EmptyHeaded: A Relational Engine for Graph Processing

CHRISTOPHER R. ABERGER, ANDREW LAMB, SUSAN TU, ANDRES NÖTZLI, KUNLE OLUKOTUN, and CHRISTOPHER RÉ, Stanford University

There are two types of high-performance graph processing engines: low- and high-level engines. Low-level engines (Galois, PowerGraph, Snap) provide optimized data structures and computation models but require users to write low-level imperative code, hence ensuring that efficiency is the burden of the user. In high-level engines, users write in query languages like datalog (SociaLite) or SQL (Grail). High-level engines are easier to use but are orders of magnitude slower than the low-level graph engines. We present EmptyHeaded, a high-level engine that supports a rich datalog-like query language and achieves performance comparable to that of low-level engines. At the core of EmptyHeaded's design is a new class of join algorithms that satisfy strong theoretical guarantees, but have thus far not achieved performance comparable to that of specialized graph processing engines. To achieve high performance, EmptyHeaded introduces a new join engine architecture, including a novel query optimizer and execution engine that leverage single-instruction multiple data (SIMD) parallelism. With this architecture, EmptyHeaded outperforms high-level approaches by up to three orders of magnitude on graph pattern queries, PageRank, and Single-Source Shortest Paths (SSSP) and is an order of magnitude faster than many low-level baselines. We validate that EmptyHeaded competes with the best-of-breed low-level engine (Galois), achieving comparable performance on PageRank and at most 3× worse performance on SSSP. Finally, we show that the EmptyHeaded design can easily be extended to accommodate a standard resource description framework (RDF) workload, the LUBM benchmark. On the LUBM benchmark, we show that EmptyHeaded can compete with and sometimes outperform two high-level, but specialized RDF baselines (TripleBit and RDF-3X), while outperforming MonetDB by up to three orders of magnitude and LogicBlox by up to two orders of magnitude.

CCS Concepts: • **Information systems → DBMS engine architectures**; **Query planning**; **Join algorithms**;

Additional Key Words and Phrases: Worst-case optimal join, generalized hypertree decomposition, GHD, graph processing, single-instruction multiple data, SIMD

We gratefully acknowledge the support of the Defense Advanced Research Projects Agency (DARPA) XDATA Program under No. FA8750-12-2-0335 and DEFT Program under No. FA8750-13-2-0039, DARPA's MEMEX program and SIMPLEX program, the National Science Foundation (NSF) CAREER Award under No. IIS-1353606, the Office of Naval Research (ONR) under awards No. N000141210041 and No. N000141310129, the National Institutes of Health Grant U54EB020405 awarded by the National Institute of Biomedical Imaging and Bioengineering (NIBIB) through funds provided by the trans-NIH Big Data to Knowledge (BD2K, http://www.bd2k.nih.gov) initiative, the Sloan Research Fellowship, the Moore Foundation, American Family Insurance, Google, and Toshiba. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA, AFRL, NSF, ONR, NIH, or the U.S. government.

Authors' addresses: C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré, Gates Computer Science, 353 Serra Mall, Stanford, CA, 94043, USA; emails: {caberger, lamb, sctu, noetzli, kunle, chrismre}@stanford.edu.

https://doi.org/10.1145/3129246

ACM Reference format:
Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4, Article 20 (October 2017), 44 pages.
https://doi.org/10.1145/3129246

1 INTRODUCTION

The massive growth in the volume of graph data from social and biological networks has created a need for efficient graph processing engines. As a result, there has been a flurry of activity around designing specialized graph analytics engines [20, 25, 40, 52, 62]. These specialized engines offer new programming models that are either (1) low-level, requiring users to write code imperatively or (2) high-level, incurring large performance gaps relative to the low-level approaches. In this work, we explore whether we can meet the performance of low-level engines while supporting a high-level relational (SQL-like) programming interface.

Low-level graph engines outperform traditional relational data processing engines on common benchmarks due to (1) asymptotically faster algorithms [50, 59] and (2) optimized data layouts that provide large constant factor runtime improvements [25]. We describe each point in detail:

(1) Low-level graph engines [20, 25, 40, 52, 62] provide iterators and domain-specific primitives, with which users can write asymptotically faster algorithms than what traditional databases or high-level approaches can provide. However, it is the burden of the user to write the query properly, which may require system-specific optimizations. Therefore, optimal algorithmic runtimes can only be achieved through the user in these low-level engines.
(2) Low-level graph engines use optimized data layouts to efficiently manage the sparse relationships common in graph data. For example, optimized sparse matrix layouts are often used to represent the edgelist relation [25]. High-level graph engines also use sparse layouts like tail-nested tables [61] to cope with sparsity.

Extending the relational interface to match these guarantees is challenging. While some have argued that traditional engines can be modified in straightforward ways to accommodate graph workloads [7, 19], order of magnitude performance gaps remain between this approach and low-level engines [40, 52, 61]. Theoretically, traditional join engines face a losing battle, as all pairwise join engines are *provably suboptimal* on many common graph queries [50]. For example, low-level specialized engines execute the "triangle listing" query, which is common in graph workloads [45, 49], in time $O(N^{3/2})$ where N is the number of edges in the graph. Any pairwise relational algebra plan takes at least $\Omega(N^2)$, which is asymptotically worse than the specialized engines by a factor of $\sqrt{N}$. This asymptotic suboptimality is often inherited by high-level graph engines, as there has not been a general way to compile these queries that obtains the correct asymptotic bound [19, 61]. Recently, new multiway join algorithms were discovered that obtain the correct asymptotic bound for any graph pattern or join [50].

These new multiway join algorithms are by themselves not enough to close the gap. LogicBlox [7] uses multiway join algorithms and has demonstrated that they can support a rich set of applications. However, LogicBlox's current engine can be orders of magnitude slower than the specialized engines on graph benchmarks (see Section 5). This leaves open the question of whether these multiway joins are destined to be slower than specialized approaches.

We argue that an engine based on multiway join algorithms can close this gap, but it requires a novel architecture (Figure 1). In this article, we present the EmptyHeaded engine, which is a

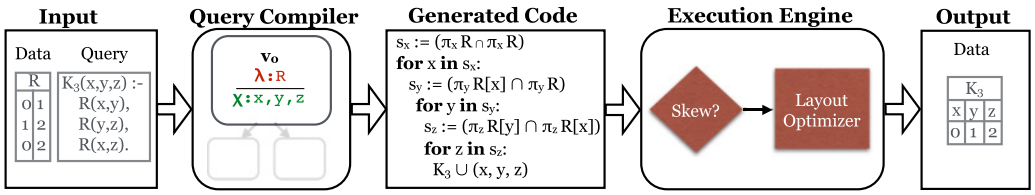


Fig. 1. The EmptyHeaded engine works in three phases: (1) the query compiler translates a high-level datalog-like query into a logical query plan represented as a GHD (a hypertree with a single node here), replacing the traditional role of relational algebra; (2) code is generated for the execution engine by translating the GHD into a series of set intersections and loops; and (3) the execution engine performs automatic algorithmic and layout decisions based upon skew in the data.

new multiway join engine. The EmptyHeaded architecture includes a novel query compiler based on *generalized hypertree decompositions* (GHDs) [14, 22] and an execution engine designed to exploit the low-level layouts necessary to increase single-instruction multiple data (SIMD) parallelism. We argue that these techniques demonstrate that multiway join engines can compete with low-level graph engines, as our prototype is faster than all tested engines on graph pattern queries (in some cases by orders of magnitude) and competitive on other common graph benchmarks.

We design EmptyHeaded around tight theoretical guarantees and data layouts optimized for SIMD parallelism.

GHDs as Query Plans. The classical approach to query planning uses relational algebra, which facilitates optimizations such as early aggregation, pushing down selections, and pushing down projections. In EmptyHeaded, we need a similar framework that supports multiway (instead of pairwise) joins. To accomplish this, based off of an initial prototype developed in our group [66], we use GHDs [22] for logical query plans in EmptyHeaded. GHDs allow one to apply the above classical optimizations to multiway joins. GHDs also have additional bookkeeping information that allows us to bound the size of intermediate results (optimally in the worst case). These bounds allow us to provide asymptotically stronger runtime guarantees than previous worst-case optimal join algorithms that do not use GHDs (including LogicBlox).[1] As these bounds depend on the data and the query it is difficult to expect users to write these algorithms in a low-level framework. Our contribution is the design of a novel query optimizer and code generator, based on GHDs, that is able to achieve the above results via a high-level query language.

Exploiting SIMD: The Battle With Skew. Optimizing relational databases for the SIMD hardware trend has become an increasingly hot research topic [41, 56, 73], as the available SIMD parallelism has been doubling consistently in each processor generation.[2] Inspired by this, we exploit the link between SIMD parallelism and worst-case optimal joins for the first time in EmptyHeaded. Our initial prototype revealed that during query execution, unoptimized set intersections often account for 95% of the overall runtime in the generic worst-case optimal join algorithm. Thus, it is critically important to optimize set intersections and the associated data layout to be well suited for

[1]LogicBlox has described a (non-public) prototype with an optimizer similar but distinct from GHDs. With these modifications, LogicBlox's relative performance improves similarly to our own. It, however, remains at least an order of magnitude slower than EmptyHeaded.

[2]The Intel Ivy Bridge architecture, which we use in this article, has a SIMD register width of 256 bits. The next generation, the Intel Skylake architecture, has 512-bit registers and a larger number of such registers.

SIMD parallelism. This is a challenging task as graph data is highly skewed, causing the runtime characteristics of set intersections to be highly varied. We explore several sophisticated (and not so sophisticated) layouts and algorithms to opportunistically increase the amount of available SIMD parallelism in the set intersection operation. Our contribution here is an automated optimizer that, all told, increases performance by up to three orders of magnitude by selecting amongst multiple data layouts and set intersection algorithms that use skew to increase the amount of available SIMD parallelism.

We choose to evaluate EmptyHeaded on graph pattern matching queries since pattern queries are naturally (and classically) expressed as join queries. We also evaluate EmptyHeaded on other common graph workloads including PageRank and Single-Source Shortest Paths (SSSP). We show that EmptyHeaded consistently outperforms the standard baselines [19] by 2–4× on PageRank and is at most 3× slower than the highly tuned implementation of Galois [52] on SSSP. However, in our high-level language these queries are expressed in 1–2 lines, while they are over 150 lines of code in Galois. For reference, a hand-coded C implementation with similar performance to Galois is 1,000 lines.

Contribution Summary. This article introduces the EmptyHeaded engine and demonstrates that a novel architecture can enable multiway join engines to compete with specialized low-level graph processing engines. We demonstrate that EmptyHeaded outperforms specialized engines on graph pattern queries while remaining competitive on other workloads. To validate our claims, we provide comparisons on standard graph benchmark queries that the specialized engines are designed to process efficiently.

A summary of our contributions and an outline is as follows:

- We describe the first worst-case optimal join processing engine to use GHDs for logical query plans. We describe how GHDs enable us to provide a tighter theoretical guarantee than previous worst-case optimal join engines (Section 3). Next, we validate that the optimizations GHDs enable can provide more than a three orders of magnitude performance advantage over previous worst-case optimal query plans (Section 5).
- We describe the architecture of the first worst-case optimal execution engine that optimizes for skew at several levels of granularity within the data. We present a series of automatic optimizers to select intersection algorithms and set layouts based on data characteristics at runtime (Section 4). We demonstrate that our automatic optimizers can result in up to a three orders of magnitude performance improvement on common graph pattern queries (Section 5).
- We validate that our general-purpose engine can compete with specialized engines on standard benchmarks in the graph domain (Section 5). We demonstrate that on cyclic graph pattern queries our approach outperforms graph engines by 2–60× and LogicBlox by three orders of magnitude. We demonstrate on PageRank and Single-Source Shortest Paths that our approach remains competitive, at most 3× off the highly tuned Galois engine (Section 5).
- We describe how EmptyHeaded can easily accommodate popular resource description framework (RDF) workloads and perform the first study of worst-case optimal join algorithms in the RDF domain (Section 6). In this study, we validate that EmptyHeaded achieves (or outperforms substantially) other popular high-level approaches on a standard RDF benchmark, the LUBM benchmark (Section 6). More precisely, we show that EmptyHeaded can be two orders of magnitude better than MonetDB, an order of magnitude better than LogicBlox, and always within an order of magnitude of TripleBit and RDF-3X on the LUBM benchmark.

