

Terascale Query Processing in the Browser: *Making GPU Acceleration More Accessible*

Jiaxin Lu*, Landon Dyken*, Yihao Sun[†], Kristopher Micinski[‡], Thomas Gilray[§], and Sidharth Kumar*

*University of Illinois at Chicago, Chicago, IL, USA, {jlu73, ldyke, sidharth}@uic.edu

[†]Utah State University, Logan, UT, USA, yihao.sun@usu.edu

[‡]Syracuse University, Syracuse, NY, USA, kkmicins@syr.edu

[§]Washington State University, Pullman, WA, USA, thomas.gilray@wsu.edu

Abstract—Recursive query computation, central to graph algorithms and relational databases, demands GPU acceleration due to inherent computational intensity. While substantial prior work addresses GPU implementations of recursive queries, requiring fixed-point evaluation, existing systems are restricted to native execution environments. We introduce **WGLog**, the first web-browser-native GPU engine for compute-bound recursive database queries. **WGLog** is built entirely on WebGPU compute shaders, a cross-platform API that enables GPU acceleration in web browsers. **WGLog** leverages two key technical innovations. First, we replace hash-table-based joins with sorted-array joins, eliminating both the repeated probing and the repeated index construction that hash-based pipelines incur across fixed-point iterations. Second, we develop an asynchronous execution pipeline using WebGPU’s indirect dispatch capability, which eliminates GPU-host synchronizations that would otherwise dominate per-iteration overhead. On representative workloads, **WGLog** delivers $1.48 - 4.68\times$ speedup over native GPU systems and orders of magnitude improvement over CPU and WebAssembly implementations.

I. INTRODUCTION

Relational query processing with SQL excels at single-pass analytical workloads but struggles with recursion. Designed for flat, acyclic computations, SQL makes recursive queries such as graph closure and transitive reachability cumbersome to express and inefficient to evaluate. Datalog [19], [23], by contrast, is a simple yet expressive declarative language purpose-built for recursion, powering state-of-the-art systems across program analysis [8], [45], network and graph analytics [13], [37], and knowledge-graph reasoning [47]. These applications share a common computational pattern in which new facts are iteratively derived from existing data until reaching a *fixed point*, a state where further rule application produces no new tuples.

At scale, recursive queries become bandwidth-bound as iterations repeatedly stream through millions of tuples [27]. What such a query costs is set by the recursion rather than by the input, since every iteration rescans the relation accumulated so far, and a graph of a few hundred thousand edges can move hundreds of gigabytes before it converges. Mainstream Datalog engines such as Soufflé [45] and DDlog [34] run on CPUs and are limited by CPU memory bandwidth when closures reach tens of millions of tuples (80 million in the largest graph we evaluate [42], [46]). GPUs are a natural alternative because their high-bandwidth memory and massive

parallelism match this access pattern. Recent CUDA systems confirm this potential. State-of-the-art GPU-based Datalog engines, mnmgJOIN [42] and GPULog [46], accelerate fixed-point computation, achieving order-of-magnitude speedups over Soufflé on large graphs. These results, however, come tied to a specific vendor ecosystem.

Both engines require NVIDIA-specific toolchains and produce native binaries, restricting deployment to systems with a CUDA driver installed. This restriction matters most where demand for local analytics is growing fastest [12], [22], [26], [43], [44]. As data processing shifts on-device for privacy and compliance, secure private computation has become a first-order design constraint. Medical, financial, and personal-data workloads often cannot be uploaded to remote servers and must be processed locally, keeping sensitive data under the user’s control. A web-based query engine directly supports this model, since computation runs entirely within the browser sandbox and data never leaves the device. Yet no existing Datalog engine serves this setting, and deploying portable Datalog outside CUDA environments remains an open problem. The gap is striking given how far GPU hardware has spread. Once confined to specialized systems, GPUs are now ubiquitous [25], with nearly all modern laptops containing powerful ones, yet no infrastructure exists to exploit them within the *portable web* environment, and the browser remains disconnected from GPU acceleration despite serving billions of users daily. A GPU-accelerated database engine in the browser would close this gap, democratizing high-performance analytics and letting users process large, sensitive datasets locally on existing hardware without specialized toolchains or expertise.

We address this gap with **WGLog**, the first GPU-accelerated database engine designed for browser execution, built entirely on WebGPU compute shaders. WebGPU [48] provides a portable GPU compute API that runs in browsers without driver installation. Instead of being a port of CUDA implementation, **WGLog** is designed from scratch. A direct port of existing CUDA pipelines to WebGPU fails for two fundamental reasons. First, a naive port would inherit the hash-based indexing that existing CUDA techniques [42], [46] maintain throughout fixed-point evaluation, a cost that surfaces in different places across the two approaches. One technique [42] incurs its overhead during lookup because a

probe terminates only upon reaching an empty slot and must therefore traverse the entire occupied run from its initial hash position, examining both matching and nonmatching entries. The traversal length increases with the degree of the probed key. The other [46] pays at construction time, since it rebuilds the index once per iteration and the tuples with the same key contend for a single entry in the hash table. Second, WebGPU’s record-submit-fence command model makes per-stage host synchronization expensive when repeated across hundreds of fixed-point iterations, unlike CUDA’s approach. To address the first problem, WGLog redesigns the recursive evaluation pipeline around sorted arrays rather than hash indexes. Our experiments show that, for the evaluated baselines and workloads, building and searching sorted arrays on the GPU costs less than constructing and probing hash indexes. Binary search directly locates each matching range, avoiding collision handling and nonmatching slot traversal the existing implementations incur. WGLog handles the second problem by observing that Datalog’s fixed-point semantics require only a termination signal rather than per-stage intermediate sizes [1]. Exploiting this, WGLog batches multiple iterations into a single WebGPU command submission using indirect dispatch, which lets the GPU compute its own dispatch parameters and frees the host from reading back intermediate sizes.

We evaluate WGLog on three representative compute-bound workloads, transitive closure (TC), same-generation (SG) queries, and triangle counting. Collectively, these workloads demonstrate terascale execution, with WGLog moving approximately 0.65 TB of GPU memory traffic and performing on the order of 0.5×10^{12} integer operations in 9.1 seconds on a laptop GPU with only 6 GB of memory. Recursive execution thus amplifies a modest resident dataset into cumulative work that exceeds the device’s memory capacity by orders of magnitude. Because WebGPU exposes no hardware performance counters, both estimates are derived from per-iteration relation sizes, and the traffic figure counts only compulsory stage transfers and is therefore a lower bound.

This paper makes the following contributions:

- 1) A WebGPU-based Datalog backend built on index nested-loop joins over sorted relations, replacing the hash-based pipelines of prior GPU engines (§IV).
- 2) A batched indirect-dispatch fixed-point loop that keeps the host out of the iteration entirely, cutting a direct port’s end-to-end time by $15.4\times$ (§V).
- 3) An implementation and evaluation on 12 TC, 4 SG, and 5 triangle-counting graphs, with cumulative speedups of $2.38\times$ over mnmJoin, $3.05\times$ over GPULog, and $25.32\times$ over Soufflé (§VII).

To our knowledge, this is the first GPU-accelerated implementation of recursive database queries for the web browser. We also compare against three WebAssembly ports of popular database engines (SQLite [44], DuckDB [11], and Ascent [35]). As expected, WGLog significantly outperforms these baselines because WebAssembly does not leverage GPU acceleration.

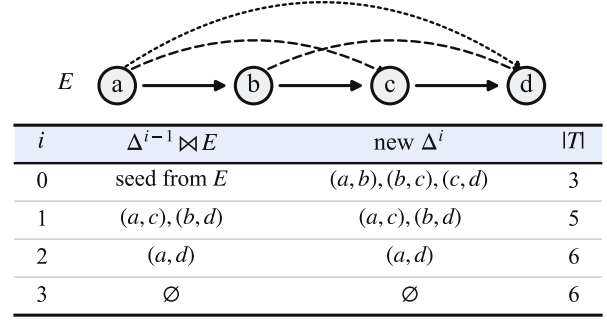


Fig. 1. Transitive Closure on the running-example chain $a \rightarrow b \rightarrow c \rightarrow d$. Top: solid arrows are the input relation E . The dashed arcs (length 2) are derived in iteration 1 and the dotted arc (length 3) in iteration 2. Bottom: Iteration 0 initializes the delta from E . Semi-naïve evaluation joins only the new delta Δ^{i-1} against E at each iteration, producing Δ^i until it becomes empty. T denotes the accumulated closure.

