Generating Fused Graph Kernels Using Relational Algebra

[Authors]
[Affiliations]

***Abstract*—Triangle counting is a fundamental graph kernel with applications from community detection to clustering coefficients. Two declarative frameworks dominate: sparse linear algebra (GraphBLAS) expresses it as masked sparse matrix–matrix multiplication (SpMM), while relational algebra expresses it as a three-way conjunctive join.**

We show that these frameworks differ in a fundamental way. The sparse linear algebra approach decomposes the computation into pairwise matrix operations (multiply then mask), exploring up to $\mathcal{O}(M^2)$ intermediate entries. The relational approach—via worst-case optimal join (WCOJ) algorithms—*fuses* all three relations into a single pass, achieving $\mathcal{O}(M^{3/2})$ time matching the AGM bound. When evaluated over CSR storage, the fused join generates code *identical* to hand-written CSR triangle counting—the same galloping-intersection inner loop that HPC programmers write by hand.

We demonstrate that SpMM is a special case of the WCOJ framework (the two-relation join), that triangle counting is where fusion strictly dominates, and that this advantage is not merely theoretical: on RMAT graphs with skewed degree distributions, the pairwise approach exhibits measurable blowup while the fused approach remains efficient. Drawing on the SPORES result of Wang et al., we show that this gap is structural: relational algebra rewrites are *complete* for optimizing linear algebra expressions, while the converse does not hold—and no amount of lazy evaluation within the pairwise framework can recover the fused kernel.

I. Introduction

Triangle counting—finding all triples of mutually adjacent vertices in a graph—is a fundamental kernel in graph analytics, underpinning clustering coefficients [1], community detection, and network characterization. It appears as a core benchmark in the GAP Benchmark Suite [2] and Graph500 [3], and is a standard test case for graph processing frameworks.

Two *declarative* frameworks compete for expressing this computation:

a) Sparse linear algebra.: GraphBLAS [4], [5] expresses graph algorithms as sparse matrix operations. Triangle counting becomes masked sparse matrix–matrix multiplication: $C\langle L\rangle = L \cdot L$, where L is the lower-triangular adjacency matrix and the mask retains only entries corresponding to actual edges [6], [7]. The programmer states *what* to compute; the runtime determines *how*. But the underlying execution decomposes into pairwise matrix operations: the SpMM enumerates all length-2 paths, then the mask filters to triangles. In the worst case, this explores $\mathcal{O}(M^2)$ intermediate entries (where M is the edge count).

b) Relational algebra.: The database community expresses the same computation as a conjunctive query: $R(x,y) \wedge R(y,z) \wedge R(x,z)$. This too is declarative, and *worst-case optimal join* (WCOJ) algorithms [8], [9] evaluate it by processing all three relations *simultaneously*—a fused, single-pass evaluation that avoids any intermediate materialization. For the triangle query, WCOJ achieves $\mathcal{O}(M^{3/2})$ time, matching the AGM bound [10].

c) The gap.: Meanwhile, the dominant HPC implementation is hand-written: store the graph in CSR, and for each edge (x,y) with $y > x$, intersect the sorted neighbor lists of x and y [11], [12]. This code is efficient but developed ad hoc, without connection to either declarative framework.

d) This paper.: We show that all three approaches are more tightly connected than previously recognized:

1) **Code equivalence via fusion.** When the WCOJ algorithm (Leapfrog Triejoin [9]) is evaluated over CSR arrays, the generated code is *identical* to hand-written CSR triangle counting—the same galloping-intersection inner loop. The relational framework *fuses* the three-relation join into a single pass; the result is the code that HPC programmers already write by hand.
2) **SpMM as a special case.** The two-relation join *is* SpMM. Triangle counting—the three-relation case—is where the fused approach strictly dominates: instead of a separate multiply and filter, it intersects all three constraints in a single nested loop.
3) **Measurable advantage on skewed graphs.** On graphs with high-degree hubs, the pairwise approach produces a large intermediate while the fused approach does not. We demonstrate this blowup experimentally.

These results connect to a deeper structural fact. Wang et al. [13] proved that relational algebra rewrite rules are *complete* for optimizing linear algebra expressions—any equivalent LA expression can be reached via RA rewrites—while the converse does not hold. The fundamental limitation is that linear algebra is restricted to pairwise (matrix) operations and cannot reason through higher-arity intermediates. Triangle counting is a concrete instance: the optimal fused kernel is a three-way relational join that pairwise matrix operations cannot express.

e) Paper organization.: Section II reviews CSR format, GraphBLAS triangle counting, the AGM bound, and how CSR supports variable-at-a-time evaluation. Section III shows that the two-relation join recovers SpMM. Section IV develops the three-way triangle join, contrasts pairwise vs. fused evaluation, and presents the intersection kernel. Section V presents the central code-equivalence result. Section VI describes a production realization. Section VII explains why lazy evaluation of the pairwise approach cannot close the gap, drawing on the SPORES completeness result. Section VIII gives experimental results. Section IX discusses related work.

II. Background and Notation

A. CSR Format

We store a graph on N vertices and M edges in Compressed Sparse Row (CSR) format: an array `rowptr`$[1{:}N{+}1]$ of row pointers and an array `colval`$[1{:}M]$ of sorted column indices. The neighbors of vertex i are `colval`$[$`rowptr`$[i]{:}$`rowptr`$[i{+}1]{-}1]$, stored in sorted order. For an undirected graph, we store both directions: if (u, v) is an edge, both v appears in row u and u appears in row v. We write $N(v)$ for the sorted neighbor list of vertex v.