2 PRELIMINARIES

We briefly review the worst-case optimal join algorithm, trie data structure, and query language at the core of the EmptyHeaded design. The worst-case optimal join algorithm, trie data structure, and query language presented here serve as building blocks for the remainder of the article.

ALGORITHM 1: Generic Worst-Case Optimal Join Algorithm

```
1   //Input: Hypergraph H=(V,E), and a tuple t.
2   Generic-Join(V,E,t):
3     if |V|=1 then return ∩_{e∈E} R_e[t].
4     Let I={v_1} // the first attribute.
5     Q ← ∅ // the return value
6     // Intersect all relations that contain v_1
7     // Only those tuples that agree with t.
8     for every t_v ∈ ∩_{e∈E:e∋v_1} π_I(R_e[t]) do
9       Q_t ← Generic-Join(V−I, E, t :: t_v)
10      Q ← Q ∪ {t_v} × Q_t
11    return Q
```

2.1 Worst-Case Optimal Join Algorithms

We briefly review worst-case optimal join algorithms, which are used in EmptyHeaded. We present these results informally and refer the reader to Ngo et al. [51] for a complete survey. The main idea is that one can place (tight) bounds on the maximum possible number of tuples returned by a query and then develop algorithms whose runtime guarantees match these worst-case bounds. For the moment, we consider only join queries (no projection or aggregation), returning to these richer queries in Section 3.

A ***hypergraph*** is a pair $H = (V, E)$, consisting of a nonempty set V of vertices, and a set E of subsets of V, the hyperedges of H. Natural join queries and graph pattern queries can be expressed as hypergraphs [22]. In particular, there is a direct correspondence between a query and its hypergraph: there is a vertex for each attribute of the query and a hyperedge for each relation. We will go freely back and forth between the query and the hypergraph that represents it.

A recent result of Atserias, Grohe, and Marx [9] (AGM) showed how to tightly bound the worst-case size of a join query using a notion called a fractional cover. Fix a hypergraph $H = (V, E)$. Let $x \in \mathbb{R}^{|E|}$ be a vector indexed by edges, i.e., with one component for each edge, such that $x \geq 0$; x is a *feasible cover* (or simply feasible) for H if

$$\text{for each } v \in V \text{ we have } \sum_{e \in E : e \ni v} x_e \geq 1.$$

A feasible cover x is also called a *fractional hypergraph cover* in the literature. AGM showed that if x is feasible, then it forms an upper bound of the query result size $|\text{OUT}|$ as follows:

$$|\text{OUT}| \leq \prod_{e \in E} |R_e|^{x_e}. \tag{1}$$

For a query Q, we denote $\text{AGM}(Q)$ as the smallest such right-hand side.[3]

Example 2.1. For simplicity, let $|R_e| = N$ for $e \in E$. Consider the triangle query, $R(x, y) \bowtie S(y, z) \bowtie T(x, z)$; a feasible cover is $x_R = x_S = 1$ and $x_T = 0$. Via Equation (1), we know that $|\text{OUT}| \leq N^2$. That is, with N tuples in each relation we cannot produce a set of output tuples that

[3] One can find the best bound, $\text{AGM}(Q)$, in polynomial time: take the log of Equation (1) and solve the linear program.

contains more than N^2. However, a tighter bound can be obtained using a different fractional cover $x = (\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$. Equation (1) yields the upper bound $N^{3/2}$. Remarkably, this bound is tight if one considers the complete graph on $\sqrt{N}$ vertexes. For this graph, this query produces $\Omega(N^{3/2})$ tuples, which shows that the optimal solution can be tight up to constant factors.

The first algorithm to have a running time matching these worst-case size bounds is the NPRR algorithm [50]. An important property for the set intersections in the NPRR algorithm is what we call the *min property*: the running time of the intersection algorithm is upper bounded by the length of the *smaller* of the two input sets. When the min property holds, a worst-case optimal running time for *any* join query is guaranteed. In fact, for *any* join query, its execution time can be upper bounded by AGM(Q). A simplified high-level description of the algorithm is presented in Algorithm 1. It was also shown that any pairwise join plan must be slower by asymptotic factors. However, we show in Section 3.1 that these optimality guarantees can be improved for non-worst-case data or more complex queries.

2.2 Input Data

EmptyHeaded stores all relations (input and output) in tries, which are multilevel data structures common in column stores and graph engines [25, 64]. Although EmptyHeaded is currently optimized for purely static data, other researchers [35] have shown that tries can provide high performance on transactional workloads. Extending EmptyHeaded in this direction is part of our future research.

Trie Annotations. The sets of values in the trie can optionally be associated with data values (1-1 mapping) that are used in aggregations. We call these associated values *annotations* [23]. For example, a two-level trie annotated with a float value represents a sparse matrix or graph with edge properties. We show in Section 5 that the trie data structure works well on a wide variety of graph workloads.

Dictionary Encoding. The tries in EmptyHeaded currently support sets containing 32-bit values. As is standard [20, 56], we use the popular database technique of dictionary encoding to build an EmptyHeaded trie from input tables of arbitrary types. Dictionary encoding maps original data values to keys of another type—in our case 32-bit unsigned integers. The order of dictionary ID assignment affects the density of the sets in the trie, and as others have shown, this can have a dramatic impact on overall performance on certain queries. Like others, we find that node ordering is powerful when coupled with pruning half the edges in an undirected graph [59]. This creates up to 3× performance difference on symmetric pattern queries like the triangle query. Unfortunately, this optimization is brittle, as the necessary symmetrical properties break with even a simple selection. On more general queries we find that node ordering typically has less than a 10% overall performance impact. We explore the effect of various node orderings in Section 4.5.

Column (Index) Order. After dictionary encoding, our 32-bit value relations are next grouped into sets of distinct values based on their parent attribute (or column). We are free to select which level corresponds to each attribute (or column) of an input relation. As with most graph engines, we simply store both orders for each edge relation. In general, we choose the order of the attributes for the trie based on a global attribute order, which is analogous to selecting a single index over the relation. The trie construction process produces tries where the sets of data values can be extremely dense, extremely sparse, or anywhere in between. Optimizing the layout of these sets based upon their data characteristics is the focus of Section 4. The complete transformation process from a standard relational table to the trie representation in EmptyHeaded is detailed in Figure 2.

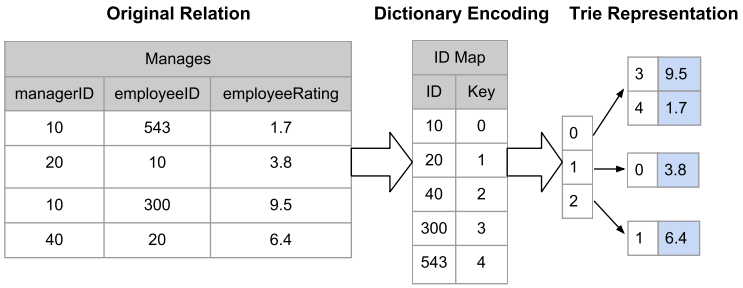


Fig. 2. EmptyHeaded transformations from a table to trie representation using attribute order (*managerID,employerID*) and *employerID* attribute annotated with *employeeRating*.

Table 1. Example Graph Queries in EmptyHeaded

Name	Query Syntax
Triangle	`Triangle(x,y,z):-R(x,y),S(y,z),T(x,z).`
4-Clique	`4Clique(x,y,z,w):-R(x,y),S(y,z),T(x,z),U(x,w),V(y,w),Q(z,w).`
Lollipop	`Lollipop(x,y,z,w):-R(x,y),S(y,z),T(x,z),U(x,w).`
Barbell	`Barbell(x,y,z,x',y',z'):-R(x,y),S(y,z),T(x,z),U(x,x'), R'(x',y'),S'(y',z'),T'(x',z').`
Count Triangle	`CntTriangle(;w:long):-R(x,y),S(x,z),T(x,z); w=<<COUNT(*)>>.`
4-Clique-Selection	`S4Clique(x,y,z,w):-R(x,y),S(y,z),T(x,z),U(x,w), V(y,w),Q(z,w),P(x,`node').`
Barbell-Selection	`SBarbell(x,y,z,x',y',z'):-R(x,y),S(y,z),T(x,z),U(x,`node'), V(`node',x'),R'(x',y'),S'(y',z'),T'(x',z').`
PageRank	`N(;w:int):-Edge(x,y); w=<<COUNT(x)>>.` `PageRank(x;y:float):-Edge(x,z); y= 1/N.` `PageRank(x;y:float)*[i=5]:-Edge(x,z),PageRank(z),InvDeg(z); y=0.15+0.85*<<SUM(z)>>.`
SSSP	`SSSP(x;y:int):-Edge(`start',x); y=1.` `SSSP(x;y:int)*:-Edge(w,x),SSSP(w); y=<<MIN(w)>>+1.`

2.3 Query Language

Our query language is inspired by datalog and supports conjunctive queries with aggregations and simple recursion (similar to LogicBlox and SociaLite). In this section, we describe the core syntax for our queries, which is sufficient to express the standard benchmarks we run in Sections 5 and 6. Table 1 shows the example queries used in Section 5. Above the first horizontal line are conjunctive queries that express joins, projections, and selections in the standard way [67]. Our language has two non-standard extensions: aggregations and a limited form of recursion. We overview both extensions next and provide an example.

Conjunctive Queries: Joins, Projections, Selections. Equality joins are expressed in EmptyHeaded as simple conjunctive queries. We show EmptyHeaded 's' syntax for two cyclic join queries in Table 1: the 3-clique query (also known as triangle or K_3), and the Barbell query (two 3-cliques

connected by a path of length 1). EmptyHeaded easily enables selections and projections in its query language as well. We enable projections through the user directly annotating which attributes appear in the head. We enable selections by directly annotating predicates on attribute values in the body (e.g., "node" for the 4-Clique-Selection query in Table 1).

Aggregation. Following Green et al. [23], tuples can be annotated in EmptyHeaded, and these annotations support aggregations from any semiring (a generalization of natural numbers equipped with a notion of addition and multiplication). This enables EmptyHeaded to support classic aggregations such as `SUM`, `MIN`, or `COUNT`, but also more sophisticated operations such as matrix multiplication. To specify the annotation, one uses a semicolon in the head of the rule, e.g., `q(x,y;z:int)` specifies that each `x,y` pair will be associated with an integer value with alias `z` similar to a `GROUP BY` in SQL. In addition, the user expresses the aggregation operation in the body of the rule. The user can specify an initialization value as any expression over the tuples' values and constants, while common aggregates have default values. Directly below the first line in Table 1, a typical triangle counting query is shown.

