Level 1 Survey

Declarative Analytics on Heterogeneous Exascale Systems

By
Ahmedur Rahman Shovon
Supervisor: Dr. Sidharth Kumar

Reported to
Dr. Ragib Hasan
Department of Computer Science
The University of Alabama at Birmingham

Semester: Spring 2023
February 26, 2023

Contents

1 Abstract

The emergence of exascale systems has brought about a paradigm shift in High-Performance Computing. These systems consist of heterogeneous programming models that combine CPUs and GPUs to perform highly parallel independent calculations. GPU computing has become popular, offering a leap in performance and power efficiency. However, using GPU architectures for GPGPU computing requires the redesign of algorithms that are traditionally executed on a deterministic sequence of steps. Datalog, a lightweight declarative logic programming language, is extensively used in deductive-database systems. It allows users to set problem rules and outline a solution using easily understandable queries. Declarative languages like Datalog can seamlessly link the formulation of human-oriented problems with efficient machine implementation. Datalog programs can be automatically compiled down to relational algebra primitives on relational tables. Recent studies show the prospect of developing multi-threaded and multi-core implementations of Datalog. As upcoming exascale systems have capabilities to leverage GPU parallelism, Datalog applications can take advantage of this opportunity to utilize the heterogeneous architecture to gain high performance, though specialized algorithms and implementations of Datalog are required for optimized use with these systems. The utilization of high-performance declarative analytics provides an exceptional prospect to seamlessly incorporate topological data analysis into high-dimensional data analysis workflows. With the aid of declarative analytics, it is possible to obtain insights from high-dimensional datasets through easy-to-express declarative rules. Overall, the integration of topological data analysis with declarative analytics can offer exceptional insights into high-dimensional data that are not readily observable using conventional analytical approaches.

2 Introduction

As part of the Exascale Computing Project (ECP), which was initiated in 2016, three new exascale systems, including the Aurora by Argonne Leadership Computing Facility (ALCF), will be launched in the 2023-2024 time frame [1]. All exascale systems consist of heterogeneous programming models, including the combination of CPUs and GPUs. GPU computing uses Graphics Processing Unit (GPU) to perform highly parallel independent calculations [2]. It became popular in the mid-2000s and influenced the High-Performance Computing paradigms. The emergence of GPGPU computing refers to the paradigm of performing General Purpose computing using GPU. GPU offers a leap in performance, and power efficiency in terms of TFlop per Watt [3]. The biggest challenge was adopting the GPU architectures, which differs significantly from the traditional multi-core CPUs. Though GPU programming offers thousands of cores, they are coupled with complex hierarchies of memory subsystems, making it challenging to utilize the GPU computing capacity efficiently. GPGPU computing requires to redesign of algorithms that are traditionally executed on a deterministic sequence of steps.

Datalog is a lightweight bottom-up declarative logic programming language. It is extensively used in deductive-database systems as it allows the users to set the problem's rules and outline a solution using easily understandable queries [4, 5, 6, 7, 8]. The queries in a Datalog program are presented as first-order Horn-clause rules. A Datalog program extends data from the input database creating the output database with all data transitively derivable via the program rules. It permits application logic to be written in a high-level manner that is suitable for non-experts leveraging artificial intelligence-based programming techniques. Thus, Datalog is being applied to many domains, including big-data analytics [9, 10, 11], machine learning [12], software analysis [13, 14, 15], graph mining [16], and deductive databases [17]. Declarative languages, like Datalog, can seamlessly link the formulation of human-oriented problems with efficient machine implementation.

Datalog programs can be automatically compiled down to relational algebra (RA) primitives (such as join, projection, aggregation, rename, and selection) on relational tables. These RA primitives are used to create effective kernels that deduce new facts from existing ones. Applications developed on top of declarative languages like Datalog can leverage data parallelism on RA primitive operations to gain high-performance [4, 5, 6, 7, 8]. Being a popular decade-old language, Datalog has multiple implementations [17]. Among them, Souffle [18] is the state-of-the-art CPU-based multi-core implementation based on OpenMP's multi-threading mechanism [19]. Recent studies show the prospect of developing multi-threaded and multi-core implementations of Datalog [20, 21, 22]. These implementations leverage CPU-based parallelism to enhance the performance of Datalog applications. As the upcoming exascale systems have the capabilities of

leveraging GPU parallelism, the Datalog applications can take advantage of this opportunity to utilize the heterogeneous architecture to gain high performance [23]. Using a GPU to accelerate Datalog queries can potentially improve performance for complex queries or massive datasets. However, this would likely require the development of specialized algorithms and implementations of Datalog that are optimized for use with heterogeneous exascale systems.

In this research survey, we will study the performance of Datalog applications on CPU-based multi-node multi-threaded architecture [17, 18, 20, 21, 22]. Then we will explore the existing literature on implementing Datalog applications and RA primitives using GPU architecture [23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36]. We will also find the research gap in recent studies and discuss the major challenges to implementing Datalog applications on heterogeneous exascale systems. As per our ongoing research, we demonstrated the feasibility of implementing Datalog applications using the RA primitives and how we can accelerate high-level logic programming language like Datalog using GPUs for graph dataset analytics [37]. In related work on graph analytics, we analyzed the impact of persistent homology to demonstrate similarity across the functional connectivity networks (FCN) acquired at different temporal sampling periods obtained from resting-state functional magnetic resonance imaging (fMRI)[38]. Furthermore, we contributed to adding Kaplan-Meier survival analyses to the integrated cancer data analysis platform [39].

The survey is organized into multiple sections to provide a comprehensive overview of the topic. Section 3 covers GPU programming models in Exascale computing systems. In Section 4, existing studies towards iterative relational algebra are discussed. Section 5 covers the revival of Datalog and modern Datalog solvers. In Section 6, common Datalog applications such as transitive closure computation and triangle counting are discussed. Section 7 focuses on the potential of topological data analysis for high-dimensional data analytics. Finally, Section 8 outlines the future research directions derived from this survey, and Section 9 concludes the survey.

3 GPU programming in exascale computing

Exaflop is a metric used to gauge the performance of a supercomputer, denoting its capability to carry out a minimum of one quintillion (10^{18}) floating point operations per second [41, 42]. As part of the Exascale Computing Project (ECP) which was initiated in 2016, three new exascale systems including the Aurora by Argonne Leadership Computing Facility (ALCF) will be launched in the 2023-2024 time frame [1]. All of the exascale systems consist of heterogeneous programming models including the combination of CPUs and GPUs. As part of the Exascale Computing Project (ECP) which was initiated in 2016, three new exascale systems including the Aurora by Argonne Leadership Computing Facility (ALCF) will be launched in the 2023-2024 time frame [1]. All of the exascale systems consist of heterogeneous programming models including the combination of CPUs and GPUs. The current ECP systems use 22 combinations of GPU programming models shown in Figure 1. From this figure, we see that the most common GPU programming models of ECP systems are CUDA and Kokkos.

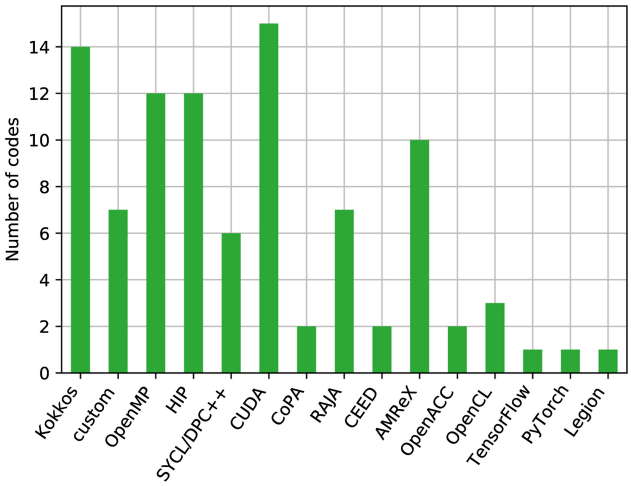


Figure 1: GPU programming models used in exascale computing project[40]

GPU computing uses Graphics Processing Unit (GPU) to perform highly parallel independent calculations [2]. It became popular in the mid-2000s and influenced the High-Performance Computing paradigms. The emergence of GPGPU computing refers to the paradigm of performing General Purpose computing using GPU. GPU offers not only a leap in performance but also power efficiency in terms of TFlop per Watt [3]. The biggest challenge was to adopt the GPU architectures which differs a lot from the traditional multicore

CPUs. Though GPU programming offers thousands of cores, they are coupled with complex hierarchies of memory subsystems which makes it challenging to utilize the GPU computing capacity efficiently. GPGPU computing requires to redesign of algorithms that are traditionally executed on a deterministic sequence of steps. We delve into the category of computing machines as per Flynn's taxonomy in Section 3.1. We then proceed to examine various GPU programming models in Section 3.2.

3.1 Category of computing machines based on Flynn's taxonomy

Michael Flynn introduced a taxonomy of computer architectures in 1966 based on the number of data items and the number of instructions they can process at the same time [43].

1. *Single Instruction, Single Data (SISD)*: This architecture is not in much use nowadays.
2. *Single Instruction, Multiple Data (SIMD)*: GPUs at the level of the Streaming Multiprocessor follow this design, e.g.: SM (Nvidia), SIMD (AMD).
3. *Multiple Instructions, Single Data (MISD)*: Fault tolerance systems use this architecture, especially in military or aerospace applications.
4. *Multiple Instructions, Multiple Data (MIMD)*: This is the most common category, which is followed by multicore machines, including GPUs. Though GPUs can be made from a collection of SM or SIMD units, they collectively behave as MIMD machines. MIMD architecture is classified in two subcategories:
   (a) *Shared-memory MIMD*: This design principle simplifies all transactions between the CPUs with a minimum amount of overhead but it is not suitable for scalability.
   (b) *Distributed memory or shared nothing MIMD*: The processors communicate by exchanging messages, which makes the communication cost high. But this design is scalable. It can be further divided into two classes:
      i. *Master-worker*
      ii. *Symmetric multiprocessing platforms*

3.2 GPU programming models

The architectural differences between traditional CPU systems and GPU systems create the necessity for new development platforms for GPGPU programming. Several popular development platforms are listed below:

3.2.1 CUDA

Nvidia introduced the Compute Unified Device Architecture (CUDA) in 2006, which is one of the first systems for GPGPU that included a unified shader pipeline to allow each Arithmetic Logic Unit (ALU) to be marshaled by a program [44]. They developed the GPUs with ALUs that complies with IEEE requirements for single-precision floating-point arithmetic operations. CUDA architecture provides both high and low-level APIs, which are available in Windows, Mac OS X, and Linux operating systems. CUDA has removed the barrier to having knowledge on OpenGL or DirectX graphics programming interfaces to perform general-purpose computing on GPUs. The CUDA programming model can be invoked using most of the popular programming languages such as C/C++, Fortan, and Python [40]. It follows the Globally Sequential Locally Parallel programming pattern shown in Figure 2. By default, the host machine continues its execution, but it can also wait for the completion of the execution of GPU threads. A CUDA program invokes parallel kernels that execute in parallel across a set of threads. CUDA spawns the threads from a hierarchy of grids and blocks. The programmer organized these threads in two levels, the first one to the three-dimensional grid and in the grid one to three-dimensional blocks. Each thread executes an instance of the kernel. Each thread has a thread ID, program counter, registers, and per-thread private memory. The capacity of the target device limits the sizes of the grids and blocks. CUDA manages the thread creation and destruction procedures that marks the worker management implicit. But the programmer needs to specify the dimension of the grid and block and workload partitioning is done explicitly. The main disadvantage of using CUDA is the device restrictions, CUDA only runs on supported Nvidia GPUs.

