Datenbank-Spektrum (2025) 25:39–50
https://doi.org/10.1007/s13222-025-00495-8

SCHWERPUNKTBEITRAG

Towards High-performance and Trusted Cloud DBMSs

Adrian Lutsch[1] · Muhammad El-Hindi[1] · Zsolt István[1] · Carsten Binnig[1,2]

Received: 15 October 2024 / Accepted: 20 January 2025 / Published online: 7 March 2025

Abstract
Cloud Database Management Systems (DBMSs), such as cloud-native analytical or serverless DBs, are experiencing rapid growth in adoption due to their flexibility and scalability. However, recent incidents with cloud providers show that the traditional model of a trusted provider/admin no longer applies to protect the customers' data. One promising solution that can prevent a sole reliance on cloud and database service providers are trusted execution environments (TEEs). While past TEEs had many limitations and caused high performance overheads, recent work shows that the support of TEEs like Intel SGX for DBMS workloads improved significantly. Thus, it is time to actively integrate TEE technologies into cloud DBMSs to achieve better security that does not rely on the cloud provider. In this paper, we discuss directions for how recent TEEs can be used to build efficient and secure databases. We summarize the recent results on Intel SGX's performance for DBMS workloads and lay out the remaining research challenges that must be addressed to achieve optimal performance and thus minimize the performance cost for additional security.

Keywords Databases · Security · Trusted Execution Environments · Intel SGX

1 Introduction

Trust Model of Cloud Databases Cloud Database Management Systems (DBMSs), such as cloud-native analytical or serverless Databases (DB)s, are experiencing rapid growth in adoption due to their flexibility and scalability [1]. This new class of DBMSs places the responsibility for data security on cloud providers, fundamentally altering traditional trust models where infrastructure providers and company administrators were considered trusted entities. Incidents like Microsoft's loss of an Azure master key [2] and regulations such as the CLOUD Act [3] forcing providers to grant government access to customer data have shown that the traditional model of a trusted provider/admin is no longer applicable. To address these trust challenges, new approaches to privacy and confidentiality for cloud DBMSs are required.

✉ Adrian Lutsch
adrian.lutsch@cs.tu-darmstadt.de

1 Systems Group, Technical University of Darmstadt, Darmstadt, Germany

2 DFKI, German Research Center for Artificial Intelligence, Darmstadt, Germany

Trusted Execution Environments One promising security solution that can prevent a sole reliance on cloud and database service providers are Trusted Execution Environments (TEEs). They are meant to create isolated processing environments where data can be processed securely without access by third parties. TEEs generally aim to provide data confidentiality and integrity, code and execution integrity, and attestation. Attestation is the act of proving the identity and integrity of the TEE and the code within it, also referred to as Trusted Computing Base (TCB). Compared to alternative approaches, such as secure multiparty computation and homomorphic encryption, TEEs can provide these security guarantees while achieving the same or nearly the same performance as traditional processing.

Intel SGXv1 and SGXv2 The first commercially available TEE built into general-purpose CPUs was Intel's Software Guard Extensions (SGX). It protects a process against other processes, the Operating System (OS), the Hypervisor, and hardware attacks [4]. Thus, it can provide isolation even in cloud settings or against a malicious OS. However, the first version of SGX was created with client applications and not with server-grade DBMSs in mind. It supported only 128 MB of secure enclave memory per system, limiting the in-memory buffer sizes and thus leading to significant performance limitations even for moderately sized databases.

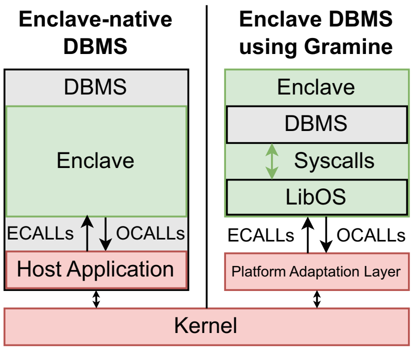

Fig. 1 Secure DBMS architecture options. Database engineers can either split their DBMS into a host application and an enclave, defining ECALLs and OCALLs required for communication (left), or they can rely on a libOS like Gramine to emulate the kernel inside the enclave and implement the enclave interface (right)

This changed with the second, server-grade generation, which we call SGXv2. It is available on server-grade Xeon processors and supports up to 512 GB of secure memory per socket [5]. However, the high isolation guarantees still introduce two significant hurdles to adoption: (1) reduced performance due to security mechanisms, such as context switches and memory encryption, (2) necessary architectural changes to split an application into a trusted enclave and an untrusted host process.

Implementation Choices To secure DBMSs with Intel SGX, there are two fundamental architecture options depicted in Fig. 1. Developers can split their database into trusted and untrusted parts and implement the interface between them, or they can leverage an adaptation layer like the Library Operating System (libOS) Gramine [6]. The first option enables fine-grained architecture control and performance optimizations. However, it requires major rewrites of existing DBMSs because direct interaction with the untrusted OS through syscalls is not allowed inside SGX enclaves, making it necessary to handle operations like storage access and networking outside the TEE. The second option lowers the engineering overhead at the cost of a larger trusted computing base and a reduced optimization potential by making the enclave environment transparent to the DBMS. The reduced optimization potential raises the question of whether a generic solution, such as the libOS, achieves acceptable trade-offs in terms of performance.

Contributions This paper compiles the current knowledge on building high-performance DBMSs along these two architectural choices and presents interesting findings for both directions. Furthermore, this paper outlines directions for future work to build high-performance enclave DBMSs. After introducing necessary background and summarizing related work in Sect. 2 and 3, our main contributions are:

1. We summarize the most important results of our two recent papers on benchmarking Intel SGXv2 for DBMSs [7, 8] and discuss strategies to reduce data processing overhead in SGX enclaves, such as low-level code changes to circumvent the overhead caused by side channel mitigations. (Sect. 4)
2. We analyze the overhead of running the in-memory DBMS Hyrise in an SGX enclave using the libOS Gramine, and find that optimizations are required for both the DBMS and the libOS to achieve optimal performance. (Sect. 5)
3. From the previous results, we derive open research and engineering challenges for high-performance trusted DBMSs in Intel SGX, such as developing an efficient query execution engine for SGXv2 and co-optimizing libOSes and DBMSs. (Sect. 6)

2 Background

Intel SGX was initially developed for client hardware and has seen two major changes since its inception. This section reviews the basics of Intel's SGX technology and discusses the most important changes introduced with the SGX2 instruction set extension [9] and with server-grade SGX [5]. Additionally, we review the basic concepts of the Gramine libOS.

