PrismX: A Single-Machine System for Querying Big Graphs

Shuhao Liu
Shenzhen Institute of Computing Sciences, China
shuhao@sics.ac.cn

Yang Liu
Beihang University, China
ly_act@buaa.edu.cn

Wenfei Fan
Shenzhen Institute of Computing Sciences, China
University of Edinburgh, UK
Beihang University, China
wenfei@inf.ed.ac.uk

ABSTRACT

We demonstrate PrismX (PRAM with SSDs as Memory eXtension), a single-machine system for graph analytics. PrismX allows users to make practical use of existing PRAM algorithms without any change. To cope with the limited DRAM capacity, it employs NVMe SSDs as memory extension. Leveraging graph preprocessing, PrismX implements a series of system optimization strategies, which automatically and transparently adapt to the runtime workload, no matter whether the computation is CPU-bound or I/O-bound. We demonstrate PrismX for its (1) ease of programming by reusing PRAM algorithms, (2) efficiency by comparing with the state-of-the-art graph systems, single-machine or multi-machine, in-memory or out-of-core; (3) parallel scalability of in-memory PRAM algorithms, reducing runtime when more CPU cores are available; and (4) applications in credit risk assessment.

PVLDB Reference Format:
Shuhao Liu, Yang Liu, and Wenfei Fan. PrismX: A Single-Machine System for Querying Big Graphs. PVLDB, 17(12): 4485 - 4488, 2024.
doi:10.14778/3685800.3685906

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SICS-Fundamental-Research-Center/Planar.

1 INTRODUCTION

Big graph analytics is often conducted by big companies using multi-machine systems with a cluster of machines. For instance, to compute connected components of a graph with billions of vertices and trillions of edges, Google employs a 1000-node cluster with 12000 processors and 128 TB of aggregated memory [12]. However, small companies typically cannot afford such enterprise clusters. Moreover, the multi-machine systems "have either a surprisingly large COST, or simply underperform one thread" [8]. This is because such systems adopt the shared-nothing architecture; the more machines are used, the heavier their communication cost is, and the less utilized the machines are due to unbalanced workload.

In light of these, a host of single-machine graph systems have been developed, classified into *in-memory* systems when graphs can be loaded entirely into the main memory of a machine, or *out-of-core* systems when graphs are too large to fit into the main memory at once. These systems typically adopt a *vertex-centric* (VC) parallel model [4, 7]. VC makes users think like a vertex: a program is "pivoted" at a vertex; it may only directly access the data at the vertex and its adjacent edges, but has to exchange information with "remote" vertices via message passing. However, it is nontrivial to program with VC for problems that are constrained by "joint" conditions on multiple vertices, *e.g.,* graph simulation [9].

To solve the problem, some systems, *e.g.,* MiniGraph [16], extend the *graph-centric* (GC) parallel model [2, 3] to single-machine systems. These systems parallelize sequential graph algorithms, handle out-of-core tasks at the subgraph level via partial and incremental evaluation, and enable beyond-neighborhood graph access; they improve the locality of out-of-core computation, reduce I/Os, and serve low-bandwidth external storage devices better than VC. The GC systems target SATA SSDs whose sequential read bandwidth is 0.5 GB/s, while the (theoretical) bandwidth of DDR5 DRAM is 51.2 GB/s, a difference of almost two orders of magnitude.

However, with the new generation of consumer-grade NVMe SSDs, the bandwidth gap between DRAM and SSDs has been significantly narrowed. A 2023 Samsung 990Pro NVMe SSD has a read bandwidth of 7 GB/s, which is only 7 times slower than DRAM. With such fast SSDs, the GC systems constantly under-utilize I/O bandwidth, and their incremental steps may become redundant.

Moreover, PRAM has been well studied for shared-memory architectures (see [13]). It allows multiple CPU cores to work in parallel, and synchronize via shared memory. A host of PRAM algorithms are already in place, and many of them guarantee the parallel scalability, *i.e.,* the more processors are used, the less runtime is taken [6].

