Computation-in-Memory Accelerators for Secure Graph Database: Opportunities and Challenges

Md Tanvir Arafin
ECE Department
Morgan State University
Baltimore, MD 21239
e-mail: mdtanvir.arafin@morgan.edu

Abstract— This work presents the challenges and opportunities for developing computing-in-memory (CIM) accelerators to support secure graph databases (GDB). First, we examine the database backend of common GDBs to understand the feasibility of CIM-based hardware architectures to speed up database queries. Then, we explore standard accelerator designs for graph computation. Next, we present the security issues of graph databases and survey how advanced cryptographic techniques such as homomorphic encryption and zero-knowledge protocols can execute privacy-preserving queries in a secure graph database. After that, we illustrate possible CIM architectures for integrating secure computation with GDB acceleration. Finally, we discuss the design overheads, useability, and potential challenges for building CIM-based accelerators for supporting data-centric calculations. Overall, we find that computing-in-memory primitives have the potential to play a crucial role in realizing the next generation of fast and secure graph databases.

I. Introduction

Graph databases (GDBs) are becoming increasingly popular in storing knowledge graphs, social network data, and chemical/biological data. Additionally, GDBs are used for graph analysis as a service (GAAS) based cloud platforms. Furthermore, the advent of graph neural networks (GNNs) and their application in image processing, natural language processing, applied chemistry, computational biology, and combinatorial optimization have made efficient data processing in GDBs a vital problem for next-generation of computation.

Big-data centric applications such as GDBs demand a move from *processor-centric* to *data-centric* systems. One such data-centric solution is the computation-in-memory (CIM) paradigm. CIM techniques have been explored over the last few decades for data-intensive problems. However, the rapid advancement in computing power in the Moore era overshadowed the earlier promises of CIM-based solutions. Data-movement cost in traditional algorithms was also compensated by clever cache design and architecture improvements. Interestingly, as we are approaching a post-Moore era and experiencing an explosion of data, CPU-centric computation is becoming bottlenecked by data-movement costs. In addition, the emergence of novel memory solutions such as resistive memory (RRAM), phase-change memory(PCM), 3D-stacked memories, and smart memory cubes have rekindled the promises of CIM-based solutions for large-scale systems.

Recent advances in CIM-based hardware accelerators can significantly impact computation performances in graph databases and other graph-related applications. It is evident that graph databases enjoy a boost by moving from traditional disk drives (HDDs) to solid-state drives (SSDs) and non-volatile memory express (NVMe) based faster storage medium (1; 2). Interestingly, these solutions (*i.e.,* faster persistent storage or near CPU memories) show a slow but steady move to processing-in-memory solutions for database systems.

On the other hand, emerging memory devices such as RRAM and PCMs provide crossbar memory structures to accelerate adjacency matrix-based graph computation. Moreover, traditional DRAMs and hybrid memory cubes (HMCs) have used near-memory processing techniques to speed up adjacency list-based algorithms. However, most reported works focus on graph engines that solve specific problems or a sub-set of similar graph algorithms. As a result, there is a void in using computation-in-memory techniques to accelerate general-purpose graph databases targeted towards large-scale data analytics.

Additionally, securing GDBs has attracted researchers due to the sensitive nature of the private data in GDBs. Common security vulnerabilities arise in GDBs from private data mining, data leaks, or malicious attacks from adversaries. Therefore, graph databases not only need to provide computation capability over large-scale data but also secure the data and computation.

Hence, this work reports the current state of graph databases and the promises of computing-in-memory for accelerating GDBs. We also discuss the security issues of traditional graph databases and how near- and in-memory computation can help solve some of the privacy and security challenges in GDBs.

II. Background

A. Graph Model

Graph representation in GDBs varies in practice. This work considers an edge-labeled graph model which is widely used in implementing popular graph databases such as Neo4j, OrientDB, and MS Graph Engine. In this graph model, edge labels are used for representing different relationships between the nodes. The nodes and edges can be stored as adjacency matrices or adjacency lists.

We consider the edge-labeled adjacency lists for storing the graph database in this work. Figure 1 details a simple example of an edge labeled graph. Here three vertices v_1, v_2, v_3 are connected via different edges $e_1, e_2, \dots e_5$. The edges have labels l_1, l_2 which represents different relationships between the connected vertices.

