Generating Fused Graph Kernels Using Relational Algebra

Authors
Affiliations

February 26, 2026

Abstract

Graph algorithms are typically implemented by hand, using operatiions over vertices and edges. Expressing them declaratively at the level of graphs enables systematic optimization of entire classes of graph computations. One successful approach exploits the correspondence between graphs and sparse matrices: GraphBLAS couches graph operations as sparse linear algebra over user-defined semirings. This paper explores an alternative: using the correspondence between graphs and *relations* to express graph algorithms, and leveraging optimizations from relational algebra—in particular worst-case optimal join (WCOJ) algorithms—to generate fused kernels equivalent to hand-written code. We show that relational algebra rewrites are strictly more powerful than linear algebra rewrites for graph computations with cyclic structure: sparse linear algebra decomposes computation into pairwise matrix products, producing intermediates up to $\mathcal{O}(M^2)$ in size for patterns such as triangles, 4-cycles, and cliques, while WCOJ fuses all 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. 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. Beyond triangles, for 4-cycles, diamonds, bow-ties, and k-cycles, no masked SpGEMM formulation exists and the pairwise gap is unavoidable for any cyclic graph pattern. Performance results confirm that the pairwise approach exhibits measurable blowup on high-degree vertices while the fused approach remains efficient.

1 Introduction

A graph is a set of vertices connected by edges. Each edge is a pair of vertices—an element of $V \times V$—and the edge set E is a binary relation over a single domain. Graphs are ubiquitous: social networks, web link structures, molecular bonds, communication topologies. Graph *algorithms* extract structure from these relations: shortest paths, connected components, communities, and—the focus of this paper—triangles.

Conceptually, a graph's edge set is a special case of a *relation* in the database sense: a set of tuples over typed attributes. A relation can connect entities of different types (e.g., authors to papers, customers to products), can have arity greater than two (ternary, quaternary, ...), and its attributes carry schema information that graphs lack. An edge set $E \subseteq V \times V$ is the degenerate case: a single binary relation where both columns range over the same untyped domain.

A *hypergraph* generalizes in a different direction: each hyperedge connects an arbitrary subset of vertices, rather than exactly two. Hypergraphs are more general than ordinary graphs but less general than relations: a hyperedge is an unordered set of vertices from a single domain, while a relation is an ordered tuple over potentially distinct, typed domains. Crucially, the relational framework admits *joins*—the ability to combine tuples from multiple relations by matching on shared attributes—and it is this join machinery, not the data model per se, that gives relational algebra its optimization power.

There is a third, equally productive analogy. A graph on N vertices can be represented as an $N \times N$ adjacency matrix A, where $A[i, j] = 1$ if edge (i, j) exists. For most real-world graphs this matrix is sparse: it has M nonzero entries out of N^2 possible, with $M \ll N^2$. The GraphBLAS project [11, 7] builds on this observation, providing a standard API that expresses graph algorithms as sparse linear algebra operations over user-defined semirings. Breadth-first search becomes sparse matrix–vector multiplication (SpMV) over the Boolean semiring; single-source shortest paths becomes SpMV over the $(\min, +)$ semiring; connected components, PageRank, and many other algorithms have natural semiring formulations [11]. This framework is powerful and productive: the programmer states *what* to compute using matrix and vector operations, and the runtime chooses *how*—which sparse data structures, which parallelization strategy, which scheduling. GraphBLAS is, in short, a declarative language for graph computation grounded in the algebra of sparse matrices.

The connection between graph algorithms and relational algebra is not new: systems such as EmptyHeaded [1] have demonstrated that worst-case optimal joins can serve as the execution engine for graph pattern queries, achieving substantial speedups over conventional graph-processing frameworks. In this paper, we make the relationship precise and show that it is strictly more powerful than the linear algebra approach for graph computations that exhibit *cyclic* structure. The distinction is fundamental: sparse linear algebra decomposes computation into pairwise matrix operations (each combining two rank-2 objects), while relational algebra admits *multiway* operations that process three or more relations simultaneously. For acyclic patterns—paths, trees, star queries—the two frameworks are equivalent. For cyclic patterns—triangles, 4-cycles, cliques—the relational framework produces fused kernels that no sequence of pairwise matrix operations can express.