II. BACKGROUND

We use Datalog to express and execute large-scale recursive relational queries. It is a declarative logic programming language, where users specify *what* facts to derive, not *how* to derive them. A program consists of rules of the form:

$$H(\bar{x}) \leftarrow B_1(\bar{x}_1), \dots, B_k(\bar{x}_k).$$

The *head* H is the fact to derive; the *body atoms* B_i are conditions that must hold; shared variables express *joins*. Each rule body represents a relational-algebra expression where the conjunction of body atoms forms a join with equality conditions on shared variables. Datalog extends relational algebra with a fixed-point operator that gives recursive rules their iterative semantics. A rule whose body references the relation it defines is *recursive*. Datalog’s semantics are based on *fixed-point computation*, where rules are repeatedly applied to the database until no new fact is derived, reaching the *fixed point* [1]. An important property of these semantics is that derived relations are *sets*, so re-deriving an existing fact does not change the database. We illustrate Datalog’s fixed-point computation with Transitive Closure (TC), the running example used throughout the paper.

A. Transitive Closure Computation—A Recursive Query

Given a directed graph with edge relation E , the transitive closure (TC) computes the reachability relation $T(x, y)$ using one base rule (left) and one recursive rule (right):

$$T(x, y) \leftarrow E(x, y). \quad T(x, z) \leftarrow T(x, y), E(y, z).$$

The base rule copies each edge into T ; the recursive rule extends a known path by one hop. Consider the four-vertex chain in Figure 1 (top). The base rule seeds T with the three input edges (a, b) , (b, c) , (c, d) . A naïve evaluator then re-applies the recursive rule to every pair in T at every iteration. *Iteration 1* derives two new pairs: (a, c) from $T(a, b) \wedge E(b, c)$, and (b, d) from $T(b, c) \wedge E(c, d)$. Both are added to T . *Iteration 2* again applies the rule to every pair currently in T , including (a, b) and (b, c) . Because E has not changed between iterations, joining $T(a, b) \wedge E(b, c)$ a second time produces (a, c) again, and $T(b, c) \wedge E(c, d)$ produces (b, d) .

again—these are duplicates of facts already in T . Only one derivation is genuinely new: (a, d) , obtained by joining the freshly-added pair $T(a, c)$ with $E(c, d)$. Because T is a set, the re-derivations of (a, c) and (b, d) contribute nothing. The engine performs the join, finds the result already in T , and discards it. *Iteration 3* produces only such discarded re-derivations, and T has reached its fixed point of six pairs.

The source of redundancy is structural. Once a pair (x, y) enters T , every subsequent iteration re-applies the rule to (x, y) and re-derives every output it can produce. On real graphs with millions of pairs in T and hundreds of iterations, this re-work dominates the cost. Semi-naïve evaluation [1], the standard optimization in Datalog engines, eliminates this redundancy.

B. Semi-Naïve Evaluation

Carrying the same transitive closure example, semi-naïve evaluation exploits the insight that each pair needs to join with E only once, immediately after it is first derived. Re-joining existing pairs in later iterations would only re-derive outputs already in T . The algorithm therefore maintains two disjoint relations. A *full relation* holds facts that have already served as join input in earlier iterations, while a small *delta* Δ^i holds the facts newly derived in iteration i , waiting to be used as input in the next. Each iteration joins only Δ^i , rather than the full relation, against E . The algorithm proceeds as follows:

- 1) The base rule seeds T 's Δ^0 with the input edges.
- 2) Iteration i joins T 's Δ^{i-1} against E , producing candidate tuples.
- 3) From those candidates, the engine removes duplicates and subtracts (set difference) those already in T 's *full*.
- 4) The surviving candidates become T 's Δ^i and are merged into the full relation.
- 5) Evaluation terminates when Δ^i is empty.

Steps 2 through 4 form the fixed-point loop and repeat until the termination test in step 5 succeeds. Termination is guaranteed, since each iteration adds only facts that are not already present and the accumulated relation is bounded, so the delta will eventually be empty.

Figure 1 (bottom) traces this process on the running-example chain. The base rule seeds Δ^0 with the three input edges (a, b) , (b, c) , (c, d) , giving $|T| = 3$. *Iteration 1* joins Δ^0 against E to produce (a, c) and (b, d) ; both are new, so $\Delta^1 = \{(a, c), (b, d)\}$ and $|T| = 5$. *Iteration 2* joins Δ^1 against E : only (a, c) extends through $E(c, d)$ to yield (a, d) , since (b, d) has no outgoing edge from d ; thus $\Delta^2 = \{(a, d)\}$ and $|T| = 6$. *Iteration 3* finds no extension for (a, d) (which also dead-ends at d), so $\Delta^3 = \emptyset$ and the evaluation terminates. Each iteration joins only the latest delta against E rather than the whole of T , so the per-iteration work scales as $|\Delta^{i-1}| \cdot |E|$ rather than $|T|^2$.

The example uses TC, but semi-naïve evaluation is general. A Datalog engine maintains a delta for every relation appearing in both a rule head and a rule body, since only those relations grow during fixed-point evaluation. A relation appearing only in rule bodies, such as E here, never grows,

so the engine joins directly against its original data. The five steps are otherwise unchanged for any program; §VI states which query shapes WGLog covers.

C. GPU Implementation of Semi-Naïve Evaluation

From an implementation point of view, each iteration of the semi-naïve loop comprises five stages: join, sort, deduplicate, set difference, and merge. All five stages are data-parallel, making them well-suited to GPU acceleration. State-of-the-art GPU systems mnmJoin [42] and GPULog [46], both built on NVIDIA CUDA, implement each stage as a distinct kernel. The join phase is the most compute-intensive and has become the focus of optimization. Both mnmJoin and GPULog implement the join using hash tables, following the classical hash join approach [27], [39]. The base relation is inserted into a GPU-resident hash table, and each delta tuple probes it to find matching edges.

Between iterations, the host reads back the delta size to decide whether the fixed point has been reached. This readback is inexpensive under CUDA, since a single `cudaMemcpy` of four bytes takes $8.8 \mu\text{s}$ in our measurements (§VII). Thus, per-iteration host coordination is essentially free, and the cost that remains is the hash index itself, which the two engines pay for in different places (§III). WebGPU, the portable alternative we target, departs from CUDA precisely on this assumption, as the command model described in §II-D makes the same four-byte readback far more expensive.

D. WebGPU Compute

WebGPU [48] provides a portable GPU compute API that runs in browsers and native runtimes, with shaders written in WGSL [49]. Implementations map WebGPU commands to platform backends such as Vulkan [24], Direct3D 12 [33], and Metal [2]. For recursive query evaluation, four aspects of WebGPU's execution model are relevant to our work.

Dispatch and workgroups. The basic unit of work is a compute dispatch, which launches a grid of *workgroups* (analogous to CUDA blocks). Each workgroup contains a fixed number of invocations (analogous to CUDA threads) that share on-chip memory and synchronize through barriers. Invocations within a workgroup are further grouped into *subgroups* (analogous to CUDA warps) that execute in lockstep and exchange values through dedicated intrinsics.

Indirect dispatch. In a standard dispatch, the host specifies the grid dimensions. In an *indirect dispatch*, the GPU reads them from a buffer that a prior kernel has written. This mechanism is critical when the output size of one pipeline stage determines the grid size of the next, which is exactly the situation in each semi-naïve iteration.

Command model. WebGPU follows a record-submit-fence model, where the host records dispatches into a command buffer, submits the buffer, and cannot observe any result until the entire buffer completes. Reading a value back requires two additional steps: copying it to a CPU-mappable *staging buffer* and issuing an asynchronous `mapAsync` call that yields the host thread until the GPU finishes. The resulting round-trip

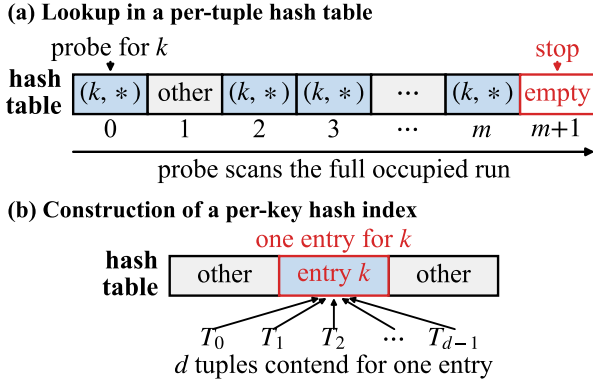


Fig. 2. What a join key of out-degree d costs each design. (a) A hash table with one slot per tuple [42] pays on every lookup, since a probe crosses the whole occupied run before it reaches an empty slot. (b) A hash index with one entry per distinct key [46] pays on construction, where the d tuples sharing the key contend for that entry.

costs ~ 2.5 ms in our measurements (§VII), roughly $285\times$ the cost of the equivalent CUDA `cudaMemcpy`.

Buffer aliasing. WebGPU forbids a single buffer from appearing in two binding slots of the same compute pass when one slot is writable. Algorithms that update data in place under CUDA must instead alternate between two buffers, using a *ping-pong* pattern.

These constraints mean that porting a CUDA-style recursive engine to WebGPU by replacing API calls one-for-one would inherit two distinct bottlenecks. The first is the hash-join bottleneck from the algorithmic design, and the second is the readback bottleneck from the command model. Together, these set the cost floor that WGLog’s redesign must beat.