Recursion. EmptyHeaded supports a simplified form of recursion similar to Kleene-star or transitive closure. Given an intensional or extensional relation R, one can write a Kleene-star rule like

$$R^*(\bar{x}) \quad :- \quad q(\bar{x}, \bar{y}).$$

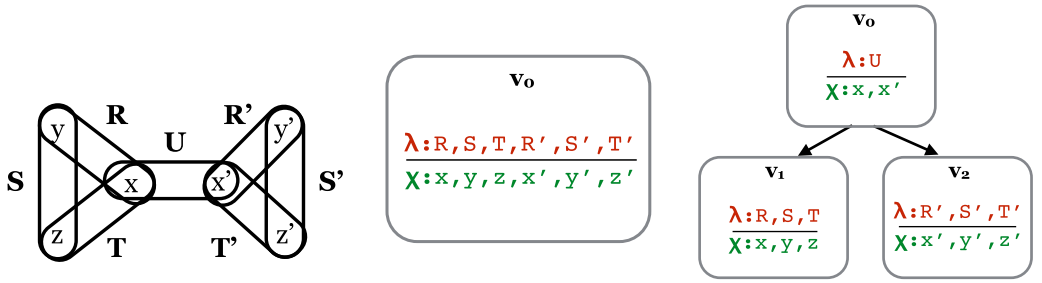
The rule $R*$ iteratively applies q to the current instantiation of R to generate new tuples which are added to R. It performs this iteration until (a) the relation does not change (a fixpoint semantic) or (b) a user-defined convergence criterion is satisfied (e.g., a number of iterations, `i=5`). Examples that capture familiar PageRank and SSSPs are below the second horizontal line in Table 1.

We illustrate how our query language works by an example for the PageRank query:

Example 2.2. Table 1 shows an example of the syntax used to express the PageRank query in EmptyHeaded . The first line specifies that we aggregate over all the edges in the graph and count the number of source nodes assuming our *Edge* relation is a two-attribute relation filled with (src, dst) pairs. For an undirected graph this simply counts the number of nodes in the graph and assigns it to the relation N which is really just a scalar integer. By definition the `COUNT` aggregation and by default the `SUM` use an initialization value of 1 if the relation is not annotated. The second line of the query defines the base case for recursion. Here we simply project away the z attributes and assign an annotation value of $1/N$ (where N is our scalar relation holding the number of nodes). Finally, the third line defines the recursive rule which joins the *Edge* and *InvDegree* relations inside the database with the new *PageRank* relation. We `SUM` over the z attribute in all of these relations. When aggregated attributes are joined with each other their annotation values are multiplied by default [28]. Therefore, we are performing a matrix-vector multiplication. After the aggregation, the corresponding expression for the annotation y is applied to each aggregated value. This is run for a fixed number (five) iterations as specified in the head.

3 QUERY COMPILER

We now present an overview of the query compiler in EmptyHeaded, which is the first worst-case optimal query compiler to enable early aggregation through its use of GHDs for logical query plans. We first discuss GHDs and their theoretical advantages. Next, we describe how we develop a simple optimizer to select a GHD (and therefore a query plan). After this, we describe how we implement the classic query optimization of pushing down selections using our GHD-based query plans. Next, we show how EmptyHeaded translates a GHD into a series of loops, aggregations, and set intersections using the generic worst-case optimal join algorithm [50]. Finally, we describe how EmptyHeaded eliminates redundant work via a simple common subexpression elimination



(a) Hypergraph (b) LogicBlox GHD (c) EmptyHeaded GHD

Fig. 3. We show the Barbell query hypergraph and two possible GHDs for the query. A node v in a GHD captures which relations should be joined with $\lambda(v)$ and which attributes should be retained with projection with $\chi(v)$.

algorithm in our GHD-based query compiler. Our contribution here is the design of a novel query compiler that provides tighter runtime guarantees than existing approaches.

3.1 Query Plans using GHDs

As in a classical database, EmptyHeaded needs an analog of relational algebra to represent logical query plans. In contrast to traditional relational algebra, EmptyHeaded has multiway join operators. A natural approach would be simply to extend relational algebra with a multiway join algorithm. Instead, we advocate replacing relational algebra with GHDs, which allow us to make non-trivial estimates on the cardinality of intermediate results. This enables optimizations, like early aggregation in EmptyHeaded, that can be asymptotically faster than existing worst-case optimal engines. We first describe the motivation for using GHDs while formally describing their advantages next.

3.1.1 Motivation. A GHD is a tree similar to the abstract syntax tree of a relational algebra expression: nodes represent a join and projection operation, and edges indicate data dependencies. A node v in a GHD captures which attributes should be retained (projection with $\chi(v)$) and which relations should be joined (with $\lambda(v)$). We consider all possible query plans (and therefore all valid GHDs), selecting the one where the sum of each node's runtime is the lowest. Given a query, there are many valid GHDs that capture the query. Finding the lowest-cost GHD is one goal of our optimizer.

Before giving the formal definition, we illustrate GHDs and their advantages by example:

Example 3.1. Figure 3(a) shows a hypergraph of the Barbell query introduced in Table 1. This query finds all pairs of triangles connected by a path of length one. Let $|\text{OUT}|$ be the size of the output data. From our definition in Section 2.1, one can check that the Barbell query has a feasible cover of $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 0, \frac{1}{2}, \frac{1}{2}, \frac{1}{2})$ with cost $6 \times \frac{1}{2} = 3$ and so runs in time $O(N^3)$. In fact, this bound is worst-case optimal because there are instances that return $\Omega(N^3)$ tuples. However, the size of the output $|\text{OUT}|$ could be much smaller.

There are multiple GHDs for the Barbell query. The simplest GHD for this query (and in fact for all queries) is a GHD with a single node containing all relations; the single node GHD for the Barbell query is shown in Figure 3(b). One can view all of LogicBlox's current query plans as a single node GHD. The single node GHD always represents a query plan which uses only the generic worst-case optimal join algorithm and no GHD optimizations. For the Barbell query, $|\text{OUT}|$ is N^3 in the worst case for the single node GHD.

Consider the alternative GHD shown in Figure 3(c). This GHD corresponds to the following alternate strategy to the above plan: first list each triangle independently using the generic worst-case optimal algorithm, say on the vertices (x, y, z), and then (x', y', z'). There are at most $O(N^{3/2})$ triangles in each of these sets and so it takes only this time. Now, for each $(x, x') \in U$ we output all the triangles that contain x or x' in the appropriate position. This approach is able to run in time $O(N^{3/2} + |\text{OUT}|)$ and essentially performs early aggregation if possible. This approach can be substantially faster when $|\text{OUT}|$ is smaller than N^3. For example, in an aggregation query $|\text{OUT}|$ is just a single scalar, and so the difference in runtime between the two GHDs can be $N^{3/2}$ where N is the size of the database. We describe how we execute this query plan in Section 3.4. This type of optimization is currently not available in the LogicBlox engine.

In general, GHDs allow us to capture this early aggregation, which can lead to dramatic runtime improvements.

3.1.2 Formal Description. We describe GHDs and their advantages formally next.

Definition 3.2. Let H be a hypergraph. A ***GHD*** of H is a triple $D = (T, \chi, \lambda)$, where

- $T(V(T), E(T))$ is a tree;
- $\chi : V(T) \rightarrow 2^{V(H)}$ is a function associating a set of vertices $\chi(v) \subseteq V(H)$ to each node v of T;
- $\lambda : V(T) \rightarrow 2^{E(H)}$ is a function associating a set of hyperedges to each vertex v of T;

such that the following properties hold:

1. For each $e \in E(H)$, there is a node $v \in V(T)$ such that $e \subseteq \chi(v)$ and $e \in \lambda(v)$.
2. For each $t \in V(H)$, the set $\{v \in V(T) | t \in \chi(v)\}$ is connected in T.
3. For every $v \in V(T)$, $\chi(v) \subseteq \cup\lambda(v)$.

A GHD can be thought of as a labeled (hyper)tree, as illustrated in Figure 3. Each node of the tree v is labeled; $\chi(v)$ describes which attributes are "returned" by the node v—this exactly captures projection in traditional relational algebra. The label $\lambda(v)$ captures the set of relations that are present in a (multiway) join at this particular node. The first property says that every edge is mapped to some node, and the second property is the famous *"running intersection property"* [5] that says any attribute must form a connected subtree. The third property is redundant for us, as any GHD violating this condition is not considered (has infinite width which we describe next).

Using GHDs, we can define a non-trivial cardinality estimate based on the sizes of the relations. For a node v, define Q_v as the query formed by joining the relations in $\lambda(v)$. The ***(fractional) width*** of a GHD is $\text{AGM}(Q_v)$, which is an upper bound on the number of tuples returned by Q_v. The ***fractional hypertree width (fhw)*** of a hypergraph H is the minimum width of all GHDs of H. Given a GHD with width w, there is a simple algorithm to run in time $O(N^w + |\text{OUT}|)$. First, run any worst-case optimal algorithm on Q_v for each node v of the GHD; each join takes time $O(N^w)$ and produces at most $O(N^w)$ tuples. Then, one is left with an acyclic query over the output of Q_v, namely, the tree itself. We then perform Yannakakis' classical algorithm [70], which for acyclic queries enables EmptyHeaded to compute the output in time linear in the input size ($O(N^w)$) plus the output size ($|\text{OUT}|$).

3.2 Choosing Logical Query Plans

We describe how EmptyHeaded chooses GHDs, explain how we leverage previous work to enable aggregations over GHDs, and describe how GHDs are used to select a global attribute ordering

in EmptyHeaded . In Section 3.3, we provide details on how the classic database optimizations of pushing down selections can be captured using GHDs.

GHD Optimizer. The EmptyHeaded query compiler selects an optimal GHD to ensure tighter theoretical runtime guarantees. It is key that the EmptyHeaded optimizer selects a GHD with the smallest width w to ensure an optimal GHD. Similar to how a traditional database pushes down projections to minimize the output size, EmptyHeaded minimizes the output size by finding the GHD with the smallest width. In contrast to pushing down projections, finding the minimum width GHD is NP-hard in the number of relations and attributes. As the number of relations and attributes is typically small (three for triangle counting), we simply brute force search GHDs of all possible widths. The algorithm EmptyHeaded uses for performing this brute force search is shown in Algorithm 2. The algorithm computes a list of GHDs by recursively computing the GHDs for all subsets of the edge set. In Section 3.3, we describe three simple rules that can be added to this process to enable pushing down selections.

Aggregations over GHDs. Previous work has investigated aggregations over hypertree decompositions [22, 54]. EmptyHeaded adopts this previous work in a straightforward way. To do this, we add a single attribute with "semiring annotations" following Green et al. [23]. EmptyHeaded simply manipulates this value as it is projected away. This general notion of aggregations over annotations enables EmptyHeaded to support traditional notions of queries with aggregations as well as a wide range of workloads outside traditional data processing, like message passing in graphical models.

Global Attribute Ordering. Once a GHD is selected, EmptyHeaded selects a global attribute ordering. The global attribute ordering determines the order in which EmptyHeaded code generates the generic worst-case optimal algorithm (Algorithm 1) and the index structure of our tries (Section 2.2). Therefore, selecting a global attribute ordering is analogous to selecting a join and index order in a traditional pairwise relational engine. The attribute order depends on the query. For the purposes of this article, we assume both trie orderings are present, and we are therefore free to select any attribute order. For graphs (two-attributes), most in-memory graph engines maintain both the matrix and its transpose in the compressed sparse row format [25, 52]. We are the first to consider selecting an attribute ordering based on a GHD and as a result we explore simple heuristics based on structural properties of the GHD. To assign an attribute order for all queries in this article, EmptyHeaded simply performs a pre-order traversal over the GHD, adding the attributes from each visited GHD node into a queue.

3.3 Pushing Down Selections

A classic database optimization is to force high selectivity operations to be processed as early as possible in a query plan [27]. In EmptyHeaded we can do this at two different granularities in our query plans: within GHD nodes and across GHD nodes.[4]

3.3.1 Within a Node. In EmptyHeaded pushing down selections within a GHD node corresponds to rearranging the attribute order for the generic worst-case optimal join algorithm that we described in Section 2.1. Recall, the attribute order determines both the order that attributes are processed in Algorithm 1 and the order in which the attributes will appear in the trie.