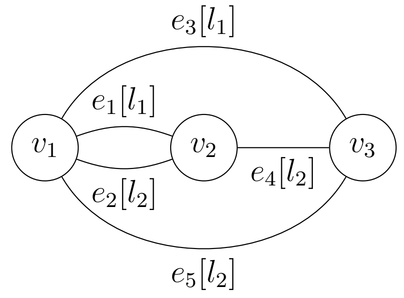

Fig. 1. Example of an edge-labelled graph. Labels for each edge is given inside the square brackets.

B. Data Store Model

We adopt a data store model similar to a common database, Neo4j, to illustrate the details of a GDB. In this model, each vertex record contains a pointer to the first edge and additional metadata such as a pointer to vertex properties. Each edge record contains the label representing the relationship between the connected vertices, pointers to the vertex pairs connected by the edge, and the pointers for previous and next edges for each vertex. The vertex and edge records for the graph at Figure 1 are shown in Table I.

TABLE I
DATA STORE MODEL FOR THE GRAPH IN FIGURE 1

Vertex	First Edge, Metadata
v_1	e_1, m_1
v_2	e_1, m_2
v_3	e_3, m_3

Edge	Label, Vertices, (Edge Pair$_1$), (Edge Pair$_2$),
e_1	$l_1, (v_1, v_2), (-, e_2), (-, e_2)$
e_2	$l_2, (v_1, v_2), (e_1, e_3), (e_1, e_4)$
e_3	$l_1, (v_1, v_3), (e_2, e_5), (-, e_4)$
e_4	$l_2, (v_2, v_3), (e_2, -), (e_3, e_5)$
e_5	$l_2, (v_1, v_3), (e_3, -), (e_4, -)$

C. Graph Traversal

Graph databases are designed for efficient traversal on a graph. For example, the datastore model presented in II-B can be implemented with doubly-linked lists. The vertex and edge values can be represented as memory locations where individual vertex and edge are stored. The metadata is not stored in the vertex table; instead, a pointer to the metadata is maintained. Similarly, a pointer to the detailed label information is kept as the label entry of the edge table. This leads to a fixed-size entry for each row in the vertex and edge table. Additionally, the edge table contains pointers for previous and next edges for each vertex.

When edge and vertex tables are stored in memory, traversing the graph becomes a trivial problem of traversing a doubly-linked list. As a result, pointer chasing becomes one of the fundamental operations for a graph database. Unfortunately, pointer chasing operations can incur significant memory access time for a large-scale graph due to cache misses. Therefore, faster memory components and better cache management techniques have traditionally been advocated for increasing performance in existing GDBs with similar data store model (1; 3; 4).

D. Graph Query Model

Graph queries on GDBs can be categorized into two fundamental types (1) graph pattern matching (GPM) and (2) graph navigation (GN) (5). Modern query languages such as Gremlin, Cypher, SPARQL *etc.* provide support for GPM and GN queries (6). Graph pattern matching queries have the same structure as the underlying GDB data structure, *e.g.,* a GPM query on an edge-labeled graph is also an edge-labeled graph. For example, finding all vertices that have the label l_2 with v_2 in Figure 1 is a basic graph pattern problem, and it can be reduced to find all instances of the pattern of Figure 2. Graph pattern-finding queries are used in practical problems, such as machine learning, pattern matching and recognition, chemical analysis, and planning problems. Complex queries such as projection, join, union, and filter can be converted into graph pattern problems and solved accordingly (5).

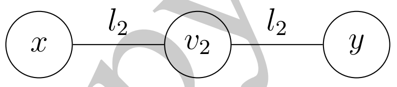

Fig. 2. Example query for Figure 1. The query returns all the values of x and y for all the sub-graphs of Figure 1 that matches this graph.

On the other hand, path queries such as finding path existence, neighbors-of-neighbors of a node, and route-finding can be solved using GN techniques. These queries are essential in solving problems in the semantic web, provenance applications. Therefore, standard graph database solutions such as Neo4j, Apache TinkerPop provide support for these queries. Adjacency list-based GDBs require fast list traversal operations for solving GN and GPM queries.

E. Computation-in-Memory

Computation-in-memory is one of the critical ideas of data-centric system design. In modern systems, data movement from memory to the CPU can incur significant power consumption and latency issues. CIM paradigm tries to move the computation to or near the data and thus, reduces the data-movement costs. Figure 3 shows a basic architecture for a CIM-integrated memory system.