Consider triangle counting, the simplest cyclic graph query and the focus of this paper. In GraphBLAS, it is expressed as 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 [4, 20]. Other cyclic queries follow the same pattern: 4-cycle counting requires computing A^2 (all length-2 paths), and k-clique counting decomposes into chains of matrix multiplications. In each case, the computation reduces to a sequence of pairwise matrix products.

The problem is intermediates. Each pairwise product can produce a result far larger than the input or the output. A star graph—a single hub connected to $\sqrt{M}$ leaves—has M edges and zero triangles, yet the SpMM $L \cdot L$ enumerates all $\binom{\sqrt{M}}{2} = \mathcal{O}(M)$ length-2 paths through the hub, every one ultimately discarded by the mask. On graphs with many high-degree vertices, the intermediate can reach $\mathcal{O}(M^2)$ entries, even though the triangle count is at most $\mathcal{O}(M^{3/2})$ by the AGM bound [3]. For 4-cycles and other cyclic patterns the situation is worse: no mask exists, and the full intermediate must be materialized (Section 8).

For the special case of triangle counting, the GraphBLAS community has developed a workaround: SuiteSparse implements a *masked dot-product SpGEMM* that, for each edge (i, j) in the mask L, computes the inner product of row i and column j directly—effectively intersecting $N(i) \cap N(j)$, the same work that the hand-written algorithm performs [7]. This avoids materializing the large intermediate and, for triangles, recovers optimal performance. But the technique is specific to triangles: the mask is the same edge set being multiplied, the output is indexed by edges (sparse), and a single inner product per edge suffices to fuse the three-way check. For other cyclic patterns—4-cycles, diamonds, bow-ties, k-cycles—at least one of these properties fails, no analogous mask exists, and the pairwise intermediate cannot be avoided (Section 8).

The relational algebra framework, by contrast, eliminates these intermediates in full generality. The triangle query, expressed as the conjunctive join $R(x,y) \wedge R(y,z) \wedge R(x,z)$, can be evaluated by *worst-case optimal join* (WCOJ) algorithms [15, 18] that process all three relations simultaneously—binding variables one at a time and intersecting candidate sets at each level, with no intermediate materialized. For the star graph, the WCOJ algorithm examines each edge, finds the neighbor-list intersection empty, and moves on—$\mathcal{O}(M)$ total work for zero triangles. In general, WCOJ achieves $\mathcal{O}(M^{3/2})$ time, matching the AGM bound and equaling the complexity of hand-written CSR triangle counting [17, 9].

Crucially, this improvement is not a matter of lazy evaluation. Deferring the SpMM—computing entries of $L \cdot L$ on demand—does not change the structure of the computation: it still decomposes a three-way query into a pairwise product followed by a filter, and every surviving entry must still be computed individually. The complexity difference comes from *eliminating the intermediate*: rather than forming all length-2 paths and then checking which ones close a triangle, the WCOJ approach checks the closing-edge constraint *during* path enumeration, pruning immediately when the intersection is empty. This is a structural difference in the evaluation plan—a different algorithm, not a different implementation of the same algorithm (Section 7).

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 [17, 9]. This code is efficient but developed ad hoc, without connection to either declarative framework.

This paper. We establish the following correspondences between sparse linear algebra, relational algebra, and hand-written CSR code:

1. **Code equivalence via fusion.** When the WCOJ algorithm (Leapfrog Triejoin [18]) 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. **Structural incompleteness of linear algebra.** Drawing on the SPORES result of Wang et al. [19], we show that this gap is not an implementation artifact but an algebraic one: relational algebra rewrite rules are *complete* for optimizing linear algebra expressions, while the converse does not hold. No amount of lazy evaluation, expression rewriting, or masking strategy within the pairwise matrix framework can recover the fused kernel.

4. **Beyond triangles.** The masked SpGEMM technique that partially closes the gap for triangles does not generalize. For 4-cycles, diamonds, bow-ties, and k-cycles, no natural mask exists, and the pairwise approach is stuck at $\mathcal{O}(M^2)$ while WCOJ achieves $\mathcal{O}(M^{3/2})$ or better. The dividing line is the *cyclicity* of the query hypergraph: any graph pattern containing a cycle requires multiway intersection for optimal evaluation.