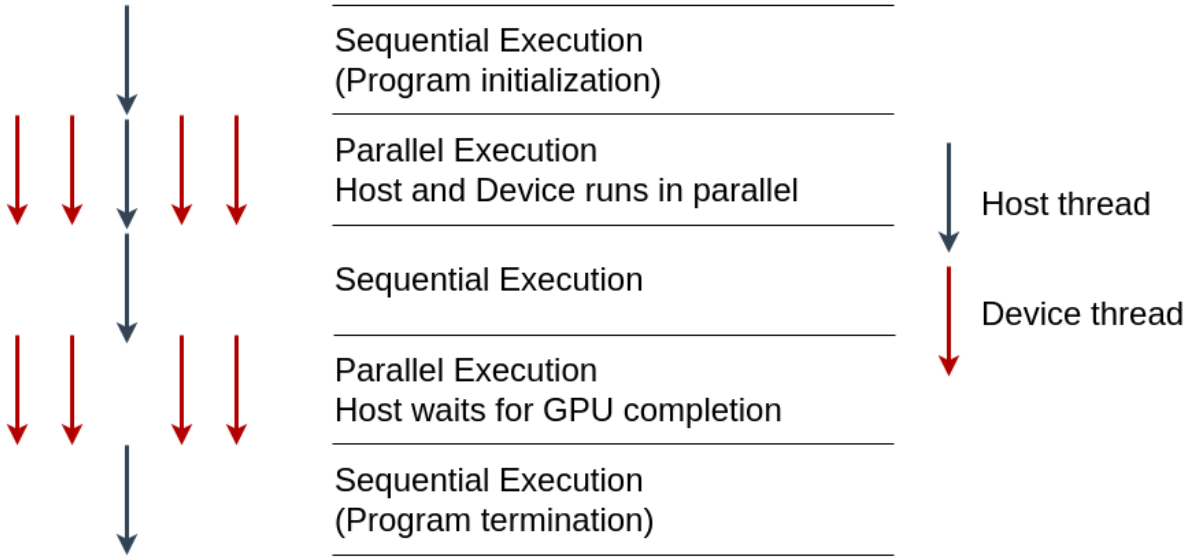


Figure 2: CUDA's Globally Sequential Locally Parallel execution model

3.2.2 OpenCL

To cope with the emerging diversity of GPU devices, OpenCL (Open Computing Language) has enhanced the boundary of GPGPU computing. It is maintained by the Khronos group [40]. As of April 2023, OpenCL latest version 3.0, an OpenCL application has two parts: one or more kernels and a host program [45]. The kernels include functions to be executed on the processing cores in a parallel fashion. The host sets up the execution environment, handles input/output, and enqueues the kernels to run. Application host code is usually written in C or C++ and kernel programs are written in a dialect of C (OpenCL C) or C++ (C++ for OpenCL). The OpenCL defines an abstract execution model and a platform model. Thus it enhances the portability of the program across multiple hardware devices. OpenCL application written for one vendor platform runs successfully on other vendors' platforms, which is ensured by the Khronos' certification program. Though OpenCL ensures the portability of the codes across different devices, it does not ensure performance portability. There are several open and closed-source implementations of OpenCL available. OpenCL had the potential to become the unified de facto standard programming model for heterogeneous many-core processors. But it is not marked as the most popular choice for GPGPU programming for several reasons. The main factor is due to the diversity of many-core architectures with regard to processing cores and memory hierarchies. Other factors include the commercial interests of vendors who prefer to use specialized programming models designed for the relevant devices to build the software ecosystem. For example, the programmers usually choose the CUDA programming model for Nvidia GPU devices as Nvidia optimizes CUDA for its own hardware that results in better performance [40].

3.2.3 OpenMP

Task-level parallelism is achieved in shared memory architecture. OpenMP (Open Multi-Processing) is an industry-standard for pragma-based shared memory multiprocessing programming model, which is maintained by the OpenMP Architecture Review Board (ARB) [19]. The latest version of the OpenMP standard is 5.2, and OpenMP Application Programming Interface (API) supports multi-platform shared memory parallel programming in C/C++ and Fortran [46]. The OpenMP API is implemented as a combination of a set of compiler directives, library routines, and environmental variables. The OpenMP compiler handles the operations related to spawning, initiating, and terminating threads. Thus, the control of the programmer over these operations is deducted, and the programmer only needs to specify compiler directives to denote a parallel region. The OpenMP model is portable across the shared memory architecture. The worker

management is done implicitly, and communication is carried out by shared address space. The execution profile of an OpenMP program follows the Globally Sequential, Locally Parallel (GSLP) structure. The most time-consuming parts of the program are parallelized by the compiler with a little assistance of the programmer.

In an OpenMP program, the parallel directive is called a Single Program Multiple Data (SPMD) directive as multiple threads run the same program. Each thread applies its logic on separate data. The model is suitable for the loop level parallelism and task parallelism supporting the Producers-Consumers semaphore. Though OpenMP is not the most flexible model for concurrency, it works well with the existing sequential structure.

3.2.4 SYCL and DPC++

SYCL is a royalty-free, cross-platform abstraction layer for enabling single-source style heterogeneous parallel programming using standard C++ [47]. In an SYCL application, the host and kernel code is contained in a single source C++ file. SYCL uses generic programming with templates and generic lambda functions to various acceleration APIs, including OpenCL. SYCL is completely standard C++, so no language extensions, pragmas or attributes are required for using SYCL targeting C++ programmers. The latest stable release of SYCL is SYCL 2000, published by the Khronos Group in 2021. The previous releases were based directly on OpenCL [48]. This release includes Unified Shared Memory (USM) model that enables code with pointers without extra buffers requirements. In addition to the support of OpenCL API, SYCL has multiple backends in development supporting diverse acceleration APIs from an increasing number of vendors (Intel, Nvidia, AMD) and hardware such as CPU, GPU, and FPGA (Field Programmable Gate Arrays)

Intel has released oneAPI HPC Toolkit that consists of optimized tools for High-Performance Computing (HPC), including DPC++ (Data Parallel C++) [49]. DPC++ is a high-level language designed for data-parallel programming productivity. It is an Intel implementation of SYCL standard with extensions for supporting explicit SIMD [50]. A DPC++ program is also a C++ program [51]. A DPC++ program can be written in a single-source style. Each of these programs are started by running on a host, which is the acronym for CPUs in GPGPU computing. The acceleration offloading is targeted to devices (GPUs, FPGAs, CPUs) for parallel processing to accelerate the completion of computing. The code for devices is specified as kernels which is a core concept of GPGPU computing. Several features like dynamic polymorphism, dynamic memory allocations, static variables, function pointers, exception handling, virtual member functions, and variadic functions are not allowed in kernel code. The rest of the standard C++ functions are allowed in the kernel code including lambdas, operator overloading, and static polymorphism.

3.2.5 Kokkos

The Kokkos C++ Performance Portability EcoSystem is a part of the US Department of Energies Exascale Project, enables writing modern C++ applications for parallel computing [52, 53]. Though several programming models like OpenMP and OpenCL ensure the portability of code among diverse devices, they do not ensure performance portability, and they fail to address memory access patterns [54]. The Kokkos ecosystem ensures performance portability on diverse many-core architectures by abstracting data parallelism and memory access patterns. Kokkos supports all three major HPC accelerator platforms [55]. It is a library-based programming model as opposed as directive-based or a language-based approach. Kokkos provides six core abstractions. The parallel execution is controlled by *Execution Spaces*, *Execution Patterns*, and *Execution Policies*. The data storage and access are controlled by *Memory Spaces*, *Memory Layouts*, and *Memory Traits*. The ecosystem includes three main components, namely the Kokkos Core programming model, the Kokkos Kernels Math libraries, and the Kokkos profiling and debugging tools [56].

The Kokkos core is a shared-memory parallel programming model that uses many-core chips. This model includes computation abstractions for frequently used parallel computing patterns, policies providing how those patterns are applied, and execution spaces denoting on which cores the parallel computation is performed [40]. The Kokkos core's architecture is shown in Figure 3. The Kokkos kernels provide linear algebra and graph algorithm libraries to achieve the best performance on every architecture. It uses vendor-specific versions of mathematical algorithms as required. The Kokkos tools provide tools infrastructure and debugging tools.

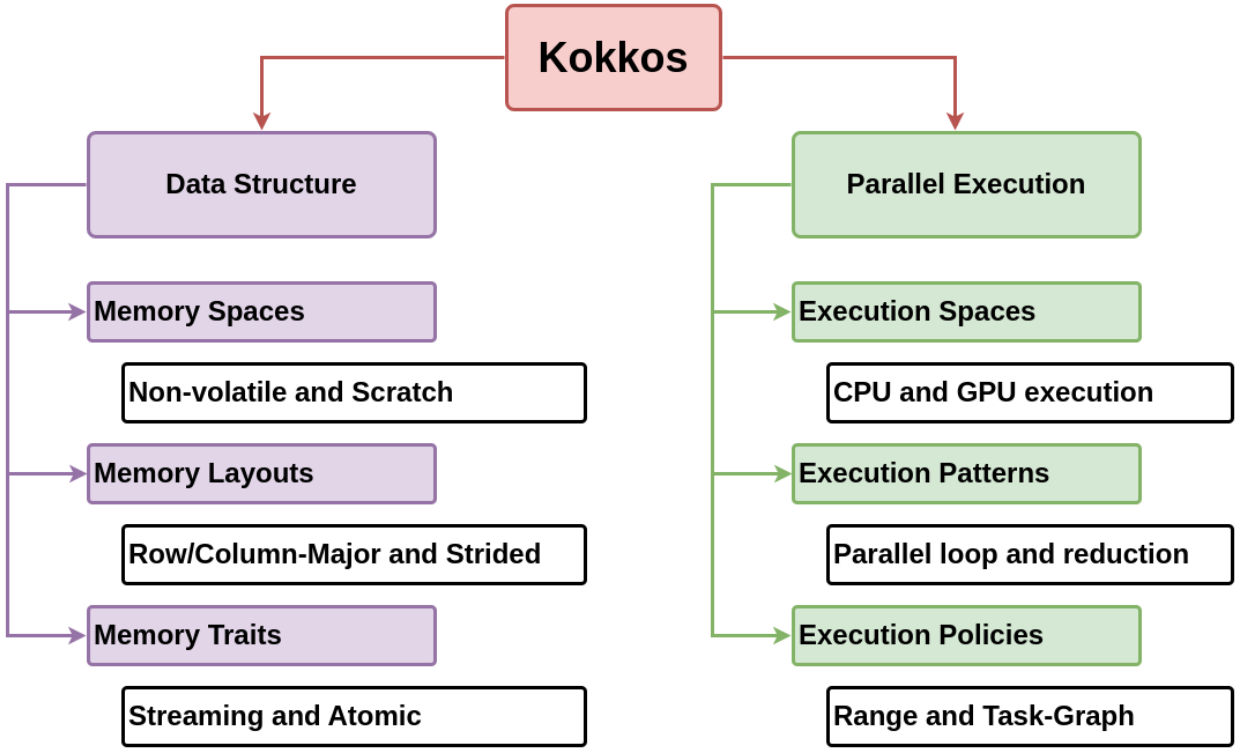


Figure 3: Kokkos data structure and parallel execution mechanism [56]

3.2.6 RAJA

The RAJA portability suite developed at Lawrence Livermore National Laboratory (LLNL) provides open-source tools for architecture and programming model portability for HPC applications [57]. RAJA is a library of C++ abstractions that enables programmers to write performance portable single-source kernels that can run on multiple hardware platforms by recompiling [50]. It insulates application source code from device-specific implementation details. It supports a variety of parallel patterns and performance tuning configurations, including simple and complex loop kernels, portable reductions, atomic operations, loop tilling, thread-local data, GPU shared memory etc. It also provides building blocks that extend the *parallel for* idiom. Existing C++ applications can adopt RAJA's programming abstractions to expose existing parallelism without complex algorithm rewrites [58]. RAJA supports multiple execution back-ends, including sequential, SIMD, Nvidia CUDA, OpenMP (CPU and device offload), AMD HIP, and SYCL (in development) [59]. RAJA portability suite consists of RAJA, Umpire, CHAI, and CAMP. Together they insulate HPC applications from the complexities of handling device architecture and enables programmers to port an existing application to multiple platforms in a single-source application code [60]. RAJA uses standard C++11 features such as lambda expressions. RAJA has similar goals and concepts found in other C++ portability abstraction models such as Kokkos. The CHAI library implements *managed array* abstraction to transfer data automatically at run time as needed to run kernels. CHAI also provides *managed pointer* that simplifies the use of virtual class hierarchies between the host and device memory systems. It relies on the Umpire package. The Upire package provides a unified, portable memory management API focused on systems with heterogeneous memory resources. The Concepts and Meta-Programming (CAMP) library ensures wide compiler compatibility across HPC systems.

4 Towards iterative relational algebra

Data management and manipulation in relational databases are made possible by the mathematical system known as relational algebra (RA). Relational algebra primitives, such as union, join, projection, aggregation,

rename, and selection, provide the fundamental operations required for manipulating and querying relational databases. Parallel computing is now necessary to efficiently execute data-intensive activities as dataset sizes keep expanding. An extension of relational algebra called parallel relational algebra enables larger-scale parallelization of operations on distributed data systems. Segmenting the data into smaller groups that may be handled simultaneously on several processors, this method reduces the processing time as a whole. Faster query processing speeds, improved scalability, and shorter execution times for data-intensive applications are some potential advantages of parallel relational algebra. To get the best performance possible in parallel computing environments, however, issues like load balancing and communication costs should be addressed. In this section, we first explore different relational algebra primitives (Section 4.1), then we talk about parallel join operation (Section 4.2) both in CPUs (Section 4.3) and GPUs (Section 4.4). Finally, we compare popular join algorithms and find their limitations (Section 4.5).

4.1 Relational algebra primitives

Primitives in relational algebra differ from those in traditional set theory. We are going to give a brief introduction to the key relational algebra primitives from the perspective of two flat relations, R and S, which have a fixed arity [20].