III. MOTIVATION

Existing GPU Datalog engines [42], [46] execute semi-naïve evaluation as a host-orchestrated pipeline of joins, sorting, duplicate removal, set difference, and merge. Expressing this pipeline through WebGPU exposes three structural problems. First, both engines rely on *hash-based indexing*, whose cost recurs at every fixed-point iteration (§III-A). Second, both depend on *fine-grained host orchestration*, as data-dependent stage sizes draw the host into every iteration to read sizes back and allocate the buffers they determine (§III-B). Third, WebGPU’s resource-binding rules prohibit the input–output aliasing that parts of the CUDA pipelines assume, forcing separate buffers or additional passes (§III-C). Because the fixed-point loop repeats the first two costs across many iterations, the resulting overhead is structural rather than attributable to an isolated slow kernel.

A. Existing Hash-Index-Based Datalog Implementations

Existing CUDA-based engines, `mnmJoin` [42] and `GPU-Log` [46], evaluate the recursive join through hash indexes but incur overheads at different stages, `mnmJoin` at lookup and `GPU-Log` at index construction. We examine each in turn.

Expensive lookup in a per-tuple hash table [42]. This design holds one slot per tuple, so a vertex of out-degree

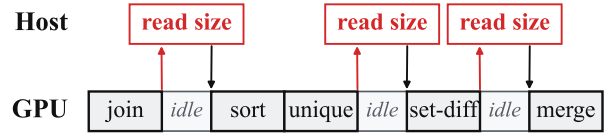


Fig. 3. One host-driven fixpoint iteration. Three of the five stages produce a size that is not known until the stage has run, so the host must read it back before the next stage can be launched, and GPU idles across each round trip.

d occupies d slots. Because insertion runs in parallel under linear probing, the entries for one key are not guaranteed to be adjacent, and a probe cannot stop when the key changes. It stops only at an empty slot, examining every occupied slot between its hash position and the end of the run, matching or not (Figure 2(a)).

Construction cost of a per-key hash index [46]. In this implementation, the hash table stores one entry per distinct join key over a sorted index array, so a key occupies one entry whatever its degree, and a probe reaches the matching range without walking a cluster. Its recurring cost instead appears during construction. A new index is built for each delta relation, and all tuples sharing a key attempt to update that key’s single entry through compare-and-swap operations that serialize under contention (Figure 2(b)).

The two engines therefore move hash-index costs rather than eliminating it. The per-tuple table pays during repeated probe traversal, whereas the per-key index pays during repeated construction and atomic coordination. WGLog avoids both hash-specific costs by performing the recursive join directly over sorted relations. This distinction is particularly important under WebGPU. WGS� offers only a weak compare-and-swap that may fail even when the compared value matches [49], so every insertion into a hash table has to be written as a retry loop. Because the relations are already sorted, WGLog locates each matching range with at most $\lceil \log_2 |E| \rceil$ comparisons and scans the range until the key changes. The recursive loop therefore requires no hash table construction or collision resolution (§IV), and §VII-C isolates the effect of this design choice.

B. Overhead of Host-Driven Fixpoints

Semi-naïve evaluation produces intermediate results whose sizes depend on the data, and in a host-orchestrated pipeline every such size becomes a point of host–device coupling. Three of the five stages produce a size the host cannot predict: the join, whose output depends on how many matches each delta tuple finds, duplicate removal, and the set difference against the accumulated relation (Figure 3). The other two impose no such constraint, since sorting preserves the tuple count and the disjoint merge produces exactly $|\Delta| + |Full|$ tuples, a quantity the host already holds.

Each data-dependent size gives the host two responsibilities inside the fixed-point loop. It must read the value back before launching the next stage with the correct grid dimensions, and, precisely because the size was unknown, it must allocate the buffer that stage will write into. Under CUDA the first is

cheap. A four-byte `cudaMemcpy` costs $8.8\mu s$ (§VII-D), so three readbacks add some $26\mu s$ per iteration. The second is not. Inside the loop the per-tuple engine allocates and frees the join output, the deduplicated delta, the temporaries that Thrust [4] and CUB [30] require in four separate primitives, and, on every iteration, a fresh buffer for the entire merged full relation. Non-compute time accounts for 68–70% of its end-to-end time on the three longest-running graphs. GPULog’s per-key pipeline shows a lower non-compute share of 21–34% because its per-iteration GPU work is heavier, but the absolute overhead remains substantial.

WebGPU substantially amplifies the first cost as well. Its record-submit-fence command model turns the same four-byte observation into a 2.503 ms round trip, $285\times$ the CUDA cost (§VII-D), so three readbacks per iteration cost 7.5 ms, and a query converging in a few hundred iterations would spend seconds inside these round trips alone. A host-driven loop is therefore not merely wasteful under WebGPU but unaffordable. Datalog’s declarative semantics, however, require the host to observe nothing beyond whether the delta has become empty. WGLog exploits this by keeping every intermediate size in device memory and fixing buffer capacities once during query initialization, so the fixed-point loop requires neither intermediate-size readbacks nor dynamic buffer allocation (§V, evaluated in §VII-D).

C. WebGPU Binding Rules Constrain In-Place Updates

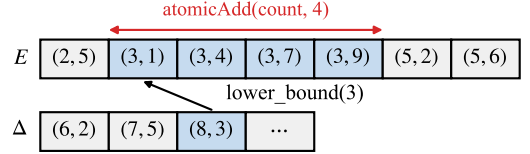
The third issue is WebGPU’s buffer binding model. CUDA implementations can often read and write the same global-memory allocation inside one kernel or across tightly coupled library calls. WebGPU imposes stricter resource aliasing rules. Overlapping regions of the same buffer cannot be bound through separate input and writable bindings within a single dispatch [48]. This restriction prevents the input–output aliasing used by several stages of the recursive Datalog pipeline, including filtering, radix-sort passes, and merge.

A direct copy-back design would preserve the single-buffer abstraction, where each stage writes a temporary output and then copies it back into the input buffer expected by the next stage. That design is simple but bandwidth-expensive, and recursive queries repeat the cost across many iterations. The alternative is a ping-pong design that alternates between two physical buffers. Ping-pong avoids the extra copy but consumes more memory and forces every stage to track which side currently holds the valid relation. WGLog addresses this restriction with ping-pong arrays, where each stage reads from one buffer set and writes to the other (§V, evaluated in §VII-E).

IV. DESIGNING A SORTED-ARRAY PIPELINE

As introduced in §II, semi-naïve evaluation iterates over a five-stage loop of join, sort, deduplicate, set difference, and merge until the delta becomes empty and the fixed point is reached. This section presents three sorted-array primitives that implement the join, duplicate removal, set difference, and merge stages without hash-based indexing. They are an

(a) Count phase



(b) Emit phase

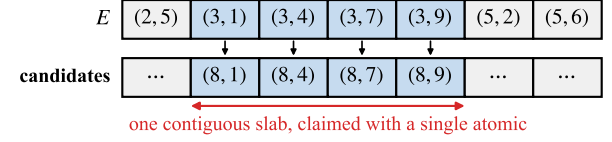


Fig. 4. Index nested-loop join, tracing the shaded delta tuple (8,3). The count phase binary-searches its matching range in E and adds the length of that range to the global counter, and the emit phase repeats the search, reserves one contiguous output slab, and writes the matches into it. Each phase issues one atomic per delta tuple.

index nested-loop join for the join stage (§IV-A), a fused deduplication and set-difference kernel (§IV-B), and a disjoint merge that incorporates the new delta into *Full* (§IV-C).

A. Index Nested-Loop Join

WGLog implements the join at the heart of every fixed-point iteration as an *index nested-loop join* [17], [36], with sorted order playing the role of the index. All relations are maintained in lexicographic order, and this ordering rests on a single invariant: the base, delta, and full relations are sorted at every stage boundary. Sorting groups all tuples sharing a join key into one contiguous range, which turns each probe into a range query. To find its matches, a probing tuple binary-searches for the start of the range and then scans sequentially over exactly the matching tuples, touching no others. A hash-based probe, by contrast, may walk a collision cluster containing nonmatching keys before isolating its matches.

This differs from a sort-merge join [7], [17]. A sort-merge join requires both inputs to be ordered by the join key so that matching groups can advance together. In recursive evaluation, however, the tuples being probed are maintained in a fixed lexicographic order that does not necessarily place the join key first. A sort-merge join would therefore require reordering those tuples before every recursive join. The index nested-loop join avoids this step because WGLog searches the sorted base relation independently for the join key of each tuple.