B. Triangle Counting on CSR

The standard CSR triangle-counting algorithm iterates over each edge (x, y) with $y > x$ and intersects the sorted neighbor lists of x and y, counting common neighbors greater than y. This is well known in the HPC community [11], [1]; we formalize it as Algorithm 3 in Section V.

C. GraphBLAS Triangle Counting

GraphBLAS [4] expresses graph algorithms as sparse linear algebra operations. It is a *declarative* framework: the programmer specifies a computation in terms of matrix operations, and the runtime (e.g., SuiteSparse:GraphBLAS [5]) selects an execution strategy.

The standard triangle-counting formulation [6], [7] is:

$$C\langle L\rangle = L \cdot L, \qquad \text{triangles} = \tfrac{1}{2} \textstyle\sum_{ij} C_{ij},$$

where L is the lower-triangular adjacency matrix and $\langle L\rangle$ denotes masking: only entries (i, j) where $L_{ij} \neq 0$ are computed or stored. The multiplication $L \cdot L$ produces all length-2 paths; the mask retains only those that close into triangles.

a) The intermediate-size problem.: Even with masking, the multiply phase may *compute* (though not necessarily store) entries that the mask will discard. Consider a star graph: a hub vertex h with degree d. The SpMM generates $\Theta(d^2)$ length-2 paths through h, of which at most $\binom{d}{2}$ could close into triangles (and in a pure star, none do). For a graph with M edges and a hub of degree $\sqrt{M}$, this produces $\Theta(M)$ wasted intermediate entries. More generally, the worst-case cost of pairwise SpMM is $\mathcal{O}(M^2)$.

D. Conjunctive Queries and the AGM Bound

A *conjunctive query* (or natural join) is a conjunction of relational atoms sharing variables. Triangle counting is the query:

$$Q(x, y, z) \;=\; R(x, y) \;\wedge\; R(y, z) \;\wedge\; R(x, z).$$

The *AGM bound* [10] establishes the maximum possible output size: for three binary relations each of size M, the output has at most $\mathcal{O}(M^{3/2})$ tuples. This bound is tight—it is achieved by Turán-like graph constructions.

A *worst-case optimal join* (WCOJ) algorithm has running time that matches the AGM bound (up to logarithmic factors) [8]. The *Leapfrog Triejoin* [9] is a WCOJ algorithm that operates on sorted data, achieving $\mathcal{O}(M^{3/2} \log M)$ time for the triangle query on a general sorted relation.

E. Variable-at-a-Time Evaluation on CSR

WCOJ algorithms evaluate conjunctive queries *variable at a time*: for each variable in a chosen ordering, they iterate over candidate values, restricting to those consistent with all relations that constrain that variable. When multiple relations constrain a variable, the candidates are the *intersection* of the relevant sorted lists.

This evaluation strategy requires two primitives: (1) sorted iteration over a relation's values for a given key, and (2) *seek*: fast-forward to the first value $\geq v$ via galloping search. These are the operations that Veldhuizen [9] formalizes as the "trie interface."

CSR provides both natively. Row access is $\mathcal{O}(1)$ via `rowptr`—no search required—and within each row, the sorted `colval` slice supports iteration and galloping search directly. No auxiliary data structures, hash tables, or allocations are needed. On a general sorted relation (e.g., COO format), finding a key's entries requires $\mathcal{O}(\log M)$ binary search; CSR eliminates this cost because it *is* the precomputed index.

III. Two-Relation Join Recovers SpMM

Before tackling the triangle query, we demonstrate a remarkable correspondence: the two-relation join $R(x, y) \bowtie S(y, z)$ is precisely sparse matrix–matrix multiplication (SpMM), $C = R \times S$.

Evaluating the join variable at a time with ordering (x, y, z):

Level	Var.	Constraints	Candidates
1	x	$R(x, \cdot)$	rows of R: $\{1, \ldots, N\}$
2	y	$R(x, y)$, $S(y, \cdot)$	$N_R(x) \cap \text{rows}(S)$
3	z	$S(y, z)$	$N_S(y)$

At each level, the variable is constrained by the relations containing it. Crucially, no level has more than one relation contributing *values*—level 2's intersection

is trivial when S covers all row indices $1{:}N$ (since $N_R(x) \cap \{1, \ldots, N\} = N_R(x)$). The join reduces to:

for $x = 1$ **to** N**:**
  for $y \in N_R(x)$**:**
    for $z \in N_S(y)$**:**
      $C[x,z] \mathrel{+}= R[x,y] \cdot S[y,z]$

This is the textbook *row-by-row SpMM* algorithm. No intersections are needed—each level's variable is constrained by at most one relation, so the evaluation is pure nested iteration.

a) Connection to GraphBLAS.: The GraphBLAS `GrB_mxm` operation computes this two-relation join. GraphBLAS triangle counting [6], [7] extends it to three relations via a mask: $C\langle L \rangle = L \cdot L$, which first computes the two-relation SpMM and then filters by the third relation. This is exactly the pairwise strategy that Section IV analyzes.