SGX Basics Intel SGX protects the integrity and confidentiality of user data and code by shielding it from the OS, the hypervisor, and attacks on the memory bus. These guarantees are achieved by 3 main components: (1) SGX creates a protected memory region in RAM, called Processor Reserved Memory (PRM), which can only be accessed via special CPU instructions [4, 10] and is not accessible to the operating system and its administrators. Inside this protected memory region, SGX maintains the Enclave Page Cache (EPC) which stores the trusted code and data of enclaves within encrypted 4 kB memory pages. These pages are only decrypted when loaded into the CPU cache for processing [4, 10]. Initially, the PRM size was limited to 128 MB. (2) The untrusted OS can only create and destroy enclaves and manage EPC by evicting encrypted enclave pages via a special CPU instruction. It cannot otherwise access enclave pages after enclave creation. The correct creation of an enclave can be checked using remote attestation [4, 10]. (3) To execute enclave code and access its data, a thread must call special CPU instructions that cause a context switch to enclave mode executing (secure) user code. SGX guarantees that only code from within the same enclave has access to the EPC pages of that enclave by adding security checks to the address translation. If the CPU is not in enclave mode, it will not translate enclave addresses, which protects the sensitive user code & data within the enclave. When the CPU leaves enclave mode, it securely stores the enclave state in EPC and clears all security-sensitive caches, especially the TLB [4, 10].

Dynamic Enclaves Extension Originally, the memory size of SGX enclaves, the page access permissions, and the maximum number of available thread contexts was fixed at compile time. The SGX2 instruction set extension introduced instructions that allow changing these characteristics at runtime. This enables more dynamic enclaves that adapt to the workload and use only as many resources as they need [9].

SGXv2—Server-grade Enclaves The capacity limitations of the PRM and the high cost of secure paging made Intel SGXv1 impractical for data-intensive applications such as DBMSs [7, 11, 12]. With the Ice Lake CPU architecture, Intel introduced server-grade SGX (SGXv2), which increases the supported PRM size to 512 GB per socket. This change allows DBMSs to hold large data sets fully in the EPC and avoids expensive enclave paging. This was achieved by replacing the previously used SGX Memory Encryption Engine with Total Memory Encryption—Multi-Key (TME-MK) [5]. In addition to changing the encryption hardware, SGXv2 replaced the integrity protection and freshness tree and the associated checks when loading encrypted enclave data into the cache with a special bit in ECC RAM [5]. Finally, enclaves can now scale across multiple sockets, increasing the number of CPU cores and the amount of memory available even further [5]. To access EPC pages on a remote socket, SGXv2 introduces the UPI Crypto Engine (UCE) that encrypts data before transferring it over Ultra Path Interconnect (UPI) [5]. In summary, these changes remove the most severe bottlenecks for data processing in enclaves and justify a reevaluation of the technology for data processing.

Developing SGX Enclaves When developing an application with Intel SGX, developers must split it into a trusted part—the enclave—and an untrusted part—the host application (cf. Fig. 1, left part). The enclave is responsible for executing security-sensitive operations over private data. The host application is required to communicate with the OS and other applications since no system calls to the (untrusted) kernel are possible inside the enclave. The interface between both parts is implemented as RPCs with so-called enclave calls (ECALLs) and outside calls (OCALLs).

Gramine Library Operating System Gramine, previously Graphene-SGX [13], enables running native Linux applications inside SGX enclaves without changes by emulating the Linux ABI. As of version 1.7, Gramine supports approximately 170 of ~360 Linux system calls, parts of the /dev, /proc, and /sys pseudo file systems, and, overall, most features required for running single applications in enclaves [14]. System calls are either routed to the libOS as library calls by using a patched libc implementation or by trapping errors inside the enclave [14]. Some system calls are implemented purely inside the libOS and thus do not require OCALLs to the underlying OS [13]. One example is the mmap call for allocating memory in fixed-size enclaves. Other system calls like file and socket access require cooperation with the untrusted OS. To implement them, Gramine forwards accesses outside of the enclave using OCALLs. If possible, the system implements additional security checks for these forwarded system calls [13]. For example, the libOS can transparently encrypt files and protect their integrity [15].

3 State of the Art

This section summarizes the state of the art in SGX-native databases for SGXv1, which came with significant performance overheads. The following sections discuss the design of high-performance and secure DBMSs for SGXv2. There are three areas of important previous work in the context of SGXv1: Secure DBMS using SGXv1, investigations of query execution performance in SGXv1, and work on running full DBMSs in SGX enclaves using adaptation layers.

DBMS Architectures for SGXv1 Several proposals have been made for secure DBMSs using the first generation of SGX. Because of the limited enclave capacity and the need to reduce the TCB, the systems differ in the components they secure within the enclave. The key value stores SPEICHER [16], ShieldStore [17], and the storage engine Enclage [18] aim to run all components entirely within the protected enclave. In contrast, the authors of StealthDB [19], Azure SQL Database Always Encrypted [20], and EncDBDB [21] minimize the part of the database that is protected inside an enclave by only evaluating simple query predicates over encrypted columns inside the SGX enclave. EnclaveDB [11] executes queries inside the enclave but keeps query parsing and optimization as well as client connections outside. Most of the proposed systems concentrate on OLTP scenarios [11, 16–20, 22] and the latency effects of enclave transitions, integrity protection, and encryption. We include a discussion of their main insights in Sect. 4.2. Only EncDBDB [21] investigated an OLAP scenario and the performance effect of in-enclave execution on range queries over dictionary-encoded data [21].

Query Execution Performance Maliszewski et al. [12] investigate the performance of join algorithms in SGXv1 enclaves. They come to the conclusion that partitioned joins like the radix join perform best given the small EPC sizes of SGXv1 enclaves. Still, the random access required for radix partitioning of data sizes larger than the EPC reduces performance by orders of magnitude [12]. Thus, they developed CrkJoin, which uses partitioning with linear accesses to reduce the EPC paging costs [23]. However, in SGXv2 enclaves with enough EPC, their linear partitioning cannot compete with classical radix partitioning [8].

Running Full DBMSs in SGX Enclaves The authors of SCONE, an adaptation layer to execute containerized applications in SGX enclaves [24], investigated the performance of running memcached and Redis inside an enclave using their framework. Redis achieved up to 61% in-enclave performance, and memcached achieved higher performance on top of SCONE because of a more efficient network encryption implementation. The authors also report that their asynchronous enclave calls improve the performance of both key-value stores by reducing the number of enclave transitions [24]. The same insight was reported by Orenbach et al. [25] for their extension of Gramine [25].

Concurrently with our work, Battiston et al. [26] did initial experiments on running DuckDB inside an SGXv2 enclave using Gramine [26]. Their investigation concentrated on storage encryption performance and memory management inside Gramine. They report a 2× overhead due to query execution overheads inside the enclave [26] and observe a bad performance when using jemalloc as their memory allocator. In this paper, we conduct a more in-depth investigation and explain the sources of query execution overheads. We chose the in-memory DBMS Hyrise for our experiments because pure in-memory processing isolates the performance effects of query processing and data access. Additionally, we investigate the memory management issues and the interaction between jemalloc and Gramine in more detail to determine the impact of different SGX aspects on query execution performance.