- *Union*: The union of the relations R and S (should have the same arity) can be expressed as:

$$R \cup S \triangleq \{(r_0 \cdots r_n) | (r_0 \cdots r_n) \in R \vee (s_0 \cdots s_n) \in S\}$$

- *Intersection*: The intersection of the relations R and S (should have the same arity) can be expressed as:

$$R \cap S \triangleq \{(r_0 \cdots r_n) | (r_0 \cdots r_n) \in R \wedge (s_0 \cdots s_n) \in S\}$$

- *Cartesian project*: The Cartesian project of the relations R and S can be expressed as:

$$R \times S \triangleq \{(r_0 \cdots r_p, s_0 \cdots s_q) | (r_0 \cdots r_p) \in R \wedge (s_0 \cdots s_q) \in S\}$$

- *Rename*: The *rename* (reordering of columns) primitive is a unary operation. Rename of two columns i and j of the relation R can be expressed as:

$$\rho_{(i,j)}(R) \triangleq \{(\cdots r_i \cdots r_j \cdots) | (\cdots r_i \cdots r_j \cdots) \in R\}$$

- *Selection*: The *selection* is also an unary operation that selects a specific set of columns from the relation. Any number of columns can be chosen in the view using *selection* primitive. For example, selecting a column i from relation R can be denoted as:

$$\sigma_i(R) \triangleq \{(r_{i_0} \cdots r_{i_n}) \in R\}$$

- *Projection*: The *projection* is a unary operation that retains only mentioned columns from a relation. Project of two columns i and j of the relation R can be expressed as:

$$\Pi_{(i,j)}(R) \triangleq \{(r_{i_0} \cdots r_{i_n}) | (r_0 \cdots r_n) \in R\}$$

- *Natural join*: The natural join operation on relation R and S for some columns can be expressed as:

$$R \bowtie S \triangleq \Pi_{R \cup S}(\sigma_{R_{a_0}=S_{a_0} \wedge R_{a_1}=S_{a_1} \wedge \cdots \wedge R_{a_n}=S_{a_n}}(R \times S))$$

  where $R_{a_0}, \ldots, R_{a_n}$ and $S_{a_0}, \ldots, S_{a_n}$ are the columns in R and S that are being joined on.

4.2 Parallel join operation

In a distributed computing context, performing the join operation simultaneously on a number of processors or machines is referred to as parallel joining. When the quantity of the input data is enormous and the join operation is computationally costly, this technique is quite helpful. Assigning each partition to a different processor enables them to carry out the join operation on each partition simultaneously. Parallel join methods divide the data into partitions. The processing time for the join operation can be greatly slashed with the help of parallel join, which makes use of the parallel processing capacity of numerous machines. Thoughtful consideration of data segmentation, load balancing, communication overheads, and fault tolerance are necessary for designing and implementing parallel join algorithms, which can be difficult. Implementing the join primitive is challenging due to the uncertain output size, which is determined by the input data properties. Furthermore, efficient joins necessitate sorting or indexing the relationships.

4.3 Join on CPUs

The processing of massive amounts of data is a challenging task, and radix hash joins and sort-merge joins have been widely discussed and compared in previous studies. Modern HPC architectures have led to advancements in distributed join algorithms that leverage methods such as workload balancing, data partitioning, and communication optimization.

4.3.1 Radix hash joins (HJ) and sort-merge joins (SMJ)

Barthels et al. discussed and compared two popular methods for executing join; radix hash joins, and sort-merge joins on distributed multi-core systems [61]. Using 4096 processor cores, they scaled their implementations and used MPI as the communication layer. The focus of this paper is on overcoming the difficulties associated with implementing advanced, distributed radix hash and sort-merge join algorithms at magnitudes that are typically limited to massively parallel scientific applications or large map-reduce batch jobs. This study provided several significant insights into the challenges of implementing state-of-the-art distributed radix hash and sort-merge join algorithms at scales typically reserved for massively parallel scientific applications or large map-reduce batch jobs. One of the key findings was that achieving optimal performance necessitates maintaining the appropriate balance between computational and communicative capabilities. Merely increasing the number of cores in a compute node may not always result in improved performance, as it can also lead to a deterioration of performance. Additionally, although both join algorithms can scale effectively to thousands of cores, communication inefficiencies can significantly impede performance. This suggests the need for further research to optimize the full capacity of large systems. The paper also highlights the fact that hash and sort-merge join algorithms exhibit distinct communication patterns that involve different communication expenses. Hence, scheduling communication between compute nodes is a crucial component. The authors indicated that the sort-merge join implementation achieves its maximum performance. Conversely, the radix hash join falls far short of its theoretical peak, yet it still outperformed the sort-merge join, confirming prior research comparing hash-based and sort-based join algorithms. It can be highlighted that the performance of multi-core distributed join algorithms is bounded by the relatively low computation capacity of CPUs in comparison with GPUs and high inter-core communication costs [62].

4.3.2 Fault tolerant hash joins

In another study, Nasibullin and Novikov introduced a fault-tolerant hash-join for relational database systems that used data replication and synchronized alive signal to detect and recover from communication, system or transaction failure [63]. Though their algorithm cannot outperform state-of-the-art distributed hash join algorithms in a failure-free situation, it can handle a single point of failure with a keeper-worker model. Though their proposed solution can be criticized for data skew characteristics and lack of using accelerators to speed up the execution.

4.3.3 Advancements in distributed join algorithms

One of the recent advancements in distributed join on multi-node systems is the development of new parallel join algorithms that can leverage the benefits of modern HPC architectures, including multi-threaded

multi-node systems and high-speed networks. These algorithms take advantage of methods like workload balancing, data partitioning, and communication optimization to provide high-performance distributed joins on massive amounts of data. A novel hash-tree-based join algorithm was introduced towards the advancements of distributed relational algebra [20]. This study demonstrated a scalable implementation of a fixed point algorithm (transitive closure) using distributed relational algebra on 32k processes with a graph size of 276B edges. This massive-scale computation was possible with the use of communication-intensive *MPI_Alltoallv* collective operation and scalable bucket-based hash tree join algorithm. They identified strong scaling for hash-tree union and join operations using distributed relational algebra. Further improvement is possible by applying adaptive load balancing and mitigating memory overflow errors if too many tuples are generated on a single iteration.

4.3.4 Mitigating load imbalance in multi-node joins

In a recent investigation, load imbalance issues were identified and tackled with regard to parallel relational algebra kernels, focusing specifically on fixed-point iterative transitive closure computation [22]. The study examined two types of load imbalance issues: spatial load imbalance, which occurs across multiple computing nodes, and temporal load imbalance, which arises from distributing newly discovered tuples in transitive closure computation. To address spatial load imbalance, the authors proposed *bucket refinement* and *bucket consolidation* techniques that can be employed to attain dynamic load balance across MPI processes. The *bucket refinement* phase was used to check if some subbuckets contained more tuples than average and refined their size to allow allocating the extra tuples. On the contrary, the *bucket consolidation* was an additional phase that ensured the refined subbuckets were combined together when the spatial load balance was resolved. To address temporal load imbalance, they introduced a buffering strategy and named it *iteration rollover*. This technique prevented memory overflow errors when a single iteration was tasked with processing too many tuples. Figure 4 shows the addition of *iteration rollover* step during the join operation. An all-to-all communication is initiated during the local-join phase once a particular threshold value is reached. This communication phase is followed by a local insertion phase where the tuples are inserted into the appropriate destination rank. The previous iteration continues precisely where it left off, as opposed to coming to an end. Next, it checks if the fixed point iteration is reached and if not it reinstates the intra-bucket communication. The *iteration rollover* step serves as a protective measure against significant instances of temporal imbalance that could decelerate evaluation due to disruptive memory swapping or cause a process to terminate. By varying the number of processes from 256 to 32,768, the efficacy of both solutions was evaluated both in combination and independently. From this study, we see that, for a specific workload, there exists a window of processes that demonstrates favorable scaling. However, once this threshold is surpassed, performance deteriorates due to less workload per process and additional communication expenses.

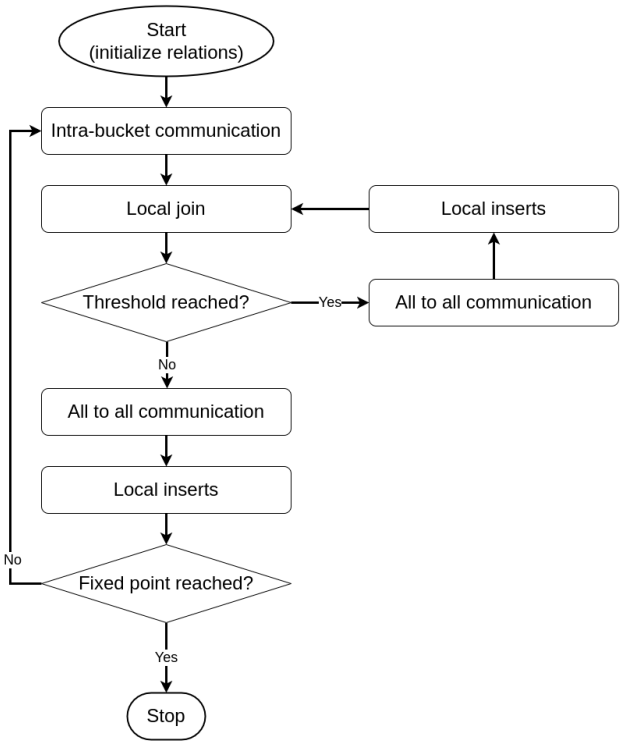


Figure 4: Handling temporal load imbalance during the join process using iteration rollover technique [22]

4.4 Join on GPUs

As we have discussed so far, the join operation forms a critical part of relational algebra that is used to merge data from two or more relations based on a column or set of columns. As GPU devices provide thousand of threads, the join operation can utilize this massive parallel architecture to process vast amounts of data in a more efficient and faster manner than a traditional CPU-based join operation [32]. By dividing the data into smaller subsets that are handled in parallel across many cores, the GPU-based join operation accomplishes this. This approach utilizes hardware resources suitable to perform computation faster using the massive number of threads. Moreover, GPUs usually have a higher memory bandwidth and a more extensive memory hierarchy, which enhances data access and reduces data transfer time. The GPU-based join operation is, therefore, an excellent tool for handling large-scale data processing tasks in relational algebra, offering significant improvements in performance and scalability.

4.4.1 Red Fox and LogiQL

Wu et al. presented a system named *Red Fox* that harnesses the power of GPUs to parallelize relational algebra operations [24]. This system serves as a high-performance accelerator core for a type of Datalog solver known as LogiQL. Red Fox can use GPU accelerators to run LogiQL's queries in conjunction with the CPU host support. After parsing and analyzing a LogiQL program, the language front-end generates an intermediate representation (IR) of the query plan. This IR represents the sequence of operations involving relational algebra and arithmetic operators that need to be executed. The RA-Kernel compiler takes this IR and creates executable CUDA implementations. These implementations are then converted into the Kernel IR and can be executed as a binary executable. The study found an order of magnitudes in computing speedup for carrying out tasks from a declarative language in comparison with the multi-threaded CPU-based implementation. In this study, the RA primitives are carried on a single-GPU environment that can fuse RA operations, such as fusing multiple joins in a single step. But they did not investigate the deduplication of tuples which had a greater impact on performance in relational join operation [20]. Moreover, they did not maintain the join result in sorted order which is required in the case of iterated relational algebra. If the join results are sorted, it can avoid costly operations like sorting the entire result at each iteration of the fixed-point iterations, which impacts the overall performance. Wu later introduced GPU-optimized multiple-predicate join that was not limited by sorting and exhibited significantly improved performance compared to binary joins [35]. Additionally, he ported a multi-predicate join algorithm that is worst-case optimal for CPU to the GPU architecture.

4.4.2 Multi-node relational learning framework using GPU parallelism

Martínez-Angeles et al. presented a relational learning framework for expediting rule coverage on GPUs to extract rules from healthcare records data [26]. They showed performance improvement for a 75% of the applications over multi-core CPU systems. But their work did not take the duplicate tuples generated from join operations efficiently, and the GPU memory management also fell short in terms of managing larger relation size. Another study designed a parallel functional control-flow analysis (CFA) encoded in Datalog utilizing highly parallel relational algebra as the foundation using GPU architecture [29]. They extended the work from *Red Fox* to enable executing fixed-point iteration. On top of that, they combined the GPU parallelism with mutli-node multi-core HPC systems with the goal of parallelizing RA primitives across multi-nodes. They found the join operation on GPU device scaled well using extended *Red Fox* and performed with similar efficiency to Soufflé. To further improve the performance, they proposed to use the Partitioned global address space (PGAS) programming model suitable for HPC systems [64]. A global address space can be divided using PGAS, and each partition can be given to a specific system processing component. With this method, shared memory access is possible between multiple processes without any complicated communication protocols. Each processing component in a PGAS system has access to both its local memory and a portion of the system's global memory. Without explicit message passing or data copying across processes, this enables direct access to data stored in remote memory locations.

4.4.3 Assessing join algorithms on GPUs

Rui et al. assessed the efficacy of identical join algorithms such as block-based Non-Indexed Nested Loop Join (NINLJ), Indexed Nested Loop Join (INLJ), Sort-Merge Join (SMJ), and Radix Hash Join (HJ) proposed in a prior study [30] on modern GPU devices [33]. The authors contend that leveraging new methodologies provided by modern GPUs can lead to even greater enhancements in join operation performance (up to 20× speedup in comparison with 7× speedup in the original study). Specifically, the study finds that SMJ outperforms HJ when executed on GPUs as the radix partitioning phase is limited to $16M$ tuple size. However, as their implementation is designed for single-GPU architecture, they are not suitable for HPC systems with multiple GPU environments. Moreover, another seven years have passed since this study and the GPU architecture has gone through rapid modifications during this time, thus raising the question of redesigning the join algorithms targeting recent GPU architecture (e.g. Nvidia Hopper architecture) [65].