The key observation: SpMM is the special case where each variable is constrained by at most one relation, requiring no intersection. Triangle counting—the three-relation case—is where multiple relations constrain the same variable, and intersection becomes essential. The fused approach intersects where the pairwise approach decomposes.

IV. Triangle Counting: Pairwise vs. Fused

We now apply variable-at-a-time evaluation to the triangle query—a three-way self-join. The contrast with pairwise evaluation (SpMM) motivates the fused approach and explains why it is strictly better. We first show how the evaluation plan is *derived* from the query structure, making the construction explicit.

A. Constructing the Evaluation Plan

The WCOJ evaluation plan for a conjunctive query is not designed by hand—it is *read off* from the query structure [9], [8]. The construction has three steps:

Step 1: Write the query as a conjunction of atoms. The triangle query is:

$$Q(x,y,z) \;=\; R(x,y) \;\wedge\; S(y,z) \;\wedge\; T(x,z).$$

Each atom is a binary relation; we treat the edge set of an undirected graph as a single relation $R = S = T$ (the self-join case, deferred to Section IV-E).

Step 2: Choose a variable ordering. Fix an ordering of the query variables—say (x,y,z). This determines the nesting of the evaluation loops: x is the outermost variable, z the innermost. (The choice of ordering affects performance but not correctness; the AGM bound and fractional hypertree width [14] guide the optimal choice.)

Step 3: At each depth, identify the participating atoms. Process the variables in order. At depth i, an atom *participates* if it contains variable x_i and all of its other variables appear at earlier depths (i.e., are already bound). The participating atoms each contribute a sorted iterator over the candidates for x_i; these iterators are intersected via the leapfrog join [9]. If only one atom participates, no intersection is needed—the evaluation simply iterates.

TABLE I
Variable-at-a-time evaluation of the triangle query, derived by the construction in Section IV-A. At depth 2, two atoms constrain z, requiring intersection. This is where the fused approach differs from pairwise SpMM.

Depth	Var.	Atoms	Candidates	Action
0	x	R, T	$\text{rows}(R) \cap \text{rows}(T)$	intersect
1	y	R, S	$N_R(x) \cap \text{rows}(S)$	intersect
2	z	S, T	$\mathbf{N_S(y) \cap N_T(x)}$	**intersect**

Applying this to the triangle query with ordering (x,y,z):

- Depth 0 (x): $R(x,y)$ contains x; its other variable y is not yet bound, but x is R's first column, so R can provide the set of x-values (its row keys). Similarly $T(x,z)$ provides x-values. $S(y,z)$ does not contain x.
  Participants: R, T. *Action:* intersect row keys of R and T.
- Depth 1 (y): x is bound (depth 0), so R provides the neighbors of x—the y-candidates. $S(y,z)$: y is S's first column, so S provides row keys. $T(x,z)$: does not contain y.
  Participants: R, S. *Action:* intersect $N_R(x)$ with rows of S.
- Depth 2 (z): y is bound (depth 1), so S provides neighbors of y—the z-candidates. $T(x,z)$: x is bound (depth 0), so T provides neighbors of x—also z-candidates. $R(x,y)$: does not contain z.
  Participants: S, T. *Action:* **intersect** $N_S(y) \cap N_T(x)$.

Table I summarizes the result.

a) Contrast with SpMM.: Apply the same construction to the two-relation join $R(x,y) \bowtie S(y,z)$ (Section III). At depth 2, only S contains z (with y already bound), so there is a single participant—no intersection, just iteration over $N_S(y)$. The difference between SpMM and the triangle query is entirely at depth 2: one participant (iterate) vs. two (intersect). This is the structural reason that the pairwise approach cannot fuse: it evaluates $R \bowtie S$ first, producing all (x,y,z) triples with $z \in N_S(y)$, and only then checks whether $z \in N_T(x)$. The fused plan checks both constraints simultaneously.

b) Generality.: This construction applies to any conjunctive query, not just triangles. Ngo et al. [8] proved that the resulting *Generic Join* algorithm is worst-case optimal (matching the AGM bound [10]); Veldhuizen's Leapfrog Triejoin [9] is its practical realization on sorted data. For aggregate queries such as triangle *counting* ($\sum_{x,y,z} R(x,y) \cdot S(y,z) \cdot T(x,z)$), the FAQ framework of

Algorithm 1 SORTEDINTERSECT: galloping merge of two sorted arrays (leapfrog join [9] for two iterators).

Require: Sorted arrays $A[a_{\text{lo}}{:}a_{\text{hi}}]$, $B[b_{\text{lo}}{:}b_{\text{hi}}]$; callback f

$i \leftarrow a_{\text{lo}}$; $j \leftarrow b_{\text{lo}}$
while $i \le a_{\text{hi}}$ **and** $j \le b_{\text{hi}}$ **do**
  $u \leftarrow A[i]$; $v \leftarrow B[j]$
  if $u = v$ **then**
    $f(u)$; $i \leftarrow i+1$; $j \leftarrow j+1$
  else if $u < v$ **then**
    $i \leftarrow \text{GALLOPGEQ}(A, v, i, a_{\text{hi}})$
  else
    $j \leftarrow \text{GALLOPGEQ}(B, u, j, b_{\text{hi}})$
  end if
end while

Abo Khamis et al. [15] generalizes the construction using variable elimination, pushing summation inside the nested loops.