What parallel model fits multi-core parallelism of a single machine better under the shared-memory architecture? Can we capitalize on the decades of work on PRAM and simplify parallel programming for graph computations? Is it possible to develop a single-machine system that performs comparably to the state-of-the-art (SOTA) multi-machine systems that employ dozens of machines?

PrismX (Section 2). To answer the questions, we develop PrismX (PRAM with SSDs as Memory eXtension), a single-machine system for graph analytics. PrismX features the following.

(1) A unified programming model. PrismX advocates reusing existing PRAM [13] algorithms for both in-memory and out-of-core graph computations. Given a query class Q, users can directly implement an existing PRAM algorithm $\mathcal{A}$ for Q using the provided parallel programming interface. The users do not have to handle out-of-core computation manually. When a graph fits into the memory of a single machine, it executes algorithm $\mathcal{A}$ with all available cores. When a graph size exceeds the DRAM capacity, it additionally

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685906

swaps data between DRAM and SSDs transparently.

Unlike VC, users of PrismX do not have to think like a vertex or develop new parallel algorithms from scratch. Unlike GC systems, there is no longer a need for incremental steps with PrismX.

(2) Preprocessing. PrismX proposes to preprocess an input graph G via *reorganization* and custom *pre-computations*, as boosts for various PRAM algorithms. It first reorganizes graph G and serializes it into a sequence of files on NVMe SSDs. It partitions G into small blocks called *vertex atoms*, groups the vertex atoms via clustering, and relabels the vertices based on topological connectivity among clusters. These enable efficient indexing and access to the graph data; moreover, they enforce a hierarchical sketch of the dependencies among graph substructures and intermediate results, which opens the way to a series of dependency-aware optimizations.

(3) New optimizations. PrismX proposes a series of strategies to maximize CPU and I/O utilization and adapt to various workloads.

For CPU-bound algorithms, PrismX automatically loads and reuses the pre-computed auxiliary results from SSDs whenever possible to skip redundant computation at the cost of extra I/Os. Moreover, by analyzing hierarchical dependencies among graph substructures, it employs a lazy, asynchronous evaluation strategy. Even though this breaks the synchronous locksteps of PRAM operations, it retains the correctness guarantees of the algorithm

For I/O-bound algorithms, PrismX supports fine-grained (vertex atom-level) selective loading of graph and auxiliary data, which reduces the redundant I/O overhead. Also, it adaptively allocates more CPU cores to parallel I/O operations. These strategies leverage the scattered and parallel I/O capabilities of NVMe SSDs, relieving the read bandwidth bottleneck at the cost of slower graph computation.

Demonstration (Section 3). We will invite demo participants to experience PrismX for its (a) ease of programming, (b) performance compared with the SOTA single-machine systems (in-memory and out-of-core) and multi-machine systems, (c) parallel scalability retained for PRAM algorithms that have the property; and (d) application in credit risk assessment in lieu of multi-machine systems.

2 SYSTEM OVERVIEW

This section presents the model and the architecture of PrismX.

2.1 Programming Model

PrismX aims to directly leverage the decade of work on PRAM for querying large graphs. PRAM supports multi-core parallelism via *single-instruction-multiple-data* (*SIMD*), *i.e.,* all processors execute the same operation simultaneously on different data. A PRAM algorithm streamlines a sequence of locksteps; each step is either a RAM operation or a read/write to a memory location (hence the name "random-access"). Unlike VC, it supports beyond-neighborhood direct memory accesses, synchronization via shared memory, and load balancing without skewed workload. A large number of PRAM algorithms have been developed, and many are parallel scalable [6].