5. **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.

Taken together, these results show that relational algebra is a more natural and more powerful mechanism for expressing graph algorithms than linear algebra. Graph edges *are* relations, and the relational framework—with its multiway join machinery—subsumes the pairwise matrix operations of sparse linear algebra while also generating kernels that linear algebra cannot express. The hand-written CSR code that HPC programmers have long used is, in fact, a worst-case optimal relational join.

Paper organization. Section 2 reviews CSR format, GraphBLAS triangle counting, the AGM bound, and how CSR supports variable-at-a-time evaluation. Section 3 shows that the two-relation join recovers SpMM. Section 4 develops the three-way triangle join, derives the WCOJ evaluation plan, and presents the intersection kernel. Section 5 presents the central code-equivalence result. Section 6 describes a production realization. Section 7 explains why lazy evaluation of the pairwise approach cannot close the gap, drawing on the SPORES completeness result. Section 8 extends the analysis to 4-cycles, diamonds, and other cyclic patterns. Section 9 gives experimental results. Sections 10–11 discuss related work and conclude.

2 Background and Notation

2.1 CSR Format

We store a graph on N vertices and M edges in Compressed Sparse Row (CSR) format: an array $\texttt{rowptr}[1{:}N{+}1]$ of row pointers and an array $\texttt{colval}[1{:}M]$ of sorted column indices. The neighbors of vertex i are $\texttt{colval}[\texttt{rowptr}[i]{:}\texttt{rowptr}[i{+}1]{-}1]$, 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.

2.2 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 [17, 12]; we formalize it as Algorithm 3 in Section 5.

2.3 GraphBLAS Triangle Counting

GraphBLAS [11] 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 [7]) selects an execution strategy.

The standard triangle-counting formulation [4, 20] 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.

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 $\mathcal{O}(M^2)$.

2.4 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* [3] 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) [15]. The *Leapfrog Triejoin* [18] 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.

2.5 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 [18] 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.

3 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.

Connection to GraphBLAS. The GraphBLAS `GrB_mxm` operation computes this two-relation join. GraphBLAS triangle counting [4, 20] 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 4 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.

4 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.

4.1 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 [18, 15]. 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 4.5).

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 [10] 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 [18]. If only one atom participates, no intersection is needed—the evaluation simply iterates.

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

Table 1: Variable-at-a-time evaluation of the triangle query, derived by the construction in Section 4.1. 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	$\mathrm{rows}(R) \cap \mathrm{rows}(T)$	intersect
1	y	R, S	$N_R(x) \cap \mathrm{rows}(S)$	intersect
2	z	S, T	$\mathbf{N_S(y) \cap N_T(x)}$	**intersect**

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). $R(x, y)$: 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). $S(y, z)$: 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 1 summarizes the result.

Contrast with SpMM. Apply the same construction to the two-relation join $R(x, y) \bowtie S(y, z)$ (Section 3). 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.

Generality. This construction applies to any conjunctive query, not just triangles. Ngo et al. [15] proved that the resulting *Generic Join* algorithm is worst-case optimal (matching the AGM bound [3]); Veldhuizen's Leapfrog Triejoin [18] 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 Abo Khamis et al. [2] generalizes the construction using variable elimination, pushing summation inside the nested loops.

4.2 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.

Algorithm 1 SORTEDINTERSECT: galloping merge of two sorted arrays (leapfrog join [18] for two iterators).

```
Require: Sorted arrays $A[a_{lo}:a_{hi}]$, $B[b_{lo}:b_{hi}]$; callback $f$
1: $i \leftarrow a_{lo}$;  $j \leftarrow b_{lo}$
2: while $i \leq a_{hi}$ and $j \leq b_{hi}$ do
3:     $u \leftarrow A[i]$;  $v \leftarrow B[j]$
4:     if $u = v$ then
5:         $f(u)$;  $i \leftarrow i + 1$;  $j \leftarrow j + 1$
6:     else if $u < v$ then
7:         $i \leftarrow$ GALLOPGEQ$(A, v, i, a_{hi})$
8:     else
9:         $j \leftarrow$ GALLOPGEQ$(B, u, j, b_{hi})$
10:    end if
11: end while
```

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.