B. Why Pairwise Evaluation Fails

A natural alternative decomposes the triangle query into two binary joins: first compute all length-2 paths $J(x,y,z) = R(x,y) \bowtie S(y,z)$ via SpMM, then filter J against $T(x,z)$.

The problem is the intermediate J. Consider a star graph: one hub vertex connected to $\sqrt{M}$ others. The SpMM produces $\sqrt{M} \times \sqrt{M} = M$ length-2 paths through the hub, most of which do not close into triangles. More generally, pairwise evaluation pays for *all* length-2 paths—cost $\mathcal{O}(M^2)$ in the worst case—before discovering that most lead nowhere.

The fused approach avoids this entirely. At depth 2, instead of freely iterating $N_S(y)$ and then filtering, it *intersects* $N_S(y)$ with $N_T(x)$. Only z-values that close the triangle survive. Dead-end paths are never explored.

This is analogous to *loop fusion* in numerical computing: the pairwise approach runs two separate passes (multiply, then filter), while the fused approach combines them into a single pass that never materializes the intermediate.

C. The Intersection Kernel

The intersection at depth 2 operates on two sorted neighbor lists. Two cursors alternate, each jumping past the other's current value (Algorithm 1). When the gap between matching values is large, galloping (exponential) search skips ahead in $\mathcal{O}(\log g)$ time rather than scanning linearly—adapting to the structure of the data. This is the *leapfrog join* of Veldhuizen [9], specialized to two iterators.

$\text{GALLOPGEQ}(A, v, \text{lo}, \text{hi})$ returns the position of the first entry $\ge v$ in $A[\text{lo:hi}]$ using exponential search followed by binary search, costing $\mathcal{O}(\log g)$ where g is the number of entries skipped [16], [17].

TABLE II
PAIRWISE (SPMM) VS. FUSED (WCOJ) TRIANGLE COUNTING.

	Pairwise (SpMM)	Fused (WCOJ)
Intermediate	$\mathcal{O}(M^2)$ worst case	None
Total cost	$\mathcal{O}(M^2)$	$\mathcal{O}(M^{3/2} \log M)$
Mechanism	Multiply then filter	Simultaneous intersection
Framework	Linear algebra	Relational algebra

HPC practitioners will recognize this as the standard galloping merge for sorted-set intersection. The WCOJ framework derives it mechanically from the query structure: the programmer specifies the triangle pattern; the construction of Section IV-A produces the intersection kernel.

D. Complexity

The AGM bound [10] establishes that for three binary relations of size M, the triangle query produces at most $\mathcal{O}(M^{3/2})$ output tuples—and this bound is tight. The bound arises from the fractional edge cover number of the query hypergraph, which for the triangle is $3/2$ [14], [10]. The Leapfrog Triejoin [9] matches this bound up to a logarithmic factor: $\mathcal{O}(M^{3/2} \log M)$ on a general sorted relation. On CSR, where row access is $\mathcal{O}(1)$ via `rowptr`, the only remaining log cost is within the intersection kernel itself.

E. Self-Join and the Standard Algorithm

When all three relations are the same graph R (stored symmetrically), the intersections at depths 0 and 1 become trivial: every vertex has a row, and every neighbor has its own adjacency list. Only the depth-2 intersection remains non-trivial:

for each vertex x**:**
  for each $y \in N(x)$**:**
    for each $z \in N(x) \cap N(y)$**:**
      count $+= 1$

Read aloud: *for each edge* (x, y)*, count the common neighbors of* x *and* y.

Without ordering, each triangle $\{a, b, c\}$ is counted six times. The constraint $x < y < z$ counts each exactly once: skip y-values $\le x$, and restrict the intersection to entries past y. This is the standard algorithm known to every HPC graph programmer—derived here mechanically from the three-way join specification via the construction of Section IV-A.

V. CODE EQUIVALENCE: FUSED JOIN ON CSR

We now arrive at the paper's central result. When the variable-at-a-time evaluation from Section IV is carried out over CSR arrays, every operation maps to a concrete array operation, and the generated triangle-counting code is *identical* to what an HPC programmer would write by hand.

Algorithm 2 INTERSECTNEIGHBORS: galloping merge of two sorted neighbor-list slices.

Require: Sorted arrays `rp` (row pointers), `cv` (column values); vertex a with start position a_{lo}; vertex b with start position b_{lo}; callback f

$a_{\text{hi}} \leftarrow \texttt{rp}[a+1]-1$; $b_{\text{hi}} \leftarrow \texttt{rp}[b+1]-1$
$i \leftarrow a_{\text{lo}}$; $j \leftarrow b_{\text{lo}}$
while $i \le a_{\text{hi}}$ **and** $j \le b_{\text{hi}}$ **do**
  $u \leftarrow \texttt{cv}[i]$; $v \leftarrow \texttt{cv}[j]$
  if $u = v$ **then**
    $f(u)$; $i \leftarrow i+1$; $j \leftarrow j+1$
  else if $u < v$ **then**
    $i \leftarrow \text{GALLOPGEQ}(\texttt{cv}, v, i, a_{\text{hi}})$
  else
    $j \leftarrow \text{GALLOPGEQ}(\texttt{cv}, u, j, b_{\text{hi}})$
  end if
end while

Algorithm 3 TRIANGLECOUNT: CSR triangle counting with $x < y < z$ ordering.

