduce the number of page copies and the amount of data to transfer to the applications. Even with the initial version of SaS-Zero, two memory copy operations are necessary for each page read or write operation: that is, one between SQLite's pager and VFS's cache and the other between VFS's cache and NAND flash chip buffer. To further reduce the memory copy operation, we modified the VFS layer to directly transfers pages between SQLite's pager and NAND flash chip buffer, thus bypassing its in-between cache.

Supporting the Share Interface: Finally, we also extended the vanilla FTL with the *SHARE* interface [48] which can guarantee the atomic page writes upon failures. With the help of *SHARE*, compared to IHDE which has to write pages redundantly, SaS could halve the amount of write to the flash storage and accordingly the time taken to flush pages upon transaction commit.

3.3 Preliminary Results and Lessons

Experimental Setup The experiments were done on a Ubuntu 18.04 LTS with 5.4.19 kernel. It was equipped with four Intel (R) Core(TM) i7-6700 3.4 GHz processor and 48GB DRAM; and Cosmos+ board as the database storage device. To evaluate the performance benefit of SaS-Zero, we measured the CPU and IO times while running two benchmarks, AndroBench [4] and Python TPC-C (Pytpcc) [5], using SQLite in IHDE and SaS modes, respectively. AndroBench is a simple update-intensive mobile benchmark with 1,024 auto-commit update statements. The results are presented in Table 1. For IHDE, the `ext4` file system in ordered mode was used, and the SQLite engine was run in `rollback` journal mode. In the case of Pytpcc, we measured the elapsed time taken to complete 10,000 transactions against initial database of 100MB (*i.e.*, 1 warehouse) in each mode. In all experiments, the size of database page was set to 16KB to match the page size of flash memory chips used in the Cosmos+ board. In addition, the buffer sizes of both SQLite modes are equally set to 32MB (*i.e.*, 2,000 pages by default). Besides, for IHDE mode, the direct I/O option was enabled to minimize the interference from ext4's page caching.

Analysis To our surprise, as shown in Table 1, SaS-Zero outperforms IHDE in terms of total execution times for both benchmarks. Though spending more CPU times than IHDE, SaS takes quite less IO time. Provided that the CPU used for IHDE and SaS was Intel i7-6770 3.4GHz and Arm Cortex-A9 1GHz, respectively, IHDE will apparently outperform SaS in terms of CPU time. But, the CPU time gap between the two modes is rather quite small considering the absolute speeds of the two CPUs used in the experiment. This is mainly because SaS, unlike IHDE, bypasses the kernel IO stacks. Meanwhile, SaS has, compared to IHDE, reduced the IO time by approximately 50% and 33% in both benchmarks, respectively. This improvement in IO time is due to two optimizations: single write journaling by share and memory copy reduction. In addition, unlike IHDE, SaS need not transfer pages between host and storage. The IO improvement ratio in Pytpcc is slightly less than that in AndroBench since Pytpcc includes read-only transactions while AndroBench does only update statements.

In addition, to evaluate the effect of the host-level IO stack optimization on IHDE, we modified SQLite so as to run on io_uring [7], which was recently incorporated into Linux mainstream, and measured the CPU and IO times taken to complete each of two benchmarks when run in the modified SQLite version. The results (two rows denoted as **IHDE-iouring**) are also included in Table 1. io_uring was designed to achieve zero-copy and to minimize the invocations of system calls by leveraging the shared ring buffer between user and kernel space, to utilize asynchronous IOs for better throughput, and to squeeze the performance of low-latency SSDs by adopting polling-based IO [7]. However, since IOs in vanilla SQLite are mostly synchronous, io_uring has less effect on reducing the IO stack overhead. Thus, to further leverage the benefit of io_uring, we have modified IHDE-iouring to equip the polling-based IO completion policy by enabling the **IORING_SETUP_IOPOLL** option. As shown in Table 1, IHDE-iouring can significantly improve the the IO stack overhead over IHDE. Nonetheless, IHDE-iouring is still considerably outperformed by SaS in terms of IO time, because SaS can achieve the single-write journaling with the help of **Share** while IHDE-iouring has to resort to redundant journaling for write atomicity. This further motivates the SaS approach.