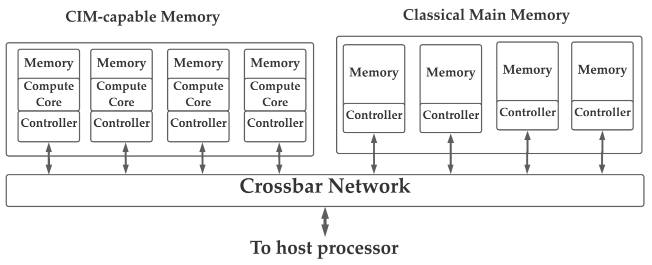

Fig. 3. Example of a CIM-integrated memory system. The compute core in the CIM-capable memory can consists of lightweight in-order core or simpler application specific logic elements.

There have been two fundamental development tracks in CIM designs – (1) near-memory and (2) in-memory computing. Near-memory computing (NMC) based designs support small in-order processors cores near the memory banks. These cores can be placed on the DIMMs of the DRAM-based main memories or on the logic layer of the 3D hybrid memory cubes (3; 4; 7). Thus, for NMC, the compute cores in Figure 3 will be small CPU cores running very near to the memory die.

In contrast, in-memory computing (IMC) solutions use novel memory systems such as resistive random access memories, phase-change memory, spin-transfer-torque magneto-resistive memories, and modified DRAMs as the memory component. These memory elements can perform basic logic and arithmetic computations. Hence, part of the memory system is repurposed for data processing, (*i.e.,* the compute cores in Figure 3 are compute-capable memory elements in IMC). As a result, in IMC-based systems, logical and arithmetic computation can be performed on the data even when the data resides in the memory. Since both IMC and NMC memories contain storage, they can accompany the main memory or replace the main memory in a system.

III. Acceleration of GDBs

With the increase in graph size, poorly designed GDBs can become notoriously slow. Moreover, the random memory access pattern of GDBs can make traversal and neighborhood-related queries cache-inefficient. Hence, there have been several proposals for accelerating graph workload using custom hardware accelerators, graphical processing units (GPUs), and computation-in-memory primitives.

A. Hardware Accelerators

Dedicated hardware accelerators can offer fast and parallel graph processing solutions (8; 9). Some graph problems are highly parallel, and proper partitioning can provide significant speedup. For example, Ozada *et al.* (9) have illustrated how standard graph algorithms can be broken in gather-scatter steps on a vertex-centric graph. Using accelerators for gather, apply, and scatter operations along with edge and vertex caches (as shown in Figure 4(a)), they showed that application-specific accelerators could significantly outperform CPU solutions in terms of power and efficiency.

Additionally, streaming-based solutions such as GraphPulse (10) exploit the locality of access principle and can deliver low power yet efficient accelerators. Streaming-based accelerators maintain on-chip memory queues, prefetcher, and edge/vertex caches, as shown in Figure 4(b)), to analyze active portions of a graph. Although these solutions provide significant speedup from the native CPU base solutions, their effectiveness is dependent on the communication speed with the main memory, as shown in Figure 4.

B. GPU-based acceleration

GPUs have also been explored for improving the performance of graph processing. It has been found that GPUs suffer worse from long latency memory operations in graph applications than non-graph applications (11; 12). Moving the memory near to the GPU kernels, ensuring heterogeneous memory access for CPU and GPU, improving cache operation, and replacing the PCIe bus with a faster and direct memory access have been explored as proposed solutions for removing the performance bottleneck of graph processing in GPUs (11; 12; 13; 14; 15).

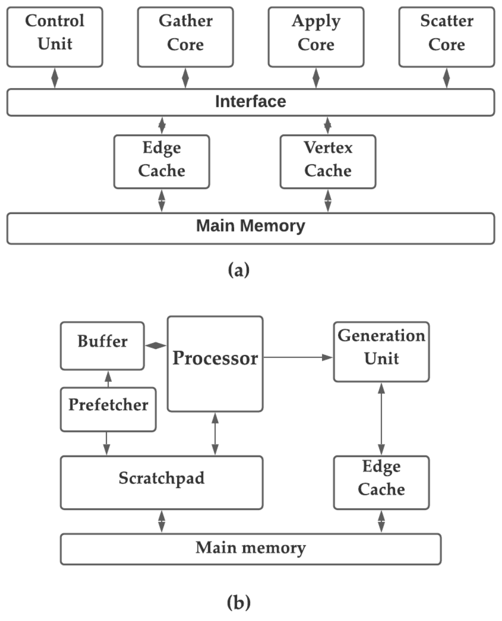

Fig. 4. Different graph accelerator designs, (a) shows a gather, apply, scatter based accelerator (9), and (b) implements a streaming-based graph asynchronous graph processor (10).