4.3 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 [18], specialized to two iterators.
GALLOPGEQ$(A, v, \text{lo}, \text{hi})$ returns the position of the first entry $\geq v$ 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 [8, 5].

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 4.1 produces the intersection kernel.

4.4 Complexity

The AGM bound [3] 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 $3/2$ [10, 3]. The Leapfrog Triejoin [18] 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.

4.5 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

Table 2: 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

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 $\leq x$, 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 4.1.

5 Code Equivalence: Fused Join on CSR

We now arrive at the paper's central result. When the variable-at-a-time evaluation from Section 4 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.

5.1 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 $\leq y$, 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.

5.2 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.

GALLOPGEQ$(A, v, \mathrm{lo}, \mathrm{hi})$ returns the position of the first entry $\geq v$ in the sorted slice $A[\mathrm{lo{:}hi}]$, using exponential search followed by binary search. Its cost is $\mathcal{O}(\log g)$ where g is the number

```
Fused Join (WCOJ)                               CSR
-----------------------------------------------------------------------------------
① for each vertex x:                   ⟷   ① for x = 1 to N

② for y in N(x), y > x:                ⟷   ② for yi = rp[x] to rp[x+1]-1
                                                  y = cv[yi]; if y <= x:  continue
③ for z in N(x) ∩ N(y),                ⟷   ③ xi = yi + 1
      z > y:                                      zi = gallop_gt(cv, y, rp[y],
                                            rp[y+1]-1)
                                                  while xi <= rp[x+1]-1 && zi <=
                                            rp[y+1]-1
                                                     xv, zv = cv[xi], cv[zi]
                                                     if xv == zv
④     count += 1                       ⟷   ④           count += 1; xi++; zi++
                                                     elseif xv < zv
                                                        xi = gallop_geq(cv, zv, xi,
                                            ...)
                                                     else
                                                        zi = gallop_geq(cv, xv, zi,
                                            ...)
```

Figure 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.

of entries skipped—the "gap." GALLOPGT is the strict variant ($> v$). These are the standard adaptive-intersection primitives [8, 5].

5.3 The Equivalence

The correspondence between the fused-join derivation (Sections 3–5) 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 1).

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 2.5), 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.**

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 $f$
 1: $a_{hi} \leftarrow \texttt{rp}[a+1]-1$;  $b_{hi} \leftarrow \texttt{rp}[b+1]-1$
 2: $i \leftarrow a_{lo}$;  $j \leftarrow b_{lo}$
 3: while $i \le a_{hi}$ and $j \le b_{hi}$ do
 4:     $u \leftarrow \texttt{cv}[i]$;  $v \leftarrow \texttt{cv}[j]$
 5:     if $u = v$ then
 6:         $f(u)$;  $i \leftarrow i+1$;  $j \leftarrow j+1$
 7:     else if $u < v$ then
 8:         $i \leftarrow$ GallopGeq$(\texttt{cv}, v, i, a_{hi})$
 9:     else
10:         $j \leftarrow$ GallopGeq$(\texttt{cv}, u, j, b_{hi})$
11:     end if
12: end while
```

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