(unit: second)		CPU	IO	Total
Androbench	IHDE	1.29	9.78	11.07
	SaS	1.52	5.13	6.65
	IHDE-iouring	0.69	9.33	10.02
Pytpcc (1 wh)	IHDE	33.4	143.4	176.8
	SaS	54.77	95.95	150.72
	IHDE-iouring	15.54	147.74	163.28

Table 1: Performance Comparison: IHDE vs. SaS

Lessons from SaS-Zero The above comparison between IHDE and SaS-Zero illustrates both the opportunities and limitations of SaS. Since it bypasses the kernel IO stacks, SaS can improve the IO efficiency as well as save the CPU cycles. In addition, various vertical optimizations made with SaS can drastically reduce the I/O time. Meanwhile, the limited computing power (*e.g.*, Arm Cortex A9) and resource (*i.e.*, less CPU cache and slower DRAM) in Cosmos+ SSD will make SaS slower than IHDE. Thus, we need to strengthen its opportunities further while compensating its limitations.

3.4 Limitations

Though innovative, the current SaS artifact has a few limitations in terms of scalability, programmability, and multi-instance support, each of which we believe could be overcome.

First, the SaS approach may not suit applications with data set too large to fit on a single SaS device. In such large-scale data processing, data has to be partitioned across multiple SaS devices, and a user-level query needs to be accordingly decomposed into subqueries over those SaS nodes. Hence, we may need to develop distributed query processing scheme coordinating multiple SaS nodes [21]. Alternatively, *sharding* can be used to make the SaS approach scalable, which splits data into many shards and distributes them across multiple database instances [52]. In practice, the sharded database architecture has been widely taken by major cloud service providers as well as database vendors as scale-out solutions for their database services [51, 52, 74, 77]. In particular, their shard managers allow scaling database backend without any sharding logic in database applications. They shed light on developing automatic and transparent sharding mechanisms for multiple SaS instances running in a shared nothing manner.

Second, developing and maintaining the software in SaS is less flexible and more time consuming than IDHE because a flexible and generic programming model for high-level languages such as C is not available from SSDs. As computational SSDs will proliferate, however, new programming environments for SSDs are expected to be defined and available in the foreseeable future [8, 19]. In particular, Samsung has already demonstrated [23, 30] that commodity SSD can offer a high-level programming model with generality and expressiveness enough to support database engine modules.

Third, one SaS device could not support multiple database instances. As a consequence of circumventing file system, the current SaS-Zero artifact simply assumes the whole NAND space as a single file, treats the SQLite file offset as LBA, and therefore supports only a single address space. This is a serious constraint that no database system is willing to make, in particular given that SSDs with the capacity of several tens TBs are not uncommon and SSD capacities are ever growing. Hence, we will investigate supporting multiple databases instance in a SaS device in the future.

4 FUTURE RESEARCH DIRECTIONS

Based on the experience with SaS-Zero, we have identified three research directions which can make SaS more competitive or can overcome some limitations of current SaS-Zero artifact.

SQL Support from SSD SaS has to support the SQL interface for both contexts of direct attached storage (DAS) and networked environment. In the case of networked SaS, we expect to be able to implant the conventional standard database interfaces such as JDBC and access-over-HTTP easily on top of the Ethernet Physical Layer (*i.e.*, Ethernet PHY). However, in the case of DAS environment, it is challenging to support tuple-oriented and set-oriented semantics on top of the dummy block interface. To be specific, SaS should be able to accept SQL statements of variable length as its input from applications and then return a set of variable-sized tuples as its output to applications. In sending SQL statements to SaS, we can emulate using the NVMe write command or by buffering SQL statements in device driver's buffer. In receiving a set of tuples from SaS, however, the existing storage interface is inappropriate for two reasons. First, since the size of query result is not known in advance, the existing storage read command can not be naively utilized. Second and more importantly, since the execution time of an SQL statements is, depending on many parameters, unpredictable and usually takes long, the interrupt-based block interface protocol is inappropriate for this scenario. To overcome the impedance mismatch between the set-oriented SQL interface and the block-oriented storage interface, we need to explore a large design space about the unit of SQL result to be returned to applications, the checking mechanism of SQL statement completion, and the number of SQL statements to be concurrently serviced by a SaS device. In addition, we need to develop a sharding mechanism which can manage large data over multiple SaS instances.