The TC query provides a concrete example. The recursive join pairs each newly discovered tuple (a, b) in the delta with every base edge (b, c) in E , producing candidates (a, c) . WGLog sorts E once at dataset load and never modifies it. The delta is maintained in lexicographic order on (a, b) , while the join uses b as its lookup key. Tuples sharing the same b are therefore not necessarily adjacent because the relation is ordered first by a . A sort-merge join would have to reorder the delta by b in every iteration. Instead, for each tuple (a, b) , WGLog binary-searches E for the first edge whose source is b , scans the contiguous range of edges (b, c) , and emits one candidate (a, c) for every match.

Two-phase join algorithm. The central difficulty of a GPU join is that its output size is not known in advance. Thousands

of threads perform the join in parallel, and each must write its results into a shared output buffer, yet the total number of candidates is not known until the count-phase lookups complete, and the output positions are assigned later during the emit phase. The TC join makes this concrete. Each delta tuple (a, b) from relation T looks up all base edges (b, c) in relation E ; because E is sorted by source vertex, these edges form one contiguous range whose length is the out-degree of vertex b , and each edge produces one candidate (a, c) . Since out-degrees vary across vertices, different delta tuples produce different numbers of candidates, and the total output size emerges only after the lookups complete. **WGLog** therefore splits the join into a count phase and an emit phase (Figure 4). Both phases assign one GPU thread per delta tuple and perform the same binary search in E , but use the resulting range differently.

In the count phase, each thread locates the first edge matching its tuple’s join key and scans to the end of the contiguous range, adding the range length to a global candidate counter. No candidate tuples are written. Once all threads finish, the counter holds the exact number of candidates the join will produce, and it is consumed on the device to configure the indirect dispatch of the subsequent radix sort. In the emit phase, each thread repeats the range lookup, obtaining the same range length d , and performs one atomic addition on a global output counter to reserve a contiguous region of d positions in the candidate array. The value returned by the atomic addition is the starting offset of that region, and the thread writes the candidates from its matching edges into consecutive positions beginning there.

What the sorted index buys. For a join key with d matching edges, **WGLog** uses the sorted edge relation as an implicit index. It locates the matching range with one binary search and then scans the d adjacent edges, requiring $O(\log |E| + d)$ work.

A hash table offers a better expected lookup bound because hashing identifies where a join key should appear in the table. With a fixed load factor and a well-distributed hash function, lookup requires a constant number of probes rather than the $O(\log |E|)$ comparisons to locate a range in a sorted relation. In the ported GPULog design, the hash table forms a per-key index. Its practical lookup advantage is small because the table uses linear probing at a load factor of 0.8, and each entry stores a hash rather than the full key, so the sorted relation must still be read to confirm a match. Consistent with this analysis, the emit phases of the two designs are within $1.05\times$.

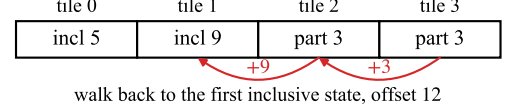
The larger performance difference appears across the complete two-phase join rather than in lookup alone. The ported per-key design computes a count for each delta tuple and converts these counts into global output offsets using a workgroup scan and decoupled lookback. In **WGLog**, the count phase instead sums the lengths of the matching ranges to obtain the total candidate count, which configures the subsequent radix-sort dispatch. The emit phase then uses one atomic operation per delta tuple to reserve a contiguous output range. This simpler two-phase path reduces cumulative count-and-join time by $1.54\times$ relative to the per-key design, even though

(a) Classify and scan

candidates	(8, 1)	(8, 1)	(8, 4)	(8, 7)	(8, 7)	(8, 9)	(9, 5)
	keep	dup	in Full	keep	dup	in Full	keep
flag	1	0	0	1	0	0	1
scan	1	1	1	2	2	2	3

one pass, two predicates, one workgroup scan

(b) Decoupled lookback



(c) Scatter survivors in input order

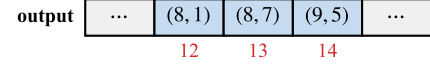


Fig. 5. Fused deduplication and set difference. The kernel classifies candidates, computes local offsets with a workgroup scan, obtains each tile’s global offset through decoupled lookback, and scatters surviving tuples in input order.

their emit-phase times are similar (§VII-C).

B. Fused Deduplication and Set Difference

After the join, the resulting candidates are sorted before becoming the next delta. Sorting supports the two filters required by semi-naïve evaluation. First, equal candidates become adjacent, so duplicate removal only needs to compare each candidate with its predecessor. Second, the accumulated full relation is sorted under the same lexicographic order, so set difference can test whether a candidate already exists through binary search. Both checks can therefore be applied while streaming the sorted candidates once. A candidate is retained only if it is the first copy among adjacent equals and does not appear in the full relation.

Fusing the filters. A direct implementation materializes a deduplicated intermediate array and then reads it again for set difference. **WGLog** avoids this intermediate write and read by fusing duplicate removal and set difference into a single tiled filtering kernel. A first dispatch computes the launch parameters from the candidate count. The second dispatch performs the fused filtering shown in Figure 5. Figure 5(a) shows one tile of sorted candidates. Each candidate receives a flag of one only when it differs from its predecessor and is absent from the full relation. A workgroup scan converts these flags into local output positions. Figure 5(b) shows how decoupled lookback adds the number of survivors produced by earlier tiles. In the example, the current tile walks back to an inclusive prefix of 9 and adds a partial count of 3, producing a global offset of 12. Figure 5(c) then scatters the surviving tuples $(8, 1)$, $(8, 7)$, and $(9, 5)$ to positions 12 through 14 while preserving their input order.

Tiled execution. The fused kernel processes candidates in fixed-size tiles of 3,072 tuples, using 256 threads with 12 tuples per thread [31]. Within each tile, threads classify candidates, assign local output positions through a workgroup prefix scan, obtain the tile’s global output offset through decoupled lookback [31], and scatter survivors in input order. Because

both the input order and the survivor order are preserved, the output remains sorted for the next stage. Across 6 TC graphs, fusion reduces cumulative execution time by 5.4%.

C. Disjoint Merge

The last stage of each iteration merges the delta into the accumulated full relation while preserving the sorted order required by the next iteration. The fused kernel has already removed tuples that appear in the full relation, so the two inputs are disjoint and can be merged directly without duplicate handling. WGLog uses merge-path partitioning [18] to divide the merge across workgroups. Merge path assigns each workgroup a diagonal in the conceptual merge grid, from which it derives the corresponding ranges in the two sorted inputs and the output range it must produce. Because the inputs are disjoint, the output contains exactly $|\Delta| + |Full|$ tuples, so every workgroup can write directly to its predetermined range.

V. WGLog’s ASYNCHRONOUS FIXED-POINT ENGINE

The sorted-array pipeline of §IV eliminates the hash index but does not by itself make the engine WebGPU-native. Host-driven stage transitions still create repeated synchronization (§III-B), and WebGPU’s binding model rules out CUDA-style input–output aliasing (§III-C). WGLog therefore follows a single engine rule. The host checks only whether the fixed point has been reached; the GPU does everything else. It computes data-dependent dispatch parameters, advances the recorded iterations, and routes intermediate relations between buffers. Three mechanisms implement this rule. One iteration passes relations, counts, and dispatch parameters entirely through device memory (§V-A). Indirect dispatch connects dependent stages without host readbacks and lets multiple iterations execute in one submission (§V-B). Ping-pong buffers provide separate I/O storage without copy-back traffic (§V-C).

A. Per-Iteration Asynchronous Pipeline

The three primitives of §IV, together with the radix sort [32] that reorders the join’s output, compose into a single fixed-point iteration (Figure 6). The join emits unsorted candidate tuples, radix sort restores their lexicographic order, the fused deduplication and set-difference kernel then extracts the new delta, and the disjoint merge folds it into the full relation. Relations, however, are not the only data flowing between stages. Each stage also produces counts that size the work its successor must launch. WGLog keeps these counts, and the dispatch arguments derived from them, resident on the GPU, so consecutive stages within an iteration proceed without any host involvement; candidate counts, filtered-delta counts, and indirect-dispatch arguments all live in device memory for the entire iteration. The next subsection extends this device-resident execution across multiple iterations, deferring host contact to a single termination check.

B. Batched Indirect-Dispatch Loop

§III-B showed that a data-dependent stage size pulls the host into the loop twice, once to read the size back before

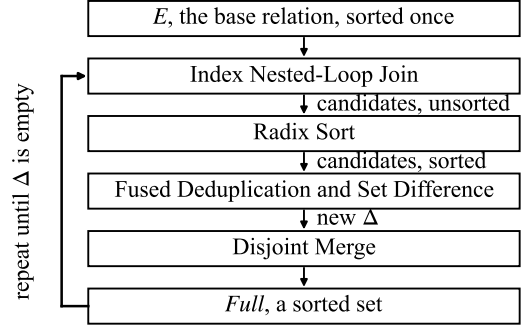


Fig. 6. Data flow of one fixed-point iteration of semi-naïve evaluation, showing all key steps.

launching the next stage, and once to allocate that stage’s output buffer. WGLog removes both requirements. The host records several consecutive fixed-point iterations into one command submission, which we call a *batch*; while a batch executes, all intermediate sizes stay in GPU-resident buffers and no stage reports its output size to the host. Allocation is likewise lifted out of the loop, as every intermediate buffer is created during query initialization with a capacity bound large enough for any iteration, so execution never allocates.