Require: CSR arrays `rp`[1:N+1], `cv`[1:nnz]

count $\leftarrow 0$
for $x = 1$ **to** N **do** ▷ ① iterate vertices
  for `yi` $=$ `rp`[x] **to** `rp`[x+1]-1 **do** ▷ ② iterate neighbors
    $y \leftarrow$ `cv`[`yi`]
    if $y \le x$ **then continue**
    end if ▷ filter $y > x$
    $a_{\text{lo}} \leftarrow$ `yi` $+ 1$ ▷ neighbors of x past y
    $b_{\text{lo}} \leftarrow \text{GALLOPGT}(\texttt{cv}, y, \texttt{rp}[y], \texttt{rp}[y+1]-1)$ ▷ neighbors of y past y
    INTERSECTNEIGHBORS(`rp`, `cv`, $x, a_{\text{lo}}, y, b_{\text{lo}}, z \mapsto$ count $+= 1$) ▷ ③④
  end for
end for
return count

A. Fused Join vs. CSR Code

Figure 1 places the fused-join pseudocode (left) next to the corresponding CSR implementation (right). Each numbered marker ①–④ connects a level of the evaluation to its CSR realization.

Two details deserve attention:

Why `xi = yi + 1`. Since `colval` is sorted within each row and `colval[yi]` $= y$, all entries after position `yi` are strictly greater than y—automatically satisfying $z > y$ on the x-side of the intersection.

Why `zi` needs `gallop_gt`. The neighbor list of y may include vertices $\le y$, so we must binary-search forward to the first entry $> y$. On the x-side this skip is free (by pointer arithmetic); on the y-side it requires a $\mathcal{O}(\log d_y)$ galloping search.

B. The Intersection Kernel

The innermost loop—marker ③ in Figure 1—is the computational hot path. We abstract it as a single function, INTERSECTNEIGHBORS, operating on two sorted slices of `colval` (Algorithm 2). The full triangle count is then Algorithm 3: two outer loops iterate over directed edges $(x \to y)$ with $y > x$, and the inner kernel intersects the neighbor lists of x and y restricted to entries past y.

$\text{GALLOPGEQ}(A, v, \text{lo}, \text{hi})$ returns the position of the first entry $\ge v$ in the sorted slice $A[\text{lo:hi}]$, using exponential search followed by binary search. Its cost is $\mathcal{O}(\log g)$ where g is the number of entries skipped—the "gap." GALLOPGT is the strict variant ($> v$). These are the standard adaptive-intersection primitives [16], [17].

C. The Equivalence

The correspondence between the fused-join derivation (Sections III–V) and the CSR code is exact. Every step is mechanically determined:

1) The query $R(x,y) \wedge R(y,z) \wedge R(x,z)$ with variable ordering (x,y,z) determines which relations constrain each variable (Table I).
2) The self-join $R = S = T$ collapses levels 1 and 2 to simple iteration, leaving level 3 as the only intersection.
3) CSR provides $\mathcal{O}(1)$ row access and sorted neighbor lists (Section II-E), converting each abstract operation to an array operation.
4) After inlining and constant folding, the result is Algorithm 3.

No design choices remain. The WCOJ framework, given the query and the storage format, *generates* the same tight inner loop that a sparse-matrix programmer would write by hand. The entire computation reduces to: **for each directed edge $(x \to y)$ with $y > x$, intersect the neighbor lists of x and y restricted to entries past y.**

VI. REALIZATION IN A PRODUCTION SYSTEM

To demonstrate that the fused-join/CSR equivalence is not merely a theoretical observation, we show how it is realized in a production relational database system that uses the Leapfrog Triejoin as its core join algorithm.

A. The TrieState Interface

The system wraps each relation in a *TrieState*—an object implementing the sorted-access primitives from Section II-E:

TrieState operation	Behavior
`iterate(Val(k))`	next key at level k
`seek_lub(v, Val(k))`	first key $\ge v$ at level k
`open(Val(k))`	descend to level k+1
`close(Val(k))`	restore state at level k

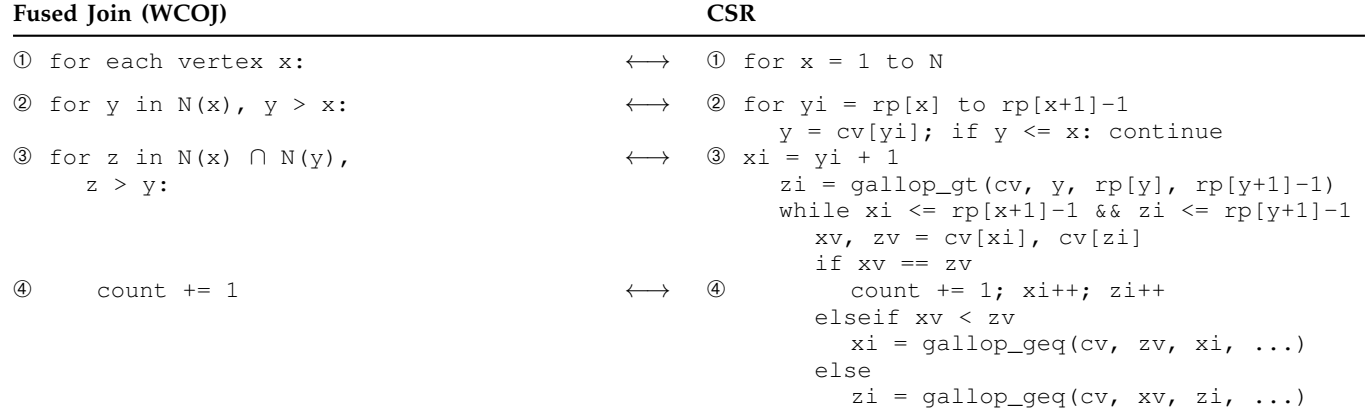

Fig. 1. Side-by-side: fused WCOJ triangle counting (left) and its CSR realization (right). Abbreviations: `rp` = `rowptr`, `cv` = `colval`. Arrows connect each level of the evaluation to its CSR implementation. The generated code is instruction-for-instruction identical to hand-written CSR triangle counting.