4 Enclave-Native DBMSs

This section compiles the important performance effects of Intel SGXv2 enclaves for designing high-performance and secure DBMSs based on [7, 8]. We split our analysis into OLAP and OLTP workloads because the different workloads stress different performance issues of Intel SGXv2, such as deactivated write position speculation in SGX (OLAP) and increased disk access latencies due to costly enclave transitions (OLTP). Based on these findings, we discuss how the design of core algorithms for executing these workloads needs to change.

Setup For all experiments, we used a server containing two Intel Ice Lake Xeon Gold 6326 CPUs with 256 GB DDR4 3200 ECC memory and 64 GB PRM per socket. The CPU has 16 cores with 48 kB/1.25 MB L1d/L2 cache per core and 24 MB L3 cache per socket. The server runs Ubuntu 22.04 with kernel 6.8, SGX SDK version 2.24, and we compile all software with GCC 12.3 using the optimization flags -O3 -march=native. The source code repositories containing all experiment configurations and execution scripts are available at [27] and [28].

4.1 Analyzing OLAP

The performance of OLAP databases, such as cloud data warehouses, depends strongly on the parallel and efficient execution of analytical queries that aggregate large volumes of data. Thus, efficient scans and joins that employ intra-operator parallelism are paramount. In the following, we review the performance of the scan and join operators and how they are affected by the memory encryption and side channel mitigation inside SGXv2 enclaves based on [8]. We also discuss aggregation operator performance, as well as memory allocation and Non-Uniform Memory Access (NUMA) in enclaves, which are known as important performance factors outside enclaves.

Scans in SGXv2 With the redesigned memory encryption and the high amount of secure memory available in SGXv2, it was unclear if memory encryption could satisfy the throughput needs of fast multi-threaded column scan algorithms employed in modern DBMSs, such as the SIMD scan [29]. Thus, we investigated the performance of running such algorithms inside SGXv2 enclaves and compared it to running the same algorithm over the same data outside enclaves in [8]. The results are summarized in Fig. 2.

The experiment shows that scans over encrypted enclave memory can achieve nearly the same throughput as scans over non-encrypted memory, even if all cores execute the scan. Thus, we conclude that memory encryption and de-

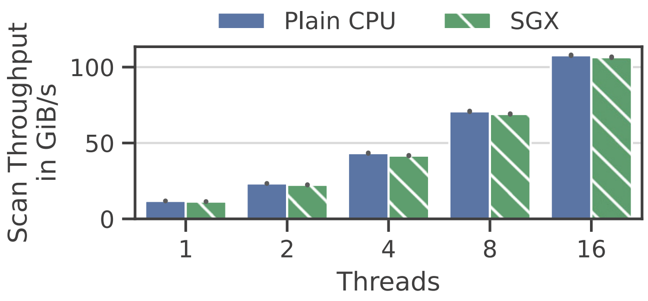

Fig. 2 Read throughput of a multi-threaded columnar range scan using SIMD instructions over 16 GB of byte-sized input values returning a boolean bit vector. The read throughput of running the scan inside the enclave reading encrypted enclave data is only 3% slower than outside the enclave and reaches the memory throughput limit. Source: [8]

cryption happen at a line rate of the memory interface. More experiments in [8] showed that this does not change with the selectivity and the write rate of the scan (for writing matching tuple indexes).

Joins in SGXv2 Joins are critical for the performance of OLAP query engines. Thus, we investigated their performance in SGXv2 enclaves and compared it to the performance of the same join implementations on the plain CPU without SGX in [8]. Our experiments revealed two main sources of overhead that mainly influence hash-based joins.

Cache Misses Expensive last-level cache misses due to encryption and TLB integrity checks are a performance problem already known in SGXv1 [12]. Micro-benchmarks in [7] and join performance benchmarks in [8] revealed that this performance issue persists with SGXv2. Especially non-partitioned hash joins with hash tables larger than the last-level cache incur last-level cache misses and thus random main memory accesses with high frequency. Thus, increased memory access latencies in SGX significantly reduce the in-enclave performance compared to the plain CPU performance [8]. To show this effect, we ran a parallel hash join with 16 threads over 3 input table sizes. The hash table is built on the smaller input table. We compare the join throughput inside an SGX enclave with the join throughput on the plain CPU. The results are depicted on the left side of Fig. 3. For the small table size, the hash join is nearly as fast inside the enclave as outside because the whole hash table fits into the CPU cache. For the larger-than-cache table sizes, the relative in-enclave performance is significantly reduced because the hash table does not fit into the cache, and resolving cache misses is more expensive in SGX enclaves.

SSB Side-channel Mitigation In addition to slower cache misses, the in-depth join performance investigation in [8] revealed that the mitigation for the Speculative Store Bypass (SSB) side-channel attack, which is enforced inside SGX enclaves and optional outside [30], can significantly reduce the performance of random access algorithms. All algorithms that determine store positions from input data, such as histogram calculation and hash table creation, are affected.

To avoid this issue, loads and stores in an algorithm must be batched or vectorized, reducing the algorithm's reliance on speculation. A guide on transforming code to achieve the split is available in [8]. The right side of Fig. 3 summarizes the performance results of the join optimizations introduced in [8]. The workload consisted of a radix partitioning hash join using a 100 MB and a 400 MB table with a 4-byte key and a 4-byte payload each. The join implementation uses two partitioning phases and no software-managed buffers with non-temporal stores. Due to the partitioning, this join is cache-friendly. Plain CPU is the baseline performance outside of the enclave without the mitigation. Activating the SSB mitigation (+Mit) decreases join performance by up to 50%. The optimized join implementation (+Opt) is faster under the mitigation than the non-optimized version without mitigation. The SGX setting shows the performance of executing the join inside the enclave without optimization. Similarly to outside the enclave, performance improves significantly when using the optimized join implementation inside the SGX enclave (SGX+Opt.). Using the optimization, the in-enclave join throughput reaches more than 90% of the optimized plain CPU baseline.

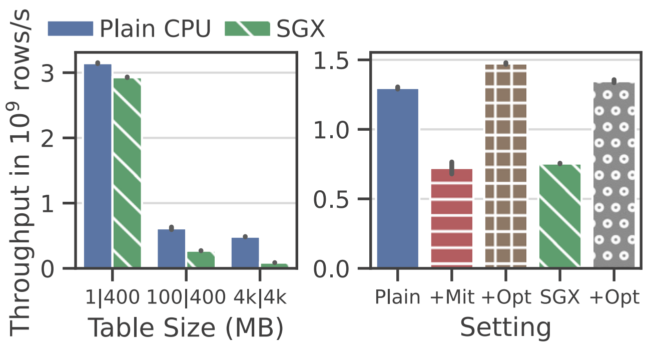

Fig. 3 Left: Hash join throughput for increasing input table sizes. More expensive cache misses cause higher throughput reduction inside the enclave. Right: The SSB mitigation reduces radix join performance and the optimization successfully counteracts the mitigation slowdown outside and inside SGX. Data source: [8]

Aggregation in SGXv2 Aggregation is the third important operator for the performance of OLAP DBMSs. Given the previous results, it can be expected that hash aggregations behave similarly to the hash join discussed above. If the hash table fits into the CPU cache, the main performance bottleneck is the SSB mitigation, and vectorization of memory reads and writes can be applied to close the performance gap to normal execution. If the hash table does not fit into the cache, the higher latency of random main memory accesses will likely decrease performance further. Hyrise TPC-H query performance insights support these assumptions (cf. Sect. 5).