PRAM naturally fits a single-machine shared-memory system, yet with a few practical considerations. PrismX introduces a unified programming model that seamlessly bridges in-memory and out-of-core graph computations, adapting PRAM to physical machines in

PrismX: WCC Algorithm

```
1  // Vertex Status: S_v = { p } where p(v) = v for each v ∈ V
2  // Input: graph G = (V, E)
3  While (!G.E.empty()) {
4    Eapply(G.E, Graft); // Merge pseudo trees.
5    Vapply(G.V, PointerJump); // Simplify pseudo tress.
6    Eapply(G.E, Contract);} // Delete unused edges.
```

PrismX: Graft

```
1  // Input: Edge e = (src -> dst)
2  min_id, max_id := FindMinAndMax(e.src, e.dst); // Find the min and max id
3  WriteMin(p[max_id], p[min_id]); // Set parent of max id to min id, which
   means larger id points to smaller id.
```

PrismX: PointerJump

```
1  // Input: Vertex v
2  While (p[v] != v) { p[v] = p[p[v]]; }
```

PrismX: Contract

```
1  // Input: Edge e = (src -> dst)
2  if p[src] == p[dst] then Delete(e);
```

Figure 1: Programming interface of PrismX.

the real world. PRAM is a theoretical model that assumes unlimited memory with unit access cost and a polynomial number of processors. However, these are beyond reach in practice, as a single machine has limited memory capacity and I/O bandwidth, and its number of CPU processors cannot scale with the size of the input graphs.

PrismX narrows the gap between theory and practice. Users can directly implement an existing PRAM algorithm $\mathcal{A}$ by virtually retaining its theoretical assumptions. They can treat the input graph G as arrays of vertices and edges and declare intermediate status variables as arrays. They can program lockstep parallel operations as if all data were already available in the shared memory.

Under the hood, PrismX transparently manages data within the memory hierarchy. It handles the graph and intermediate results in small *vertex atoms*, each of which is a collection of data associated with a single vertex. With in-memory atoms, it executes the PRAM algorithm $\mathcal{A}$ using all available CPU cores for multi-core parallelism. For atoms residing in SSD, it preemptively swaps them between DRAM and SSD storage, ensuring continuous, balanced CPU and I/O operations without explicit user intervention. Once the atoms are loaded into memory, it proceeds to run $\mathcal{A}$ on the atoms.

Programming interface. PrismX provides three sets of primitives.

(1) Status declaration: Consider a graph $G = (V, E, L)$, where V (resp. E and L) is a finite set of vertices (resp. edges and labels). Given G, PrismX organizes graph G as a set of data arrays for V, E, and L. A PrismX program $\mathcal{A}$ declares and initializes *status variables* for G, which serve as intermediate results of $\mathcal{A}$.

(2) Parallel operators: The lockstep operations of $\mathcal{A}$ are directly streamlined as if users were programming with PRAM. Operator VApply(f_V) (resp. EApply(f_E)) applies a unit function f_V (resp. f_E) to a set of vertices (resp. edges) in parallel, where the unit function can invoke any data accessors and mutators to be defined next.

(3) Data accessors and mutators: PrismX provides APIs for users to manipulate G and its associated status variables, including read/write accesses to k-hop neighbors, and vertex/edge attributes.

*Example: weakly connected components (*WCC*).* A PrismX program for WCC is exemplified in Figure 1. It implements the PRAM algorithm $\mathcal{A}$ of [11]. To find the WCC of a subgraph $F_i = (V_i, E_i)$, $\mathcal{A}$ partitions $|V_i|$ into disjoint sets, where vertices within a set are weakly connected to each other. The set membership of a vertex v

is marked by the flag $p(v)$, declared as a vertex status (Line 1).

Each iteration of $\mathcal{A}$ merges vertex sets based on inter-set edges ("Graft", Line 4), makes membership flags consistent ("PointerJump", Line 5), and removes edges internal to a set ("Contract", Line 6). It proceeds until all edges in E_i are removed (Line 3).

2.2 System Architecture

As shown in Fig. 2, PrismX is implemented with programming interface, a preprocessor, data processing modules and control modules.

(1) APIs. PrismX provides users with primitives for status variable declarations, synchronized parallel operators for lockstep concurrent graph manipulations, and a collection of convenient functions for data access and mutations, as remarked in Section 2.1.