Tighter Integration of DB engine and FTL SaS-Zero has removed the intervening IO stacks between DB engine and FTL, thus eliminating significant run-time overheads from the stacks. We will further realize the full performance potential of SaS by tightly integrating DB engine with FTL. To be specific, by making DB engine aware of the underlying SSD architecture and the data layout and also by allowing DB engine to control the physical placement of data in SSD and to schedule the data accesses [31, 64], we can further optimize most major database module such as storage manager, buffer manager, query processor, and transaction manager [18, 35]. For instance, by placing data pages with different lifetimes in different flash blocks [33], SaS can drastically improve SaS in terms of write amplification, performance, and lifespan, thus realizing the concept of multi-streamed SSD truly. As another example, we can accelerate B-tree traversal by adding the functionality of reading multiple pages at once to the FTL interface [56]. Also, to support multi-tenant database in one SaS device, we plan to introduce another address translation layer between SQLite engine and FTL, which is aware of database instances (*i.e.*, name or ID).

Hardware-assisted acceleration Next-generation SSDs will be armed with richer computing resources. For instance, the new version of Cosmos+ board, Daisy OpenSSD [15], will use server-grade A53 CPU, larger FPGA logic gates, 100G Ethernet, and battery-backed NVM. To make SaS perform comparable to IHDE, it is compelling to accelerate core database functionalities by leveraging such hardware features. In this regard, we plan three hardware-assisted accelerations. First, based on the performance breakdowns from the experiments in Section 3.3, we identified three parallelizable tasks as candidates for FPGA implementations: SQL parsing, B-tree traversal [39], and query processing operators (*e.g.*, sorting and joins[43]). Second, we expect to boost the transaction processing by utilizing a small amount of NVM in novel ways. Byte-addressable NVM can realize the full potential of logical logging [53]. In addition, we expect to further reduce the amount of writes to flash memory by caching the small number of hot dirty pages in NVM. These optimizations will also shorten the recovery time and prolong the lifespan of the flash storage [53]. Third, we will extend SaS to support the replication mechanism where the master can propagate its update to its slave SaS devices efficiently and consistently with the help of 100G Ethernet, RDMA, and NVM. For instance, the primary SaS can directly store its new SQL statements to its slaves' NVM over the RDMA interface [42, 53].

5 CONCLUDING REMARKS

In this paper, we envisioned the idea of SaS and argued it is high noon for SSDs to evolve from dummy block devices to substrates for running database engines. In addition, we also demonstrated that moderate prototyping of SaS using a real SSD can perform comparable to IHDE. Finally, based on this experiment, we set out several interesting directions for future SaS research.

ACKNOWLEDGMENT

We thank the anonymous PVLDB reviewers for their valuable comments. This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1802-07.

REFERENCES

[1] Daniel Abadi et al. 2020. The Seattle Report on Database Research. *SIGMOD Rec.* 48, 4 (Feb. 2020), 44–53. https://doi.org/10.1145/3385658.3385668

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*. USENIX Association, USA, 923–935.

[3] Amazon Web Services 2020. How Aurora Serverless Works. https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-serverless.how-it-works.html.

[4] AndroBench 2011. AndroBench (SQLite Benchmark). http://www.androbench.org/wiki/AndroBench.

[5] apavlo 2011. py-tpcc. https://github.com/apavlo/py-tpcc.git.

[6] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's Talk About Storage amp; Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*.

[7] Jens Axboe. 2019. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf?source=techstories.org.