Memory Allocation Memory allocation in enclaves differs from the allocation for normal processes. Initially, Intel SGX enclaves had a fixed size, and all memory was pre-allocated during enclave creation. Thus, enclave creation time is slow, but every call to malloc inside it is fast because it only returns already allocated and pre-faulted enclave memory without OS interaction. With the SGX2 instruction set and the necessary kernel and SDK support, it became possible to dynamically increase and decrease an enclave's size [9]. This is called Enclave Dynamic Memory Management (EDMM) and can be managed transparently by the SGX SDK malloc implementation or explicitly by the enclave developer. However, experiments in [8] showed that EDMM significantly decreases the performance of query execution if memory allocation during query execution requires increasing the enclave size because enclave malloc requires multiple expensive instructions [9]. Thus, memory allocation inside enclaves can be both slower (if EDMM is required) and faster (if pages are already allocated to the enclave) than in the native environment.

NUMA The availability of Intel SGX on multi-socket server hardware and the option to use the PRM of multiple sockets for one enclave are new features of SGXv2. Thus, enclaves can now experience Non-Uniform Memory Access (NUMA) which increases the latency of accessing memory connected to another socket. Experiments in [7, 8] showed that cross-NUMA memory accesses are more expensive in SGX enclaves because of the Ultra Path Interconnect (UPI) encryption: Latency of random remote memory accesses is increased by 46% in SGX enclaves [7], column scans on a remote NUMA region are 4 to 23% slower [8], and joins executed on cores of two sockets while data is only stored in the memory of one socket can be slower than only using the cores of the socket where the data is stored [8]. Although an enclave can use memory on multiple sockets, the location of memory allocations is transparent to enclaves, and no APIs for memory allocation in a specific NUMA region are available within enclaves. This is because allocating physical memory to enclaves is a task of the untrusted OS. This prevents optimizations for NUMA in SGX without trusting the OS.

4.2 Analyzing OLTP

OLTP databases are characterized by many short-running lookup, update, and insert transactions. Thus, their performance depends on low-latency disk and network access. To optimize query performance, OLTP databases use indexes like B-Trees. In the following, we first review important architecture effects directly impacting OLTP performance by increasing transaction latency and then summarize insights on B-Tree performance in an SGXv2 enclave.

Enclave Transitions To execute enclave code, the CPU must switch from normal mode to enclave mode. When returning from the enclave or when the enclave must access operating system services or hardware, the CPU mode must switch back. Thus, a write transaction in an OLTP database requires at least 3 transitions: one to copy the transaction into the enclave, one to write the updates to disk, and one to return an acknowledgment to the client via the network only accessible outside the enclave. These enclave transitions are expensive because they must enforce the SGX security guarantees. This requires flushing caches, especially the TLB. Additionally, copying function call parameters can prolong effective transition time. This problem is well-known from SGXv1 [24, 25, 31, 32] and there are 3 main solutions for the issue: (1) Reduce the number of enclave transitions, for example, by batching or by reducing reliance on the OS. (2) Reduce the size of copied data during enclave transitions, for example, by transferring only pointers to normal system memory [31]. (3) Use so-called switchless enclave calls [24, 25, 32], that replace enclave transitions with queues containing calls and data that are polled by worker threads inside and outside of the enclave. All of these approaches influence the latency of enclave calls, but especially the second and third approaches also affect security guarantees and place additional responsibility onto the enclave developers. Additionally, the queueing approach of switchless enclave calls requires developers to define how many threads run inside the enclave and outside to service the queues.

Synchronization Database systems must latch internal data structures to prevent corruption while concurrently processing transactions. In normal DBMSs, this kind of thread synchronization is often supported by the OS to optimize CPU utilization and prevent scheduling issues. Waiting threads are sent to sleep via a system call and can be woken up by other threads, for example, when they unlock a mutex. In SGX enclaves, this design requires threads to leave the enclave before executing the required syscalls. The issue with this approach is that enclave transitions are up to two orders of magnitude more expensive than syscalls [31]. In the case of contended latches, as they can occur in query execution algorithms or when latching in-memory data structures, the cost of enclave transitions can thus dominate the whole algorithm runtime [8, 12]. To solve this problem, enclave developers should prefer lock-free designs, atomics, or spinlocks over the SGX SDK mutex, which is implemented with the design described above, for short critical sections [8, 23]. For critical sections that involve enclave transitions and are thus longer than an enclave transition by design, the SGX SDK mutex might still be a good fit.

Storage Since OS and storage HW are untrusted in the SGX security model, storing databases on disks requires encryption and integrity protection. In [7], we investigated the performance of the two APIs in the SGX SDK that can be used for data sealing—encryption with keys that are either bound to the enclave identity or the enclave signer identity. The first option is the sgx_seal_data function. It encrypts and protects the integrity of a buffer with AES GCM and writes the output to another buffer. The enclave developer is then responsible for copying the encrypted data out of the enclave with an OCALL and saving it to a file. The integrity protection of this encryption method only applies to the data in the buffer. Thus, a file consisting of multiple encrypted blocks is not protected against switching blocks out or selectively reverting information in some blocks.

The second API for file encryption in the SGX SDK is the "SGX Protected File System Library". It provides enclave developers with a similar file API to the standard libc file API, including functions like sgx_fwrite and sgx_fread. In contrast to the sgx_seal_data function, sgx_fwrite encrypts the file in blocks and protects the whole file integrity using a Merkel hash tree. This protects the file against block swapping and selective rollback attacks. Furthermore, the Protected File System Library transparently encrypts and stores blocks in files. The enclave developer is not involved in copying the encrypted data out of the enclave and to the file. Finally, the library enables file recovery after enclave crashes by writing a recovery file before writing to the actual file.

I/O Experiment To investigate the performance costs associated with the two file I/O approaches presented above, we determined the number of CPU cycles per byte stored on disk when using the secure file approaches and compared the result with the fwrite and fflush functions in the standard C library [7]. The results are depicted in Fig. 4. It shows that all methods of writing files improve in terms of cycles per byte when batching large writes. Furthermore, it shows that the additional protections of the SGX Protected File System Library have significant costs compared to the sgx_seal_data+OCALL approach. The overhead is especially large for small writes of 2 bytes, where sgx_fwrite+sgx_fflush is 20 times slower than sgx_seal_data+OCALL. This can be attributed to the large overhead of writing a full 4 kB page plus the metadata page first to a recovery file and then to the protected file. Although the overhead decreases with write size, even for large batches of 8 MB per write, sgx_fwrite+sgx_fflush is still 4 times slower than untrusted fwrite+fflush.