```
Require: CSR arrays rp[1:$N$+1], cv[1:nnz]
 1: count $\leftarrow 0$
 2: for $x = 1$ to $N$ do                                   ▷ ① iterate vertices
 3:     for yi $=$ rp[$x$] to rp[$x$+1]$-1$ do               ▷ ② iterate neighbors
 4:         $y \leftarrow$ cv[yi]
 5:         if $y \le x$ then continue
 6:         end if                                          ▷ filter $y > x$
 7:         $a_{lo} \leftarrow$ yi $+ 1$                        ▷ neighbors of $x$ past $y$
 8:         $b_{lo} \leftarrow$ GallopGt(cv, $y$, rp[$y$], rp[$y$+1]$-1$)   ▷ neighbors of $y$ past $y$
 9:         IntersectNeighbors(rp, cv, $x$, $a_{lo}$, $y$, $b_{lo}$, $z \mapsto$ count += 1)   ▷ ③④
10:     end for
11: end for
12: return count
```

6 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.

6.1 The TrieState Interface

The system wraps each relation in a *TrieState*—an object implementing the sorted-access primitives from Section 2.5:

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

Table 3: 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.

6.2 Storage Backends and Overhead

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

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.

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 5). The abstraction imposes zero overhead.

7 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.

7.1 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.

7.2 A Formal Basis: The SPORES Result

Wang et al. [19] 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.

7.3 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 5–6 demonstrate, the resulting code is not some exotic database construction—it is the same CSR intersection loop that HPC programmers already write.

8 Beyond Triangles: Cyclic Patterns in General

Triangle counting is the simplest cyclic graph query, but the gap between pairwise and fused evaluation widens for larger patterns. This section applies the WCOJ construction of Section 4.1 to additional cyclic queries, showing that the masked SpGEMM technique—which partially closes the gap for triangles—does not generalize.

8.1 Why Masked SpGEMM Is Triangle-Specific

Recall the GraphBLAS triangle-counting formulation: $C\langle L\rangle = L \cdot L$. SuiteSparse implements this via *dot-product masked SpGEMM*: for each edge (i, j) in the mask L, compute the inner product of row i and column j of L. This effectively intersects $N(i) \cap N(j)$ for each edge—the same work that the WCOJ fused kernel performs.

This implementation succeeds because of three properties specific to triangles:

1. The mask is the *same* edge set L being multiplied—known in advance, with M entries.
2. The output is indexed by edges (sparse), not by arbitrary vertex pairs (potentially dense).
3. A single masked inner product per edge suffices to fuse the three-way check.

For other cyclic patterns, at least one of these properties fails.

8.2 4-Cycle (Square) Counting

The 4-cycle query asks for all tuples (a, b, c, d) such that edges (a, b), (b, c), (c, d), and (a, d) all exist:

$$Q_4(a, b, c, d) \;=\; R(a, b) \;\wedge\; R(b, c) \;\wedge\; R(c, d) \;\wedge\; R(a, d).$$

The linear algebra approach. The standard formulation computes the 2-path matrix $P = A^2$, where $P[a, c]$ counts the number of vertices b such that (a, b) and (b, c) are edges. Each pair of distinct 2-paths between the same endpoints forms a 4-cycle, so the count is $\sum_{a<c} \binom{P[a,c]}{2}$.

The problem: **there is no natural mask.** The computation requires $P[a, c]$ for every pair (a, c) with two or more common neighbors. Which pairs are those? Unknown until P is computed. And P can be *dense*: a star graph with hub degree d produces $\binom{d}{2}$ nonzero entries in P, even though A has only d edges. The "mask" would be $\{(a, c) : P[a, c] \geq 2\}$—potentially $\mathcal{O}(n^2)$ entries, exactly the blowup that masking was supposed to prevent.

The WCOJ approach. Applying the construction of Section 4.1 with variable ordering (a, b, d, c):

Depth	Var.	Atoms	Candidates	Action
0	a	$R(a, b)$, $R(a, d)$	vertices	iterate
1	b	$R(a, b)$	$N(a)$	iterate
2	d	$R(a, d)$	$N(a)$, $d \neq b$	iterate
3	c	$R(b, c)$, $R(c, d)$	$\mathbf{N(b) \cap N(d)}$	**intersect**

At depth 3, WCOJ intersects $N(b) \cap N(d)$—only c-values that close the 4-cycle survive. Triples (a, b, d) where b and d share no common neighbor are pruned immediately. The pairwise approach computes all of A^2 first, then sifts through it.

Complexity. The AGM bound for the 4-cycle with M-edge relations is $\mathcal{O}(M^{3/2})$; the best known WCOJ-style algorithm (PANDA [2]) achieves $\mathcal{O}(M^{3/2})$. Any pairwise decomposition requires $\mathcal{O}(M^2)$ in the worst case, because computing the full 2-path matrix is unavoidable.