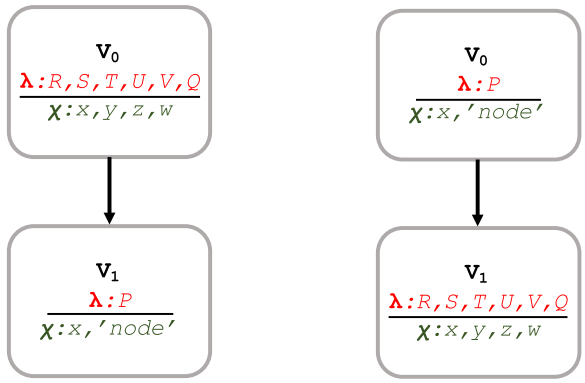
[4]Recall EmptyHeaded executes a GHD query plan in two phases: (1) the generic worst-case optimal join algorithm runs inside of each node in the GHD and (2) the final result is computed by passing intermediate results across nodes. The phases directly correspond to the two granularities at which we push down selections.

ALGORITHM 2: Enumerating all GHDs via Brute Force Search

```
1   // Input: Hypergraph H = (V, E) and a set of parent edges P.
2   // Output: A list of GHDs in the form of (GHD node, subtree) pairs.
3   GHD-Enumeration(V, E, P):
4     GHDs = []
5     // Iterate over all subset combinations of edges.
6     for {C | C ⊆ E} do
7       // The remaining edges, not in C.
8       R = E - C
9       // If the running intersection property is broken, the GHD is
10      // not valid. The check makes sure that all attributes in the
11      // parent and subtree of a specified GHD node also appear
12      // within the specified GHD node. Here we use ∪ on a set
13      // of edges to indicate the union of their attributes.
14      if not ( (∪P) ∩ (∪R) ⊆ ∪C ) then continue
15      // Consider each subgraph of the remaining edges. For each
16      // subgraph, recursively enumerate all possible GHDs.
17      PartitionChildren = []
18      for Pr in Partition(R) do
19        PartitionChildren += [GHD-Enumeration(∪Pr, Pr, C)]
20      // Consider all possible combinations of subtrees by calling the
21      // method below. For each, construct a GHD with C as the root.
22      for Ch in Subtree-Combinations(PartitionChildren) do
23        GHDs += [(C, Ch)]
24    return GHDs
25
26  // Input: A list of lists of GHDs: for each partition of a hypergraph
27  // (the outer list), all possible decompositions for that partition
28  // (the inner lists).
29  // Output: A list where each member of this list is a list that
30  // contains one subtree from each partition.
31  Subtree-Combinations(PartitionChildren):
32    ChildrenCombinations = []
33    if |PartitionChildren| > 0 then
34      if |PartitionChildren| == 1 then
35        // If there is only one partition, for each of the possible GHDs
36        // of this partition, add a combination with just this GHD.
37        for Ch in PartitionChildren[0] do
38          ChildrenCombinations += [[Ch]]
39      else
40        // Recursively generate combinations for the partitions after
41        // the first one.
42        RemainingCombinations = Subtree-Combinations(PartitionChildren[1:])
43        // If there is more than one partition, each subtree in the
44        // first partition is combined with each list of subtrees in
45        // the recursively generated combinations for the remaining
46        // partitions.
47        for Ch in PartitionChildren[0] do
48          for C in RemainingCombinations do
49              FinalCombination = [Ch] + C
50              ChildrenCombinations += [FinalCombination]
51    return ChildrenCombinations
```

Example 3.3. Consider a query where the relation R has one selected attribute and another (x) to be materialized in the output:

```
OUT(x) :- R('node',x).
```



(a) GHD without pushing down (b) GHD with pushing down

Fig. 4. We show two possible GHDs for the 4-clique selection query.

For this trivial query we produce a single node GHD containing attributes `('node',x)`. An attribute ordering of `[x,'node']` in this node means that `'x'` is the first level of the trie and `'node'` is the second level of the trie. Thus, the worst-case optimal join algorithm would probe the second level of the trie for each `'node'` attribute to determine if there was a corresponding value of `'node'`. This is much less efficient than selecting the attribute ordering of `['node',x]`, where EmptyHeaded can simply perform a lookup in the first level of the trie (to find if a value of `'node'` exists) and, if successful, return the corresponding second level as the result.

3.3.2 Across Nodes. Pushing down selections across nodes in EmptyHeaded's query plans corresponds to changing the criteria for choosing a GHD described in Section 3.2. Our goal is to have high-selectivity or low-cardinality nodes be pushed down as far as possible in the GHD so that they are executed earlier in our bottom-up pass. We accomplish this by adding three additional steps to our GHD optimizer:

(1) Find optimal GHDs $\mathcal{T}$ with respect to fhw, changing V in the AGM constraint to be only the attributes without selections.
(2) Let R_s be some relations with selections and let R_t be the relations that we plan to place in a subtree. If for each $e \in R_s$, there exists $e' \in R_t$ such that e' covers e's unselected attributes, include R_s in the subtree for R_t. This means that we may duplicate some members of R_s to include them in multiple subtrees.
(3) Of the GHDs $\mathcal{T}$, choose a $T \in \mathcal{T}$ with maximal selection depth, where selection depth is the sum of the distances from selections to the root of the GHD.

In Figure 4, we show two possible GHDs for the 4-clique selection query: one produced without these additional rules to push down selections (Figure 4(a)) and one produced with these additional rules to push down selections (Figure 4(b)). The GHD shown in Figure 4(b) can result in up to four orders of magnitude faster overall query runtime than (see Section 5) the GHD shown in Figure 4(a) on real datasets. Additionally, these simple rules to push down selections can speed up RDF queries by up to 234× (see Section 6).

3.4 Code Generation

EmptyHeaded's code generator converts the selected GHD for each query into optimized C++ code that uses the operators in Table 2. We choose to implement code generation in EmptyHeaded

Table 2. Execution Engine Operations

	Operation	Description
Trie (R)	$R[t]$	Returns the set matching tuple $t \in R$.
	$R \leftarrow R \cup t \times xs$	Appends elements in set xs to tuple $t \in R$.
Set (xs)	for x in xs	Iterates through the elements x of a set xs.
	$xs \cap ys$	Returns the intersection of sets xs and ys.

as it is has been shown to be an efficient technique to translate high-level query plans into code optimized for modern hardware [47].

3.4.1 Code Generation API. We first describe the storage-engine operations which serve as the basic high-level application programming interface (API) for our generated code. Our trie data structure offers a standard, simple API for traversals and set intersections that is sufficient to express the worst-case optimal join algorithm detailed in Algorithm 1. The key operation over the trie is to return a set of values that match a specified tuple predicate (see Table 2). This operation is typically performed while traversing the trie, so EmptyHeaded provides an optimized iterator interface. The set of values retrieved from the trie can be intersected with other sets or iterated over using the operations in Table 2.

3.4.2 GHD Translation. The goal of code generation is to translate a GHD to the operations in Table 2. Each GHD node $v \in V(T)$ is associated with a trie described by the attribute ordering in $\chi(v)$. Unlike previous worst-case optimal join engines, there are two phases to our algorithm: (1) within nodes of $V(T)$ and (2) between nodes $V(T)$.

Within a Node. For each $v \in V(T)$, we run the generic worst-case optimal algorithm shown in Algorithm 1. Suppose Q_v is the triangle query.

Example 3.4. Consider the triangle query. The hypergraph is $V = \{X, Y, Z\}$ and $E = \{R, S, T\}$. In the first call, the loop body generates a loop with body Generic-Join$(\{Y, Z\}, E, t_X)$. In turn, with two more calls this generates

for $t_X \in \pi_X R \cap \pi_X T$ do
  for $t_Y \in \pi_Y R[t_X] \cap \pi_Y S$ do
    $Q \leftarrow Q \cup (t_x, t_y) \times (\pi_Z S[t_Y] \cap \pi_Z T[t_X])$.

Across Nodes. Recall Yannakakis' seminal algorithm [70]: we first perform a "bottom-up" pass, which is a reverse level-order traversal of T. For each $v \in V(T)$, the algorithm computes Q_v and passes its results to the parent node. Between nodes (v_0, v_1) we pass the relations projected onto the shared attributes $\chi(v_0) \cap \chi(v_1)$. Then, the result is constructed by walking the tree "top-down" and collecting each result.

Recursion. EmptyHeaded supports both naive and semi-naive evaluation to handle recursion. For naive recursion, EmptyHeaded's optimizer produces a (potentially infinite) linear chain GHD with the output of one GHD node serving as the input to its parent GHD node. We run naive recursion for PageRank in Table 1. This boils down to a simple unrolling of the join algorithm. Naive recursion is not an acceptable solution in applications such as SSSP where work is continually being eliminated. To detect when EmptyHeaded should run semi-naive recursion, we check

if the aggregation is monotonically increasing or decreasing with a `MIN` or `MAX` operator. We use semi-naive recursion for SSSP.

Example 3.5. For the Barbell query (see Figure 3(c)), we first run Algorithm 1 on nodes v_1 and v_2; then we project their results on x and x' and pass them to node v_0. This is part of the "bottom-up" pass. We then execute Algorithm 1 on node v_0 which now contains the results (triangles) of its children. Algorithm 1 executes here by simply checking for pairs of (x, x') from its children that are in U. To perform the "top-down" pass, for each matching pair, we append (y, z) from v_1 and (y', z') from v_2.

3.5 Eliminating Redundant Work

An artifact of our GHD-based query compiler is that it is possible to produce a GHD query plan with two identical nodes and therefore a query plan with redundant work. To address this we add a simple form of common subexpression elimination to our query compiler. This enables the elimination of redundant work across GHD nodes and across phases of code generation.

Our query compiler performs a simple analysis to determine if two GHD nodes are identical and therefore if redundant work can be eliminated. For each GHD node in the "bottom-up" pass of Yannakakis' algorithm, we scan a list of the previously computed GHD nodes to determine if the result of the current node has already been computed. We use the conditions below to determine if two GHD nodes are equivalent in the Barbell query. Recognizing this provides a 2× performance increase on the Barbell query.

We say that two GHD nodes produce equivalent results in the "bottom-up pass" if

(1) the two nodes contain identical join patterns on the same input relations;
(2) the two nodes contain identical aggregations, selections, and projections;
(3) the results from each of their subtrees are identical.

We can also eliminate the "top-down" pass of Yannakakis' algorithm if all the attributes appearing in the result also appear in the root node. This determines if the final query result is present after the "bottom-up" phase of Yannakakis' algorithm. For example, if we perform a `COUNT` query on all attributes, the "top-down" pass in general is unnecessary. We found eliminating the top-down pass provided a 10% performance improvement on the Barbell query.

4 EXECUTION ENGINE

The EmptyHeaded execution engine runs code generated from the query compiler. The goal of the engine is to fully utilize SIMD parallelism, which is challenging because graph data is often skewed in several distinct ways. The density of data values is almost never constant: some parts of the relation are dense while others are sparse. We call this *density skew*.[5] Further, the cardinality of the data values in graph neighborhoods is highly varied. We call this *cardinality skew*. Table 3 shows the graph datasets used in this article along with their cardinality and density skew. A novel aspect of EmptyHeaded is that it automatically copes with both density and cardinality skew through optimizers that select among different data layouts and intersection algorithms to maximize SIMD parallelism.

Making these layout and algorithm choices is challenging, as the optimal choice depends both on characteristics of the data, such as density and cardinality, and characteristics of the query. In this section, we describe how EmptyHeaded makes such decisions while providing validation

[5]We measure density skew using the Pearson's first coefficient of skew defined as $3\sigma^{-1}(mean - mode)$ where σ is the standard deviation.