[8] Antonio Barbalace and Jaeyoung Do. 2021. Computational Storage: Where Are We Today?. In *Proceedings of the 11th Conference on Innovative Data Systems Research (CIDR '21)*.

[9] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition.* http://dx.doi.org/10.2200/S00516ED2V01Y201306CAC024

[10] H. Boral and D. DeWitt. 1983. Database Machines: An Idea Whose Time has Passed? A Critique of the Future of Database Machines. In *Proceedings of International Workshop on Database Machines*.

[11] W. Cao, Y. Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, F. Zhu, and Tong Zhang. 2020. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*. USENIX Association, USA.

[12] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. 2012. Providing Safe, User Space Access to Fast, Solid State Disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*.

[13] Swati Chawdhary and Sandeep Kumar Ananthapalli. 2019. NVMe-oF Ethernet SSD. Storage Developer Conference (SNIA India).

[14] Rakesh Cheerla. 2019. Computational SSDs. https://www.snia.org/sites/default/files/SDCEMEA/2019/Presentations/Computational_SSDs_Final.pdf.

[15] CRZ Technology 2020. Daisy OpenSSD. http://www.mangoboard.com/main/view.asp?idx=1061&pageNo=1&cate1=9&cate2=150&cate3=181.

[16] debiman 2020. nvme-io-passtru man page. https://manpages.debian.org/testing/nvme-cli/nvme-io-passthru.1.en.html.

[17] Samsung Memory Division. 2020. SmartSSD: Computational Storage Drive. https://samsungsemiconductor-us.com/smartssd/.

[18] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1221–1230.

[19] Jaeyoung Do, Sudipta Sengupta, and Steven Swanson. 2019. Programmable Solid-State Storage in Future Cloud Datacenters. *Commun. ACM* 62, 6 (2019), 54–62.

[20] Gregory R. Ganger. 2001. Blurring the Line Between OSes and Storage Devices. Technical Report CMU-CS-01-166, Carnegie Mellon University.

[21] V. Giannakouris, N. Papailiou, D. Tsoumakos, and N. Koziris. 2016. MuSQLE: Distributed SQL query execution over multiple engine environments. In *2016 IEEE International Conference on Big Data (Big Data)*.

[22] Jim Gray. 2002. Data Centric Computing. In *Usenix Conference on File and Storage Technology (Keynote) (USENIX FAST '02)*.

[23] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, 153–165.

[24] David Cock Mohsen Ewaida Kaan Kara Dario Korolija David Sidler Zeke Wang Gustavo Alonso, Timothy Roscoe. 2020. Tackling Hardware/Software co-design from a database perspective. In *Proceedings of the 10th Conference on Innovative Data Systems Research (CIDR '20)*.

[25] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *Proceedings of the 10th Conference on Innovative Data Systems Research (CIDR '20)*.

[26] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *Proceedings of the 10th Conference on Innovative Data Systems Research (CIDR 2020)*.

[27] Man-Qing Hu, Geng-Xin Liu, Xiao-He Liu, and Yi-Fei Jian. 2016/12. Design and Implementation of the LwIP in Embedded System. In *Proceedings of the 2nd Annual International Conference on Electronics, Electrical Engineering and Information Science (EEEIS 2016)*. Atlantis Press, 175–180. https://doi.org/10.2991/eeeis-16.2017.24

[28] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-Scale E-Commerce Transaction Processing. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. 651–665. https://doi.org/10.1145/3299869.3314041

[29] INTEL. 2020. Storage Performance Development Kit. https://spdk.io/.

[30] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2016. YourSQL: a high performance database system leveraging in storage computing. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 924–935.

[31] Aarati Kakaraparthy, Jignesh M. Patel, Kwanghyun Park, and Brian P. Kroth. 2020. Optimizing Databases by Learning Hidden Parameters of Solid State Drives. *Proceedings of VLDB Endowment* 13, 4 (2020).