C. CIM-based Acceleration

Two of the critical bottlenecks in query resolution for GDBs are the data movement from the memory to the processing units and pointer chasing as described in Section II-C. Therefore, in CPU-centric design, faster memory access improves the performance of databases. For example, simply replacing non-NVMe SSDs with NVMe-enabled SSDs increases the throughput of database management systems by 1.3x (16). Neo4j has reported over 1500x speedup in terms of IOPS (input/output operations per second) when moving from throughput optimized HDD to a locally attached NVMe drive (1). On the other hand, main memory components that sit relatively closer to the CPU, such as Intel Optane, can provide as much as 10x improvement than traditional DRAMs for database operations (2). Ahn *et al.* (3) have demonstrated that CIM-based near-memory computation can offer faster and more efficient pointer chasing. Additionally, Choe *et. al.* have demonstrated that concurrent data structures link-lists, skip lists, and FIFO queues enjoy significant throughput improvement in near-memory computing hardware than traditional processing architectures (7). These promising results mandate a closer inspection of near-memory data processing for accelerating graph database applications.

IV. Security Issues of Graph Databases

There have been several reports on the security vulnerabilities of graph databases (17; 18). Commonly deployed GDBs such as Neo4j, RedisGraph, and OrientDB have been found to contain a wide range of issues from basic authentication vulnerability to remote code executions (17). A partial list of recent vulnerabilities in common GDBs is given in Table II.

Given the vulnerabilities of existing GDBs, one of the critical concerns is to secure sensitive private data in such databases. Security aware coding and software development

TABLE II
LIST OF RECENTLY DISCOVERED VULNERABILITIES IN COMMODITY GDBS

Attack Type	GDB	Publication
Privilege Escalation	Neo4j v4.2 & v4.3	CVE-2021-34802
Remote code execution	Neo4j 3.4.18 OrientDB 2.2.22	CVE-2021-34371 CVE-2017-11467
Authentication	Neo4j 3.4.x OrientDB	CVE-2018-18389 CVE-2019-25021
Code injection	SAP OrientDB v3.0	CVE-2020-6230
Denial-of-service	RedisGraph v2.2.1	CVE-2020-35668

practices and advanced authentication techniques have limitations and cannot provide security guarantees to the data owner. As a result, privacy-preserving secure graph computations has become an active research area for exploring opportunities in security solutions for GDBs.

A. Threat Model

GDBs have several stakeholders, such as (1) the data owner who provides private data, (2) the database owner who maintains the database, and (3) the cloud service provider who queries the DB to provide relevant information to the user. For simple GDBs problems, the data owner, GDB owner, and computation service provider are usually the same entity. However, as the database grows, data might be generated at the edge by the end-users, the GDB might be stored in the cloud, and the query performed on the data might be done on third-party servers. This complex ecosystem creates a significant problem: *how can different stakeholders provide efficient computation and security guarantee to each other?*

In this work, let us assume a simple threat model – an honest client C who owns the data and queries on the data, and a cloud server S who stores the data and performs the query computation on the GDB. S is not trustworthy, *i.e.,* S might want to infer information regarding the data from the queries and the stored data. Hence, for the most straightforward instance of secure GDB, the goal for the designer is to provide privacy-preserving data store and query response from S.

B. Privacy Preserving Solutions for GDBs

A secure GDB should cover both the front-end entities (*i.e.,* external adversaries) and backend service providers from accessing and manipulating private data. As a result, privacy-preserving computation and storage solutions become imperative for secure GDBs. Given the threat model in Section IV-A, the information regarding the stored data as well as the queries should be encrypted to support data privacy.

The most straightforward solution to this problem is to use an encrypted database and provide encrypted query support for the DB. For example, Aburawi *et al.* has demonstrated how CryptDB(19), an encrypted database solution, can be repurposed for supporting GDBs (20). However, since CryptDB is essentially a SQL-based design, native graph-DBs cannot directly support such solutions. Therefore, graph-oriented cryptographic primitives are required for efficient privacy-preserving GDB queries.