Table 3. Graph Datasets that are Used in the Experiments

Dataset	Nodes [M]	Dir. Edges [M]	Undir. Edges [M]	Density Skew	Card. Skew	Description
Google+ [38]	0.11	13.7	12.2	1.17	1.17	User network
Higgs [38]	0.4	14.9	12.5	0.23	0.46	Tweets about Higgs Boson
LiveJournal [39]	4.8	68.5	43.4	0.09	0.97	User network
Orkut [46]	3.1	117.2	117.2	0.08	1.46	User network
Patents [1]	3.8	16.5	16.5	0.09	2.22	Citation network
Twitter [34]	41.7	1,468.4	757.8	0.12	0.07	Follower network

for our decisions. We begin in Sections 4.1 and 4.2 by describing the set layouts (Section 4.1) and intersection algorithms (Section 4.2) that were tested in EmptyHeaded. This serves as necessary background for the tradeoff studies and optimizers we present in Sections 4.3 and 4.4. In Section 4.3, we explore how to use different set intersection algorithms to cope with cardinality skew in the data. Based on this exploration, we present the simple optimizer EmptyHeaded uses to select among set intersect algorithms. In Section 4.4, we explore the proper granularity at which to make layout decisions to cope with density skew. Again, based on this exploration, we present the simple layout optimizer that EmptyHeaded uses to select among set layouts and show that it is close to an unachievable optimal. Finally, in Section 4.5, we discuss how different node orderings can change the density skew (and potentially the cardinality skew[6]) of the graph. Although this is well known, we go on to validate that, with the optimizations presented in this section, EmptyHeaded is able to mitigate the effects of these orderings by achieving robust performance regardless of the ordering.

4.1 Layouts

We describe the set layouts that were tested in the EmptyHeaded engine and how their associated values are stored. These layouts are necessary to understand the SIMD set intersection algorithms presented in Section 4.2 and are at the core of our study in Section 4.4, where we use these layouts to exploit SIMD parallelism in the presence of density skew. In Section 4.4, we show that a combination of a simple 32-bit unsigned integer (`uint`) layout and a simple bit vector layout (`bitset`) yields the highest performance in our experiments. For dense data, the `bitset` layout makes it trivial to take advantage of SIMD parallelism but causes a quadratic blowup in memory usage for sparse data. The `uint` layout represents sparse data efficiently but makes extracting SIMD parallelism challenging.

In the following, we describe in detail the `bitset` layout in EmptyHeaded and three additional set layouts that we tested: `pshort`, `varint`, and `bitpacked`. The `pshort` layout groups values with a common upper 16-bit prefix together and stores each prefix only once. The `varint` and `bitpacked` layouts use difference encoding[7] for compression and have been shown to both

[6]The cardinality skew can change on symmetric queries, such as the triangle query, where node neighborhoods can be pruned to avoid duplicate results.

[7]Difference encoding encodes the difference between successive values in a sorted list of values (x_1, $\delta_2 = x_2 - x_1$, $\delta_3 = x_3 - x_2$, ...) instead of the original values (x_1, x_2, x_3, ...). The original array can be reconstructed by computing prefix

compress better and be up to an order of magnitude faster than compression tools such as LZO, Google Snappy, FastLZ, LZ4, or gzip [37]. As such, all layouts in this section (potentially) enable compression of the data, which is of interest as data compression has been shown to sometimes increase overall query performance by decreasing the memory bandwidth in graph applications [63]. Although interesting, we are not concerned with reducing memory usage—main memory sizes are consistently increasing and persistent storage is plentiful. Most importantly, some of these layouts (namely, `bitset` and `pshort`) are well suited for SIMD parallelism, enabling EmptyHeaded to potentially achieve higher computational performance in the generic worst -case optimal join algorithm. Note that we omit a description of the `uint` layout as it is just an array of sorted 32-bit unsigned integers. Finally, we conclude with a brief discussion of how associated data values are added to these set layouts.

`bitset`. The `bitset` layout stores a set of pairs (offset, bit vector). Each offset stores the index of the smallest value in the corresponding bit vector. Thus, the layout is a compromise between sparse and dense layouts. We refer to the number of bits in the bit vector as the *block size*. EmptyHeaded supports block sizes that are powers of two with a default of 256.[8] As shown, we pack the offsets contiguously, which allows us to regard the offsets as a `uint` layout; in turn, this allows EmptyHeaded to use the same algorithm to intersect the offsets as it does for the `uint` layout. An example of the `bitset` layout that contains n blocks and a sequence of offsets (o_1–o_n) and blocks (b_1–b_n) is shown below. The offsets store the start offset for values in the bit vector.

n	o_1	…	o_n	b_1	…	b_n

`pshort`. The Prefix Short (`pshort`) layout exploits the fact that values that are close to each other share a common prefix. The layout consists of partitions of values with a common 16-bit prefix. For each partition, the layout stores the common prefix and the number of values in the partition. Below we show an example of the `pshort` layout with a single partition:

$$S = \{65536, 65636, 65736\}$$

0 15	16 31	32 47	48 63	64 79
$v_1[31..16]$	length	$v_1[15..0]$	$v_2[15..0]$	$v_3[15..0]$
1	3	0	100	200

`varint`. The `varint` layout uses variable byte encoding, which is a popular technique first proposed by Thiel and Heaps in 1972 [36]. The `varint` layout encodes the differences between data values into units of bytes where the lower 7 bits store the data and the 8th-bit indicates whether the data extends to another byte or not. The decoding procedure reads bytes sequentially. If the 8th bit is 0 it outputs the data value and if the 8th bit is 1 the decoder appends the data from this byte to the output data value and moves on to the next byte. This layout is simple to implement and reasonably efficient [36]. Below we show an example of the `varint` layout:

$$S = \{0, 2, 4\} \qquad Diff = \{0, 2, 2\}$$

0 31	32 38	39	40 46	47	48 54	55
$\lvert S\rvert$	$\delta_1[6..0]$	c	$\delta_2[6..0]$	c	$\delta_3[6..0]$	c
3	0	0	2	0	2	0

sums ($x_i = x_1 + \sum_{n=2}^{i} x_n$). The benefit of this approach is that the differences are always smaller than the original values, allowing for more aggressive compression.

[8] The width of an AVX register.

bitpacked. The bitpacked layout partitions a set into blocks and compresses them individually. First, the layout determines the maximum bits of entropy of the values in each block b and then encodes each value of the block using b bits. Previous work in the literature [37] showed that this technique can be adapted to encode and decode values efficiently by packing and unpacking values at the granularity of SIMD registers rather than each value individually. Although Lemire et al. propose several variations of the layout, we chose to implement the bitpacked with the fastest encoding and decoding algorithms at the cost of a worse compression ratio. Thus, instead of computing and packing the deltas sequentially, the bitpacked layout computes deltas at the granularity of a SIMD register:

$$(\delta_5, \delta_6, \delta_7, \delta_8) = (x_5, x_6, x_7, x_8) - (x_1, x_2, x_3, x_4).$$

Next, each delta is packed to the minimum bit width of its block SIMD register at a time, rather than sequentially. In EmptyHeaded, we use one partition for the whole set. The deltas for each neighborhood are computed by starting our difference encoding from the first element in the set. For the tail of the neighborhood that does not fit in a SIMD register we use the varint encoding scheme. An example of the bitpacked layout is below.

$$S = \{0, 2, 8\}, \qquad Diff = \{0, 2, 6\}$$

0 31	32 39	40 42	43 45	46 48
$\|S\|$	bits/elem	δ_1[2..0]	δ_2[2..0]	δ_3[2..0]
3	3	0	2	6

Associated Values. Our sets need to be able to store associated values such as pointers to the next level of the trie or annotations of arbitrary types. In EmptyHeaded, the associated values for each set also use different underlying data layouts based on the type of the underlying set. For the bitset layout we store the associated values as a dense vector (where associated values are accessed based upon the data value in the set). For all the remaining layouts, we store the associated values as a sparse vector where the associated values are accessed based upon the index of the value in the set.

4.2 Intersections

We present an overview of the intersection algorithms EmptyHeaded uses for each layout. This serves as the background for our cardinality skew study and set intersection algorithm optimizer in Section 4.3. We remind the reader that the *min property* presented in Section 2.1 must hold for set intersections so that a worst-case optimal runtime can be guaranteed in EmptyHeaded.

uint $\cap$ *uint*. For the uint layout, we implemented and tested five state-of-the-art SIMD set intersections:

- **SIMDShuffling** iterates through both sets block-wise and compares blocks of values using SIMD shuffles and comparisons [29].
- **V1** iterates through the smaller set one-by-one and checks each value against a block of values in the larger set using SIMD comparisons [37].
- **V3** is similar to V1 but performs a binary search on four blocks of data in the larger set (each the size of a SIMD register) to identify potential matches [37].
- **SIMDGalloping** is similar to V1 but performs a scalar binary search in the larger set to find a block of data with a potential match and then uses SIMD comparisons [37].
- **BMiss** uses SIMD instructions to compare parts of blocks of values and filter potential matches, then uses scalar comparisons to check the full values of the partial matches [26].

For uint intersections, we found that the size of two sets being intersected may be drastically different. This is *cardinality skew*. So-called *galloping* algorithms [68] allow one to run in time proportional to the size of the smaller set, which copes with cardinality skew. However, for sets that are of similar size, galloping algorithms may have additional overhead. We empirically show this in Section 4.3.

bitset ∩ bitset. Our bitset is conceptually a two-layer structure of offsets and blocks. Offsets are stored using uint sets. Each offset determines the start of the corresponding block. To compute the intersection, we first find the common blocks between the bitsets by intersecting the offsets using a uint intersection followed by SIMD AND instructions to intersect matching blocks. In the best case, i.e., when all bits in the register are 1, a single hardware instruction computes the intersection of 256 values.

uint ∩ bitset. To compute the intersection between a uint and a bitset, we first intersect the uint values with the offsets in the bitset. We do this to check if it is possible that some value in a bitset block matches a uint value. As bitset block sizes are powers of two in EmptyHeaded , this can be accomplished by masking out the lower bits of each uint value in the comparison. This check may result in false positives, so, for each matching uint and bitset block we check whether the corresponding bitset blocks contain the uint value by probing the block. We store the result in a uint layout as the intersection of two sets can be at most as dense as the sparser set.[9] Notice that this algorithm satisfies the min property with a constant determined by the block size.

pshort ∩ pshort. The pshort intersection uses a set intersection algorithm previously proposed in the literature [60]. This algorithm depends on the range of the data and therefore does not preserve the min property, but can process more elements per cycle than the SIMDShuffling algorithm. The pshort intersection uses the ×86 STNII (String and Text processing New Instruction) comparison instruction allowing for a full comparison of 8 shorts, with a common 16-bit prefix, in one cycle. The pshort representation also enables jumps over chunks that do not share a common 16-bit prefix.

uint ∩ pshort. For the uint and pshort set intersection, we again take advantage of the STNII SIMD instruction. We compare the upper 16-bit prefixes of the values and shuffle the uint representation if there is a match. Next, we compare the lower 16 bits of each set, eight elements at a time using the STNII instruction.

varint *and bitpacked*. Developing set intersections for the varint and bitpacked types is challenging because of the complex decoding and the irregular access pattern of the set intersection. As a consequence, EmptyHeaded decodes the neighborhood into an array of integers and then uses the uint intersection algorithms when operating on a neighborhood represented in the varint or bitpacked representations.

4.3 Cardinality Skew

The skew in the degree distribution of graph data causes set intersections to operate on sets with different cardinalities. The most interesting tradeoff space for cardinality skew is exploring the performance of various uint intersection algorithms on sets of different sizes. Therefore, in this section we compare the five different SIMD algorithms for uint set intersections from Section 4.2 and present a simple optimizer to select among them based on our results.

[9]Estimating data characteristics like output cardinality a priori is a hard problem [13] and we found it is too costly to reinspect the data after each operation.