(2) Preprocessor. As mentioned in Section 1, PrismX employs graph preprocessing techniques to enhance runtime performance. During graph reorganization, it partitions the input graph G into vertex atoms, organizes these atoms into clusters based on their topological connectivity, and then relabels vertices and edges to reflect these connectivities. By establishing a hierarchical representation of graph substructures and their dependencies, PrismX sets the stage for dependency-aware system optimizations that can significantly boost the performance of various PRAM algorithms.

(3) Data processing modules. PrismX implements a data pipeline to continuously read from and write to the NVMe SSDs. It partitions the memory into (a) an off-stage area as a buffering space for vertex atom loading and discharging, and (b) an on-stage area as shared memory for executing PRAM programs. These areas are used to overlap CPU and I/O operations, dynamically adjusted across lockstep operations. PrismX increases the size of the off-stage area when the lockstep is I/O-bound, *i.e.,* when the I/O cost dominates; otherwise, it increases the budget for the on-stage area if the lockstep is CPU-bound. It also adopts incremental writing to reduce I/O.

(4) Control modules. PrismX supports the following: (1) ConfigMng maintains system configs and the profiling results for cost estimation; (2) DependencyMng keeps track of the topological dependencies among clusters of vertex atoms; and (3) Scheduler actively monitors runtime statistics to decide system bottlenecks. Based on these statistics, it implements a series of optimization strategies, including reusing pre-computed data, coordinating lazy evaluations, selective loading of graph and auxiliary data, and CPU resource scheduling.

2.3 Optimizations

PrismX adopts different strategies to optimize CPU- and I/O-bound computations. It also ensures low-overhead simulation of PRAM.

Lazy evaluations. In addressing the limitations of synchronous lockstep operations, PrismX adopts a lazy, asynchronous evaluation strategy. This method allows the system to temporarily break the synchronization barriers after PRAM locksteps, while still retaining its correctness. It defers certain computations until the corresponding intermediate results are explicitly required, thereby reducing repetitive updates. To this end, it exploits hierarchical dependencies within the graph structure, prioritizing computations based on their immediate necessity and potential impact on overall performance.

Selective data loading. For I/O-bound workloads, PrismX introduces a selective loading mechanism. By enabling fine-grained access to vertex atom-level data, it can load only the active graph and auxiliary data required for a given lockstep operation. This targeted approach reduces unnecessary I/O operations, since it is common for a part of the graph to remain inactive for an operation. Moreover, it may opt to recompute some pre-computed auxiliary data if the cost of loading it from storage exceeds the cost of recomputation.

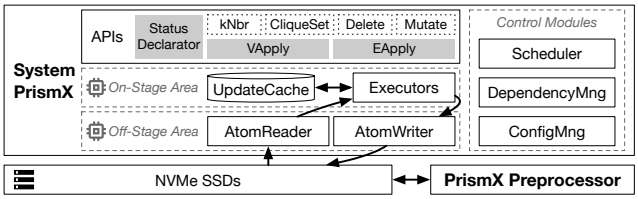

Figure 2: PrismX system architecture.

CPU resource scheduling. PrismX dynamically allocates CPU resources between parallel computation and I/O operations. By assigning more CPU cores to handle scattered reads/writes over NVMe SSDs, it mitigates the read bottleneck for I/O-bound algorithms. To this end, it implements an adaptive algorithm that continuously assesses the workload between CPU and I/O demands, ensuring balanced computation and data transfer operations.

PRAM simulation. PRAM assumes a polynomial number of processors. PrismX simulates PRAM with a machine of p CPU cores.

Synchronization. Planar (1) places an implicit synchronization barrier after each parallel VApply or EApply; and (2) ensures that all read accesses *precede* any write within each parallel operator.

Load balancing. Each parallel operator can generate a number of parallel tasks. To allocate these tasks to p processors with a balanced workload, we employ a size-p thread pool in Executors. Initially, all generated tasks are placed in a task queue. Each thread then polls the queue whenever it becomes idle and executes the obtained task.

Lock-free parallelism. This is to further reduce write contention and speed up parallel processing. Consider concurrent writes to data Φ. If Φ cannot be implemented as an atomic data structure, PrismX adopts the copy-on-write technique. That is, whenever a thread is to modify Φ, it creates a thread-local copy Φ' of Φ, writes a new value to Φ', and makes an atomic switch from the old to the new.