TABLE III
ABSTRACTION LAYERS FROM QUERY TO MACHINE CODE. EACH LAYER ADDS FUNCTIONALITY; THE RIGHTMOST COLUMN SHOWS WHAT OVERHEAD IT INTRODUCES IN THE INTERSECTION KERNEL.

Layer	Adds	Kernel overhead
Fused join (WCOJ)	Correctness, optimality	—
TrieState interface	Pluggable backends	Dispatch
B+ tree w/ spans	Page-level access	Page boundaries
CSR arrays	Flat, contiguous	**Zero**

For the triangle query, three TrieStates are created over the same edge relation—one for each atom $R(x,y)$, $R(y,z)$, $R(x,z)$—and handed to a *TrieStateConjunction*. The conjunction uses the variable-to-relation bindings (known at compile time) to generate, via metaprogramming, a specialized nested-loop program that calls only the abstract TrieState operations.

B. Storage Backends and Overhead

The TrieState interface decouples the join algorithm from storage. Different backends incur different overhead in the intersection inner loop:

a) B+ tree storage.: The default storage uses B+ trees accessed through a contiguous span iterator (CSI), which returns data in *spans*—contiguous arrays of keys from a single B+ tree page. Within a span, the generated `seek_lub` performs a galloping search on a contiguous array—the same GALLOPGEQ from Algorithm 2. The remaining overhead is page boundaries: when a neighbor list crosses a span boundary, the iterator must advance to the next B+ tree page.

b) CSR storage.: With CSR arrays, each row's neighbor list is a single contiguous slice—no page boundaries, no multi-span handling. The TrieState operations become trivial: `iterate(Val(1))` increments the row index; `open(Val(1))` reads `rowptr` to get the column range; `seek_lub(v, Val(2))` calls GALLOPGEQ on the `colval` slice.

After the metaprogramming system inlines these operations and the compiler performs constant folding, the generated code for the triangle query is instruction-for-instruction Algorithm 3—the same code that an HPC programmer writes by hand (Section V). The abstraction imposes zero overhead.

VII. WHY LAZY EVALUATION CANNOT CLOSE THE GAP

A natural objection to the preceding analysis is that the pairwise blowup is an implementation artifact: if the SpMM were evaluated *lazily*—fusing the multiply and mask phases without materializing the intermediate—the $\mathcal{O}(M^2)$ cost would disappear. This section explains why lazy evaluation within the linear algebra framework is fundamentally insufficient.

A. The Structural Limitation

The pairwise SpMM approach computes $C\langle L\rangle = L \cdot L$ in two conceptual phases: (1) for each pair (i,k), sum over shared indices j to form C_{ik}, and (2) discard entries where $L_{ik} = 0$. Lazy evaluation can fuse these phases, checking the mask before accumulating. But the fused pairwise computation still enumerates the *same* set of (i,j,k) triples—it merely avoids storing the non-triangles. The *work* is unchanged: every length-2 path $(i \to j \to k)$ is visited, regardless of whether the closing edge (i,k) exists.

The WCOJ approach operates differently. At the innermost level, it intersects $N(x) \cap N(y)$—only visiting z-values that satisfy *both* constraints simultaneously. Length-2 paths that do not close into triangles are never explored, because the intersection prunes them before they are enumerated.

Lazy evaluation of the pairwise plan eliminates the *materialization* cost but not the *enumeration* cost. The WCOJ plan eliminates both.

B. A Formal Basis: The SPORES Result

Wang et al. [13] formalized this limitation. Their SPORES system optimizes linear algebra (LA) expressions by lifting them into relational algebra (RA), applying RA equivalence rules, and lowering the result back to LA. The central theorem is:

> *RA rewrite rules are complete for LA optimization*: any equivalent LA expression can be reached via RA rewrites. The converse does not hold—LA rewrite rules alone cannot reach all equivalent LA expressions.

The reason is structural. LA expressions are restricted to operations on matrices (two-index objects). RA can introduce intermediate expressions with *three or more free variables*—higher-arity relations that have no matrix representation. These higher-arity intermediates serve as "stepping stones" to reach LA expressions that are unreachable by pairwise matrix rewrites alone.

For triangle counting, the fused three-way intersection is precisely such a higher-arity operation: it simultaneously binds x, y, and z across three relations. No sequence of pairwise matrix operations—however cleverly fused or lazily evaluated—can express this simultaneous three-way constraint. The limitation is not in the implementation but in the *algebra*: pairwise operations are simply less expressive than multiway operations for cyclic patterns.