Indirect dispatch. Whenever a stage produces a data-dependent number of tuples, it writes that count to a 4-byte device buffer. A lightweight kernel reads the count, computes the workgroup dimensions the next stage requires, and writes them into an indirect-dispatch buffer, from which the next stage reads its launch configuration instead of receiving it from the host. The count also marks how many positions in the preallocated output buffer hold valid tuples. Because both the count and the dispatch arguments stay on the GPU, consecutive stages proceed without host–device synchronization, and the same mechanism carries the new delta size into the next iteration, letting all iterations recorded in a batch execute without returning to the host.

Adaptive batching. Because every dispatch, within and across iterations, reads its parameters from device memory, the host can record K consecutive iterations into a single command buffer and submit them as one unit, which the GPU then executes in sequence. After the batch completes, the host reads back the recorded delta sizes and checks whether any reached zero. This single batch-level observation replaces the per-stage readbacks of a host-driven loop. On SF.cedge [10], transitive-closure computation converges in 287 iterations. WGLog caps the batch at $K_{\max} = 30$ iterations, so a fixed-size schedule reduces the 287-iteration loop to $\lceil 287/30 \rceil = 10$ termination checks.

A larger K amortizes the per-batch submission cost over more iterations but risks overshoot, since iterations recorded after the fixed point is reached dispatch on an empty delta and waste GPU time, a waste that dominates on datasets with short fixed-point computations where K may exceed the total iteration count. For transitive closure, WGLog starts from $K_{\max} = 30$ and halves the next batch whenever a completed batch shows strong delta decay, with a last-to-first size ratio below 0.1. Because the halving is monotonic, the batch shrinks

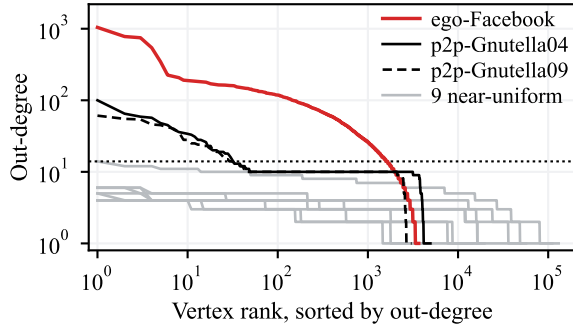


Fig. 7. Out-degree distributions of the 12 TC graphs. Maximum out-degree ranges from 4 to 1,043, with nine graphs at or below 14. Nonmatching probe visits remain at 37–67%, so degree mainly affects probe-tail length.

to one or two iterations near convergence, bounding the final empty work.

C. Ping-Pong Buffers

§III-C explained that WebGPU rejects overlapping input and output bindings within one compute pass. The engine therefore maintains two buffer sets, A and B, for each relation. A stage reads from one set, writes to the other, and the output set becomes the input for the next stage. This ping-pong rule governs the entire recursive pipeline. The fused filter reads candidates from one side and writes the new delta to the other and, the disjoint merge does the same for the full relation, giving one consistent buffer-management policy for every stage that produces a new relation. The benefit is that no stage copies its result back into the buffer it read from. The cost is a second buffer set per relation, a memory tradeoff that §VII-E measures.

VI. GENERALIZATION ACROSS DATALOG QUERIES

Datalog can express a wide range of queries beyond transitive closure, including graph reachability, ancestor relationships, social-network analysis, knowledge-graph reasoning, and triangle or clique enumeration [1], [8], [13], [37], [47]. The design in §§IV–V is not tied to the TC rule. Its sorted-array operators can be reused across different Datalog queries. We illustrate this generality with Same Generation and Triangle Counting. Same-generation exercises the complete recursive engine under a different join structure, while triangle counting isolates the index nested-loop join non-recursively.

Same Generation is a canonical recursive query over a base *parent* relation, defined by a seed rule and a recursive rule:

$$\begin{aligned} sg(X, Y) &\leftarrow \text{parent}(X, P), \text{parent}(Y, P), X \neq Y, \\ sg(X, Y) &\leftarrow \text{parent}(X, P_x), \text{parent}(Y, P_y), sg(P_x, P_y). \end{aligned}$$

The query derives pairs of nodes at the same generational depth. The seed rule produces sibling pairs that share a parent, while the recursive rule extends an existing pair (P_x, P_y) to pairs of their children. For each delta tuple (P_x, P_y) , the join locates the two ranges of children associated with P_x and P_y in the sorted *parent* relation. If the ranges contain k_x and k_y tuples, the join emits a cross product of $k_x k_y$ candidate pairs. TC instead searches one matching range of size d and emits

d candidates. Supporting SG changes the seed and the join kernel structure. The seed scans each parent group and emits all sibling pairs within that group. The recursive join performs two binary searches and enumerates the cross product of the resulting ranges. The fused deduplication and set-difference kernel, disjoint merge, and batched fixed-point loop are reused unchanged (§§IV–V-B).

Triangle Counting is a non-recursive three-way self-join over the edge relation:

$$\text{triangle}(a, b, c) \leftarrow \text{edge}(a, b), \text{edge}(b, c), \text{edge}(c, a).$$

Because the rule is non-recursive, it reuses the index nested-loop join without delta maintenance, set difference, merge, or the batched fixed-point loop. For each edge (a, b) , the kernel first locates the range of edges (b, c) and then checks whether each closing edge (c, a) exists.

VII. EVALUATION

Evaluation platform. We use an NVIDIA GeForce RTX 3060 Laptop GPU with 6 GB of memory. We intentionally choose a consumer-grade laptop GPU because WGLog targets browser-hosted analytics on personal hardware rather than datacenter accelerators. The same machine runs Chromium with WebGPU’s Dawn/D3D12 backend and native CUDA 13.1 with the Thrust and CUB libraries. WebGPU subgroup operations and timestamp queries are enabled.

Datasets. We evaluate two recursive workloads, Transitive Closure (TC) and Same-Generation (SG), on graphs from SNAP [28] and SuiteSparse [10]. We select graphs that fit within the laptop GPU’s 6 GB memory and cover a range of relation sizes and degree distributions. The TC suite contains 12 graphs with closure sizes ranging from 146K to 80M tuples, while the SG suite contains 4 graphs with closure sizes ranging from 23K to 66M tuples. As shown in Fig. 7, the datasets exhibit varying degrees of skewness, allowing us to evaluate whether WGLog remains effective across different graph structures. The triangle-counting microbenchmark in §VII-C uses 5 SNAP graphs whose maximum out-degrees range from 343 to 2,540.

Baselines. For GPU comparison, we use mnmgJOIN [42] and GPULog [46], two CUDA engines based on hash indexing. For CPU comparison, we use Soufflé [45] as a native compiled baseline and sql.js [44], Ascent-WASM [35], and DuckDB-WASM [11] as browser-based baselines.

Methodology. For each production configuration and dataset, we perform 10 warmup executions followed by 100 timed executions and report the arithmetic mean. TC uses an adaptive batch size capped at $K_{\max} = 30$ because its iteration counts vary widely and large batches can overshoot near convergence. SG uses a fixed batch size of $K = 10$ because its evaluated runs require fewer iterations and adaptive shrinking provides little additional benefit. The larger ablation sweeps also use 10 warmup executions followed by 100 timed executions per configuration. End-to-end time measures the wall-clock duration of the fixed-point loop and excludes binary input loading, CPU lexicographic sorting, and initial

TABLE I

TC END-TO-END TIME (MS) ON 12 GRAPHS. $|TC|$ IS THE CLOSURE SIZE IN TUPLES. SPEEDUPS BELOW $1\times$ INDICATE WGL_{OG} IS SLOWER. THE SOUFFLÉ COLUMN, IN BOLD, IS THE COMPILED CPU DATALOG REFERENCE, AND WGL_{OG} OUTPERFORMS IT ON EVERY GRAPH BUT THE SMALLEST.