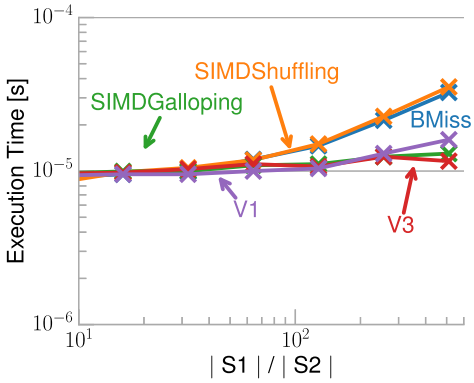


Fig. 5. Intersection time of `uint` intersection algorithms for different ratios of set cardinalities.

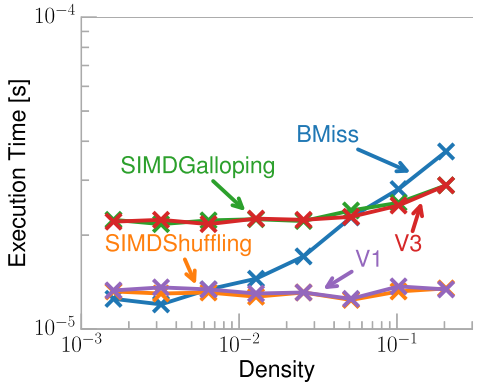


Fig. 6. Intersection time of `uint` intersection algorithms for different densities.

4.3.1 Tradeoffs. To test cardinality skew we set the range of the sets to 1M and set the cardinality of one set to 64 while changing the cardinality of the other set. Confirming the findings of others [26, 29, 37, 60], we find that SIMDGalloping and V3 algorithms outperform other intersection algorithms by more than 5× with a crossover point at a cardinality ratio of 1:32. Figure 5 shows that the SIMDGalloping and V3 algorithm outperform all other algorithms when the cardinality difference between the two sets becomes large. In contrast to the other algorithms, SIMDGalloping runs in time proportional to the size of the smaller set. Thus, SIMDGalloping is more efficient when the cardinalities of the sets are different.

We also vary the range of numbers that we place in a set from 10K to 1.2M while fixing the cardinality at 2,048. Figure 6 shows the execution time for sets of a fixed cardinality with varying ranges of numbers. BMiss is up to 5× slower when the sets have a small range and a high output cardinality. When the range of values is large and the output cardinality is small, the algorithm outperforms all other algorithms by up to 20%. Figure 6 shows that the V1 and SIMDShuflling algorithms outperform all other algorithms, by more than 2×, when the sets have a low density.

4.3.2 `uint` ∩ `uint` Algorithm Optimizer. We find that no one algorithm dominates the others, so EmptyHeaded switches dynamically between `uint` algorithms. Based on these results, we select the SIMDShuflling algorithm by default but when the ratio between the cardinality of the two sets became over 1:32, like others [26, 37], we select the SIMDGalloping algorithm. Because the sets in graph data are typically sparse, we found the impact of selecting SIMDGalloping on graph datasets to be minimal, often under a 5% total performance impact. Still we use this simple optimizer to select among these two `uint` intersection algorithms in EmptyHeaded.

4.4 Density Skew

In this section, we present our study that serves as the foundation for the layout optimizer in EmptyHeaded. To perform this study we considered all set intersection algorithm and layout combinations from Sections 4.1 and 4.2. We first show in Section 4.4.1 that using layouts and associated algorithm combinations other than those on the `uint` and `bitset` layouts resulted in no significant performance advantage. Next, in Section 4.4.2, we present our tradeoff study for making layout choices between the `uint` and `bitset` representations at three different granularities: the relation level, the set level, and the block level. We evaluate this tradeoff space and show that the

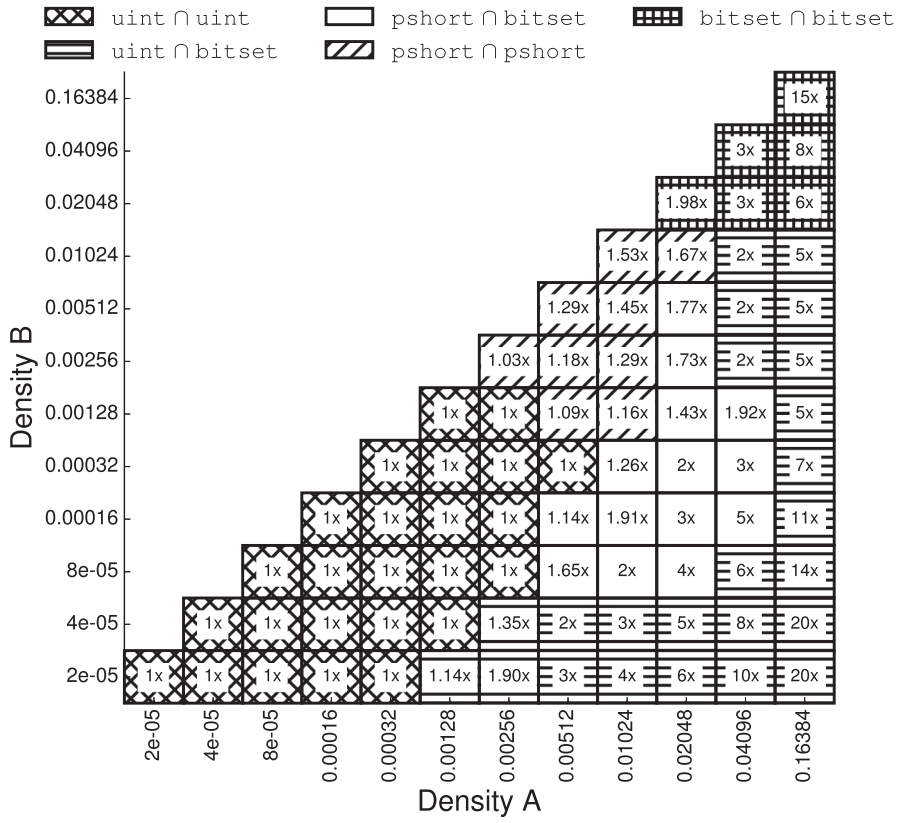


Fig. 7. Best performing layouts for set intersections with relative performance over `uint`.

set level is the best choice. We finish in Section 4.4.4 by presenting the simple set layout optimizer that EmptyHeaded uses to cope with density skew.

4.4.1 Eliminating Complexity. Figure 7 displays the best performing layout combinations and their relative performance increase compared to the best performing `uint` algorithm while changing the density of the input sets in a fixed range of 1M. Unsurprisingly, the `varint` and `bitpacked` representations never achieve the best performance. In fact, on real data, we found the `varint` and `bitpacked` types typically perform the triangle counting query 2× slower due to the decoding step not outweighing the slight memory bandwidth decrease.[10] Finally, our experiments on synthetic data show only moderate performance gains from using the `pshort` layout and on real data we found that it is rarely a good choice for a set in combination with other representations. Even worse, these layouts are all expensive to build when compared with the simple `uint` layout (see Table 4). Based on these results, we focus on only exploiting the tradeoffs for the `uint` and `bitset` intersections and layouts for the remainder of this section.

[10]The most we were able to compress real (randomly ordered) graphs with the `varint` and `bitpacked` layouts was close to 3×, therefore using 11.54 bits per edge (32 bits per edge is default with the `uint` layout). Still, on average these layouts use 19.75 bits per edge and could be as high as 28.50 bits per edge. Even worse, compressing real graphs provides no guarantee on the reduction in conflict cache misses due to random memory accesses (the way to decrease memory bandwidth) for complex join queries such as the triangle query.

Table 4. Construction Times in Seconds

Layout	Patents	LiveJournal	Higgs	Orkut	Google+
`uint`	0.93	1.71	0.36	3.16	0.30
`pshort`	0.99	1.89	0.37	3.65	0.30
`bitpacked`	1.81	2.90	0.52	4.46	0.36
`varint`	2.15	4.86	1.25	11.76	1.12

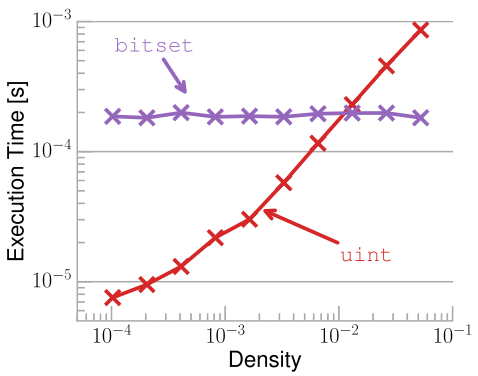


Fig. 8. Intersection time of `uint` and `bitset` layouts for different densities.

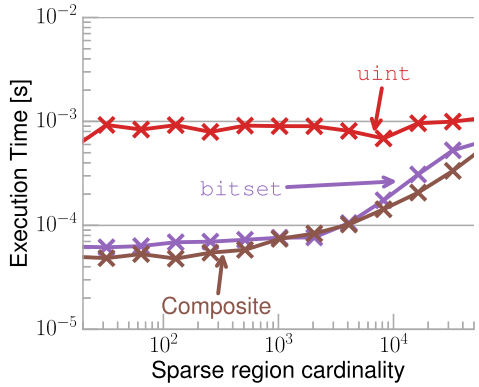


Fig. 9. Intersection time of layouts for sets with different densities in a region.

4.4.2 Tradeoffs. We study the tradeoff space of optimizing for density skew by comparing the performance of set representation decisions between the `uint` and `bitset` layouts in our trie data structure at three levels: the relation level, the set level, and the block level.

Relation Level. Set layout decisions at the relation level force the data in all relations to be stored using the same layout and therefore do not address density skew. The simplest layout in memory is to store all sets in every trie using the `uint` layout. Unfortunately, it is difficult to fully exploit SIMD parallelism using this layout, as only four elements fit in a single SIMD register.[11] In contrast, the `bitset` layout can store up to 256 elements in a single SIMD register. However, the `bitset` layout is inefficient on sparse data and can result in a quadratic blowup of memory usage. Therefore, one would expect `uint` to be well suited for sparse sets and `bitset` for dense sets. Figure 8 illustrates this trend. Because of the sparsity in real-world data, we found that `uint` provides the best performance at the relation level.

Set Level. Real-world data often has a large amount of density skew, so both the `uint` and `bitset` layouts are useful. At the set level we simply decide on a per-set level if the entire set should be represented using a `uint` or a `bitset` layout. Furthermore, we found that our `uint` and `bitset` intersection can provide up to a 6× performance increase over the best homogeneous `uint` intersection and a 132× increase over a homogeneous `bitset` intersection. We show in Sections 5.3 and 4.4.3 that the impact of mixing layouts at the set level on real data can increase overall query performance by over an order of magnitude.

[11] In the Intel Ivy Bridge architecture only SSE instructions contain integer comparison mechanisms; therefore, we are forced to restrict ourselves to a 128-bit register width.

Table 5. Relative Time of the Level Optimizers on Triangle Counting Compared to the Oracle

Dataset	Relation Level	Set Level	Block Level
Google+	7.3×	1.1×	3.2×
Higgs	1.6×	1.4×	2.4×
LiveJournal	1.3×	1.4×	2.0×
Orkut	1.4×	1.4×	2.0×
Patents	1.2×	1.6×	1.9×

Block Level. Selecting a layout at the set level might be too coarse if there is internal skew. For example, set level layout decisions are too coarse-grained to optimally exploit a set with a large sparse region followed by a dense region. Ideally, we would like to treat dense regions separately from sparse ones. To deal with skew at a finer granularity, we propose a *composite set* layout that regards the domain as a series of fixed-sized blocks; we represent sparse blocks using the `uint` layout in a single `uint` region and dense blocks using the `bitset` layout in a single `bitset` region. Conceptually this is similar to our `bitset` layout, but now when encoding a set using the composite set, the system checks the density of each block (again with a default size of 256) and decides in which region to store it. To benchmark the performance of our composite type, we generate sets with internal skew by having two regions: (1) a region with a fixed range of 5M; we change the density of this region by varying the cardinality from 16 to 131K; and (2) a dense region with 500K consecutive values. We show in Figure 9 that our composite layout outperforms the `bitset` by up to 2× when the sparse region is sparse. As the sparse region gets denser, the performance gap between the `bitset` and composite layouts increases. The `uint` type is not competitive in the range of data we present because the dense region is best represented using the `bitset` representation.