C. Implications

This has a concrete consequence for the sparse linear algebra community. Efforts to improve GraphBLAS triangle counting through better SpMM implementations, smarter masking strategies, or lazy evaluation of expression trees are optimizing within an algebraic framework that is provably incomplete. For triangle counting (and more generally, for any cyclic join pattern), the optimal algorithm lives outside the space of pairwise matrix expressions.

The relational algebra framework reaches it naturally. And as Sections V–VI demonstrate, the resulting code is not some exotic database construction—it is the same CSR intersection loop that HPC programmers already write.

VIII. Experimental Evaluation

We evaluate two claims: (A) the fused WCOJ join over CSR matches hand-written CSR performance, and (B) the pairwise (SpMM) approach exhibits measurable blowup on skewed graphs.

A. Setup

a) Hardware.:

TABLE IV
Triangle counting time (seconds) on RMAT graphs.

Scale	Edges	CSR (C++)	CSR (Jl)	WCOJ/CSR	WCOJ/B+	EH
16						
18						
20						
22						

b) Graphs.: We use RMAT/Kronecker graphs at scales 16, 18, 20, and 22 (edge factor 16), following the Graph500 specification [3]. Edges are symmetrized and self-loops removed. For the blowup experiment (Thread B), we additionally construct graphs with controlled degree skew by varying the RMAT parameters (a, b, c, d)—increasing a concentrates edges on fewer hub vertices.

c) Implementations.: We compare triangle-counting implementations across frameworks:

1) **CSR (C++)**: hand-written CSR with galloping intersection, compiled with `gcc -O3`.
2) **CSR (Julia)**: equivalent Julia implementation.
3) **WCOJ/CSR**: the production system's TrieStateConjunction with CSR-backed TrieStates (Section VI).
4) **WCOJ/B+tree**: same conjunction with B+ tree storage and span-level intersection.
5) **EmptyHeaded**: the WCOJ-based relational engine of Aberger et al. [18], using its native triangle counting query.
6) **GraphBLAS**: SuiteSparse:GraphBLAS [5], $C\langle L\rangle = L \cdot L$ formulation.

All experiments are single-threaded to isolate algorithmic differences from parallelization effects.

B. Thread A: Code Equivalence

We expect the WCOJ/CSR column to match hand-written CSR within measurement noise, confirming that the abstraction imposes zero overhead. The WCOJ/B+tree column should show measurable overhead from page-boundary handling in the intersection kernel.

C. Thread B: Pairwise Blowup

The pairwise approach (GraphBLAS $C\langle L\rangle = L \cdot L$) computes all length-2 paths before filtering. On graphs with high-degree hubs, the number of length-2 paths grows quadratically in the hub degree, while the fused WCOJ approach intersects eagerly and avoids dead-end paths.

We vary the RMAT skew parameter to control the maximum hub degree and measure the ratio of GraphBLAS time to WCOJ/CSR time. As the skew increases (more power-law-like), we expect this ratio to grow, confirming that the $\mathcal{O}(M^2)$ vs. $\mathcal{O}(M^{3/2})$ gap from Table II is not merely theoretical.

a) Caveat.: SuiteSparse:GraphBLAS is a highly optimized, mature implementation; our WCOJ/CSR is a single-threaded research prototype. Absolute times are not directly comparable—the meaningful quantity is how each approach *scales* with degree skew.

D. Analysis

IX. Related Work

a) GraphBLAS and sparse linear algebra.: GraphBLAS [4], [19] provides a standard API for expressing graph algorithms as sparse matrix operations. SuiteSparse:GraphBLAS [5] is the most widely used implementation. Triangle counting via masked SpMM [6], [7] is the standard GraphBLAS formulation. Our work shows that this corresponds to the pairwise join strategy, and that the WCOJ framework subsumes it: the two-relation join *is* SpMM, while the three-relation case fuses the computation that GraphBLAS decomposes into multiply and mask.

b) Worst-case optimal joins.: The AGM bound [10] on conjunctive query output size motivated WCOJ algorithms matching this bound [8]. Veldhuizen's Leapfrog Triejoin [9] operates on sorted data and is the basis of our analysis. The survey by Ngo et al. [20] provides an accessible introduction.

c) WCOJ for graph processing.: EmptyHeaded [18] is a graph-processing engine built on WCOJ, demonstrating competitive performance with hand-tuned graph systems. Mhedhbi and Salihoglu [21] combine binary and worst-case optimal joins for subgraph queries. Our contribution is complementary: rather than building a new system, we show that WCOJ on CSR generates *exactly* the code that existing HPC implementations use.

d) Relational vs. linear algebra optimization.: Wang et al. [13] introduced SPORES, which optimizes linear algebra expressions by lifting them into relational algebra, applying RA rewrite rules, and lowering the result back to LA. Their key result is that this process is *complete*: any equivalent LA expression can be reached via RA rewrites. The converse does not hold—LA rewrite rules alone cannot reach all equivalent LA expressions, because they cannot reason through intermediate forms with more than two free variables. Our work provides a concrete instance of this gap: the fused triangle-counting kernel is a three-way relational join that has no natural decomposition into pairwise matrix operations. We develop this point further in Section VII.

e) Set intersection.: The galloping-merge intersection kernel at the core of both the fused join and hand-written CSR triangle counting has been studied extensively [16], [17]. Our work does not contribute new intersection algorithms; rather, we show that the WCOJ framework derives these algorithms mechanically from a declarative query specification.

f) Triangle counting in HPC.: The HPC community has developed highly optimized triangle-counting implementations using CSR [11], degree ordering [1], and GPU acceleration [12]. These are orthogonal to WCOJ: degree ordering reduces the work per edge and can be applied within the fused-join framework; GPU parallelism applies to the intersection kernel regardless of how it was derived.