4.4.4 A hybrid join algorithm for GPUs

Guo et al. proposed a parallel hybrid join algorithm (PHYJ) by combining SMJ with HJ join algorithms alluding to GPU architecture [66]. They introduced a pipeline technique that is able to reduce host-to-device and device-to-host bidirectional communication costs by fusing data communication with GPU execution on setting the input and output relation [67]. On a single GPU evaluation, their hybrid algorithm achieved up to 1.72× speedup over the state-of-the-art join algorithms. One of the fundamental improvements of the hybrid PHYJ algorithm is the handling of skewed data. However, the paper does not provide information on whether the proposed algorithm can be applied to multiple GPUs or distributed systems.

4.4.5 Multi-GPU join algorithms for GPUs

Single GPU-based join implementations cannot handle large input relations due to the limitation of the GPU global memory. Thus, multi-GPU implementation has brought a solution to the scalability issue of the join primitive [34]. They identified the low transfer rate between CPU and GPUs and complex data generation pattern in join operation as the main challenges in designing efficient algorithms targeting multiple-GPUs. To address these challenges, the authors proposed three algorithms: nested loop, global sort-merge, and hybrid joins which are suitable for different cases of multi-GPU join algorithms. The research indicated that the algorithms exhibit strong scalability, and utilizing multiple GPUs results in noteworthy enhancements in performance (up to 25× and 2.8× over state-of-the-art multi-core CPUs and multi-GPUs implementations correspondingly). The authors did not discuss one of the key aspects of parallel join, which is removing duplicates immediately while they are generated in the iterative process. In another study, Paul et al. presented *MG-Join*, which is a single node multi-GPU partition-based hashjoin [36]. To reduce the communication cost between multiple GPUs, they proposed a novel routing technique that dynamically chose the route to reduce congestion. Both of these studies failed to discuss the design principle of multi-GPUs join with the multi-node architecture of current supercomputing systems.

4.4.6 WarpDrive: a single-node multi-GPU hashing implementation for hashjoin

Jünger et al. presented WarpDrive, which showed that even though it requires more arithmetic operations, attaining better memory coalescing for accesses to GPU DRAM may enhance performance [27]. They discussed the limitations of single-GPU hashing implementations due to the limitation of global GPU memory, which bounded the size of hash maps that can be supported. To address this issue, they proposed a scalable and distributed single-node multi-GPU implementation that supports the construction and querying of billions of key-value pairs. They provided a novel subwarp-based probing scheme that enabled coalesced memory access, mitigating the high latency associated with irregular access patterns. The authors reported achieving $1.4B$ insertions rate per second in single-GPU and it increased up to $4.3B$ insertions on multi-GPUs. This study outperformed CUDPP library by 2.8×. But this implementation did not provide any method to handle duplicate values [68] and was limited to *32 bit* single value hashtables. In a follow-up to WarpDrive, they presented WarpCore, a framework that enabled the building of purpose-built GPU hash tables which supported 64 bit keys but still suffered from the deduplication issues [69].

4.4.7 Multi-core multi-GPUs Leapfrog Triejoin with secondary storage

Zinn et al. explore the use of general-purpose join algorithms on multi-core multi-GPUs heterogeneous systems focusing on triangle counting problem [25]. They also proposed a novel technique to handle large relations as input which cannot fit in CPU RAM or GPU's global memory but rather be placed in a secondary solid-state disk (SSD) storage. Specifically, they investigated on Leapfrog Triejoin (LFTJ) algorithm, which is a worst-case-optimal algorithm. The authors introduced a novel technique called *boxing* for partitioning and feeding out-of-core input data to the LFTJ algorithm. The results show that this approach reduces I/O cost and demonstrates competitive performance on heterogeneous systems. The proposed framework is a complete query engine that can run any relational join, making it useful beyond just triangle counting. However, the paper did not discuss how duplicate tuples are handled during the join operation. Moreover, they did not address the optimization needed to apply LFTJ on multi-node systems.

4.5 Comparison of join algorithms

Algorithm	**Type**	**Pros**	**Cons**	**Dataset size**
NINLJ [30]	Nested Loop	Simple to implement	High time complexity, low efficiency	Small-sized tables
INLJ [34]	Nested Loop	Uses indexes for faster performance	High memory usage, high time complexity	Small to medium tables with index
SMJ [33]	Divide and Conquer	Scalable, handles non-equality joins	Requires sorting, high memory usage	Medium-sized tables
HJ [61]	Hash	Efficient for equality joins, scalable	Requires partitioning, not suitable for non-equality joins	Large tables with equality join
LFTJ [25]	Trie	Efficient for complex queries, scalable, handles non-equality joins	High memory usage, complex implementation	Large tables with complex queries
PHYJ [66]	SMJ, HJ combined	Reduce communication cost, can handle skewed data	Complex implementation, requires tuning	Large tables with complex queries

Table 1: A comparison of different algorithms for joining data in parallel

Parallel join algorithms are used in parallel relational algebra to efficiently combine data from multiple relations, which is a fundamental operation in many data processing tasks. Table 1 presents a useful comparison of various algorithms for joining data in parallel. The algorithms are grouped by type and their respective pros and cons are outlined. Additionally, the table indicates the optimal dataset size for each algorithm. For example, the nested loop join (NLJ) is a simple algorithm to implement, but it is not particularly efficient for large datasets. Indexed nested loop join (INLJ) improves upon the performance of NLJ by using indexes, making it a suitable choice for small to medium-sized tables with an index. Sort merge join (SMJ) is a scalable algorithm that can handle non-equality joins, but requires sorting, which can be costly for larger datasets. Hash join (HJ) is efficient for equality joins and does not require sorting, but it does require partitioning, which can be challenging to implement. Leapfrog trie join (LFTJ) is a scalable algorithm that can handle non-equality joins, but its implementation is complex and it requires high memory usage. Hybrid phyj join (PHYJ) is a combination of SMJ and HJ that can reduce communication costs and handle skewed data, but its implementation is complex and requires tuning. Ultimately, choosing the best algorithm for a particular dataset depends on various factors, such as dataset size, number of join columns, and available hardware resources. However, there is a growing need for more effective parallel join algorithms that can handle even larger datasets as the amount of data being processed grows and the need for faster

computation becomes more urgent. An effective parallel join algorithm based on iterated relational algebra that is appropriate for multi-GPU systems in heterogeneous supercomputers is particularly needed. Such an algorithm would be a helpful tool for a variety of data-intensive applications since it would enable more swift and effective declarative analytics on high-dimensional data.

5 Datalog

Datalog is a declarative logic programming language that uses a lightweight bottom-up approach [70, 5, 6]. Its primary use case is in deductive-database systems, where it enables users to define problem rules and outline solutions with easily understandable queries [7, 8]. Datalog programs present queries as first-order Horn-clause rules (also known as the intentional database), and the program extends input database data to create an output database with all data that is derivable using the program rules (also known as the extensional database) [31]. Datalog's high-level, artificial intelligence-based programming techniques make it an ideal choice for non-experts in various domains, including big-data analytics [9, 10, 11], machine learning [12], software analysis [13, 14, 15], graph mining [16], and deductive databases [17]. Declarative languages such as Datalog enable seamless linkage of human-oriented problem formulation with efficient machine implementation.

When a Datalog program is executed, it makes the intentional database (output) concrete by inferring it explicitly. This intentional database supplements the extensional database with all the facts that can be deduced through the program's rules in a transitive manner. Figure 5 shows the execution strategy of Datalog programs that extends data from input data creating the output database with all data transitively deliverable via the set of program rules defined in the program. Here's an example of a Datalog rule with an explanation that represents an advisor-student relationship:

Figure 5: Datalog execution strategy

$$advisor(X, Y) :- teaches(X, C), enrolled(Y, C).$$

In Datalog, rules are written in the form of *head :- body*, where the *head* is a predicate and the *body* is a conjunction of predicates or literals. This rule states that for any persons X and Y, if X teaches a course C and Y is enrolled in course C, then X is the advisor of Y. In this rule, *advisor(X,Y)* is the head predicate, which defines the relationship between X and Y as advisor-student. The body of the rule contains two predicates: *teaches(X,C)* and *enrolled(Y,C)*. These predicates represent the conditions that must be satisfied for the rule to be true. Specifically, the rule requires that X teaches a course C and Y is enrolled in the same course. Using this rule, we can infer the advisor of a student based on their enrollment in a course and the instructor of that course. For example, if we have facts such as *teaches(Bob,CS101)* and *enrolled(Alice,CS101)*, we can infer that *advisor(Bob,Alice)* is also true. This allows us to query the database and retrieve information about advisor-student relationships that were not explicitly stated in the original facts.

The remaining part of this section delves into the revival of Datalog in recent years (Section 5.1), explores various Datalog implementations (Section 5.2), and provides a comparative analysis of these solvers, highlighting their respective limitations (Section 5.3).

5.1 Revival of Datalog

Although there was significant interest in Datalog in the database systems community during the 1980s and early 1990s, a perceived lack of useful applications at the time resulted in Datalog research becoming dormant for an extended period [71, 72]. A brief timeline of how Datalog evolved over the years is shown in 6. Datalog was introduced through a series of papers and workshops in the 1970s. The concept of Horn clauses, which formed the basis of Datalog, was introduced by John McCarthy in 1970. Robert Kowalski's paper in 1972 introduced Prolog, a logic programming language that is based on Horn clauses. David Maier

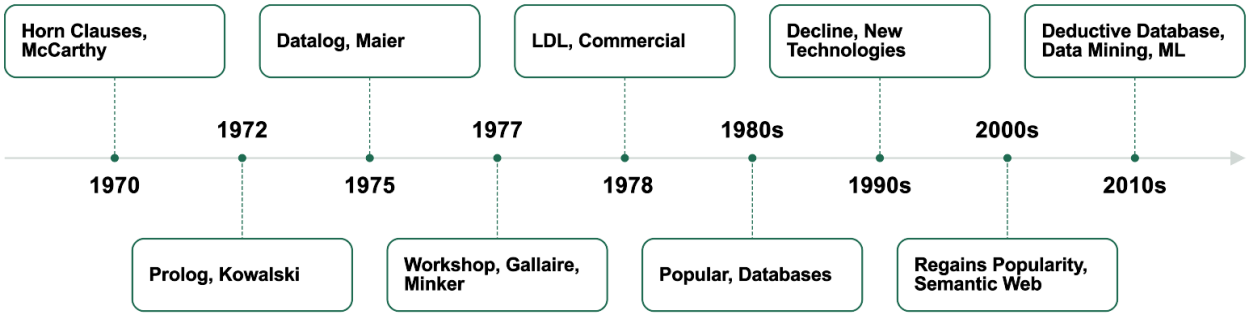


Figure 6: A timeline of Datalog's evolution over the years

coined the term *Datalog* in 1975. After a couple of years, in 1978 the first commercial version of Daatalog called LDL was released, and the language became increasingly popular in the 1980s. During the 1900s, its popularity declined due to the introduction of new database technologies, such as object-oriented databases and relational databases with triggers. In the 2000s, Datalog regained its popularity due to its simple expression nature and efficiency. Maier et al. provided an overview of the history and main concepts of Datalog, including its use for database definition and querying [17]. They explored the language's roots in logic languages, databases, artificial intelligence, and expert systems, and considered extensions such as constraints, updates, and object-oriented features. They also discussed approaches to Datalog evaluation as well as early implementations of Datalog and similar deductive database systems. They outlined the reasons for the decline and revival of interest in the language and provided several current extensions of Datalog. Ceri et al. presented an in-depth discussion on Datalog [5]. They discussed its syntax and semantics, and usage for relational databases. They presented the taxonomy of optimization methods to gather efficient evaluations of Datalog queries such as top-down approach, bottom-up approach, magic sets (logical rewriting), and counting (algebraic rewriting) technique. Additionally, they discussed several enhancements of Datalog to extend Datalog's applications in practical aspects. In recent years several novel applications, including data integration, declarative networking, program analysis, information extraction, network monitoring, security, and cloud computing, have seen an increase in their use of Datalog [73]. The use of Datalog as a high-level abstraction for querying relational structures and graphs, as well as the implementation of incremental view maintenance and recursive query execution based on the relational model, formal reasoning, and analysis, are shared characteristics of these systems [74]. Most of the application domains use the base of Datalog language, and it is modified and improved to suit the needs of the domain.

5.2 Modern implementations of Datalog

Modern implementations of Datalog have been optimized for efficiency and scalability, making it a popular choice for applications such as data analytics and knowledge representation. Most of these implementations support multi-core systems. In this section, we discuss a few modern implementations of Datalog.