[32] D. H. Kang, C. Min, S. Lee, and Y. I. Eom. 2020. Making Application-Level Crash Consistency Practical on Flash Storage. *IEEE Transactions on Parallel and Distributed Systems* 31, 5 (2020).

[33] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The Multi-streamed Solid-State Drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14) (HotStorage'14)*. USENIX Association, USA, 13.

[34] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Yang-Suk Kee, and Moonwook Oh. 2014. Durable Write Cache in Flash Memory SSD for Relational and NoSQL Databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 529–540.

[35] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. 2013. X-FTL: Transactional FTL for SQLite Databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, 97–108.

[36] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. 1998. A Case for Intelligent Disks (IDISKs). *SIGMOD Record* 27, 3 (Sept. 1998), 42–52.

[37] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. NVMeDirect: A User-Space I/O Framework for Application-Specific Optimization on NVMe SSDs. In *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'16)*.

[38] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. 2017. Enlightening the I/O Path: A Holistic Approach for Application Performance. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST'17)*.

[39] Kim, Kangnyeon and Johnson, Ryan and Pandis, Ippokratis. 2019. BionicDB: Fast and Power-Efficient OLTP on FPGA.. In *EDBT*. 301–312.

[40] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. 2020. Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems. *ACM Transactions on Storage* 16, 3, Article 15 (July 2020).

[41] Feifei Li. 2019. Cloud-Native Database Systems at Alibaba: Opportunities and Challenges. *Proceedings of VLDB Endowment* 12, 12 (Aug. 2019), 2263–2272.

[42] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. 2016. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 355–370.

[43] Woods Louis, Istvan Zsolt, and Alonso Gustavo. 2014. Ibex—An Intelligent Storage Engine with Support for Advanced SQL Off-loading. *Proceedings of the VLDB Endowment* (2014), 963–974.

[44] P. Bonnet M. Bjørling, J. Gonzalez. 2017. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*. USENIX Association, USA, 359–373.

[45] Changwoo Min, Woon-Hak Kang, Taesoo Kim, Sang-Won Lee, and Young Ik Eom. 2015. Lightweight Application-Level Crash Consistency on Transactional Flash Storage. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*.

[46] Max Mühlhäuser, Christian Meurisch, Michael Stein, Jörg Daubert, Julius Von Willich, Jan Riemann, and Lin Wang. 2020. Street Lamps as a Platform. *Commun. ACM* 6 (June 2020), 75–83.

[47] nvme-user 2013. NVM-Express user utilites and test program. http://git.infradead.org/users/kbusch/nvme-user.git.

[48] Gihwan Oh, Chiyoung Seo, Ravi Mayuram, Yang-Suk Kee, and Sang-Won Lee. 2016. SHARE Interface in Flash Storeage for Relational and NoSQL Database. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 343–354.

[49] OPENSSD TEAM 2019. The OpenSSD Project. http://www.openssd.io.

[50] Microsoft(White Paper). 2020. Azure SQL Edge: A data engine optimized for IoT workloads on edge devices. https://azure.microsoft.com/mediahandler/files/resourcefiles/azure-sql-edge-whitepaper/White_Paper_AzureSQLEdge_2019.pdf.

[51] Oracle (White Paper). 2019. Oracle Shariding Database 12c Release - Oracle Sharding. https://www.oracle.com/a/tech/docs/sharding-wp-12c.pdf.

[52] WikipediaMicrosoft(White Paper). 2021. Shard (database architecture). https://en.wikipedia.org/wiki/Shard_(database_architecture).

[53] Park, Jong-Hyeok and Oh, Gihwan and Lee, Sang-Won. 2017. SQL Statement Logging for Making SQLite Truly Lite. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 513–525.

[54] Rekha Pitchumani and Yang suk Kee. 2020. Hybrid Data Reliability for Emerging Key-Value Storage Devices. In *18th USENIX Conference on File and Storage Technologies (FAST '20)*.

[55] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. 2001. Active Disks for Large-Scale Data Processing. *Computer* 34, 6 (June 2001), 68–74.