X. Conclusion

We have shown that two declarative frameworks—sparse linear algebra and relational algebra—target the same CSR storage and the same sorted-intersection primitives, but differ fundamentally in what they can express. Sparse linear algebra decomposes triangle counting into pairwise matrix operations (SpMM + mask); relational algebra fuses the three-relation join into a single pass. The fused kernel, derived mechanically from the Leapfrog Triejoin, is instruction-for-instruction identical to the hand-written CSR triangle counting code that HPC programmers have long used.

This equivalence has two implications. First, the hand-written CSR algorithm—previously an ad hoc construction—is in fact a worst-case optimal join, inheriting the $\mathcal{O}(M^{3/2})$ AGM bound guarantee. Second, the pairwise SpMM approach is provably suboptimal for triangle counting: it cannot avoid the $\mathcal{O}(M^2)$ intermediate that the fused approach eliminates. This limitation is structural, not an implementation artifact—Wang et al. [13] proved that relational algebra rewrites are strictly more powerful than linear algebra rewrites, and no amount of lazy evaluation within the pairwise linear algebra framework can recover the fused kernel (Section VII).

These results suggest a broader program: worst-case optimal joins provide a principled path from declarative graph-pattern queries to provably optimal code, with zero abstraction overhead when the underlying storage is CSR. Extensions to k-clique counting, subgraph enumeration, and more complex motif queries are natural—the WCOJ framework handles arbitrary cyclic join patterns, a capability that pairwise matrix operations fundamentally lack.

References

[1] M. Latapy, "Main-memory triangle computations for very large (sparse (power-law)) graphs," *Theoretical Computer Science*, vol. 407, no. 1–3, pp. 458–473, 2008.

[2] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," in *Proc. IEEE International Symposium on Workload Characterization (IISWC)*, 2015.

[3] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," *Cray User's Group (CUG)*, 2010.

[4] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, J. Moreira, J. Owens, C. Yang, M. Zalewski, and T. Mattson, "Mathematical foundations of the GraphBLAS," in *Proc. IEEE High Performance Extreme Computing Conference (HPEC)*, 2016, pp. 1–9.

[5] T. A. Davis, "Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra," *ACM Transactions on Mathematical Software*, vol. 45, no. 4, pp. 44:1–44:25, 2019.

[6] M. M. Aznaveh, J. Chen, T. A. Davis, B. Hegyi, S. P. Kolodziej, T. G. Mattson, and G. Parimalarangan, "Parallel triangle counting and enumeration using matrix algebra," in *Proc. IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 1–10.

[7] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, "Fast linear algebra-based triangle counting with KokkosKernels," in *Proc. IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–7.

[8] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case optimal join algorithms," *Journal of the ACM*, vol. 65, no. 3, pp. 16:1–16:40, 2018.

[9] T. L. Veldhuizen, "Leapfrog triejoin: A simple, worst-case optimal join algorithm," in *Proc. International Conference on Database Theory (ICDT)*, 2014, pp. 96–106.

[10] A. Atserias, M. Grohe, and D. Marx, "Size bounds and query plans for relational joins," *SIAM Journal on Computing*, vol. 42, no. 4, pp. 1737–1767, 2013.

[11] J. Shun and K. Tangwongsan, "Multicore triangle computations without tuning," in *Proc. IEEE International Conference on Data Engineering (ICDE)*, 2015, pp. 149–160.

[12] O. Green and D. A. Bader, "Faster clustering coefficient using vertex covers," in *Proc. IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2014, pp. 1–8.

[13] Y. R. Wang, S. Hutchison, J. Leang, B. Howe, and D. Suciu, "SPORES: Sum-product optimization via relational equality saturation for large scale linear algebra," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 1919–1932, 2020.

[14] M. Grohe and D. Marx, "Constraint solving via fractional edge covers," *ACM Transactions on Algorithms*, vol. 11, no. 1, pp. 4:1–4:20, 2014.

[15] M. Abo Khamis, H. Q. Ngo, and A. Rudra, "FAQ: Questions asked frequently," in *Proc. ACM Symposium on Principles of Database Systems (PODS)*, 2016, pp. 13–28.

[16] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Adaptive set intersections, unions, and differences," in *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2000, pp. 743–752.

[17] R. Baeza-Yates and A. Salinger, "Fast intersection algorithms for sorted sequences," in *Algorithms and Applications*, 2004, pp. 45–61.

[18] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré, "EmptyHeaded: A relational engine for graph processing," in *Proc. ACM SIGMOD International Conference on Management of Data*, 2017, pp. 431–446.

[19] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[20] H. Q. Ngo, C. Ré, and A. Rudra, "Skew strikes back: New developments in the theory of join algorithms," *SIGMOD Record*, vol. 42, no. 4, pp. 5–16, 2013.

[21] A. Mhedhbi and S. Salihoglu, "Optimizing subgraph queries by combining binary and worst-case optimal joins," *Proc. VLDB Endowment*, vol. 12, no. 11, pp. 1692–1704, 2019.