5.2.1 LogicBlox

Aref et al. showed the gradual advances in logic programming and presented LogiQL language under their developed LogicBlox system for business analytics based on Datalog [4]. This extensive study shifted the focus of Datalog research back to semi-naïve evaluation targeting high-performance shared-memory data structure. The semantics of LogiQL is easy to understand and does not use nulls which causes hard-to-spot mistakes in query languages. They introduced a mature product (LogicBlox) and integrated with enterprise applications that earned revenue of over $300B. This system proved to generate massive financial benefits by using Datalog-like declarative language for business analytics.

5.2.2 Soufflé

Soufflé is a recent implementation for Datalog that utilizes a pragma directive-based OpenMP programming model utilizing parallelism from multi-core CPU systems using multiple threads [18]. It is marked as a

state-of-the-art Datalog implementation for multi-core CPU systems that relies on single-node. As it relies on a single-node it supports only a limited number of threads for task-level parallelism. Moreover, its scaling is impeded by internal locking [75]. It also faces challenges due to its coarse-grained approach to parallelism. Despite being considered a state-of-the-art analysis platform, Soufflé has a fundamental limitation in that it lacks the ability to provide data parallelism [76]. This limits its ability to operate beyond a single node, and therefore may not be ideal for large-scale data processing and reasoning tasks that require distributed computing.

5.2.3 RadLog

RadLog, which is also commonly referred to as BigDatalog, has introduced a proposal for the scaling of deduction on multi-thread machines and clusters. This is achieved through the utilization of Hadoop and the map-reduce paradigm for distributed programming [11]. The proposal aims to allow for more efficient and effective processing of data, especially for larger data sets that require a distributed computing approach. With this proposal, RadLog aims to provide a more practical and scalable solution for deduction in multi-thread environments, leading to improved performance and faster processing times. The limitations of mapreduce algorithms have become increasingly apparent as they suffer from a bottleneck in many-to-one collective communication, particularly in a hierarchical system. This has led to a realization that they are inadequate for high-performance parallel-computing environments [77]. Thus, RadLog is proven not to be suitable for recent heterogeneous supercomputers, and in a particular experiment, it performs 1000× slower than modern data level parallel Datalog implementation [76].

5.2.4 PRAM

Parallel Relational Algebra Machine (PRAM) is the first data-parallel Datalog solver that leverages parallel relational algebra with a compiler that generates its intermediate representation (IR) [76]. The paper presents a methodology for constructing data-parallel deductive databases that leverages recent advancements in parallel relational algebra. The approach creates state-of-the-art data-parallel semantics for Datalog. This is the first Datalog solver that utilizes Message Passing Interface (MPI) to develop a data-parallel Datalog solver. The experiments show 1.7× performance enhancement for a single-node architecture on Theta supercomputer compared to Soufflé with a 88% scaling capacity for 64 cores CPU. Though they cannot achieve as performance gain as Soufflé on Amazon Web Services (AWS), they showed significant improvement in scalability for an increment of the CPU cores. They view their methodology as a significant advancement towards the implementation of high-performance logical-inference engines.

5.2.5 SLOG

Gilray et al. introduced SLOG, a high-performance implementation of Datalog that provides a compiler, REPL, and runtime [75]. The compiler is written in Racket that translates the SLOG's source code to C++ code that utilizes a parallel relational algebra backend alongside generating incremental intermediate representations. SLOG uses PRAM as relational algebra backend. The study shows the steps of an iterated transitive closure (TC) computation using SLOG's backend. It includes three steps: local join, all-to-all communication between processes, and listing of the new facts in the proper process. The workload is distributed among the processes, and relational algebra primitives are executed in parallel. The new facts from local joins are transferred to their managing processes using an all-to-all communication. The last step checks for duplicate tuples, and if new tuples are found, they are inserted. The fixed-point iteration is stopped in case of no new tuples in any iteration.

This is the outcome of an incremental approach toward developing an MPI-based multi-node parallel implementation for Datalog [20, 21, 22]. Though Soufflé outperforms SLOG for lower core counts (<60 cores), SLOG outperforms both Soufflé and RadLog for higher core counts in a single-node experiment with better scalability. SLOG opens new opportunities to run data-parallel Datalog on multi-node HPC clusters enabling massing parallel analytics. Though it does not appear to utilize the massive computing power from GPU devices in heterogeneous supercomputing systems. Further development can be done in the direction of developing a multi-node multi-gpu based parallel relational algebra backend for SLOG.

5.2.6 Datalog on GPU

Martínez-Angeles et al. presented a single GPU based Datalog engine that utilized CUDA programming model [78]. They outperformed non-GPU based systems for join and transitive closure computation. They illustrated the organization of the Datalog engine on GPU. The architecture of their GPU-based Datalog engine utilizes a number of GPU kernels. During the rule assessment process, the selection and self-joining kernels are initially employed for each subgoal pair to eliminate unnecessary tuples immediately. The subsequent step involves the use of join and projection kernels. After the completion of rule evaluation, the kernels designed for deduplication process are invoked. They achieved up to 40× performance enhancement in compare with single CPU based system for transitive closure computation. They also identified the deduplication process at each iteration as the most costly operation in transitive closure computation at the first set of iterations, and in subsequent iterations, the join became the most costly operation. The drawback of this single-gpu based implementation is the limitation of relation size as it cannot manage relations larger than the total memory of GPU global memory. As this is based on in-memory systems, the performance is also restricted by the available GPU memory [79]. Though they discussed a few aspects of memory management, there was no mention of whether their implementation could be applicable to distributed platforms. Modern heterogeneous systems contain heterogeneous computing environments where a large number of CPUs are connected with a large number of GPUs with a high-speed interconnected network. It arises the need for the development of a Datalog backend that can take advantage of the massive multi-GPU parallelism in a multi-node environment.

5.3 Assessment of Datalog solvers

Solver	**Technique**	**Data Parallelism**	**Scalability**	**Performance**	**Limitations**
LogicBlox [4]	Shared-memory	No	High	Optimized for shared memory	Limited number of threads for task-level parallelism
Soufflé [18]	Multi-core	No	Limited	Optimized for single node shared memory	Impeded scaling due to internal locking
RadLog [11]	MapReduce	Yes	Limited	Optimized for MapReduce based distributed system	Inadequate for high-performance parallel-computing environments
PRAM [76]	Data-parallel	Yes	High	Optimized for multi-core multi-node systems	Cannot achieve as much performance gain as Soufflé on AWS
SLOG [75]	Data-parallel	Yes	High	Optimized for multi-core multi-node systems	No use of GPU parallelism

Table 2: Comparison of modern Datalog implementations

Table 2 compares the state-of-the-art Datalog solvers. The table shows a comparison of various modern implementations of Datalog, a logic programming language commonly used in data analytics and knowledge representation. The implementations discussed include LogicBlox, Soufflé, RadLog, PRAM, and SLOG. The table presents a comparison of various Datalog solvers based on their technique, data parallelism, scalability, performance, and limitations. LogicBlox is optimized for shared memory and has high scalability, but its task-level parallelism is limited due to a restricted number of threads. Soufflé is optimized for single-node shared memory, but its scaling is limited due to internal locking. RadLog is suitable for distributed systems and optimized for MapReduce, but it is inadequate for high-performance parallel-computing environments. PRAM and SLOG are optimized for multi-core multi-node systems and offer high scalability with data-parallelism. However, PRAM cannot achieve the same performance gain as Soufflé on AWS, and SLOG does not utilize GPU parallelism. Ultimately, the choice of Datalog solver depends on the specific requirements

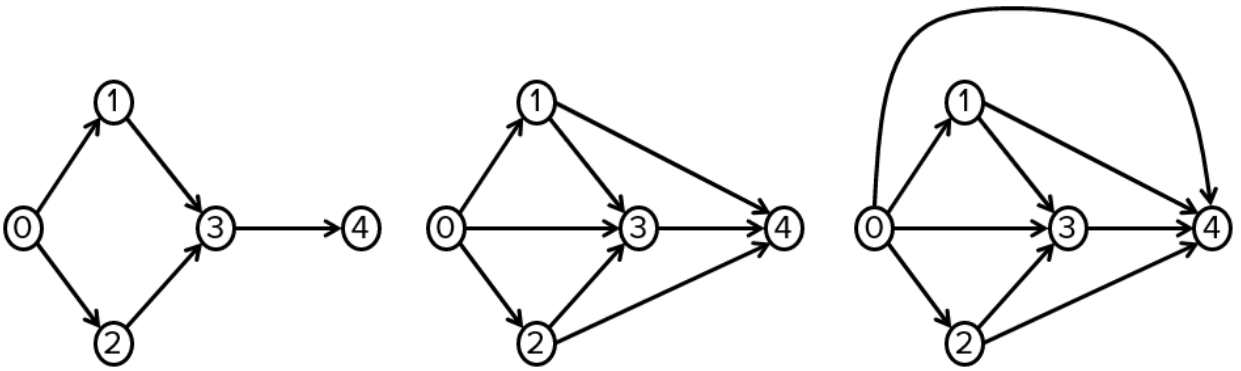
of the use case, including the hardware environment and the size and complexity of the dataset. Given the limitations of current Datalog solvers, it is clear that a multi-GPU, multi-core Datalog solver is needed to address the challenges of processing large and complex datasets. This would enable efficient data parallelism, high scalability, and optimized performance across heterogeneous exascale computing systems.

6 Declarative analytics using Datalog applications

Declarative analytics is a method of data analysis that turns complex computations into logic rules by using declarative languages like Datalog. Due to its simplicity and expressiveness, Datalog in particular has been drawing attention as a potent tool for declarative analytics. Users of Datalog can define logical rules and queries that describe the connections between different data pieces, and the system will automatically infer the outcomes. This method has a number of advantages, including higher productivity, clearer code, and portability because the same query may be executed on many platforms with little to no changes. Furthermore, the declarative nature of Datalog makes it simple to optimize and parallelize queries, improving both performance and scalability. Declarative analytics utilizing Datalog is thus gaining popularity across a variety of industries, including data analytics, machine learning, and graph analytics, to mention a few.

This section begins with a formal introduction to the most common Datalog applications, transitive closure computation (Section 6.1) and triangle counting (Section 6.2). Subsequently, we examine existing studies that explore the development of transitive closure computation on the path to using Datalog as a declarative analytics engine (Section 6.3).

6.1 Transitive closure computation using Datalog



0	1
0	2
1	3
2	3
3	4

0	1
0	2
1	3
2	3
3	4
0	3
1	4
2	4

0	1
0	2
1	3
2	3
3	4
0	3
1	4
2	4
0	4

Figure 7: Iterated transitive closure computation [20]. The top row represents the incremental path finding of a directed graph. The bottom row shows the newly found paths in the edgelist. The left side represents the original graph. Middle portion shows the new paths in one hop distance and the right portion depicts the new paths of two hop distance. The iteration stops as no new paths can be found.

Efficiently computing transitive closures is a crucial subproblem in the efficient computation of general recursive queries. Thus, transitive closure computation is regarded as an ideal application of logic programming languages such as Datalog. The recursive Datalog program rules presented below compute the transitive closure, denoted as A, of a given graph B [80]:

$$A(p,q) \text{ :- } B(p,q).$$
$$A(p,r) \text{ :- } A(p,q), B(q,r).$$

The initial rule serves as a foundation, stating that any edge that connects vertex p to vertex q in the graph A implies that there exists a direct path from p to q. On the other hand, the second rule is recursive and requires multiple iterations until it reaches a stable state where the value for A conforms to all the rules. The iterative process for the second rule involves repeating it until A is complete. The first rule inserts all tuples from B into A. The second rule invokes a kernel that operates on RA primitives such as join, rename, projection, and union. This kernel is repeatedly invoked until no new reachable paths can be inferred. In a single iteration of the process, A and B on a common column q and list in all possible triples (p,q,r). In the next operation, the common column q is projected out, resulting in the unique set of (p,r) tuples. These tuples are later on union with the existing tuples in A and A is updated with newly discovered tuples. Figure 7 illustrates an example of transitive closure computation. The input graph contains five vertices and five directional edges (Figure 7 left column). At the first iteration, three new paths are discovered of length 2, which are - $(0,3)$, $(1,4)$ and $(2,4)$ (Figure 7 middle column). Later these paths are unioned with A. As the Datalog rules are recursive in nature and follow incremental evaluation, these newly discovered paths construct the input set for the next iteration [81, 82]. In the next iteration, it finds one new path of length 3 - $(0,4)$ and the result set A is updated accordingly (Figure 7 right column). The newly discovered paths then construct the input set for the next iteration. This process remains in operation until meeting a fixed point; in this case, no discovery of new paths with incremental distance. Thus, the transitive closure computation can be seen as a good use case of an iterative process that utilizes relational algebra primitives at each iteration with the opportunity to parallelize the work to gain performance enhancement.