There has been significant progress in computation on encrypted graphs over the last decade (21; 22; 23; 24; 25; 26; 27; 28). Fully homomorphic encryption (FHE), oblivious RAM, garbled circuits, and verifiable computation primitives are used for designing simple graph queries over the encrypted graph. Specifically, additive homomorphic encryption can provide a simple yet effective solution for secure GDB updates. An example of such update techniques is given in Algorithm 1 (21). Here, an arithmetic black-box construction is employed that allows S and C to store elements of a ring Z_m securely in the DB. Assume that elements inside the square bracket represent encrypted elements, and additive and multiplication homomorphic encryption (such as Paillier encryption) are supported on these elements. Then, if C wants to update an edge or vertex list $[l]$ at $[k]$-th location with $[z]$, s/he can construct an update vector $[e]$ where $e_i = 1$ if $i == k$, else $e_i = 0$. Then, C can securely share $[e]$ and $[z]$ with S, and S can run Algorithm 1 on the GDB to update the kth element.

Algorithm 1 Update an encrypted list $[l]$ at $[k]$-th location with $[z]$ (21)

```
1: procedure $[l]_{[k]} \leftarrow update([z])$
2:     For $i$ in $1 : length(l)$
3:         $[l]_i \leftarrow [1]_i + [e]_i.[z] - [e]_i.[l]_i$
```

This update scheme has a time complexity of $O(n)$, where n is the number of elements in the list. Aly *et al.* have also demonstrated secure shortest path, and maximal flow using this same construction (21). This simple example illustrates the increased computational complexity for security. Each update requires additional $O(n)$ homomorphic additions and $O(n)$ multiplications. Thus, accelerators build for FHE techniques, specifically, number-theoretic-transform (NTT) accelerators, can expedite such calculation.

SecGDB is another secure GBD solution (22) that uses structural encryption schemes to design GDBs specialized for exact shortest distance queries. The key idea of the work is to use structured encryption for faster search on an encrypted dataset. Additive Paillier encryption, garbled circuits, and Fibonacci heaps are employed in SecGDB for resolving privacy-preserving GN queries. This design shows adaptively schematic security and efficiency in practice.

Finally, Cui *et al.* (24) have recently demonstrated oblivious filter-based protected graph database search. Their design targets encrypted edge labeled graph database and uses Ulmann's algorithm (29) for developing efficient GPM and GN query resolution. One core concept is that the server S shares the encrypted graph details $[G]$ with the client C. The client can check the validity of the graph $[G]$ and generate the query graph. The query problem is resolved on the client-side, where the client creates encrypted subqueries for every step of Ulmann's algorithm, and the server S responds to the subquery. Since the subquery responses are returned from an encrypted database, the server S cannot infer any information from the responses.

C. Open Problems in securing GBDs

Although techniques for encrypted graph database management exist as shown in Section IV-B, state-of-the-art GDBs do not practice advanced cryptographic tools or methods to address security issues. Our studies find that the following technical challenges exist on the road to privacy-preserving graph computations.

C1. Computation overhead: Cryptographic computation on encrypted data incurs a large computation overhead. Easing this overhead using improved algorithms and supporting hardware can make secure GDBs cost-effective.
C2. Weak-security models: Weaker security models with implicit trust on the server or trusted third parties provide a simpler alternative for reducing computation overhead. In practice, this is the status-quo where the user puts faith in the service providers' basic authentication and security guarantees.
C3. Non-uniform standards: Developing a security solution for GDB is also challenging due to the lack of industry-standard on NoSQL databases. Different GDBs support different data store and query languages, making improvising a single solution for these DBs extremely difficult.

V. CIM for Securing GDBs

As discussed in Section II-E, CIM-based computation can provide in-memory multiplication and parallel logic operation useful in designing FHE or multiparty computation-based security primitives. Thus, CIMs have the potential to accelerate processes in a secure GDB. In Figure 5, we present a vision for the system architecture of a CIM-based graph accelerator that supports encrypted databases.

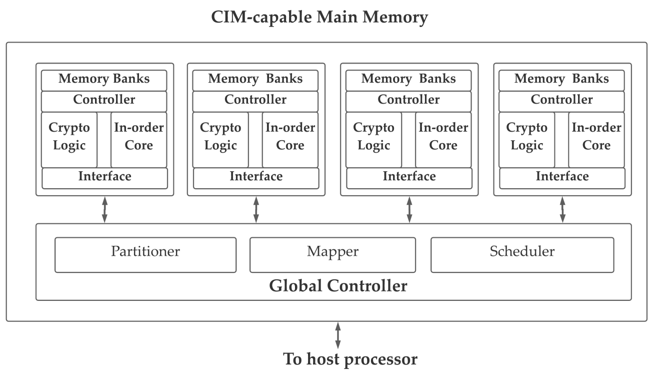

Fig. 5. System architecture for CIM-based memory system for graph acceleration.