8.3 Other Cyclic Patterns

The same analysis applies to every cyclic subgraph pattern:

Diamond (K_4 minus one edge). Four vertices, five edges. The query has two cycles. Pairwise: $\mathcal{O}(M^2)$. WCOJ: $\mathcal{O}(M^{3/2})$.

Bow-tie (two triangles sharing a vertex). Five variables, six constraints, two triangular cycles joined at a hub vertex. The pairwise approach requires computing per-vertex triangle counts and then combining—multiple stages, compounding intermediates. WCOJ processes all five variables with intersections at the cyclic depths, fusing both triangles into a single pass.

k-cycle (C_k for $k \geq 3$). The AGM bound gives $\mathcal{O}(M^{k/2})$; pairwise decomposition is stuck at $\mathcal{O}(M^2)$ because any tree decomposition of a k-cycle has width $\lceil k/2 \rceil$. The gap grows with k.

k-clique (K_k). The AGM bound is $\mathcal{O}(M^{k/2})$, which pairwise plans also achieve, so the worst-case gap is in constant factors. However, the WCOJ approach still avoids intermediate materialization and benefits from intersection pruning on real-world data.

8.4 The General Criterion

The dividing line is the **cyclicity of the query hypergraph**. Grohe and Marx [10] established:

- *Acyclic queries* (fractional hypertree width $= 1$): Yannakakis's algorithm computes the join in $\mathcal{O}(N + |\text{output}|)$ time using semijoins. Binary join plans suffice. WCOJ provides no asymptotic benefit.
- *Cyclic queries* (fractional hypertree width > 1): Any binary join-project plan is polynomially slower than the AGM bound in the worst case.

Ngo et al. [16] proved the stronger statement: for cyclic queries, "any join-project plan is destined to be slower than the best possible run time by a polynomial factor in the data size."

In graph terms: acyclic queries are path/tree pattern matching (two-way joins suffice). Every query involving a *cycle*—triangles, squares, diamonds, cliques, bow-ties, wheels—requires multi-way intersection for optimal evaluation. Since sparse linear algebra operates via pairwise matrix products, it is structurally limited to binary join plans. The masked SpGEMM trick recovers optimal performance for triangles as a special case, but the general cyclic case requires the WCOJ framework.

Table 4 summarizes the complexity landscape.

9 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.

9.1 Setup

Hardware.

Table 4: Pairwise vs. WCOJ complexity for cyclic graph patterns. M = number of edges in each relation.

Pattern	Query	Pairwise	WCOJ	Gap
2-path (acyclic)	$R \bowtie S$	$\mathcal{O}(M)$	$\mathcal{O}(M)$	—
Triangle (C_3)	3-way	$\mathcal{O}(M^2)$	$\mathcal{O}(M^{3/2})$	$M^{1/2}$
4-cycle (C_4)	4-way	$\mathcal{O}(M^2)$	$\mathcal{O}(M^{3/2})$	$M^{1/2}$
Diamond ($K_4 - e$)	5-way	$\mathcal{O}(M^2)$	$\mathcal{O}(M^{3/2})$	$M^{1/2}$
k-cycle (C_k)	k-way	$\mathcal{O}(M^2)$	$\mathcal{O}(M^{2-\frac{1}{\lceil k/2 \rceil}})$	polynomial
k-clique (K_k)	$\binom{k}{2}$-way	$\mathcal{O}(M^{k/2})$	$\mathcal{O}(M^{k/2})$	const.

Table 5: Triangle counting time (seconds) on RMAT graphs.

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

Graphs. We use RMAT/Kronecker graphs at scales 16, 18, 20, and 22 (edge factor 16), following the Graph500 specification [14]. 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.

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 6).
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. [1], using its native triangle counting query.
6. **GraphBLAS**: SuiteSparse:GraphBLAS [7], $C\langle L \rangle = L \cdot L$ formulation.

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

9.2 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.

9.3 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 2 is not merely theoretical.

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.

9.4 Analysis

10 Related Work

GraphBLAS and sparse linear algebra. GraphBLAS [11, 6] provides a standard API for expressing graph algorithms as sparse matrix operations. SuiteSparse:GraphBLAS [7] is the most widely used implementation. Triangle counting via masked SpMM [4, 20] 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.

Worst-case optimal joins. The AGM bound [3] on conjunctive query output size motivated WCOJ algorithms matching this bound [15]. Veldhuizen's Leapfrog Triejoin [18] operates on sorted data and is the basis of our analysis. The survey by Ngo et al. [16] provides an accessible introduction.