3 DEMONSTRATION OVERVIEW

This section presents our demonstration setting and plan.

3.1 Demonstration Setting

Algorithms. To demonstrate how PrismX works, we have implemented PRAM algorithms WCC, Graph Convolutional Networks (GCN), PageRank (PR) and k-hop neighborhood counting (kCount), which are widely used in application such as network analysis, routing protocols, Web search, and credit risk assessment, respectively.

Graphs. We will use (1) medium `web-sk` with 50M nodes and 1.9B edges; and (2) large `clue-web` with 1.7B nodes and 7.9B edges.

System comparison. We will compare PrismX with the SOTA single-machine and multi-machine systems: (1) out-of-core systems MiniGraph [16], Blaze [5] and GridGraph [15]; (2) in-memory Galois [10] and CoroGraph [14]; and (3) multi-machine Gluon [1] and GraphScope [2] (the open-source version of GRAPE [3]).

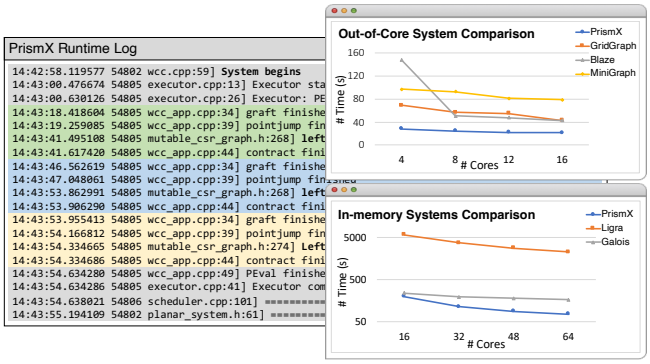

Figure 3: A snapshot of analytics panel (WCC, single-machine).

Environment. For out-of-core workloads, we will use a workstation powered by an Intel Core i7-11700@2.50GHz CPU (16 cores, 16MB LLC) and 64GB of DDR4-3200 memory. A 2TB Samsung 990Pro NVMe SSD will be used as memory extension, with an average sequential read throughput of 6.9GB/s. For in-memory executions, we will use a server with 4× Intel Xeon Gold 5320@2.20GHz CPUs (26 cores, 39MB LLC each) and 512GB of DDR4-2933 memory.

To compare with multi-machine systems, we will showcase remote cloud deployments. We will run PrismX on a single 8-vCPU 32GB-memory instance. Gluon will use multiple instances of the same type, while GraphScope will use multiple 8-vCPU 64GB-memory instances since it requires >32 GB in all demonstrations.

Setup. We will configure PrismX to use 32GB of memory as buffer, and show how the buffer is dynamically adjusted across locksteps.

3.2 Demonstration Plan

We will invite participants to experience the following.

(1) Ease of programming. Given the PRAM algorithm $\mathcal{A}$ [11] for, *e.g.,* WCC, we will walk the participants through the process of implementing $\mathcal{A}$ with the interface of PrismX (as shown in Figure 1). One can experience how straightforward PrismX primitives are.

(2) Performance. We will demonstrate the efficiency and scalability of PrismX, and the impact of key factors on its performance.

Comparison with single-machine systems. Users are invited to run PrismX programs and the SOTA systems in a uniform setup. Table 1 compares the average performance for WCC. Over large `clue-web`, PrismX outperforms the SOTA in-memory Galois by 2.20×, and out-of-core Blaze by 4.58×. In-memory system CoroGraph cannot handle graphs at this scale. Over `web-sk`, PrismX beats Galois, CoroGraph and Blaze by 1.61×, 2.55× and 1.98×, respectively.

Comparison with multi-machine systems. We will invite participants to witness the benefit of PrismX over the SOTA GraphScope and Gluon, for different query classes. As shown in Table 1 for WCC, single-node PrismX outperforms a 10-node cluster by 1.50–4.53×. It highlights the cost effectiveness of a single-machine system.