4.4.3 Evaluation. We introduce the concept of an oracle optimizer to properly evaluate our representation decisions by providing a lower bound for our overall query runtime. We use a comparison to the oracle optimizer and a study of the overheads associated with making decisions at the set and block level to validate that EmptyHeaded should make decisions between the `uint` and `bitset` layouts at the set level. We finish by presenting our set level optimizer which is used for all the experiments in Sections 5 and 6.

Oracle Optimizer. The oracle optimizer provides a lower bound baseline to evaluate our system's performance at different granularities. The performance of the oracle is not achievable in practice because the oracle is allowed to choose any representation and intersection combination while assuming perfect knowledge of the cost of each intersection. We implement the oracle optimizer by sweeping the space of all representation and algorithm combinations that EmptyHeaded considers while only counting the cost of the fastest combination for each intersection.

To determine if our system should make representation decisions at a graph, set, or block level we compare each approach on the triangle counting query to the time of the oracle optimizer. We found that on real graph data choosing representations at a set level provided the best overall performance. Table 5 demonstrates that choosing at the set level is at most 1.6× off the optimal performance. Choosing at the graph and block levels can be up to 7.3× and 3.2× slower than the oracle, respectively. Representation decisions at the graph level do not optimize at all for density skew and therefore are the least robust across graph datasets. Representation decisions at the block level are fine-grained and compensate for density skew but are too fine-grained on the graph datasets we consider and do not outweigh their increased overhead. Real graph data often has

Table 6. Set Level and Block Level Optimizer Overheads on Triangle Counting

Dataset	Set Optimizer	Block Optimizer
Google+	4%	5%
Higgs	1%	6%
LiveJournal	4%	12%
Orkut	3%	8%
Patents	10%	24%

a high density skew across sets making the middle level set optimizer perform the most robust across the graph datasets we consider.

Optimizer Overhead. Overhead is unavoidable when making fine-grained representation decisions at the set level or block level. At the set level we must incur an extra conditional check on the type of the set before performing any operation over the set. At the block level we must call four set intersection functions (the cross product of types in our composite type) and merge the final `uint` outputs into a single array to maintain the property that this is a sorted set representation. A natural question to ask is: are these overheads substantial on real graph data and real queries?

To evaluate the overhead of optimizers at the block and set levels, we modify both optimizers to always pick the `uint` representation and compare the execution time for these optimizers to graph level selection of a `uint` (no overhead). Table 6 shows the relative overhead of both optimizers across different datasets on the triangle counting query. The overhead ranges from 1% to 10% of the total runtime for our set level optimizer and from 5% to 25% for our block level optimizer. The amount of overhead we pay for each dataset is linked to its size and density skew as these are the two factors that can amortize this overhead. For example, the small Patents dataset with a low density skew of 0.09 consistently has the highest overhead at each level. The block level optimizer overhead is more pronounced on graph data due to the fact that the majority of sets in a graph are extremely sparse or extremely dense. Thus, the sets do not contain a high enough level of internal skew to outweigh the cost making such fine-grained decisions when compared to the set level optimizer. Thus, set level representations come at a lower cost than block level decisions and enable larger performance gains than both the relation level and block level decisions.

4.4.4 Set Layout Optimizer. Based on the results of our tradeoff study, we present the simple optimizer used in the EmptyHeaded engine to automatically select between the `bitset` and `uint` at the set level. The set optimizer in EmptyHeaded selects the layout for a set in isolation based on its cardinality and range. It selects the `bitset` layout when each value in the set consumes at most as much space as a SIMD (AVX) register and the `uint` layout otherwise. The optimizer uses the `bitset` layout with a block size equal to the range of the data in the set. We find this to be more effective than a fixed block size since it lacks the overhead of storing multiple offsets which is not necessary when the entire set is stored as a `bitset`. Our simple optimizer is shown in Algorithm 3.

4.5 The Impact of Node Ordering

As is standard, EmptyHeaded encodes the nodes of a graph as unique unsigned integers using the dictionary encoding technique presented in Section 2.2. Interestingly, a side effect of this technique is that the order in which nodes are encoded can affect both the cardinality skew and density skew of the processed data. Therefore, it is important to consider the node ordering used during

dictionary encoding, as this can change the heuristics used by the optimizers in Sections 4.3 and 4.4. We provide an overview of dictionary encoding and node ordering in Section 4.5.1, and an evaluation of its impact in Section 4.5.2.

ALGORITHM 3: Set layout optimizer

```
def get_layout_type(S):
  inverse_density = S.range/|S|
  if inverse_density < SIMD_register_size:
    return bitset
  else:
    return uint
```

4.5.1 Overview. Dictionary encoding is a common technique used in online analytical processing (OLAP) and graph engines to compress the columns of a relation. Graph analytic engines use the same technique on edge relations and refer to the assignment of entries in the dictionary as node ordering. Formally, there is a mapping $\pi(v)$ that assigns each node v a unique integer, which implicitly orders the nodes. The choice of $\pi(v)$ affects the internal skew of the sets in a graph. Node ordering may impact the performance of a class of symmetrical queries like triangle without selections [15, 59]. The key idea of this method is that given an undirected graph, one creates a new directed graph that preserves the number of triangles by defining a unique orientation of the edges. This new directed graph has the property that no node has a degree more than $\sqrt{N}$ where N is the number of edges in the original graph. Node ordering in combination with this symmetry technique affects the density and cardinality skew of the data. Because EmptyHeaded maps each node to an integer value, it is natural to consider the performance implications of these mappings.

4.5.2 Evaluation. We explore the impact of node ordering on query performance using triangle counting query on synthetically generated power-law graphs with different power-law exponents. We generate the data using the Snap Random Power-Law graph generator [40] and vary the Power-Law degree exponents from 1 to 3. We find that the best ordering can achieve over an order of magnitude better performance than the worst ordering on symmetrical queries such as triangle counting.

We consider the following orderings:

- **Random:** random ordering of vertices. We use this as a baseline to measure the impact of the different orderings.
- **BFS:** labels the nodes in breadth-first order.
- **Strong-Runs** first sorts the node by degree and then starting from the highest degree node, the algorithm assigns continuous numbers to the neighbors of each node. This ordering can be seen as an approximation of breadth-first search (BFS).
- **Degree:** this ordering is a simple ordering by descending degree which is widely used in existing graph systems.
- **Rev-Degree** labels the nodes by ascending degree.
- **Shingle:** an ordering scheme based on the similarity of neighborhoods [16].

In addition to these orderings, we propose a hybrid ordering algorithm `hybrid` that first labels nodes using BFS followed by sorting by descending degree. Nodes with equal degree retain their BFS ordering with respect to each other. The `hybrid` ordering is inspired by our findings that ordering by degree and BFS provided the highest performance on symmetrical queries. Figure 10

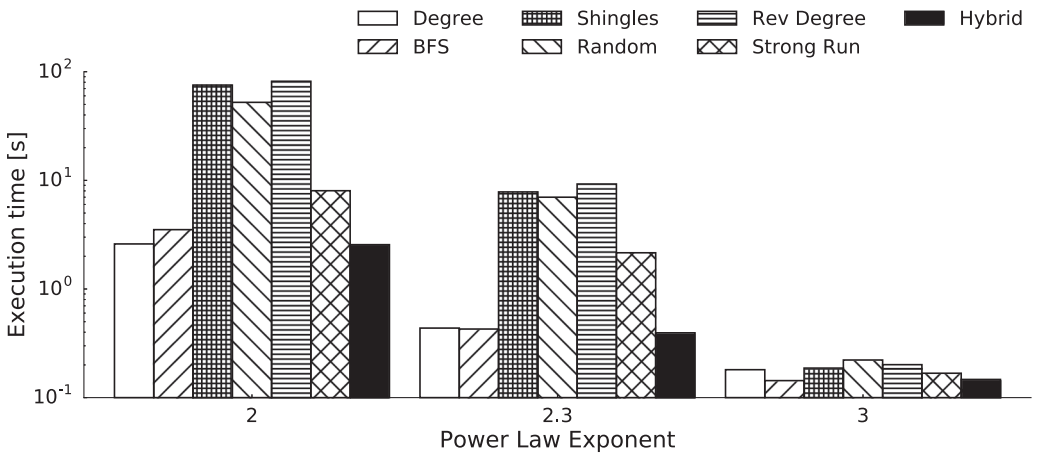


Fig. 10. Effect of data ordering on triangle counting with synthetic data.

Table 7. Node Ordering Times in Seconds on Two Popular Graph Datasets

Ordering	Higgs	LiveJournal
Shingles	1.67	9.14
`hybrid`	3.77	24.41
BFS	2.42	15.80
Degree	1.43	9.93
Reverse Degree	1.40	8.47
Strong Run	2.69	21.67

shows that graphs with a low power-law coefficient achieve the best performance through ordering by degree and that a BFS ordering works best on graphs with a high power-law coefficient. Figure 10 shows the performance of `hybrid` ordering and how it tracks the performance of BFS or degree where each is optimal. We find this ordering to be the most robust ordering for symmetrical queries.

Each ordering incurs the cost of performing the actual ordering of the data. Table 7 shows examples of node ordering times in EmptyHeaded . The execution time of the BFS ordering grows linearly with the number of edges, while sorting by degree and reverse degree depend on the number of nodes. The cost of the `hybrid` ordering is the sum of the costs of the BFS ordering and ordering by degree.

In total, we found that, although considering node orderings is an interesting problem, the optimizers to select among intersections and layouts presented in this section often mitigated the effects of node ordering. For example, Table 8 shows that a random ordering in EmptyHeaded often outperforms a compressed sparse row (CSR) layout that is sorted by degree. As such, we found that considering node orderings is not as crucial of a problem in EmptyHeaded as it is in other graph engines which do not implement our optimizations.

5 EXPERIMENTS

We compare EmptyHeaded against state-of-the-art high- and low-level specialized graph engines on standard graph benchmarks. Additionally, we compare EmptyHeaded to several state-of-the-art

Table 8. Slow Down of a Random Ordering to a CSR Layout Sorted by Degree for the Triangle Counting Query

	Default		Symmetry Breaking	
Dataset	`uint`	EmptyHeaded	`uint`	EmptyHeaded
Google+	1.8×	0.9×	1.0×	0.5×
Higgs	3.0×	2.0×	0.9×	0.6×
LiveJournal	1.7×	1.8×	1.2×	1.2×
Orkut	1.4×	1.5×	1.1×	1.1×
Patents	1.9×	1.8×	1.2×	1.3×

relational database engines which use a pairwise join algorithm on a standard graph benchmark. We show that by using our optimizations from Section 3 and Section 4, EmptyHeaded is able to compete with specialized graph engines while outperforming pairwise relational database engines on standard graph benchmarks.

5.1 Experiment Setup

We describe the datasets, comparison engines, metrics, and experiment setting used to validate that EmptyHeaded competes with specialized engines in Sections 5.2, 5.3, and 5.5.

5.1.1 Datasets. Table 3 provides a list of the six popular datasets that we use in our comparison to other graph analytics engines. LiveJournal, Orkut, and Patents are graphs with a low amount of density skew, and Patents is a much smaller graph in comparison to the others. Twitter is one of the largest publicly available datasets and is a standard benchmarking dataset that contains a modest amount of density skew. Higgs is a medium-sized graph with a modest amount of density skew. Google+ is a graph with a large amount of density skew.