These results exemplify that different security features, such as swap protection and crash resistance, can incur high costs for enclave-native DBMSs. Giving up on some of these guarantees or implementing them by other means inside the DBMS (e.g., page IDs and checksums against swapping and corruption of pages) can speed up transaction processing by orders of magnitude.

B-Trees in SGXv2 The performance of indexes like B-Trees is especially important for OLTP workloads with high insert, update, and lookup frequencies. Thus, [7] investigated the performance of in-memory B-Tree inserts, updates, and lookups in SGXv2 enclaves relative to the plain CPU. Since B-Trees combine linear access patterns inside of nodes with random lookups of node pointers, they combine patterns with low and high overhead in one data structure and workload (cf. Sect. 4.1). To determine the slowdown caused by using B-Trees in SGXv2 enclaves, we ran various YCSB-like workloads with different read and write rates using our implementation of an in-memory B-Tree with 4 kB nodes. We measured the throughput of the workload running inside the enclave relative to the same workload running outside the enclave (right, green bars). For this paper, we additionally ran the workloads outside the enclave with the SSB mitigation (cf. Sect. 4.1) enabled (left, red bars). The results are depicted in Fig. 5. They show that the mitigation does not negatively affect the B-tree performance and that relative in-enclave performance decreases with higher read rates. We attribute this to the fact that a considerable part of the runtime for write-heavy workloads is spent on moving data inside nodes to make space for the new keys and then copying the new value. This makes the write-heavy workload up to 2.5× slower in terms of operations per second than the pure read workload. Since copy performance inside SGX enclaves is the same as outside and dominates the

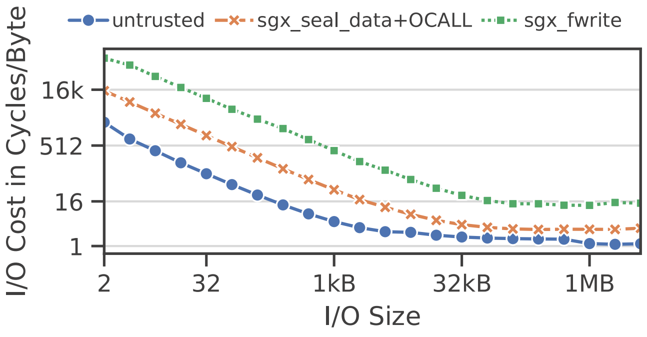

Fig. 4 I/O cost in cycles/byte of the two trusted I/O variants (sgx_fwrite and sgx_seal_data+OCALL) vs. untrusted I/O (fwrite). Source: [7]

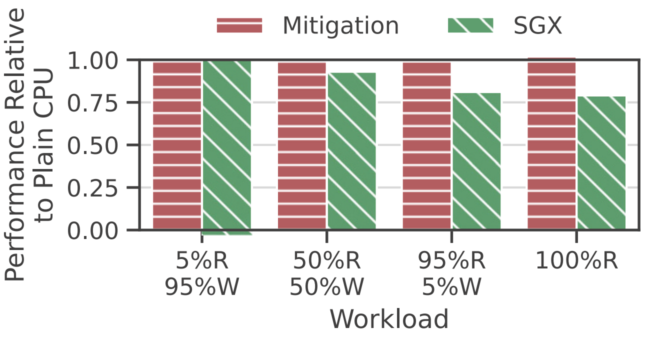

Fig. 5 Throughput of four B-Tree read & update workloads with the SSB mitigation enabled and inside an SGXv2 enclave, relative to the plain CPU throughput. The mitigation does not influence the performance. Costly cache misses cause up to 25% slowdown in SGX. Source: [7]

runtime, it hides the random access overheads. In contrast, the performance of the read-heavy workloads is limited by random memory access for tree traversal, revealing the random access overheads inside the enclave.

Summary Previous investigations revealed multiple effects that influence the performance of DBMSs in SGXv2. For OLAP DBMSs, especially expensive cache misses and the SSB mitigation require changes to query execution algorithms. Enclave dynamic memory management must be managed explicitly if required, and optimizations for NUMA require trust in the OS. For OLTP DBMSs, slow enclave transitions can become the bottleneck of query processing. To achieve optimal performance under this limitation, implementations of thread synchronization, storage, and networking must be adapted. The performance of an in-memory B-Tree index inside an SGXv2 enclave is reduced by expensive cache misses and random memory access, mostly for fast, read-heavy workloads.

5 Full DBMSs in SGXv2

As introduced previously, an enclave-native DBMS requires major rewrites of core parts of existing DBMSs because certain operations like storage access and network need to be handled outside the TEE in the untrusted host application. Another option is to use a library OS and run the full DBMS unmodified in the enclave. For this option, the question remains how the library OS abstraction influences the overall DBMS performance. To answer this question, we ran the Hyrise DBMS inside an SGXv2 enclave using the Gramine Library Operating System [6]. The code and results can be found on Github [33].

Hyrise DBMS We chose Hyrise [34] as an open source columnar in-memory DBMS. We used the included TPC-H benchmark runner to determine query latency and throughput inside the enclave and outside and compare these settings. Hyrise supports inter-query parallelism, where different queries can be executed concurrently. Thus, adding more cores and concurrent clients increases the throughput in queries per second but does not reduce query latency. To understand the performance overheads, we thus used throughput in queries per second as the performance measure for our experiments.

Out-of-the-box Performance In the first experiment, we compiled both Gramine and Hyrise from their source and executed the TPC-H benchmark using 8 cores and 8 clients scheduling queries. We chose this 50:50 core split to prevent NUMA effects and have sufficient free cores on the same NUMA node for managing libOS threads. Each query was repeatedly executed for 10 s. Each experiment was repeated 10 times and we report the arithmetic mean. Since the benchmark failed with out-of-memory errors at scale factor 10, we reduced the scale factor to 5. The results are depicted in Fig. 6. They show that, on average, the execution inside the enclave reaches only 4% of the throughput outside of the enclave without Gramine.

Reasons for Performance Overhead We investigated the issue using the perf record functionality integrated into debug builds of Gramine. This feature stores the current call stack on every asynchronous enclave exit, effectively sampling which functions are executed inside the enclave. Figure 7 summarizes the runtime percentage estimations from the sampling. As shown, 60 to 70% of the application runtime is spent in the libOS, of which most of the time is spent in memory management functions (not shown in the figure). This hints at a highly problematic interaction between Hyrise and Gramine.