Dataset	Iter.	$ TC $	WGL _{OG}	mnmgJOIN	GPULog	Soufflé	speedup vs. mnmgJOIN / GPULog / Soufflé
OL.cedge	64	146,120	68.538	54.83	94.78	65.9	$0.80\times / 1.38\times / \mathbf{0.96\times}$
cal.cedge	195	501,755	155.366	332.48	318.50	203.4	$2.14\times / 2.05\times / \mathbf{1.31\times}$
TG.cedge	58	481,121	55.551	83.88	103.32	233.7	$1.51\times / 1.86\times / \mathbf{4.21\times}$
p2p-Gnutella09	20	21,402,960	239.319	313.51	1131.98	13,751.6	$1.31\times / 4.73\times / \mathbf{57.46\times}$
p2p-Gnutella04	26	47,059,527	537.248	789.75	2654.20	34,477.6	$1.47\times / 4.94\times / \mathbf{64.17\times}$
cti	53	6,859,653	127.389	252.23	401.28	3,959.2	$1.98\times / 3.15\times / \mathbf{31.08\times}$
fe_sphere	188	78,557,912	1965.439	4972.56	5778.39	63,779.5	$2.53\times / 2.94\times / \mathbf{32.45\times}$
ego-Facebook	17	2,508,102	149.445	164.39	547.17	3,683.0	$1.10\times / 3.66\times / \mathbf{24.64\times}$
luxembourg_osm	426	5,022,084	598.227	1208.42	1375.92	2,929.5	$2.02\times / 2.30\times / \mathbf{4.90\times}$
wing	11	329,438	38.758	15.50	28.29	192.3	$0.40\times / 0.73\times / \mathbf{4.96\times}$
delaunay_n16	101	6,137,959	208.760	405.00	500.34	3,008.9	$1.94\times / 2.40\times / \mathbf{14.41\times}$
SF.cedge	287	80,498,014	2467.678	7180.45	7211.61	41,107.4	$2.91\times / 2.92\times / \mathbf{16.66\times}$

TABLE II
SG END-TO-END TIME IN MS ON 4 GRAPHS.

Dataset	Iter.	$ SG $	WGL _{OG}	mnmgJOIN	GPULog	vs. mnmgJOIN / GPULog
OL.cedge	56	285,431	62.317	101.58	207.37	$1.63\times / 3.33\times$
cal.cedge	58	23,519	58.142	52.91	153.49	$0.91\times / 2.64\times$
TG.cedge	54	608,090	63.325	108.29	228.60	$1.71\times / 3.61\times$
fe_ocean	77	65,941,441	1998.453	2977.80	9614.56	$1.49\times / 4.81\times$

GPU preprocessing. Per-stage GPU time is measured using WebGPU timestamp queries.

A. End-to-End vs. GPU SOTA

We compare WGL_{OG} against the two CUDA GPU baselines on the 12 TC graphs and the 4 SG graphs. Tables I and II report end-to-end time and per-dataset speedup. Table I also includes Soufflé as a compiled native CPU reference.

Cumulatively, WGL_{OG} is $2.38\times$ faster than mnmgJOIN and $3.05\times$ faster than GPULog on TC. On SG, the corresponding speedups are $1.48\times$ and $4.68\times$. Against Soufflé, WGL_{OG} is $25.32\times$ faster cumulatively on TC and faster on 11 of the 12 graphs. The only exception is the smallest workload, OL.cedge, where WGL_{OG} is $1.04\times$ slower.

We make three observations. *First*, the largest absolute savings occur on the largest TC workloads. The four largest closures, p2p-Gnutella09, p2p-Gnutella04, fe_sphere, and SF.cedge, contain 21–80M tuples and account for $\sim 78\%$ of cumulative TC time, and on them WGL_{OG} achieves 1.31 – $2.91\times$ speedups over mnmgJOIN and 2.92 – $4.94\times$ over GPULog. This is where the structural costs bite at scale, as large closures inflate the relation data the hash-based pipelines must process, and long runs repeatedly incur host-driven synchronization.

Second, different graphs emphasize different parts of the design. ego-Facebook converges in only 17 iterations, so its $3.66\times$ speedup over GPULog is unlikely to come primarily from multi-iteration batching; it is instead consistent with the recurring index-construction and materialization costs of §III-A, examined further in §VII-C. luxembourg_osm shows the opposite pattern, running 426 iterations with little work in each, and its $2.02\times$ speedup over mnmgJOIN is consistent with amortizing host synchronization across batches, an effect the submission-policy ablation in §VII-D measures directly.

Third, workloads with little useful GPU work can favor native CUDA. On OL.cedge, whose closure holds only 146K tuples, and wing, which converges in 11 iterations, WGL_{OG} runs $1.25\times$ and $2.50\times$ slower than mnmgJOIN, since browser and submission overheads are not amortized by enough computation. These small workloads contribute little to cumulative runtime, however, and across the larger TC workloads WGL_{OG} retains a cumulative advantage of 9.16 s over mnmgJOIN.

SG generalization. The SG results show different trends against the two GPU baselines. Against mnmgJOIN, WGL_{OG} achieves a cumulative speedup of $1.48\times$ on SG, compared with $2.38\times$ on TC. SG’s bilinear join performs more GPU work per iteration than TC’s single-range join. This increases the fraction of execution spent on join computation and reduces the relative contribution of the host-side synchronization removed by WGL_{OG}. The resulting speedup over mnmgJOIN is therefore smaller on SG.

Against GPULog, WGL_{OG} achieves a larger cumulative speedup of $4.68\times$ on SG. SG can generate large intermediate outputs because each recursive join emits the cross product of two matching ranges. The result is consistent with higher costs from materializing these outputs and rebuilding GPULog’s per-key hash index in each iteration.

B. Comparison to Browser-Based Recursion Engines

Browser-based deployment of recursive queries currently relies primarily on WebAssembly CPU runtimes, which compile database engines to WASM and execute them through the browser’s WebAssembly runtime. We compare WGL_{OG} against three such runtimes drawn from the SQL, Datalog, and analytical-database families. These baselines are sql.js [44], which ports SQLite, Ascent-WASM [35], which ports the Rust Datalog engine Ascent, and DuckDB-WASM [11]. All three share WGL_{OG}’s zero-install deployment context but lack GPU acceleration. Table III reports their cumulative end-to-end time and the corresponding speedup of WGL_{OG}.

We make two observations. *First*, all three baselines execute recursive queries on the CPU, while WGL_{OG} executes their data-parallel operators on the GPU. WGL_{OG} outperforms them by more than an order of magnitude. The speedups range from 14 – $132\times$ on TC and 24 – $309\times$ on SG. DuckDB-WASM and Ascent-WASM are the most competitive browser baselines, but

TABLE III
CUMULATIVE END-TO-END TIME IN SECONDS FOR BROWSER-BASED WASM BASELINES. THE WGLog REFERENCE TIMES ARE 6.61 s FOR TC AND 2.18 s FOR SG.

Baseline	TC time	TC speedup	SG time	SG speedup
sql.js (browser)	873.00	132.0×	673.64	308.7×
Ascent-WASM (browser)	140.93	21.31×	60.96	27.94×
DuckDB-WASM (browser)	94.92	14.35×	52.97	24.28×

TABLE IV
JOIN STRATEGIES IN THE SAME WebGPU PIPELINE. TIMES ARE CUMULATIVE MILLISECONDS OVER SIX TC GRAPHS, WITH SLOWDOWNS RELATIVE TO WGLog IN PARENTHESES.

Join Strategy	Count and Join	End-to-End
Per-tuple hash [42]	97.25 (1.74×	825.45 (1.065×
Per-key hash [46]	86.13 (1.54×	791.64 (1.022×
Index nested-loop (WGLog)	55.76	774.72

both remain substantially slower than WGLog on the evaluated recursive workloads.

Second, the performance gap grows on SG. SG’s bilinear recursive join generates a cross product of two matching ranges, exposing more parallel candidate-generation work to the GPU than TC’s single-range join. The larger SG speedups are consistent with WGLog benefiting more from GPU execution as the amount of data-parallel work per iteration increases.

C. Sensitivity Analysis—Sorted-Array Pipeline

We isolate contribution 1, the replacement of hash-based joins with a sorted-array pipeline, through two complementary experiments. The within-pipeline ablation (Table IV) ports each published join strategy into WGLog’s recursive engine while holding every other stage fixed. The triangle-counting microbenchmark (Table V) compares the strategies on a non-recursive query, removing loop overhead. Of the two baselines, the comparison with mnmgJOIN isolates the lookup strategy more directly, since both implementations stream probes without materializing the two-hop relation, whereas the comparison with GPULog additionally reflects the cost of writing and reading its materialized `two_hop` intermediate.

We make two observations. *First*, the index nested-loop join outperforms both hash-based strategies when every other stage is held fixed. Across the six graphs, the per-tuple hash table requires 1.74× as much count-and-join time as WGLog, and the per-key hash index 1.54×; on `ego-Facebook`, these slowdowns rise to 2.70× and 1.57×. The per-tuple design pays for traversing occupied slots during lookup, while the per-key design reduces this traversal but must construct its index, making it the closer alternative. Both results are consistent with §III-A. The end-to-end gaps are smaller, 1.065× and 1.022×, because count and join account for only 7.2% of WGLog’s cumulative time on these graphs, and the radix sort and set operations, identical across configurations, dilute the join-level difference. The join advantage thus remains visible but cannot dominate a pipeline that spends most of its time elsewhere; the command-model ablation in §VII-D measures the recursive loop’s contribution separately.