6.2 Triangle counting using Datalog

Datalog provides a simple solution for triangle counting, which is a widely used graph-mining application for analyzing social networks. Despite the availability of several tailored algorithms for calculating triangles, they usually require a significant learning effort [83, 84]. In contrast, Datalog offers a straightforward approach to encoding triangle counting. The recursive Datalog program rules presented below compute the triangle counting of a given graph A [80].

$$2c(p,q) \text{ :- } A(p,q), p < q.$$
$$2c(p,q) \text{ :- } A(q,p), p < q.$$
$$T(p,q,r) \text{ :- } 2c(p,q), 2c(q,r), 2c(p,r).$$

The initial two rules generate a relationship called $2c$. These rules eliminate all self-looping vertices from the input graph A. They also sort the edges of the graph. During the compilation, these rules will be compiled down into a copy rule, including a filter operator and will generate each tuple of A into $2c$ while satisfying the condition $p < q$. The last rule will be translated to two relative join operations. To compute all triangles in the graph, the first join operation involves connecting the second column of the graph with the first column. This operation corresponds to joining the graph with itself, and it computes paths of length two. The output of this join operation is an intermediate relation that is then joined with $2c$ to compute all triangles. The resulting relation $T(p,q,r)$ represents all triangles in the graph, and it can be used to count the total number of triangles present in the input graph.

6.3 Datalog applications targeting heterogeneous systems

Leveraging the GPU parallelism capabilities of upcoming exascale systems can provide a significant opportunity for Datalog applications to improve performance using heterogeneous architecture. This can potentially be achieved by utilizing GPUs to accelerate Datalog queries, particularly for complex queries

and large datasets. However, developing specialized algorithms and implementations of Datalog that are optimized for heterogeneous exascale systems may be necessary.

A GPU-based scalable parallel algorithm was presented on solving the triangle counting problem using a novel list intersection algorithm called *intersection path* extending the *merge path* algorithm [23]. They used two types of parallelism; partitioning the input vertices and finding the triangles on those partitioned vertices. They achieved up to 32× speedup against state-of-the-art CPU solution. As they design the solution targeting a single GPU system, it is not possible to handle graphs that exceed the memory capacity of the GPU. In another study, Wang et al. proposed an all-source breadth fast search-based triangle counting solution leveraging massive GPU parallelism [85]. They pruned the invalid nodes during the joining phase, which reduced the computational work in subsequent phases. They claimed to outperform all distributed CPU-based implementations for the same task alongside the state-of-the-art GPU solutions. However, this is also a single GPU-based solution without any indication of its characteristics on multi-GPU systems popular with the current supercomputing systems.

Green et al. introduced a novel algorithm called Anti-section Transitive Closure (ATC) for finding the transitive closure of a sparse relation [28]. The algorithm utilizes Nvidia's GPU to distribute the computation across available GPU threads. The authors use dual-round and hash-based operations to discover the reachable paths from dynamic graph data structure using a novel edge operator called anti-section. The hash-based ATC surpassed the performance of the cutting-edge multi-thread CPU and single-GPU-based implementation. They also considered handling duplicated tuples with a justifiable memory overhead without deteriorating the performance. This study's main limitation is that it fails to identify any research direction for the implementation of multi-GPU systems, which is essential to implement a multi-node multi-GPU version of the transitive closure computation.

Recent studies indicate that it may be possible to develop Datalog implementations with multiple threads and cores [21, 22]. These implementations use multi-node multi-threaded CPU-based parallelism to enhance Datalog application performance. With the advent of exascale systems, which have the ability to leverage GPU parallelism, there is an opportunity to take advantage of the heterogeneous architecture to achieve even higher performance for Datalog applications. This can be achieved by utilizing GPUs to accelerate Datalog queries, which has the potential to improve performance for complex queries or large datasets. However, achieving this will likely require developing specialized algorithms and optimized implementations of Datalog that are designed for use with heterogeneous exascale systems.

In roads to develop a multi-node multi-GPU based backend for Datalog targeting the heterogeneous systems, a recent investigation utilized off-the-shelf Python dataframe both in CPU (Pandas) and GPU (cuDF) architecture to demonstrate the possibilities of using GPU parallelism for Datalog applications [80]. This investigation found up to 71× speedup for transitive closure computation using GPU-based cuDF library over CPU-based Pandas RA primitives APIs. In another experiment, it got up to 147× speedup for triangle counting of various types of relations using the same set of libraries. This study is limited by its single GPU architecture, which makes it unable to process relations that exceed the memory size of the GPU. Additionally, GPU kernel fusion cannot be utilized due to the sequential execution order of the RA primitives in the libraries. Despite these limitations, the significant performance improvements suggest that expanding this study to a multi-node multi-GPU architecture could be worthwhile.

7 Topological data analysis for high dimensional data analytics

The field of topological data analysis (TDA) is an evolving area that has recently been utilized in high-dimensional data analytics fields such as functional neuroimaging. Graph-analytics is a popular technique for studying Functional Connectivity Networks (FCN). However, more advanced TDA tools, such as persistent homology (PH), have recently been applied to the study of brain networks [**?**]. Functional Connectivity Networks (FCN) are obtained from Resting-State Functional Magnetic Resonance Imaging (rs-fMRI) by using measures of time series association such as Pearson's correlation. These networks are powerful tools for investigating brain connectivity patterns and are commonly used in neuroscience research. Multi-site fMRI investigations have become more popular during the last few years. This is so that faster participant recruitment and greater sample sizes, which increase statistical power, are made possible by such investigations [86, 87]. This holds particular significance when investigating rare brain diseases and

varied demographics. [88]. Multi-site rs-fMRI studies have the advantage of enabling faster participant recruitment and larger sample sizes [89]. However, FCNs are sensitive to data acquisition parameters such as sampling period, which can introduce non-neural variability. This non-neural variability is compounded when pooled data from different acquisition protocols and scanners are used, which can negate the advantages of larger sample sizes [90, 91]. Therefore, careful consideration of data acquisition parameters is necessary to ensure the accuracy and reliability of FCN studies. The topology of brain networks is an important aspect of understanding brain function and connectivity patterns. It is crucial to ensure that this topology is preserved, regardless of the sampling density used in data acquisition. Metrics that capture topology may be statistically similar across different sampling periods, reducing the impact of non-neural variability. Additionally, persistent homology (PH) has been shown to be robust to changes in data acquisition parameters, such as sampling period. This suggests that PH may be a useful tool for studying brain networks, even in cases where the data has been acquired under different conditions. Overall, preserving the topology of brain networks is crucial for accurate and reliable studies of brain connectivity, and it is important to consider metrics and methods that can reduce non-neural variability.

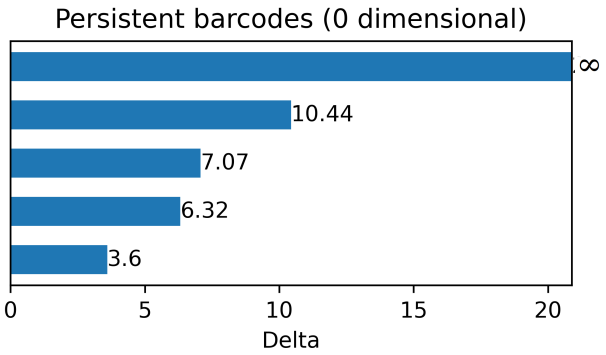
In this section, we explore the concept of persistent homology (Section 7.1) and how it can be represented through persistent barcodes (Section 7.2). To illustrate this, we provide an example of computing 0-dimensional barcodes (Section 7.3). Additionally, we investigate the potential applications of topological data analysis in the analysis of brain images (Section 7.4).

7.1 Persistent homology

Persistent homology is a method utilized in topological data analysis to examine how topological features evolve within a dataset [92]. It offers a means of extracting information regarding the shape and composition of complex data by gauging the endurance of homological features across a series of parameter values [93]. It is an emerging tool in studying complex networks, including brain networks [94]. PH-based methods have shown promising results in modeling transitions between brain states in fMRI data [95]. By implementing persistent homology, analysts can acquire knowledge of the fundamental structure of high-dimensional datasets, ranging from biological to social networks, and enhance comprehension of their operations [96]. Studying the connections between various components of brain networks is often carried out through the use of simplicial complexes [97]. These complexes represent topological spaces as collections of simplices, including points, line segments, and triangles, among others. By implementing simplicial complexes, researchers are able to detect topological characteristics and determine the significance of relationships across multiple thresholds [98]. The term topological features refers to the homology groups of a metric space that describe the connected components, tunnels, and voids in dimensions zero, one, and two, respectively [99]. Through analyzing these topological features, researchers can obtain insights into the structure of complex networks and the manner in which different parts of the network are connected. Employing simplicial complexes for the study of complex brain networks can provide valuable information that may be difficult or unfeasible to acquire using other techniques [100].

7.2 Barcodes and persistent diagrams

Persistent homology (PH) can be used to analyze graphs and networks. PH is a mathematical tool that tracks the creation and destruction of features in a graph by assigning each feature a level of significance or persistence. These births and deaths of features are represented in the form of barcodes, which serve as a fingerprint for the graph [101]. A persistent diagram is a representation of topological features obtained from barcodes, displayed as points in a two-dimensional plane. The x-axis represents the birth time of the feature, while the y-axis represents its death time. The persistent diagram captures the persistence of each topological feature, meaning how long it exists in the metric space. The topological characteristics of several data sets may be compared using the persistence diagram in order to spot similarities and differences between them as well as to deduce higher-dimensional topological qualities of the space. The presence of metrics like Wasserstein distance (WD) makes this fingerprint useful since it can effectively measure the statistical difference between two barcodes [102, 103]. This ability is particularly valuable as the WD can withstand small data perturbations, enabling reliable comparisons and the establishment of similarities between persistent diagrams.



Adjacency matrix

0.00	8.54	12.36	7.07	10.44
8.54	0.00	20.88	3.60	18.00
12.36	20.88	0.00	18.78	6.32
7.07	3.60	18.78	0.00	15.13
10.44	18.00	6.32	15.13	0.00

Figure 8: 0-dimensional barcodes generation for small dataset. The extracted barcodes are illustrated on the left for the given adjacency matrix on the right.

7.3 Computing 0-dimensional barcodes

To identify the 0-dimensional barcodes of an adjacency matrix M containing pairwise distances of n vertices in a point cloud, persistent homology can be utilized as described in [104]. One method is to utilize a modified breadth-first search to determine the number of disjoint components, with a typical runtime complexity of $O(|V| + |E|)$ where V and E are the numbers of vertices and edges, respectively. An example of the process for a small dataset, represented by a 5×5 adjacency matrix, is shown in Figure 8 as per [105]. At $\delta = 0$, there are five disconnected vertices, meaning there are five connected components in the simplicial complex, and hence five bars are born. At $\delta = 3.6$, one edge is grown, resulting in a decrease in the number of connected components and finishing one bar. At $\delta = 6.32$, another edge is grown, causing the number of connected components to decrease again and another bar to finish. At $\delta = 7.07$ and $\delta = 10.44$, two edges are connected in sequence, resulting in two bars being born. At this point, there is only one single component in the point cloud, and hence the upper bar grows to infinity.

7.4 Robustness of Brain Network Persistence: TDA Approach

In recent years, the usage of Persistent homology (PH) to analyze FCNs from rs-fMRI data has gained popularity [106]. One major benefit of PH is that it may provide a barcode that is a compact representation of the topological characteristics of an FCN. To mitigate the non-neural variability issue, which is added because of different sampling periods (data acquisition parameters), studies have found the robustness of TDA based approach to preserve the topology of brain networks of the same subject [107]. Building upon this, studies have developed an end-to-end TDA pipeline that utilizes PH-based techniques to analyze FCN networks on publicly available datasets such as Enhanced Nathan Kline Institute Rockland Sample database (NKI-RS) [108]. This dataset was obtained using a 3T Siemens Magnetom Tim Trio scanner. The rs-fMRI data were acquired from each subject using 3 acquisition protocols with 3 temporal sampling rates ($645ms$, $1400ms$, $2500ms$). In total, there were 316 subjects, and the mean time series from 113 brain regions were integrated into the dataset. For each fMRI scan, one weighted network was generated. A recent study showed the robustness of TDA approach in high-dimensional brain FCNs analysis, stating that the topology of the same subject's FCN was statistically similar, despite variations in temporal sampling periods [38].