Analyzing the Root Cause The source of the issue is a very high amount of mmap and munmap calls issued by the jemalloc [35] allocator used by Hyrise. The high number of (de-)allocations trigger a performance bug in the Gramine memory management implementation. We identified and fixed this bug. This significantly improves the performance

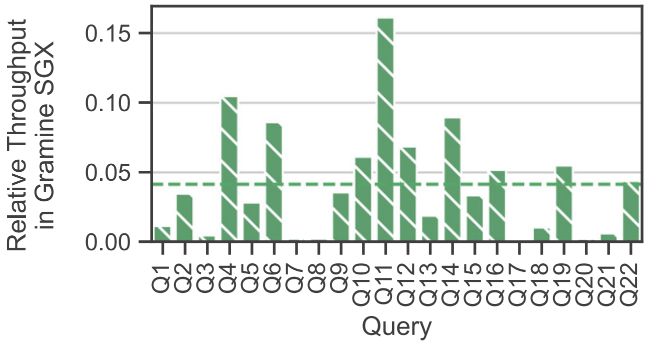

Fig. 6 Relative throughput of running TPC-H queries with Hyrise+ Gramine inside an SGX enclave compared to native execution without Gramine. Queries are repeatedly issued for 10 s by 8 clients in parallel. TPC-H scale factor 5

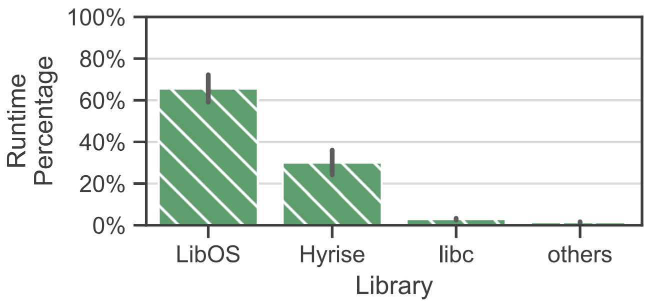

Fig. 7 Part of benchmark runtime spent per library. 97% of the time spent in the libOS is spent in the memory management subsystem

at TPC-H scale factor 5. However, it is still not possible to run higher scale factors.

As such, we investigated the issue further and found that larger TPC-H scale factors can be executed inside Gramine without running out of memory when using the optimized glibc malloc provided by Gramine instead of jemalloc. This memory allocator uses fewer mmap calls and thus causes less memory fragmentation on the libOS side. Additionally, it replaces system calls with direct and faster library calls to the libOS. Thus, in addition to fixing the performance bug in Gramine memory management, we suggest using the optimized glibc malloc instead of the Hyrise default jemalloc. We see this as a first step of co-optimization for both systems.

Performance with Fixes With the optimizations applied, the TPC-H benchmark can be executed at scale factor 10 inside the enclave and achieves a relative throughput of 56% on average over the 22 queries when compared to Hyrise outside the enclave (cf. Fig. 8, green bars). The percentage of runtime of Gramine functionality during the benchmark drops from 60 to 70% previously (cf. Fig. 7) to between 10 and 15%.

To find the source of the remaining slowdown, we ran the TPC-H benchmark outside the TEE while enabling the mitigation for the SSB side-channel attack, which we previously identified as a major source of in-enclave overhead (cf. Sect. 4.1). As shown in Fig. 8 (red bars), the mitigation decreases the throughput by 33% on average. Thus, running Hyrise inside an enclave using Gramine (green bar) only causes an 11% performance drop when compared to the performance of running Hyrise outside the TEE with the mitigation enabled (red bar).

A remaining observation from Fig. 8 is that some queries have very low relative performance (e.g. Q1 and Q21), whereas others achieve nearly the same performance inside the enclave, such as Q11 and Q17. After investigating these queries on the operator level, we conclude that Q1 and Q21 are slowed down by large aggregations (Q1) and joins (Q21), which suffer from slow random main memory accesses and missing store position speculation. In contrast, Q11 and Q17 reduce input data before it is used in joins or aggregations, increasing cache-friendliness. Applying and evaluating our previously identified optimization for joins to further improve the Hyrise performance inside the enclave (cf. Sect. 4.1) is an interesting area of future work.

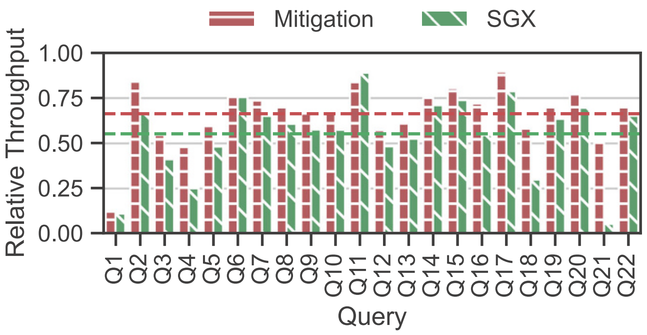

Fig. 8 Comparison of throughput relative to normal execution outside the enclave: Hyrise with mitigation and Hyrise in Gramine SGX with memory management fixes applied. TPC-H at SF 10

Summary Running Hyrise inside an SGXv2 enclave using Gramine is possible but has unusable performance out of the box. The main culprit is memory management inside the enclave, which can be improved by using the malloc functionality of the Gramine glibc and by fixing a bug in the libOS. With the fixes applied, the SSB mitigation reduces the performance of Hyrise even more than the libOS. Thus, improvements to the query execution operators are more pressing than improvements to other performance factors. First experiments in this work on applying SGX-specific optimizations (cf. Sect. 4.1) to the Hyrise query execution operators showed promising results, but we leave a comprehensive integration and evaluation of those optimizations for future work.

6 Research Directions

Related work and the recent results of SGX performance studies summarized previously already address several important questions when building high-performance and secure DBMSs. Still, some open questions for future research on SGXv2 for databases remain. We first summarize the open research questions for purpose-built enclave-native DBMSs and then outline research directions for database systems in library operating systems.

6.1 Enclave-native DBMSs

For enclave-native DBMSs in SGXv2, running as much of the DBMS inside the enclave as possible gives the best security guarantees and is now feasible due to the enlarged PRM. While related work focused on efficient and low-latency enclave transitions, networking, and storage (cf. Sect. 3), we see the open challenges in efficient query execution and usage of new SGX features like Dynamic Memory Management and NUMA.

Efficient Query Execution The results of the join operator benchmarks (Fig. 3) and running Hyrise under the effect of the SSB mitigation (Fig. 8) have shown that the performance gain of an optimization for the SSB mitigation can be large. Thus, one open area of future work is designing and implementing optimized versions of all necessary operators in a query engine. Random access operators like hash join, grouping, and intersection have the highest optimization potential. Experiments on sorting algorithms we conducted did not reveal overheads. This raises the question if efficient sort-based implementations for algorithms like aggregation [36, 37] and join [38] are generally a better choice in SGX enclaves because SGX hardware characteristics do not slow them down. Thus, this line of work can create operator implementations that combine principles of cache optimization with new optimizations for missing store position speculation.