(3) Parallel scalability. One can observe the parallel scalability of PrismX by varying query classes and the number of CPU cores. As shown in Figure 3, the system panel will also visualize system speedups under different parallelism settings. We will see that, *e.g.,* using 4× cores for in-memory workloads, PrismX gets a 2.60× speedup, better than the 2.54× of Ligra and the 1.43× of Galois.

Table 1: System performance comparison for WCC.

Graph	Type	System	Time (s)	I/O (GB)
web-sk	In-Memory	PrismX	**3.1**	N/A
		CoroGraph	7.9 (2.55×)	N/A
		Galois	5.0 (1.61×)	N/A
	Out-of-Core	PrismX	**21.6**	**8.3**
		MiniGraph	78.9 (3.65×)	21.5 (2.59×)
		Blaze	42.8 (1.98×)	31.3 (3.77×)
		GridGraph	42.6 (1.97×)	12.5 (1.51×)
	Multi-Machine	PrismX (1 node)	**24.7**	**8.3**
		GraphScope (10 node)	37.0 (1.50×)	N/A
		Gluon (10 node)	112.0 (4.53×)	N/A
clue-web	In-Memory	PrismX	**76.4**	N/A
		Galois	168.3 (2.20×)	N/A
		Ligra	2290.0 (29.97×)	N/A
	Out-of-Core	PrismX	**304.7**	**170.2**
		Blaze	1396.7 (4.58×)	237.4 (1.39×)

(4) Application. Participants are also invited to experience the application of PrismX in credit risk assessment. A user A is likely to default on a loan if A has direct or indirect transactions with at least δ high-risk users. In a graph G where a transaction is abstracted as an edge between users, it is to flag any user who has at least δ high-risk users in its two-hop neighborhood. We adopt 2Count to identify such flagged users. We will demonstrate the PrismX program for this application, and visualize the findings over `web-sk` as G.

ACKNOWLEDGMENTS

This work is partially supported by China NSFC 62225202.

REFERENCES

[1] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A Communication-Optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *PLDI*.
[2] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, Youyang Yao, Qiang Yin, Wenyuan Yu, Kai Zeng, Kun Zhao, Jingren Zhou, Diwen Zhu, and Rong Zhu. 2021. GraphScope: A Unified Engine For Big Graph Processing. *PVLDB* 14, 12 (2021), 2879–2892.
[3] Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, XiaoJian Luo, Qiang Yin, Ping Lu, Yang Cao, and Ruiqi Xu. 2018. Parallelizing Sequential Graph Computations. *ACM Trans. Database Syst.* 43, 18 (2018).
[4] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*.
[5] J. Kim and S. Swanson. 2022. Blaze: Fast graph processing on fast SSDs. In *SC*.
[6] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. 1990. A complexity theory of efficient parallel algorithms. *Theor. Comput. Sci.* 71, 1 (1990), 95–132.
[7] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *SIGMOD*. 135–146.
[8] Frank McSherry, Michael Isard, and Derek Gordon Murray. 2015. Scalability! But at what COST?. In *HotOS*.
[9] Robin Milner. 1989. *Communication and Concurrency*. Prentice Hall.
[10] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *SOSP*. ACM, 456–471.
[11] Yossi Shiloach and Uzi Vishkin. 1982. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms* 3, 1 (1982), 57–67.
[12] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. 2018. Shortcutting label propagation for distributed connected components. In *WSDM*.
[13] Leslie G. Valiant. 1990. General Purpose Parallel Architectures. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. 943–972.
[14] Xiangyu Zhi, Xiao Yan, Bo Tang, Ziyao Yin, Yanchao Zhu, and Minqi Zhou. 2023. CoroGraph: Bridging Cache Efficiency and Work Efficiency for Graph Algorithm Execution. *PVLDB* 17, 4 (2023).
[15] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *USENIX ATC*. 375–386.
[16] Xiaoke Zhu, Yang Liu, Shuhao Liu, and Wenfei Fan. 2023. MiniGraph: Querying big graphs with a single machine. In *PVLDB*, Vol. 16. 2172–2185.