WCOJ for graph processing. EmptyHeaded [1] is a graph-processing engine built on WCOJ, demonstrating competitive performance with hand-tuned graph systems. Mhedhbi and Salihoglu [13] 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.

Relational vs. linear algebra optimization. Wang et al. [19] 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 7.

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 [8, 5]. Our work does not contribute new intersection algorithms; rather, we show that the WCOJ framework derives these algorithms mechanically from a declarative query specification.

Triangle counting in HPC. The HPC community has developed highly optimized triangle-counting implementations using CSR [17], degree ordering [12], and GPU acceleration [9]. 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.

11 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. [19] 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 7).

These results extend beyond triangles. As Section 8 demonstrates, the masked SpGEMM technique that partially closes the gap for triangles does not generalize: for 4-cycles, diamonds, bowties, and k-cycles, no natural mask exists, and the pairwise approach is stuck at $\mathcal{O}(M^2)$ while WCOJ achieves $\mathcal{O}(M^{3/2})$ or better. The dividing line is the cyclicity of the query hypergraph—any graph pattern containing a cycle requires multiway intersection for optimal evaluation, and pairwise matrix operations are structurally limited to binary join plans.

Worst-case optimal joins thus provide a principled path from declarative graph-pattern queries to provably optimal code, with zero abstraction overhead when the underlying storage is CSR. The WCOJ framework handles arbitrary cyclic join patterns, a capability that pairwise matrix operations fundamentally lack.

References

[1] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. EmptyHeaded: A relational engine for graph processing. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 431–446, 2017.

[2] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: Questions asked frequently. In *Proc. ACM Symposium on Principles of Database Systems (PODS)*, pages 13–28, 2016.

[3] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.

[4] Mohsen Mahmoudi Aznaveh, Jinhao Chen, Timothy A. Davis, Bence Hegyi, Scott P. Kolodziej, Timothy G. Mattson, and Gabor Parimalarangan. Parallel triangle counting and enumeration using matrix algebra. In *Proc. IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1–10, 2020.

[5] Ricardo Baeza-Yates and Alejandro Salinger. Fast intersection algorithms for sorted sequences. In *Algorithms and Applications*, pages 45–61, 2004.

[6] Aydin Buluç and John R. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[7] Timothy A. Davis. Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software*, 45(4):44:1–44:25, 2019.

[8] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 743–752, 2000.

[9] Oded Green and David A. Bader. Faster clustering coefficient using vertex covers. In *Proc. IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 1–8, 2014.

[10] Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. *ACM Transactions on Algorithms*, 11(1):4:1–4:20, 2014.

[11] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Jose Moreira, John Owens, Carl Yang, Marcin Zalewski, and Timothy Mattson. Mathematical foundations of the GraphBLAS. In *Proc. IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, 2016.

[12] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, 407(1–3):458–473, 2008.

[13] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endowment*, 12(11):1692–1704, 2019.

[14] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. Introducing the Graph 500. *Cray User's Group (CUG)*, 2010.

[15] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *Journal of the ACM*, 65(3):16:1–16:40, 2018.

[16] Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: New developments in the theory of join algorithms. *SIGMOD Record*, 42(4):5–16, 2013.

[17] Julian Shun and Kanat Tangwongsan. Multicore triangle computations without tuning. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, pages 149–160, 2015.

[18] Todd L. Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory (ICDT)*, pages 96–106, 2014.

[19] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. SPORES: Sum-product optimization via relational equality saturation for large scale linear algebra. *Proc. VLDB Endowment*, 13(12):1919–1932, 2020.

[20] Michael M. Wolf, Mehmet Deveci, Jonathan W. Berry, Simon D. Hammond, and Sivasankaran Rajamanickam. Fast linear algebra-based triangle counting with KokkosKernels. In *Proc. IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.