Dynamic Memory Management Depending on the situation, DBMSs that can allocate and release memory dynamically are required. For SGX-based DBMSs, this requirement also applies to the EPC, and DBMSs should thus be able to enlarge and shrink their enclave dynamically. Although previous experiments have highlighted that dynamically enlarging an enclave during query execution increases query runtime by more than one order of magnitude [8], optimized strategies are still missing. These optimized strategies must explicitly manage enclave memory to prevent synchronous allocation and shrink the enclave if memory consumption decreases.

NUMA Database servers that scale beyond single-socket CPUs optimize for the characteristics of NUMA to achieve better performance. This is done by pinning threads to specific CPU cores and allocating buffers in specific NUMA regions. Both operations are not supported inside SGX enclaves. However, if enclave database developers trust the OS to allocate memory in the local NUMA region and pin threads correctly, enclave memory allocations in a specific NUMA region should be possible when using explicit EDMM. Future work in this area should investigate (1) if this approach works as expected, (2) if performance inside the enclave benefits to the same degree as outside, (3) if this approach has implications for data confidentiality, and, (4) if required, how confidentiality can be preserved.

6.2 LibOS-based DBMSs

Due to the reduced engineering overhead, running existing DBMSs on top of abstraction layers like Gramine remains a valid option. In this case, we see multiple avenues for optimizations in both the DBMS and the libOS in addition to optimizations for the query execution.

Memory Management Our experiments in Sect. 5 showed that memory management can greatly impact performance. On the DBMS side, choosing a memory allocator that works well with the libOS is crucial. But, as the performance issue we discovered in the Gramine memory management shows, libOSes can also be optimized for the memory allocation patterns of DBMSs. Another known problem in memory management is the requirement to zero the memory inside the libOS before it can be given to the application via mmap [26]. This is expensive compared to the copy-on-write tricks a real OS can apply and can cause high latencies. Thus, optimizations in the libOS or the DBMS could improve performance. Finally, a libOS in SGXv2 enclaves should enable the DBMS to optimize for NUMA and efficiently support EDMM to achieve optimal performance.

Storage and Networking Especially for OLAP DBMSs where storage and network latency are not the primary performance concerns, libOSes can reduce the engineering work required to create an enclave DBMS. Gramine allows applications inside the enclave to access the host file system through mounts. These mounts can be configured to automatically encrypt and decrypt files in a specific directory. This can reduce the effort required inside the DBMS, but the generic encryption scheme of the libOS is probably subpar for DBMS usage. Regarding networking, DBMSs can likely use the libOS functionality without changes, but the number of worker threads must be chosen carefully. Further investigations are required to determine the validity of these approaches.

Optimization Hurdles A downside of using libOSes for DBMSs is that the complexity and generality of a full library operating system can make it hard to find root causes and provide fixes as functionalities are now split between DBMS and libOS and problems occur due to non-optimal interactions of both. For example, optimizing the DBMS inside the enclave with specialized ECALLs and OCALLs or queuing techniques is impossible when a libOS manages the interface. Future research could thus investigate semi-transparent approaches where the application inside a libOS can incrementally replace the syscall interface with its own optimized enclave interface.

7 Conclusion

Previous research has shown that Intel SGXv2 can enable high-performance and trusted DBMSs in the cloud. We summarized recent work on query processing in SGXv1 and SGXv2, investigated the viability of using library operating systems for DBMSs in SGXv2, and laid out future research directions to achieve practical, secure DBMSs. We see several remaining research and engineering challenges, especially those connected to new hardware features, such as NUMA and dynamic enclave memory management. However, we believe that with the recent generation of SGX, secure and high-performance cloud DBMSs for OLAP and OLTP are within reach.

Funding This work was supported by SAP SE. Additional funding was provided by the German Federal Ministry for Economic Affairs and Climate Action under Grant Agreement Number 01MK21002K and by the National Research Center for Applied Cybersecurity ATHENE.

Author Contribution All authors contributed to the study conception and design. Material preparation, data collection and analysis, and writing of the first draft of the manuscript were performed by Adrian Lutsch. All authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding Open Access funding enabled and organized by Projekt DEAL.

Conflict of interest A. Lutsch, M. El-Hindi, Z. István and C. Binnig certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.

References

1. Pritchard S (2024) Trends in the cloud database market. https://www.computerweekly.com/feature/Trends-in-the-cloud-database-market. Accessed 2024-10-14
2. Cyber Safety Review Board (2024) Review of the summer 2023 Microsoft exchange online intrusion. https://www.cisa.gov/resources-tools/resources/CSRB-Review-Summer-2023-MEO-Intrusion
3. CLOUD Act. Wikipedia (2024). Accessed 2024-10-07
4. Costan V, Devadas S (2016) Intel SGX Explained. https://eprint.iacr.org/2016/086.pdf
5. Johnson S, Makaram R, Santoni A, Scarlata V (2021) Supporting Intel SGX on Multi-Socket Platforms. Intel Corporation. https://www.intel.com/content/www/us/en/content-details/843058/supporting-intel-sgx-on-multisocket-platforms.html. Accessed 2025-01-08
6. Gramine Project (2025) Gramine—a Library OS for Unmodified Applications. https://gramineproject.io/. Accessed 2025-01-08
7. El-Hindi M, Ziegler T, Heinrich M, Lutsch A, Zhao Z, Binnig C (2022) Benchmarking the second generation of intel SGX hardware. In: Data management on new hardware DaMoN'22. Association for Computing Machinery, New York, pp 1–8 https://doi.org/10.1145/3533737.3535098
8. Lutsch A, El-Hindi M, Heinrich M, Ritter D, István Z, Binnig C (2025) Benchmarking Analytical Query Processing in Intel SGXv2. In: Simitsis A, Kemme B, Queralt A, Romero O, Jovanovic P (eds) Proceedings 28th International Conference on Extending Database Technology EDBT 2025, Barcelona, March 25–28, 2025 OpenProceedings.org, Konstanz, pp 516–528 https://doi.org/10.48786/EDBT.2025.41
9. McKeen F, Alexandrovich I, Anati I, Caspi D, Johnson S, Leslie-Hurd R, Rozas C (2016) Intel® Software Guard Extensions (Intel® SGX) Support for Dynamic Memory Management Inside an Enclave. In: Proceedings of the Hardware and Architectural Support for Security And Privacy 2016 HASP '16. Association for Computing Machinery, New York, pp 1–9 https://doi.org/10.1145/2948618.2954331
10. McKeen F, Alexandrovich I, Berenzon A, Rozas CV, Shafi H, Shanbhogue V, Savagaonkar UR (2013) Innovative instructions and software model for isolated execution. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy HASP '13. Association for Computing Machinery, New York, p 1 https://doi.org/10.1145/2487726.2488368
11. Priebe C, Vaswani K, Costa M (2018) EnclaveDB: A Secure Database Using SGX. In: 2018 IEEE Symposium on Security and Privacy (SP). IEEE, San Francisco, pp 264–278 https://doi.org/10.1109/SP.2018.00025
12. Maliszewski K, Quiané-Ruiz J-A, Traub J, Markl V (2021) What is the price for joining securely? benchmarking equi-joins in trusted execution environments. Proc VLDB Endow 15(3):659–672. https://doi.org/10.14778/3494124.3494146
13. Tsai C-C, Porter DE, Vij M (2017) Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In: 2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association, Santa Clara, pp 645–658 (https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai)
14. Gramine Project (2025) Gramine Documentation—Gramine Features. https://gramine.readthedocs.io/en/stable/devel/features.html. Accessed 2025-01-08
15. Gramine Project (2025) Gramine Documentation. https://gramine.readthedocs.io/en/stable/. Accessed 2025-01-08
16. Bailleu M, Thalheim J, Bhatotia P, Fetzer C, Honda M, Vaswani K (2019) SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In: 17th USENIX Conference on File and Storage Technologies FAST 19, pp 173–190 (https://www.usenix.org/conference/fast19/presentation/bailleu)
17. Kim T, Park J, Woo J, Jeon S, Huh J (2019) ShieldStore: Shielded In-memory Key-value Storage with SGX. In: Proceedings of the Fourteenth EuroSys Conference EuroSys '19. Association for Computing Machinery, New York, pp 1–15 https://doi.org/10.1145/3302424.3303951
18. Sun Y, Wang S, Li H, Li F (2021) Building enclave-native storage engines for practical encrypted databases. Proc VLDB Endow 14(6):1019–1032. https://doi.org/10.14778/3447689.3447705
19. Vinayagamurthy D, Gribov A, Gorbunov S (2019) StealthDB: A Scalable Encrypted Database with Full SQL Query Support. Proc Priv Enhancing Technol 2019(3):370–388. https://doi.org/10.2478/popets-2019-0052
20. Antonopoulos P, Arasu A, Singh KD, Eguro K, Gupta N, Jain R, Kaushik R, Kodavalla H, Kossmann D, Ogg N, Ramamurthy R, Szymaszek J, Trimmer J, Vaswani K, Venkatesan R, Zwilling M (2020) Azure SQL Database Always Encrypted. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data SIGMOD '20. Association for Computing Machinery, New York, pp 1511–1525 https://doi.org/10.1145/3318464.3386141
21. Fuhry B, Jayanth Jain HA, Kerschbaum F (2021) EncDBDB: Searchable Encrypted, Fast, Compressed, In-Memory Database Using Enclaves. In: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp 438–450 https://doi.org/10.1109/DSN48987.2021.00054