A pipeline is shown in Figure 9 illustrating the TDA approach to analyze rs-fMRI data from NKI-RS database captured within the same sampling rate. Initially, this pipeline calculates the pairwise Wasserstein distance between the persistence diagrams of individuals in each of the data cohorts resulting in three adjacency matrices (of size 316×316). A 316×316 adjacency matrix represents a high-dimensional space with 316 dimensions. Analyzing and interpreting high-dimensional data in a statistically meaningful way can be a difficult task. Thus a popular dimensionality reduction scheme called 2-component multidimensional scaling (MDS) is applied which provides 3 reduced size adjacency matrices (of size 316×2). Then a K-means clustering with the silhouette coefficient is applied for cluster analysis [109, 110]. This pipeline removes non-neural variability in multi-site case-control studies, leading to improvements in effect sizes in group

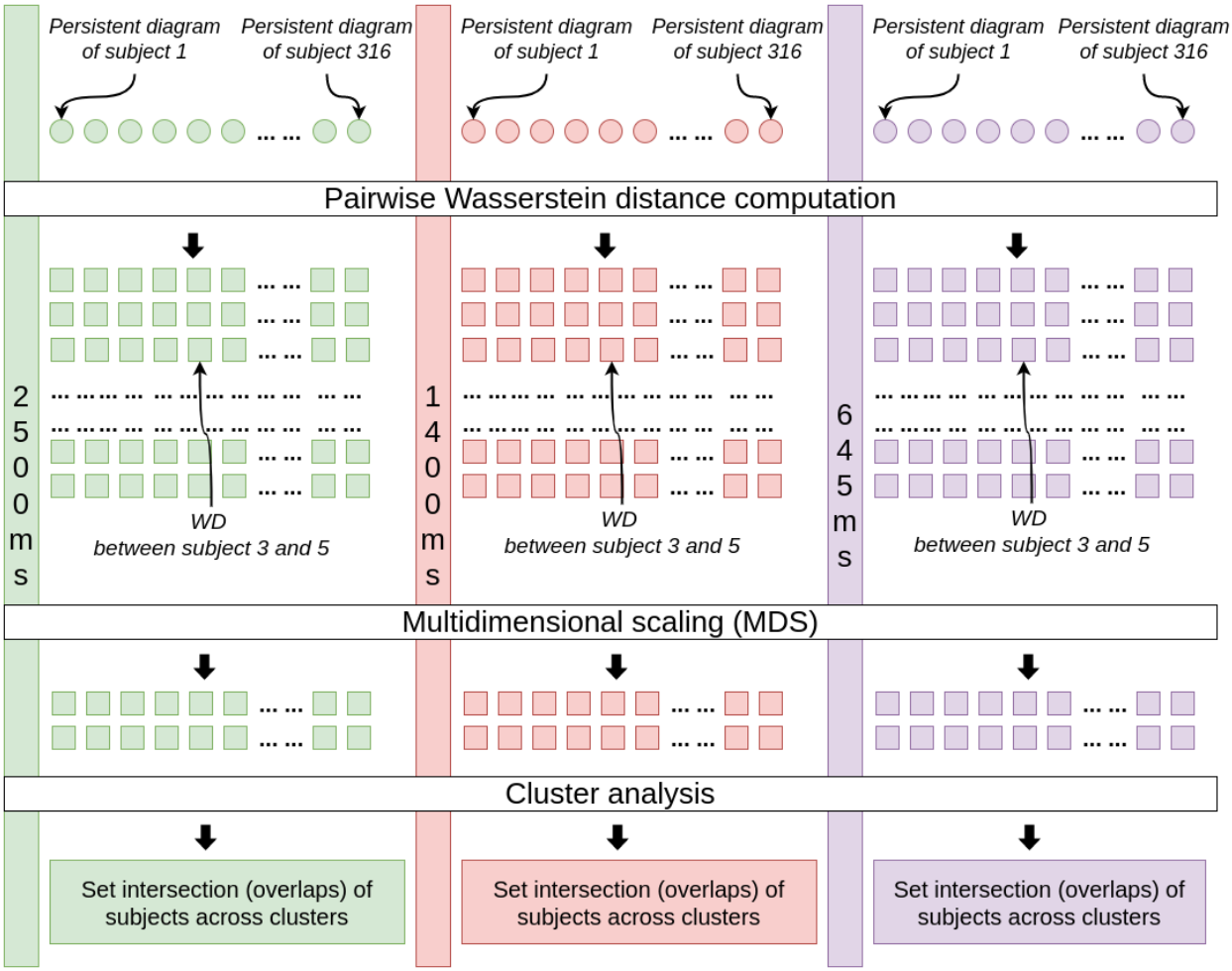


Figure 9: TDA pipeline to analyze different subjects within the same cohort. The same data cohort means the subjects' FCNs are captured using identical temporal sampling [38].

comparisons. Thus, the combination of PH and TDA-based techniques holds great promise for advancing the understanding of FCN networks and their underlying topology.

8 Future research direction

Datalog has been gaining attention as a powerful tool for declarative analytics due to its expressive capabilities and simplicity. Recent studies highlight the significance of creating a scalable backend for Datalog, which can utilize multi-node multi-GPU heterogeneous systems. Our research will focus on developing a scalable Datalog backend that can take advantage of modern computing systems.

In the step towards developing a multi-node multi-GPU backend for Datalog, we will develop novel algorithms to perform iterated relational algebra operations tailored for GPU. We will resolve the issues for iterated operations (Section 4), such as inefficient operation fusion, GPU memory management, and facts deduplication while implementing the RA primitives which are necessary for developing Datalog applications. We will also consider the limitations listed in Table 2 We will develop novel algorithms for implementing relational joins on GPUs. Unlike existing works on standalone joins from standard relational algebra, the new algorithm will be optimized for recursive joins supported by deductive database systems and formalized in Datalog-based languages. This will also require efficient data structures for GPUs to avoid costly operations like sorting the entire relation at each iteration of the iterative process. We will describe and benchmark

techniques for performing iterative relation algebra (RA) computations on GPU devices. The research throughput will be a scalable GPU-based implementation of iterative join tailored for Datalog applications. We will explore key primitives such as join, union, deduplication, and sorting in the first phase and then we will discuss the other aspects of Datalog, such as filter, map, aggregation, and antijoin in an incremental study. We will address the significant challenges in the efficient implementation of relational algebra systems for very large relations. We will evaluate our contributions using Datalog applications such as transitive closure computation, and triangle counting to name a few. In the next iteration, we will extend the state-of-the-art multi-node CPU-based Datalog-like language SLOG to leverage our GPU-based solutions.

In the field of topological data analysis, we will continue our work on exploring persistent homology-based techniques to handle non-neural variability issues generated by using different sampling rates from brain FCNs embedded in rs-fMRI datasets. We will research on designing novel clustering algorithms targeting brain FCNs. We aim to develop novel TDA based analysis pipeline to accommodate temporal dynamics-based time series data. This pipeline will enable us to extract important features from FCNs and characterize their temporal evolution over time. By combining TDA with machine learning algorithms, we will be able to uncover hidden patterns and relationships within large and complex rs-fMRI datasets. Ultimately, our goal is to develop a comprehensive framework that can provide a more detailed understanding of brain function and help diagnose and treat neurological disorders.

We aim to develop and implement a robust and efficient topological data analysis (TDA) methodology utilizing Datalog-like declarative languages on supercomputers. This will leverage the power of supercomputers to enable large-scale and high-dimensional TDA of complex datasets such as brain networks. To achieve this goal, we will explore the use of advanced algorithms and techniques for TDA, including persistent homology, and integrate them with Datalog-enabled declarative analytics for efficient computation and analysis. The expected outcome of our research is a novel declarative analytics technique that can provide deeper insights from complex high-dimensional datasets and support decision-making in diverse applications.

9 Conclusion

This survey has reviewed several research works related to Datalog applications on heterogeneous systems and topological data analysis for brain networks. It first reviewed GPU programming models in exascale settings. Then it reviewed the iterative relational algebra primitives, specifically the join operation on CPUs and GPUs. Later on it analyzes Datalog, it's evaluation and modern Datalog solvers. Additionally, the survey also reviewed the usage of topological data analysis for high-dimensional data. Finally, the future research direction toward developing a novel Datalog backend for the heterogeneous system is discussed along with the possible application of topological data analysis using high-performance Datalog engines. The integration of topological data analysis with high-performance declarative analytics can provide unparalleled insights into high-dimensional data that may be difficult to discern using traditional analytical methods.

References

[1] Thomas M Evans, Andrew Siegel, Erik W Draeger, Jack Deslippe, Marianne M Francois, Timothy C Germann, William E Hart, and Daniel F Martin. A survey of software implementations used by application codes in the exascale computing project. *The international journal of high performance computing applications*, 36(1):5–12, 2022.

[2] Mantas Levinas. What is gpu computing and how is it applied today?, Nov 2020.

[3] Gerassimos Barlas. Chapter 6 - gpu programming. In Gerassimos Barlas, editor, *Multicore and GPU Programming*, pages 391–526. Morgan Kaufmann, Boston, 2015.

[4] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the logicblox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1371–1382, New York, NY, USA, 2015. Association for Computing Machinery.

[5] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. What you always wanted to know about datalog(and never dared to ask). *IEEE transactions on knowledge and data engineering*, 1(1):146–166, 1989.

[6] Oege De Moor, Georg Gottlob, Tim Furche, and Andrew Sellers. *Datalog Reloaded: First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702. Springer, 2012.

[7] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1213–1216, 2011.

[8] Jeffrey D Ullman. *Principles of database systems*. Galgotia publications, 1983.

[9] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, et al. Demonstration of the myria big data management service. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 881–884, 2014.

[10] Jiwon Seo, Stephen Guo, and Monica S Lam. Socialite: Datalog extensions for efficient social network analysis. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 278–289. IEEE, 2013.

[11] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1135–1149, 2016.

[12] Raymond J Mooney. Inductive logic programming for natural language processing. In *International conference on inductive logic programming*, pages 1–22. Springer, 1996.

[13] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 243–262, 2009.

[14] Yannis Smaragdakis, George Balatsouras, and George Kastrinis. Set-based pre-processing for points-to analysis. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 253–270, 2013.

[15] Elnar Hajiyev, Mathieu Verbaere, and Oege De Moor. Codequest: Scalable source code queries with datalog. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2006.

[16] Alexander Shkapsky, Kai Zeng, and Carlo Zaniolo. Graph queries in a next-generation datalog system. *Proceedings of the VLDB Endowment*, 6(12):1258–1261, 2013.

[17] David Maier, K Tuncay Tekle, Michael Kifer, and David S Warren. Datalog: concepts, history, and outlook. In *Declarative Logic Programming: Theory, Systems, and Applications*, pages 3–100. 2018.

[18] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*, pages 422–430. Springer, 2016.

[19] Gerassimos Barlas. Chapter 4 - shared-memory programming: Openmp. In Gerassimos Barlas, editor, *Multicore and GPU Programming*, pages 165–238. Morgan Kaufmann, Boston, 2015.

[20] Sidharth Kumar and Thomas Gilray. Distributed relational algebra at scale. In *International Conference on High Performance Computing, Data, and Analytics (HiPC). IEEE*, 2019.

[21] Thomas Gilray, Sidharth Kumar, and Kristopher Micinski. Compiling data-parallel datalog. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 23–35, 2021.

[22] Sidharth Kumar and Thomas Gilray. Load-balancing parallel relational algebra. In *International Conference on High Performance Computing*, pages 288–308. Springer, 2020.

[23] Oded Green, Pavan Yalamanchili, and Lluís-Miquel Munguía. Fast triangle counting on the gpu. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, pages 1–8, 2014.

[24] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. Red fox: An execution environment for relational query processing on gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 44–54, 2014.

[25] Daniel Zinn, Haicheng Wu, Jin Wang, Molham Aref, and Sudhakar Yalamanchili. General-purpose join algorithms for large graph triangle listing on heterogeneous systems. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, pages 12–21, 2016.

[26] Carlos Alberto Martínez-Angeles, Haicheng Wu, Inês Dutra, Vítor Santos Costa, and Jorge Buenabad-Chávez. Relational learning with gpus: Accelerating rule coverage. *International Journal of Parallel Programming*, 44(3):663–685, 2016.

[27] Daniel Jünger, Christian Hundt, and Bertil Schmidt. Warpdrive: Massively parallel hashing on multi-gpu nodes. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 441–450. IEEE, 2018.

[28] Oded Green, Zhihui Du, Sanyamee Patel, Zehui Xie, Hang Liu, and David A Bader. Anti-section transitive closure. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 192–201. IEEE, 2021.

[29] THOMAS GILRAY and SIDHARTH KUMAR. Toward parallel cfa with datalog, mpi, and cuda. In *Scheme and Functional Programming Workshop*, 2017.

[30] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 511–524, 2008.

[31] Peter A Boncz, Stefan Manegold, Martin L Kersten, et al. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, volume 99, pages 54–65, 1999.

[32] Çağrı Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. Main-memory hash joins on modern processor architectures. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1754–1766, 2014.

[33] Ran Rui, Hao Li, and Yi-Cheng Tu. Join algorithms on gpus: A revisit after seven years. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 2541–2550. IEEE, 2015.

[34] Ran Rui, Hao Li, and Yi Cheng Tu. Efficient join algorithms for large database tables in a multi-gpu environment. *Proceedings of the VLDB Endowment*, 14:708–720, 2020.

[35] Haicheng Wu. *Acceleration and execution of relational queries using general purpose graphics processing unit (GPGPU).* PhD thesis, Georgia Institute of Technology, 2015.

[36] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. Mg-join: A scalable join for massively parallel multi-gpu architectures. pages 1413–1425. Association for Computing Machinery, 2021.

[37] Ahmedur Rahman Shovon, Landon Richard Dyken, Oded Green, Thomas Gilray, and Sidharth Kumar. Accelerating datalog applications with cudf. In *2022 IEEE/ACM 11th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 41–45, 2022.

[38] Sidharth Kumar, Ahmedur Rahman Shovon, and Gopikrishna Deshpande. The invariance of persistent homology of brain networks to data acquisition-related non-neural variability in resting state fmri. *Human Brain Mapping*, in review.

[39] Darshan Shimoga Chandrashekar, Santhosh Kumar Karthikeyan, Praveen Kumar Korla, Henalben Patel, Ahmedur Rahman Shovon, Mohammad Athar, George J Netto, Zhaohui S Qin, Sidharth Kumar, Upender Manne, et al. Ualcan: An update to the integrated cancer data analysis platform. *Neoplasia*, 25:18–27, 2022.