Second, the numbers show the advantage persists without the recursive loop. WGLog outperforms mnmgJOIN on all five graphs by 1.34–11.96×, and GPULog by 23.0–1,210× on

TABLE V
TRIANGLE-COUNTING TIME ON FIVE SNAP GRAPHS.

Dataset	edges	max-deg	WGLog	mnmgJOIN	GPULog
email-Enron	367,662	1,383	37.78	450.91	45,724.4
ca-AstroPh	396,100	504	17.82	173.47	710.7
com-amazon	1,851,744	549	26.29	35.20	605.9
soc-Slashdot0902	870,161	2,540	152.04	1,819.17	OOM
com-dblp	2,099,732	343	38.45	187.83	1,235.3

the four graphs that complete, though the latter also includes GPULog’s materialization cost. mnmgJOIN’s slowdown grows with the number of closing-edge checks per input edge, which equals the number of two-hop paths per edge. As this value rises from 11.5 on `com-amazon` to 140 on `email-Enron`, the slowdown climbs monotonically from 1.34× to 11.94×, consistent with unsuccessful hash probes traversing occupied slots before reporting absence (§III-A). GPULog additionally pays to materialize a `two_hop` intermediate of 21–51M tuples on the completed datasets, and on `soc-Slashdot0902`, with 870K edges and a maximum out-degree of 2,540, the intermediate exceeds the 6 GB of GPU memory and execution aborts. WGLog never materializes this intermediate, locating the edges leaving b for each input edge (a, b) in $O(\log |E| + d)$ work, where d is the range size, and checking each closing edge (c, a) with an $O(\log |E|)$ binary search.

D. Sensitivity Analysis—Batched Indirect-Dispatch Loop

We isolate contribution 2 through two experiments. The submission-policy ablation in Figure 8 changes when the host reads GPU state and how many fixed-point iterations are included in each submission while keeping all GPU kernels unchanged. The batch-size sweep in Figure 9 varies the number of consecutive fixed-point iterations recorded before each termination check and guides the production settings for TC and SG.

We make three observations about submission policy. *First*, reading a small control value through the host is expensive in WebGPU. A 4-byte readback costs 2.503 ms under WebGPU, compared with only 8.8 μs under CUDA, making the WebGPU operation 285× more expensive. This cost accumulates when a host-driven implementation reads intermediate sizes after every data-dependent stage. On `OL.cedge`, a naïve port that performs three readbacks per iteration takes 1273.6 ms.

Second, removing intermediate readbacks gives a 4.70× speedup. With $K = 1$, all dependent stages within an iteration obtain their dispatch sizes through indirect dispatch and execute without intermediate host involvement. The host reads back only the delta size needed to test termination after the iteration. This change reduces execution time from 1273.6 ms to 271.1 ms, a 78.7% reduction in execution time.

Third, batching several iterations provides a further 3.27× improvement over $K = 1$. A fixed batch size of $K = 10$ reduces execution time from 271.1 ms to 82.9 ms, while $K = 30$ reaches 82.6 ms. The latter is a cumulative 15.4× speedup relative to the naïve port. Although indirect dispatch removes synchronization between stages, one termination readback after every iteration remains expensive. Recording several iterations before the next termination check amortizes this remaining cost.

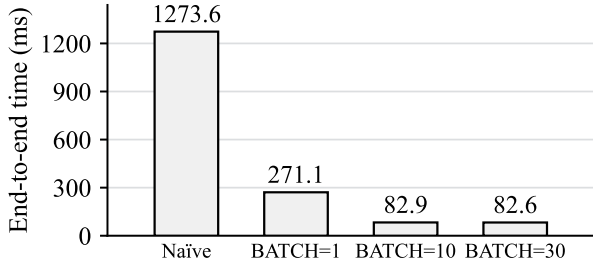


Fig. 8. Effect of removing host readbacks and batching WebGPU submissions on OL.cedge. Results average five cold runs from one dedicated driver and are not comparable with the warmed production time in Table I.

Batch-size selection. The TC sweep shows that the preferred fixed batch size varies across graphs. Queries with few iterations favor $K = 10$ – 30 , medium-length runs favor $K = 30$ – 40 , and the longest runs continue improving up to $K = 50$. Although some long-running graphs continue to benefit from larger batches, we set $K_{\max} = 30$ because it is the largest tested cap that improves all six representative TC graphs over $K = 20$. Fixed $K = 40$ regresses three graphs, while $K = 50$ regresses four. Adaptive shrinking then cuts cumulative time another 2.2% relative to fixed $K = 30$ by reducing overshoot near convergence. Across all 12 TC datasets, the resulting policy improves cumulative time by 7.8% over the previous adaptive policy capped at $K_{\max} = 10$.

For SG, a fixed batch size of $K = 10$ performs best on three of the four evaluated graphs. They require only 54–77 fixed-point iterations, leaving less opportunity to amortize termination checks through larger batches. A fixed size of $K = 30$ increases cumulative SG time by 4.8% because extra recorded iterations increase overshoot.

E. Ping-Pong vs. Copy-Back

WebGPU does not allow binding overlapping regions of one buffer as both a read-only input and a writable output within the same compute pass, so an implementation must either copy each result back into the original buffer or alternate between two buffers. Across 100 filter passes over 64M elements, ping-pong execution takes 221.78 ms while copy-back takes 390.10 ms, with the extra transfer making copy-back 75.9% slower. WGLog therefore uses ping-pong buffers for the repeatedly updated relations in the fixed-point loop, at the cost of an additional 1,280 MiB of VRAM (20.8% of capacity).

F. Portability

To demonstrate portability of WGLog, we run the same TypeScript and WGS� application used for the Windows Dawn/D3D12 experiments in Chrome on an M5 Pro MacBook with WebGPU’s Metal backend, without any code or configuration changes. On the 12-graph TC suite, WGLog completes in 4.13 s, compared with 50.72 s for DuckDB-WASM [11], 78.75 s for Ascent-WASM [35], and 463.82 s for sql.js [44], yielding speedups of 12.3 $\times$, 19.1 $\times$, and 112.3 $\times$.

VIII. RELATED WORK

Datalog systems. Several systems accelerate Datalog and Datalog-based neurosymbolic programs on GPUs [6], [15],

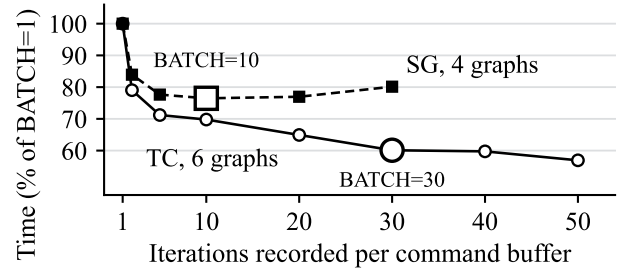


Fig. 9. Batch-size sweep for TC and SG. TC uses an adaptive policy capped at $K_{\max} = 30$, while SG uses a fixed batch size of $K = 10$.

[29], [41], [42], [46]. CPU, incremental, and distributed Datalog systems explore complementary execution models [16], [34], [40], [45]. WGLog differs by targeting portable, browser-native GPU execution through WebGPU.

GPU graph, database, and execution systems. Gunrock [50], GraphBLAST [51], and cuGraph [14] provide CUDA implementations of predefined graph algorithm families. HeavyDB [21] and Crystal [38] accelerate non-recursive analytical queries on GPUs. WGLog instead evaluates recursive relational rules whose intermediate and accumulated relations evolve across fixed-point iterations. CUDA streams, CUDA Graphs [9], persistent kernels [20], and Groute [5] reduce host overhead in iterative GPU workloads. These mechanisms depend on CUDA-specific runtime support and cannot be applied directly to WebGPU, where host readbacks are expensive and WGS� provides no cross-workgroup forward-progress guarantee. WGLog instead uses indirect dispatch and multi-iteration command batching (§V-B).

GPU primitives and browser analytics. WGLog implements merge-path partitioning [18] and decoupled-lookback scan [31] in WGS�. CUDA libraries such as Thrust [4], CUB [30], and ModernGPU [3] provide related primitives but cannot be called directly from WebGPU. WebGPU [48] provides a portable GPU interface across browser backends. To our knowledge, WGLog is the first WebGPU implementation of a recursive database query engine. Prior browser engines such as sql.js [44], DuckDB-WASM [11], and Ascent-WASM [35] execute on the CPU through WebAssembly. §VII-B quantifies the resulting performance gap.

IX. CONCLUSION

This paper presented WGLog, a browser-native GPU engine for recursive Datalog evaluation. Its sorted-array pipeline avoids hash indexes, while its batched indirect-dispatch loop keeps data-dependent execution on the GPU and amortizes WebGPU synchronization across fixed-point iterations. Across TC and SG, WGLog outperforms existing engines while executing unchanged across D3D12 and Metal, showing that high-performance recursive analytics can be delivered as a portable browser application. By meeting users in the browser, WGLog extends GPU-accelerated analytics beyond CUDA installations to the GPUs already present in commodity laptops, requiring no driver setup, toolchain, or specialized expertise, and allowing sensitive data to be processed entirely on-device. WGLog is open source at <https://github.com/harp-lab/webDLog>.