22. Wang Y, Shen Y, Su C, Ma J, Liu L, Dong X (2020) CryptSQLite: SQLite With High Data Security. IEEE Trans Comput 69(5):666–678. https://doi.org/10.1109/TC.2019.2963303
23. Maliszewski K, Quiané-Ruiz J-A, Markl V (2023) Cracking-Like Join for Trusted Execution Environments. Proc VLDB Endow 16(9):2330–2343. https://doi.org/10.14778/3598581.3598602
24. Arnautov S, Trach B, Gregor F, Knauth T, Martin A, Priebe C, Lind J, Muthukumaran D, O'Keeffe D, Stillwell ML, Goltzsche D, Eyers D, Kapitza R, Pietzuch P, Fetzer C (2016) SCONE: Secure Linux Containers with Intel SGX. In: 12th USENIX Symposium on Operating Systems Design and Implementation OSDI 16, pp 689–703 (https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov)
25. Orenbach M, Lifshits P, Minkin M, Silberstein M (2017) Eleos: ExitLess OS Services for SGX Enclaves. In: Proceedings of the Twelfth European Conference on Computer Systems EuroSys '17. Association for Computing Machinery, New York, pp 238–253 https://doi.org/10.1145/3064176.3064219
26. Battiston I, Felius L, Ansmink S, Kuiper L, Boncz P (2024) DuckDB-SGX2: The Good, The Bad and The Ugly within Confidential Analytical Query Processing. In: Proceedings of the 20th International Workshop on Data Management on New Hardware DaMoN '24. Association for Computing Machinery, New York, pp 1–5 https://doi.org/10.1145/3662010.3663447
27. El-Hindit et al (2022) second_gen_sgx_benchmark. https://github.com/DataManagementLab/second_gen_sgx_benchmark. Accessed 2025-01-10
28. Lutsch et al (2024) sgxv2-analytical-query-processing-benchmarks. https://github.com/DataManagementLab/sgxv2-analytical-query-processing-benchmarks. Accessed 2025-01-10
29. Willhalm T, Popovici N, Boshmaf Y, Plattner H, Zeier A, Schaffner J (2009) SIMD-scan: Ultra fast in-memory table scan using on-chip vector processing units. Proc VLDB Endow 2(1):385–394. https://doi.org/10.14778/1687627.1687671
30. Intel Corporation (2018) Speculative Store Bypass / CVE-2018-3639 / INTEL-SA-00115. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-store-bypass.html. Accessed 2024-05-10
31. Weisse O, Bertacco V, Austin T (2017) Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In: Proceedings of the 44th Annual International Symposium on Computer Architecture ISCA '17. Association for Computing Machinery, New York, pp 81–93 https://doi.org/10.1145/3079856.3080208
32. Tian H, Zhang Q, Yan S, Rudnitsky A, Shacham L, Yariv R, Milshten N (2018) Switchless Calls Made Practical in Intel SGX. In: Proceedings of the 3rd Workshop on System Software for Trusted Execution SysTEX '18. Association for Computing Machinery, New York, pp 22–27 https://doi.org/10.1145/3268935.3268942
33. Lutsch et al (2025) full-DBMS-in-SGX-experiments. https://github.com/DataManagementLab/full-DBMS-in-SGX-experiments. Accessed 2025-01-10
34. Dreseler M, Kossmann J, Boissier M, Klauck S, Uflacker M, Plattner H (2019) Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In: Herschel M, Galhardas H, Reinwald B, Fundulaki I, Binnig C, Kaoudi Z (eds) Advances in Database Technology 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, March 26–29, 2019 OpenProceedings.org, Konstanz, pp 313–324 https://doi.org/10.5441/002/EDBT.2019.28
35. Jemalloc (2024) https://jemalloc.net/. Accessed 2024-12-28
36. Müller I, Sanders P, Lacurie A, Lehner W, Färber F (2015) Cache-Efficient Aggregation: Hashing Is Sorting. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data SIGMOD '15. Association for Computing Machinery, New York, pp 1123–1136 https://doi.org/10.1145/2723372.2747644
37. Do T, Graefe G, Naughton J (2023) Efficient sorting, duplicate removal, grouping, and aggregation. ACM Trans Database Syst 47(4):16–11635. https://doi.org/10.1145/3568027
38. Albutiu M-C, Kemper A, Neumann T (2012) Massively parallel sort-merge joins in main memory multi-core database systems. Proc VLDB Endow 5(10):1064–1075. https://doi.org/10.14778/2336664.2336678