The design is based on a redesigned hybrid memory cube (HMC) for the main memory. Choe et al. (7) and Ahn et al. (3) have demonstrated that modified HMC vaults are excellent for fast list traversal operations. The logic core in each vault consists of a simple in-order processor for essential data operation and an additional cryptographic logic accelerator. This logic component can be a cryptographic key-storage or an NTT-accelerator based on the security design. Since we assume an adjacency list-based store, list traversal operations will be accelerated by the in-order core. The memory interfaces are connected to a crossbar and provide communication capabilities between the cores. A global controller can support graph partitioning, mapping, and operation scheduling to multiple vaults. It should be noted that the architecture can also represent IMC designs, where the crypto logic core and the in-order processor can be replaced with compute-capable memory units. Such systems can open up some excellent opportunities in privacy-preserving graph computation, as discussed below.

1) Hardware-root-of-trust: CIM and near-memory processing can hide data access patterns from the host CPU. Thus, CIM-based computation's security model should significantly differ from traditional security models based on von-Neumann architectures. This can also provide additional security opportunities. For example, cryptographic logic can be employed to contain keys for a given vault. Data-owners can trust only the CIM hardware and send encrypted data using the vault's key. For larger datasets, hash trees can be constructed using the keys, and the user can share the encrypted data with the known vaults. This is a weak security model; however, it moves the trust from the server S to the server hardware.
2) Support for homomorphic computation: Efficient NTT accelerators can be constructed on the crypto logic fabric of the HMC. This moves the FHE computation from the processor to the memory. Clients can create encrypted queries such as the ones discussed in (24; 21) and the vaults will provide computation on encrypted data. On the other hand, if we consider an IMC system, compute-capable memory can provide fast arithmetic and logic computation on encrypted data. For example, RRAM crossbars can perform a large number of parallel multiplications simultaneously (30). Such designs can enable faster execution of gather, apply, scatter-based parallel algorithms on encrypted GDBs.
3) Secure enclaves: The in-order processor and the crypto logic core can provide constructions for secure enclaves that can certify the computation performed in the graph database.
4) Ease of integration: Main-memory components such as DRAMs are easy to replace in a computing system. Properly designed CIMs should be able to provide backward compatibility to DRAM-based systems and introduce in-memory computing capacity to the system.

In addition to the obstacles mentioned in C1-C3, developing systems such as Figure 5 has several challenges. These challenges arise from load balancing, concurrency, hardware-software co-design, and security issues. First, queries on a large-scale graph can be sporadic and non-local. Inefficient partitioning and storage can cause load unbalanced to specific CIM components. This issue can get worsen for encrypted query resolutions. Therefore, a proper load-balancing technique is essential for an efficient GDB.

Second, concurrent data structures need to be investigated in detail to harness the full potential of parallel processing in CIM. Additionally, support of cryptographic computation on CIM-based concurrent data structure needs to be developed.

Third, CIM-based designs have an opportunity to replace conventional DRAM without incurring significant system changes. HMC-based structures also support the coexistence of DRAM and CIM modules. Therefore, improved hardware-software co-development for handling graph problems with simple in-order cores in CIM is required to accelerate their adoption in practice.

Finally, computing-in-memory opportunities can open up new security and trust issues (31). Main memory systems are vulnerable to hardware and software side attacks (32; 33).

Thus, the CIM-baed system must consider the security implications of its design and implementation before wide-scale adaptation.

VI. Conclusion

Security and privacy issues are critical for existing GDBs. Unfortunately, privacy-preserving computations in GDBs require significant computation overhead, which can become intractable with common system design approaches. We find that recent advances in computation-in-memory-based data-processing and cryptography can offer viable solutions for secure graph computation in GDBs. This work should guide and encourage computer architects, GDB designers, and security researchers in developing the next generation of privacy-preserving large-scale graph processing techniques.