[56] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. 2011. B+-Tree Index Optimization by Exploiting Internal Parallelism of Flash-Based Solid State Drives. *Proc. VLDB Endow.* 5, 4 (Dec. 2011), 286–297.

[57] Utku Sirin, Raja Appuswamy, and Anastasia Ailamaki. 2016. OLTP on a Server-Grade ARM: Power, Throughput and Latency Comparison. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN '16)*. Association for Computing Machinery, New York, NY, USA, Article 10.

[58] SQLite 2017. Well-Known Users of SQLite. http://www.sqlite.org/famous.html.

[59] SQLite 2020. Dynamic Memory Allocation in SQLite. https://www.sqlite.org/malloc.html.

[60] SQLite 2020. The SQLite OS Interface or "VFS". https://sqlite.org/vfs.html.

[61] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. 2017. Crail: A High-Performance I/O Architecture for Distributed Data Processing. *IEEE Data Engineering Bulletin* 40, 1 (2017), 38–49.

[62] Steven Swanson and Adrian M. Caulfield. 2013. *Refactor, reduce, recycle:Restructiong the I/O Stack for the Future of Storage*. Vol. 46. 52–59 pages.

[63] Alexander S. Szalay and Jose A. Blakeley. 2009. Gray's Laws: Database-centric Computing in Science. In *The Fourth Paradigm: Data-Intensive Scientific Discovery*, Tony Hey, Stewart Tansley, and Kristin Tolle (Eds.). Microsoft Research.

[64] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie S. Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. 2018. FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*.

[65] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter J. Desnoyers, and Yan Solihin. 2013. Active flash: Towards energy efficient, in-situ data analytics on extreme scale machines. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. USENIX Association, USA, 119–132.

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

[67] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Building Blocks for Persistent Memory. *VLDB Journal* 29, 6 (2020).

[68] 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. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1041–1052.

[69] Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin A. McGrail, Peng Wang, Diaohan Luo, Jun Yuan, Jianmin Wang, and Jiaguang Sun. 2020. Apache IoTDB: Time-Series Database for Internet of Things. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 2901–2904.

[70] Ronald Weiss. 2012. *Oracle White paper: A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server.* Technical Report. Oracle corp.

[71] Marianne Winslett. 2003. Jim Gray Speaks Out. *SIGMOD Rec.* 32, 1 (March 2003), 53–61.

[72] Jisoo Yang, Dave B. Minturn, and Frank Hady. 2012. When Poll Is Better than Interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX Association, USA, 3.

[73] Jinsoo Yoo, Joontaek Oh, Seongjin Lee, Youjip Won, Jin-Yong Ha, Jongsung Lee, and Junseok Shim. 2018. OrcFS: Orchestrated File System for Flash Storage. *ACM Transactions on Storage* 14, 2, Article 17 (April 2018), 26 pages.

[74] Chen Yu. 2018. Horizontally scaling a MySQL database backend with Cloud SQL and ProxySQL. https://cloud.google.com/community/tutorials/horizontally-scale-mysql-database-backend-with-google-cloud-sql-and-proxysql.

[75] X. Yu, M. Youill, M. Woicik, A. Ghanem, M. Serafini, A. Aboulnaga, and M. Stonebraker. 2020. PushdownDB: Accelerating a DBMS Using S3 Computation. In *IEEE 36th International Conference on Data Engineering (ICDE '20)*. 1802–1805.

[76] Matei Zaharia. 2019. Lessons from Large-Scale Software as a Service at Databricks. In *Proceedings of the ACM Symposium on Cloud Computing (SocC '19)*. Association for Computing Machinery, New York, NY, USA, 101.

[77] Lei Zeng. 2020. Sharding with Amazon Relational Database Service. https://aws.amazon.com/ko/blogs/database/sharding-with-amazon-relational-database-service/.

[78] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Understanding the Effect of Data Center Resource Disaggregation on Production DBMSs. *Proceedings of VLDB Endowment* 13, 9 (2020).