REFERENCES

- [1] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.
- [2] Apple, “Metal programming guide,” <https://developer.apple.com/metal/>.
- [3] S. Baxter, “ModernGPU: Patterns and techniques for GPU computing,” <https://moderngpu.github.io/>, 2013.
- [4] N. Bell and J. Hoberock, “Thrust: A productivity-oriented library for CUDA,” in *GPU Computing Gems Jade Edition*, W. mei W. Hwu, Ed. Morgan Kaufmann, 2012, pp. 359–371.
- [5] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, “Groute: An asynchronous multi-GPU programming model for irregular computations,” in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’17)*. ACM, 2017, pp. 235–248.
- [6] P. Biberstein, Z. Li, J. Devietti, and M. Naik, “Lobster: A gpu-accelerated framework for neurosymbolic programming,” in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2026.
- [7] M. W. Blasgen and K. P. Eswaran, “Storage and access in relational data bases,” *IBM Systems Journal*, vol. 16, no. 4, pp. 363–377, 1977.
- [8] M. Bravenboer and Y. 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 (OOPSLA ’09)*. ACM, 2009, pp. 243–262.
- [9] N. Corporation, “CUDA C++ programming guide,” <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>, 2026.
- [10] T. A. Davis and Y. Hu, “The University of Florida sparse matrix collection,” *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, 2011.
- [11] DuckDB Foundation, “DuckDB-Wasm: An in-process analytical database for the browser,” <https://github.com/duckdb/duckdb-wasm>.
- [12] ElectricSQL, “PGLite: Embeddable Postgres compiled to WebAssembly,” <https://github.com/electric-sql/pglite>, 2024.
- [13] W. Fan, C. Hu, and C. Tian, “Incremental graph computations: Doable and undoable,” *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–30, 2023.
- [14] A. Fender, B. Rees, and J. Eaton, “RAPIDS cuGraph,” in *Massive Graph Analytics*. Chapman and Hall/CRC, 2022, ch. 22.
- [15] T. Gilray and S. Kumar, “Distributed relational algebra at scale,” in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2019, pp. 12–22.
- [16] T. Gilray, A. Sahebhamri, Y. Sun, S. Kunapaneni, S. Kumar, and K. Micinski, “Datalog with first-class facts,” *Proceedings of the VLDB Endowment*, vol. 18, no. 3, pp. 651–665, 2024.
- [17] G. Graefe, “Query evaluation techniques for large databases,” *ACM Computing Surveys*, vol. 25, no. 2, pp. 73–169, 1993.
- [18] O. Green, R. McColl, and D. A. Bader, “GPU merge path: A GPU merging algorithm,” in *Proceedings of the 26th ACM International Conference on Supercomputing (ICS ’12)*. ACM, 2012, pp. 331–340.
- [19] T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou, “Datalog and recursive query processing,” *Foundations and Trends in Databases*, vol. 5, no. 2, pp. 105–195, 2013.
- [20] K. Gupta, J. A. Stuart, and J. D. Owens, “A study of persistent threads style GPU programming for GPGPU workloads,” in *Innovative Parallel Computing (InPar)*. IEEE, 2012.
- [21] HEAVY.AI, “HeavyDB: GPU-accelerated SQL database engine,” <https://github.com/heavyai/heavydb>, 2022, formerly MapD and OmniSciDB.
- [22] J. Heer and D. Moritz, “Mosaic: An architecture for scalable and interoperable data views,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 30, no. 1, pp. 436–446, 2024.
- [23] S. S. Huang, T. J. Green, and B. T. Loo, “Datalog and emerging applications: an interactive tutorial,” in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011.
- [24] Khronos Group, “Vulkan specification,” <https://www.khronos.org/vulkan/>, 2026.
- [25] D. Kirk *et al.*, “Nvidia cuda software and gpu parallel computing architecture,” in *ISMM*, vol. 7, 2007, pp. 103–104.
- [26] A. Kohn, D. Moritz, M. Raasveldt, H. Mühleisen, and T. Neumann, “DuckDB-Wasm: Fast analytical processing for the web,” *Proceedings of the VLDB Endowment*, vol. 15, no. 12, pp. 3574–3577, 2022.
- [27] S. Kumar and T. Gilray, “Distributed relational algebra at scale,” in *International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, vol. 1, 2019.
- [28] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection,” <http://snap.stanford.edu/data>, Jun. 2014.
- [29] Z. Li, J. Huang, J. Liu, and M. Naik, “Neurosymbolic programming in scallop: Principles and practice,” *Foundations and Trends in Programming Languages*, vol. 8, no. 2, pp. 118–249, 2024.
- [30] D. Merrill, “CUB: Cooperative primitives for CUDA C++,” <https://github.com/NVIDIA/cccl/tree/main/cub>, 2024, NVIDIA Corporation.
- [31] D. Merrill and M. Garland, “Single-pass parallel prefix scan with decoupled look-back,” in *GPU Technology Conference (GTC)*. NVIDIA.
- [32] D. Merrill and A. Grimshaw, “High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing,” *Parallel Processing Letters*, vol. 21, no. 2, pp. 245–272.
- [33] Microsoft, “Direct3D 12 programming guide,” <https://learn.microsoft.com/en-us/windows/win32/direct3d12/>, 2026.
- [34] L. Ryzhyk and M. Budiu, “Differential Datalog,” in *Datalog 2.0 2019*, 2019, pp. 56–67.
- [35] A. Sahebhamri, T. Gilray, and K. Micinski, “Seamless deductive inference via macros,” in *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC ’22)*. ACM, 2022, pp. 77–88.
- [36] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, “Access path selection in a relational database management system,” in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*. ACM, 1979, pp. 23–34.
- [37] J. Seo, S. Guo, and M. S. Lam, “Socialite: Datalog extensions for efficient social network analysis,” in *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE ’13)*. IEEE, 2013, pp. 278–289.
- [38] A. Shanbhag, S. Madden, and X. Yu, “A study of the fundamental performance characteristics of GPUs and CPUs for database analytics,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, 2020, pp. 1617–1632.
- [39] L. D. Shapiro, “Join processing in database systems with large main memories,” *ACM Transactions on Database Systems*, vol. 11, no. 3, pp. 239–264, 1986.
- [40] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo, “Big data analytics with Datalog queries on Spark,” in *Proceedings of the 2016 International Conference on Management of Data (SIGMOD ’16)*. ACM, 2016, pp. 1135–1149.
- [41] A. R. Shovon, L. R. Dyken, O. Green, T. Gilray, and S. Kumar, “Accelerating datalog applications with cudf,” in *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms*, 2022, pp. 41–45.
- [42] A. R. Shovon, T. Gilray, K. Micinski, and S. Kumar, “Towards iterative relational algebra on the GPU,” in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 1009–1016. [Online]. Available: <https://www.usenix.org/conference/atc23/presentation/shovon>
- [43] SQLite Consortium, “SQLite Wasm: The SQLite3 WebAssembly/JS subproject,” <https://sqlite.org/wasm>, 2024.
- [44] sql.js contributors, “sql.js: A Javascript SQLite library compiled to WebAssembly,” <https://github.com/sql-js/sql.js>, 2024.
- [45] P. Subotić, H. Jordan, L. Chang, and B. Scholz, “On fast large-scale program analysis in Datalog,” in *Proceedings of the 25th International Conference on Compiler Construction (CC ’16)*. ACM, pp. 196–206.
- [46] Y. Sun, A. R. Shovon, T. Gilray, S. Kumar, and K. K. Micinski, “Optimizing Datalog for the GPU,” in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS ’25)*. Rotterdam, Netherlands: ACM, Mar. 2025, pp. 762–776.
- [47] J. Urbani, C. J. H. Jacobs, and M. Krötzsch, “Column-oriented Datalog materialization for large knowledge graphs,” in *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI ’16)*. AAAI Press, 2016, pp. 258–264.
- [48] W3C, “WebGPU,” <https://www.w3.org/TR/webgpu/>, 2026, W3C Candidate Recommendation Draft.
- [49] W3C, “WebGPU Shading Language,” <https://www.w3.org/TR/WGSL/>, 2026, W3C Candidate Recommendation Draft.
- [50] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, “Gunrock: A high-performance graph processing library on the GPU,” in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’16)*. ACM, pp. 11:1–11:12.
- [51] C. Yang, A. Buluç, and J. D. Owens, “GraphBLAST: A high-performance linear algebra-based graph framework on the GPU,” *ACM Transactions on Mathematical Software*, vol. 48, no. 1, pp. 1:1–1:51.