References

[1] J. Phoulchand. (2021) Diagnose storage performance issues. [Online]. Available: https://neo4j.com/developer/kb/diagnose-storage-performance-issues/

[2] L. Hughes. (2019) Top tier bank achieves nearly 10x speed improvement using kx kdb+ v4.0 and optane memory. [Online]. Available: http://www.odbms.org/2020/08/top-tier-bank-achieves-nearly-10x-speed-improvement-using-kx-kdb-v4-0-and-optane-memory/

[3] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 336–348.

[4] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 105–117.

[5] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč, "Foundations of modern query languages for graph databases," *ACM Computing Surveys (CSUR)*, vol. 50, no. 5, pp. 1–40, 2017.

[6] M. Besta, E. Peter, R. Gerstenberger, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler, "Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries," *arXiv preprint arXiv:1910.09017*, 2019.

[7] J. Choe, A. Huang, T. Moreshet, M. Herlihy, and R. I. Bahar, "Concurrent data structures with near-data-processing: An architecture-aware implementation," in *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, 2019, pp. 297–308.

[8] D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan, "Graphreduce: processing large-scale graphs on accelerator-based systems," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12.

[9] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 166–177, 2016.

[10] S. Rahman, N. Abu-Ghazaleh, and R. Gupta, "Graphpulse: An event-driven hardware accelerator for asynchronous graph processing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 908–921.

[11] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua, "Graph processing on gpus: A survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, pp. 1–35, 2018.

[12] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on gpus: Where are the bottlenecks?" in *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2014, pp. 140–149.

[13] V. Vineet and P. Narayanan, "Cuda cuts: Fast graph cuts on the gpu," in *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*. IEEE, 2008, pp. 1–8.

[14] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[15] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2012, pp. 141–151.

[16] J. Arulraj, M. Perron, and A. Pavlo, "Write-behind logging," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 337–348, 2016.

[17] M. Hernández Boza. and A. Muñoz Muñoz., "(in) security in graph databases - analysis and data leaks," in *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications - SECRYPT, (ICETE 2017)*, INSTICC. SciTePress, 2017, pp. 303–310.

[18] G. Hurlburt, "High tech, high sec.: Security concerns in graph databases," *IT Professional*, vol. 17, no. 1, pp. 58–61, 2015.

[19] R. A. Popa, C. M. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: Protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 85–100.

[20] N. Aburawi, "Cryptdb mechanism on graph databases," Ph.D. dissertation, University of Liverpool, 2020.

[21] A. Aly, E. Cuvelier, S. Mawet, O. Pereira, and M. Van Vyve, "Securely solving simple combinatorial graph problems," in *International Conference on Financial Cryptography and Data Security*. Springer, 2013, pp. 239–257.

[22] Q. Wang, K. Ren, M. Du, Q. Li, and A. Mohaisen, "Secgdb: Graph encryption for exact shortest distance queries with efficient updates," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 79–97.

[23] S. Lai, X. Yuan, S.-F. Sun, J. K. Liu, Y. Liu, and D. Liu, "Graphse^2: An encrypted graph database for privacy-preserving social search," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 41–54.

[24] J. Cui, C. Chen, and L. Wang, "Highly-scalable protected graph database search with oblivious filter." *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 1415, 2020.

[25] P. Yi, Z. Fan, and S. Yin, "Privacy-preserving reachability query services for sparse graphs," in *2014 IEEE 30th International Conference on Data Engineering Workshops*. IEEE, 2014, pp. 32–35.

[26] X. Meng, S. Kamara, K. Nissim, and G. Kollios, "Grecs: Graph encryption for approximate shortest distance queries," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 504–517.

[27] S. Sharma, J. Powers, and K. Chen, "Privategraph: Privacy-preserving spectral analysis of encrypted graphs in the cloud," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 5, pp. 981–995, 2018.

[28] C. Chen, J. Cui, G. Liu, J. Wu, and L. Wang, "Survey and open problems in privacy preserving knowledge graph: Merging, query, representation, completion and applications," *arXiv preprint arXiv:2011.10180*, 2020.

[29] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.

[30] Z. Lu, M. T. Arafin, and G. Qu, "Rime: A scalable and energy-efficient processing-in-memory architecture for floating-point operations," in *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2021, pp. 120–125.

[31] M. T. Arafin and Z. Lu, "Security challenges of processing-in-memory systems," in *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, 2020, pp. 229–234.

[32] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[33] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: A remote software-induced fault attack in javascript," in *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer, 2016, pp. 300–321.