[40] Jianbin Fang, Chun Huang, Tao Tang, and Zheng Wang. Parallel programming models for heterogeneous many-cores: a comprehensive survey. *CCF Transactions on High Performance Computing*, 2(4):382–400, 2020.

[41] Rick Merritt. What is an exaflop?, Sep 2022.

[42] Griffon Webstudios. Why countries are competing to build supercomputers?, Aug 2021.

[43] Gerassimos Barlas. Chapter 1 - introduction. In Gerassimos Barlas, editor, *Multicore and GPU Programming*, pages 1–26. Morgan Kaufmann, Boston, 2015.

[44] Jason. Sanders. *CUDA by example : an introduction to general-purpose GPU programming.* Addison-Wesley, Upper Saddle River, NJ, 2011.

[45] Neil Trevett, Andrew Richards, Mark Butler, Jeff McVeigh, Anshuman Bhat, Balaji Calidas, Vincent Hindriksen, and Weijin Dai. Opencl - the open standard for parallel programming of heterogeneous systems, Jul 2013.

[46] The OpenMP ARB, Sep 2021.

[47] CodinGame SA. Introduction to sycl, 2020.

[48] Michael Wong, Nevin Liber, Sanzio Bassini, Andrew Richards, Mark Butler, Jeff McVeigh, Brandon Cook, Hideki Sugimoto, Cyril Cordoba, Thomas Fahringer, and et al. Sycl - c++ single-source heterogeneous programming for acceleration offload, Jan 2014.

[49] Intel Corporation. Intel® oneapi hpc toolkit: Cluster and hpc development tools, 2021.

[50] Argonne Leadership Computing Facility. 2021 alcf computational performance workshop, 2021.

[51] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. *Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL.* Springer Nature, 2021.

[52] Christian Trott, Damien Lebrun-Grandie, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S Hollman, Daniel Alejandro Ibanez, Nevin Liber, Jonathan Madsen, Jeff Scott Miles, David Zoeller Poliakoff, Amy Jo Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. Kokkos 3: Programming model extensions for the exascale era. *IEEE transactions on parallel and distributed systems*, 33(4):1–1, 2021.

[53] Christian Trott, Luc Berger-Vergiat, David Poliakoff, Sivasankaran Rajamanickam, Damien Lebrun-Grandie, Jonathan Madsen, Nader Al Awar, Milos Gligoric, Galen Shipman, and Geoff Womeldorff. The kokkos ecosystem: Comprehensive performance portability for high performance computing. *Computing in science & engineering*, 23(5):10–18, 2021.

[54] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing*, 74(12):3202–3216, 2014.

[55] Jeff R. Hammond, Michael Kinsner, and James Brodman. A comparative analysis of kokkos and sycl as heterogeneous, parallel programming models for c++ applications. In *Proceedings of the International Workshop on OpenCL*, IWOCL'19, New York, NY, USA, 2019. Association for Computing Machinery.

[56] Exascale Computing Project. The kokkos c++ performance portability ecosystem, 2017.

[57] Arturo Vargas. An overview of the raja portability suite, Mar 2021.

[58] David Beckingsale, Richard Hornung, Tom Scogland, and Arturo Vargas. Performance portable c++ programming with raja. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 455–456, New York, NY, USA, 2019. Association for Computing Machinery.

[59] Richard Hornung. Raja portability suite: Enabling performance portable cpu and gpu hpc applications, 2021.

[60] David A. Beckingsale, Thomas RW Scogland, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, Brian S. Ryujin, and et al. Raja: Portable performance for large-scale scientific applications. *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019.

[61] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. Distributed join algorithms on thousands of cores. *Proc. VLDB Endow.*, 10(5):517–528, jan 2017.

[62] Chengxin Guo, Hong Chen, Feng Zhang, and Cuiping Li. Distributed join algorithms on multi-cpu clusters with gpudirect rdma. *Proceedings of the 48th International Conference on Parallel Processing*, 2019.

[63] Arsen Nasibullin and Boris Novikov. Fault tolerant distributed hash-join in relational databases. In *Fifth Conference on Software Engineering and Information Management (SEIM-2020)*, pages 11–17, 2020.

[64] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned global address space languages. *ACM Computing Surveys (CSUR)*, 47(4):1–27, 2015.

[65] Anne C Elster and Tor A Haugdahl. Nvidia hopper gpu and grace cpu highlights. *Computing in Science & Engineering*, 24(2):95–100, 2022.

[66] Chengxin Guo, Hong Chen, Feng Zhang, and Cuiping Li. Parallel hybrid join algorithm on gpu. *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1572–1579, 2019.

[67] Hongzhi Wang, Ning Li, Zheng ke Wang, and Jianing Li. Gpu-based efficient join algorithms on hadoop. *The Journal of Supercomputing*, 77:292 – 321, 2020.

[68] Oded Green. Hashgraph—scalable hash tables using a sparse graph data structure. *ACM Transactions on Parallel Computing (TOPC)*, 8:1 – 17, 2019.

[69] Daniel Jünger, Robin Kobus, André Müller, Christian Hundt, Kai Xu, Weiguo Liu, and Bertil Schmidt. Warpcore: A library for fast hash tables on gpus. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 11–20, 2020.

[70] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, sep 2001.

[71] Yehoshua Sagiv. Optimizing datalog programs. Technical report, Stanford, CA, USA, 1986.

[72] Michael Stonebraker. *Readings in database systems*. Morgan Kaufmann Publishers Inc., 1988.

[73] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: An interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 1213–1216, New York, NY, USA, 2011. Association for Computing Machinery.

[74] Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. Parallel materialisation of datalog programs in centralised, main-memory rdf systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014.

[75] Thomas Gilray, Arash Sahebolamri, Sidharth Kumar, and Kristopher Micinski. Higher-order, data-parallel structured deduction. *arXiv preprint arXiv:2211.11573*, 2022.

[76] Thomas Gilray, Sidharth Kumar, and Kristopher Micinski. Compiling data-parallel datalog. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, CC 2021, page 23–35, New York, NY, USA, 2021. Association for Computing Machinery.

[77] Jorge L Reyes-Ortiz, Luca Oneto, and Davide Anguita. Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf. *Procedia Computer Science*, 53:121–130, 2015.

[78] Carlos Alberto Martínez-Angeles, Inês Dutra, Vítor Santos Costa, and Jorge Buenabad-Chávez. A datalog engine for gpus. In *Declarative Programming and Knowledge Management: Declarative Programming Days, KDPD 2013, Unifying INAP, WFLP, and WLP, Kiel, Germany, September 11-13, 2013, Revised Selected Papers 0*, pages 152–168. Springer, 2014.

[79] Grigoris Antoniou, Sotiris Batsakis, Raghava Mutharaju, Jeff Z. Pan, Guilin Qi, Ilias Tachmazidis, Jacopo Urbani, and Zhangquan Zhou. A survey of large-scale reasoning on the web of data. *The Knowledge Engineering Review*, 33, 2018.

[80] Ahmedur Rahman Shovon, Landon Richard Dyken, Oded Green, Thomas Gilray, and Sidharth Kumar. Accelerating datalog applications with cudf. In *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 41–45. IEEE, 2022.

[81] Francois Bancilhon. Naive evaluation of recursively defined relations. In *On Knowledge Base Management Systems*, pages 165–178. Springer, 1986.

[82] Marianne Shaw, Paraschos Koutris, Bill Howe, and Dan Suciu. Optimizing large-scale semi-naïve datalog evaluation in hadoop. In *International Datalog 2.0 Workshop*, pages 165–176. Springer, 2012.

[83] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 16–24, 2008.

[84] Mohammad Al Hasan and Vachik S Dave. Triangle counting in large networks: a review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(2):e1226, 2018.

[85] Leyuan Wang and John Douglas Owens. Fast bfs-based triangle counting on gpus. *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2019.

[86] Bharat B Biswal, Maarten Mennes, Xi-Nian Zuo, Suril Gohel, Clare Kelly, Steve M Smith, Christian F Beckmann, Jonathan S Adelstein, Randy L Buckner, Stan Colcombe, et al. Toward discovery science of human brain function. *Proceedings of the national academy of sciences*, 107(10):4734–4739, 2010.

[87] Stephanie Noble, Dustin Scheinost, Emily S Finn, Xilin Shen, Xenophon Papademetris, Sarah C McEwen, Carrie E Bearden, Jean Addington, Bradley Goodyear, Kristin S Cadenhead, et al. Multisite reliability of mr-based functional connectivity. *Neuroimage*, 146:959–970, 2017.

[88] Christian Dansereau, Yassine Benhajali, Celine Risterucci, Emilio Merlo Pich, Pierre Orban, Douglas Arnold, and Pierre Bellec. Statistical power and prediction accuracy in multisite resting-state fmri connectivity. *Neuroimage*, 149:220–232, 2017.

[89] Hugo G. Schnack and René S. Kahn. Detecting neuroimaging biomarkers for psychiatric disorders: Sample size matters. *Frontiers in Psychiatry*, 7, 2016.

[90] Najmeh Khalili-Mahani, Serge ARB Rombouts, Matthias JP van Osch, Eugene P Duff, Felix Carbonell, Lisa D Nickerson, Lino Becerra, Albert Dahan, Alan C Evans, Jean-Paul Soucy, et al. Biomarkers, designs, and interpretations of resting-state fmri in translational pharmacological research: A review of state-of-the-art, challenges, and opportunities for studying brain chemistry. *Human brain mapping*, 38(4):2276–2325, 2017.

[91] Alexandre Abraham, Michael P. Milham, Adriana Di Martino, R. Cameron Craddock, Dimitris Samaras, Bertrand Thirion, and Gael Varoquaux. Deriving reproducible biomarkers from multi-site resting-state data: An autism-based example. *NeuroImage*, 147:736–745, 2017.

[92] Robert W Ghrist. *Elementary applied topology*, volume 1. Createspace Seattle, 2014.

[93] Alice Patania, Francesco Vaccarino, and Giovanni Petri. Topological analysis of data. *EPJ Data Science*, 6:1–6, 2017.

[94] H. Lee, M. K. Chung, H. Kang, B. Kim, and D. S. Lee. Discriminative persistent homology of brain networks. In *2011 IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pages 841–844, 2011.

[95] Manish Saggar, Olaf Sporns, Javier Gonzalez-Castillo, Peter A. Bandettini, Gunnar Carlsson, Gary Glover, and Allan L. Reiss. Towards a new approach to reveal dynamical organization of the brain using topological data analysis. *Nature Communications*, 9(1):1399, Apr 2018.

[96] Y. Dabaghian, F. Mémoli, L. Frank, and G. Carlsson. A topological paradigm for hippocampal spatial map formation using persistent homology. *PLOS Computational Biology*, 8(8):1–14, 08 2012.

[97] H Edelsbrunner and J Harer. Surveys on discrete and computational geometry. *Contemporary Mathematics (American Mathematical Society, Providence, RI)*, 453:257–282, 2008.

[98] Herbert Edelsbrunner and John Harer. *Computational Topology: An Introduction*. 2010.

[99] Steve Y Oudot. *Persistence theory: from quiver representations to data analysis*, volume 209. American Mathematical Society Providence, 2015.

[100] B. Cassidy, C. Rae, and V. Solo. Brain activity: Conditional dissimilarity and persistent homology. In *2015 IEEE 12th International Symposium on Biomedical Imaging (ISBI)*, pages 1356–1359, 2015.

[101] Robert Ghrist. Barcodes: the persistent topology of data. *Bulletin of the American Mathematical Society*, 45(1):61–75, 2008.

[102] SS Vallender. Calculation of the wasserstein distance between probability distributions on the line. *Theory of Probability & Its Applications*, 18(4):784–786, 1974.

[103] Herbert Edelsbrunner. Persistent homology: theory and practice. 2013.

[104] Afra J Zomorodian. *Topology for computing*, volume 16. Cambridge university press, 2005.

[105] Ashley Suh, Mustafa Hajij, Bei Wang, Carlos Scheidegger, and Paul Rosen. Persistent homology guided force-directed graph layouts. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):697–707, January 2020.

[106] Tananun Songdechakraiwut, Li Shen, and Moo Chung. Topological learning and its application to multimodal brain network integration. In *Medical Image Computing and Computer Assisted Intervention–MICCAI 2021: 24th International Conference, Strasbourg, France, September 27–October 1, 2021, Proceedings, Part II 24*, pages 166–176. Springer, 2021.

[107] Soumya Das, Hernando Ombao, and Moo K Chung. Topological data analysis for functional brain networks. *arXiv preprint arXiv:2210.09092*, 2022.

[108] Kate Brody Nooner, Stanley Colcombe, Russell Tobe, Maarten Mennes, Melissa Benedict, Alexis Moreno, Laura Panek, Shaquanna Brown, Stephen Zavitz, Qingyang Li, et al. The nki-rockland sample: a model for accelerating the pace of discovery science in psychiatry. *Frontiers in neuroscience*, 6:152, 2012.

[109] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[110] Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.