5.1.2 Comparison Engines. We compare EmptyHeaded against popular high- and low-level engines in the graph domain. We also compare to the high-level LogicBlox engine on each query, as it is the first commercial database with a worst-case optimal join optimizer. Finally, we also compare EmptyHeaded to several popular, general-purpose, and high-level relational pairwise join databases.

Low-Level Engines. We benchmark several graph analytic engines and compare their performance. The engines that we compare to are PowerGraph v2.2 [20], the latest release of commercial graph tool (CGT-X), and Snap-R [40]. Each system provides highly optimized shared-memory implementations of the triangle counting query. Other shared-memory graph engines such as Ligra [62] and Galois [52] do not provide optimized implementations of the triangle query and requires one to write queries by hand. We do provide a comparison to Galois v2.2.1 on PageRank and SSSP. Galois has been shown to achieve performance similar to that of Intel's hand-coded implementations [58] on these queries.

High-Level Engines. We compare to LogicBlox v4.3.4 on all queries since LogicBlox is the first general-purpose commercial engine to provide similar worst-case optimal join guarantees. LogicBlox also provides a relational model that makes complex queries easy and succinct to express. It is important to note that LogicBlox is a full-featured commercial system (supports transactions, updates, etc.) and therefore incurs inefficiencies that EmptyHeaded does not. Regardless, we demonstrate that using GHDs as the intermediate representation in EmptyHeaded's query compiler not only provides tighter theoretical guarantees, but provides more than a three orders of magnitude performance improvement over LogicBlox. We further demonstrate that our set layouts account

for over an order of magnitude performance advantage over the LogicBlox design. We also compare to SociaLite [61] on each query as it also provides high-level language optimizers, making the queries as succinct and easy to express as in EmptyHeaded. Unlike LogicBlox, SociaLite does not use a worst-case optimal join optimizer and therefore suffers large performance gaps on graph pattern queries. Our experimental setup of the LogicBlox and SociaLite engines was verified by an engineer from each system and our results are in line with previous findings [53, 58, 61].

Pairwise Join Engines. We compare to HyPer v0.5.0 [30] as HyPer is a state-of-the-art in-memory relational database management system (RDBMS). HyPeR is a high-performance engine that outperforms most other popular database engines on commodity hardware, like that which is used in this article. We also compare to version 9.3.5 of PostgreSQL and MonetDB (Jul2015-SP1 release). We configured the sizes of PostgreSQL and MonetDB's buffers to be more than an order of magnitude larger than the input size in uncompressed text form. Before running the queries, we created indices and used `ANALYZE` to collect statistics (both excluded from the running time). We stored the edge relation on a RAM disk using `tablespaces` for PostgreSQL and `tmpfs` for MonetDB. These relational engines are full-featured commercial strength systems (support transactions, etc.) and therefore incur inefficiencies that EmptyHeaded does not.

Omitted Comparisons. We compared EmptyHeaded to GraphX [21] which is a graph engine designed for scale-out performance and Neo4j which is a commercial graph database engine. Both GraphX and Neo4j were consistently several orders of magnitude slower than EmptyHeaded's performance in a shared-memory setting. We also compared to a commercial column store database engine but they were consistently over three orders of magnitude off of EmptyHeaded's performance. We exclude a comparison to the Grail method [19] as this approach in a SQL Server has been shown to be comparable to or sometimes worse than PowerGraph [20] when the entire dataset can easily fit in-memory (like we consider in this article). It should be noted that the Grail approach in a persistent database has been shown to be more robust than in-memory engines, such as EmptyHeaded and PowerGraph, when the entire dataset does not fit easily in-memory [19].

5.1.3 Metrics. We measure the performance of EmptyHeaded and other engines. For end-to-end performance, we measure the wall-clock time for each system to complete each query. This measurement excludes the time used for data loading, outputting the result, data statistics collection, and index creation for all engines. We repeat each measurement seven times, eliminate the lowest and the highest value, and report the average. Between each measurement of the *low-level* engines we wipe the caches and re-load the data to avoid intermediate results that each engine might store. For the *high-level* engines we perform runs back-to-back, eliminating the first run which can be an order of magnitude worse than the remaining runs. We do not include compilation times in our measurements. Low-level graph engines run as a stand-alone program (no compilation time) and we discard the compilation time for high-level engines (by excluding their first run, which includes compilation time). Nevertheless, our unoptimized compilation process (under two seconds for all queries in this article) is often faster than other high-level engines' (Socialite or LogicBlox).

5.1.4 Experiment Setting. EmptyHeaded is an in-memory engine that runs and is evaluated on a single node server. As such, we ran all experiments on a single machine with a total of 48 cores on four Intel Xeon E5-4657L v2 CPUs and 1TB of RAM. We compiled the C++ engines (EmptyHeaded, Snap-R, PowerGraph, TripleBit) with g++ 4.9.3 (-O3) and ran the Java-based engines (CGT-X, LogicBlox, SociaLite) on OpenJDK 7u65 on Ubuntu 12.04 LTS. For all engines, we chose buffer and heap sizes that were at least an order of magnitude larger than the dataset itself to avoid garbage collection.

Table 9. Triangle Counting Runtime (in Seconds) for EmptyHeaded and Relative Slowdown for Other Engines Including PowerGraph, a Commercial Graph Tool (CGT-X), Snap-Ringo, SociaLite, and LogicBlox

		Low-Level			High-Level	
Dataset	EmptyHeaded	PowerGraph	CGT-X	Snap-Ringo	SociaLite	LogicBlox
Google+	**0.31**	8.40×	62.19×	4.18×	1390.75×	83.74×
Higgs	**0.15**	3.25×	57.96×	5.84×	387.41×	29.13×
LiveJournal	**0.48**	5.17×	3.85×	10.72×	225.97×	23.53×
Orkut	**2.36**	2.94×	-	4.09×	191.84×	19.24×
Patents	**0.14**	10.20×	7.45×	22.14×	49.12×	27.82×
Twitter	**56.81**	4.40×	-	2.22×	t/o	30.60×

48 threads used for all engines. "-" indicates the engine does not process over 70 million edges. "t/o" indicates the engine ran for over 30 minutes.

5.2 Experimental Results

We provide a comparison to specialized graph analytics engines on several standard workloads. We demonstrate that EmptyHeaded outperforms the graph analytics engines by 2 to 60× on graph pattern queries while remaining competitive on PageRank and SSSP.

5.2.1 Graph Pattern Queries. We first focus on the triangle counting query as it is a standard graph pattern benchmark with hand-tuned implementations provided in both high- and low-level engines. Furthermore, the triangle counting query is widely used in graph processing applications and is a common subgraph query pattern [45, 49]. To be fair to the low-level frameworks, we compare the triangle query only to frameworks that provide a hand-tuned implementation. Although we have a high-level optimizer, we outperform the graph analytics engines by 2 to 60× on the triangle counting query.

As is the standard, we run each engine on pruned versions of these datasets, where each undirected edge is pruned such that $src_{id} > dst_{id}$ and id's are assigned based upon the degree of the node. This process (describe more in Section 4.5) is standard as it limits the size of the intersected sets and has been shown to empirically work well [59]. Nearly every graph engine implements pruning in this fashion for the triangle query.

Takeaways. The results from this experiment are in Table 9. On very sparse datasets with low density skew (such as the Patents dataset) our performance gains are modest as it is best to represent all sets in the graph using the `uint` layout, which is what many competitor engines already do. As expected, on datasets with a larger degree of density skew, our performance gains become much more pronounced. For example, on the Google+ dataset, with a high density skew, our set level optimizer selects 41% of the neighborhood sets to be `bitsets` and achieves over an order of magnitude performance gain over representing all sets as `uints`. LogicBlox performs well in comparison to CGT-X on the Higgs dataset, which has a large amount of cardinality skew, as they use a Leapfrog Triejoin algorithm [68] that optimizes for cardinality skew by obeying the min property of set intersection. EmptyHeaded similarly obeys the min property by selecting amongst set intersection algorithms based on cardinality skew. In Section 5.3, we demonstrate that over a two orders of magnitude performance gain comes from our set layout and intersection algorithm choices.

Omitted Comparison. We do not compare to Galois on the triangle counting query, as Galois does not provide an implementation and implementing it ourselves would require us to write a custom set intersection in Galois (where >95% of the runtime goes). We describe how to implement

Table 10. Runtime for Five Iterations of PageRank (in Seconds) Using 48 Threads for All Engines

		Low-Level				High-Level	
Dataset	EmptyHeaded	Galois	PowerGraph	CGT-X	Snap-Ringo	SociaLite	LogicBlox
Google+	0.10	**0.021**	0.24	1.65	0.24	1.25	7.03
Higgs	0.08	**0.049**	0.5	2.24	0.32	1.78	7.72
LiveJournal	0.58	**0.51**	4.32	-	1.37	5.09	25.03
Orkut	0.65	**0.59**	4.48	-	1.15	17.52	75.11
Patents	**0.41**	0.78	3.12	4.45	1.06	10.42	17.86
Twitter	**15.41**	17.98	57.00	-	27.92	367.32	442.85

"-" indicates the engine does not process over 70 million edges. The other engines include Galois, PowerGraph, a Commercial Graph Tool (CGT-X), Snap-Ringo, SociaLite, and LogicBlox.

Table 11. SSSP Runtime (in Seconds) Using 48 Threads for All Engines

		Low-Level			High-Level	
Dataset	EmptyHeaded	Galois	PowerGraph	CGT-X	SociaLite	LogicBlox
Google+	0.024	**0.008**	0.22	0.51	0.27	41.81
Higgs	0.035	**0.017**	0.34	0.91	0.85	58.68
LiveJournal	0.19	**0.062**	1.80	-	3.40	102.83
Orkut	0.24	**0.079**	2.30	-	7.33	215.25
Patents	0.15	**0.054**	1.40	4.70	3.97	159.12
Twitter	7.87	**2.52**	36.90	-	x	379.16

"-" indicates the engine does not process over 70 million edges. The other engines include Galois, PowerGraph, a commercial graph tool (CGT-X), SociaLite, and LogicBlox. "x" indicates the engine did not compute the query properly.

high-performance set intersections in depth in Section 4, and EmptyHeaded's triangle counting numbers are comparable to Intel's hand-coded numbers, which are slightly (10%–20%) faster than the Galois implementation [58]. We provide a comparison to Galois on SSSP and PageRank in Section 5.2.2.

5.2.2 Graph Analytics Queries. Although EmptyHeaded is capable of expressing a variety of different workloads, we benchmark PageRank and SSSP as they are common graph benchmarks. In addition, these benchmarks illustrate the capability of EmptyHeaded to process broader workloads that relational engines typically do not process efficiently: (1) linear algebra operations (in PageRank) and (2) transitive closure (in SSSP). We run each query on undirected versions of the graph datasets and demonstrate competitive performance compared to specialized graph engines. Our results suggest that our approach is competitive outside of classic join workloads.

PageRank. As shown in Table 10, we are consistently 2–4× faster than standard low-level baselines and more than an order of magnitude faster than the high-level baselines on the PageRank query. We observe competitive performance with Galois (271 lines of code), a highly tuned shared-memory graph engine, as seen in Table 10, while expressing the query in three lines of code (Table 1). There is room for improvement on this query in EmptyHeaded since double buffering and the elimination of redundant joins would enable EmptyHeaded to achieve performance closer to the bare metal performance, which is necessary to outperform Galois.

Single-Source Shortest Paths. We compare EmptyHeaded's performance to LogicBlox and specialized engines in Table 11 for SSSP while omitting a comparison to Snap-R. Snap-R does not implement a parallel version of the algorithm and is over three orders of magnitude slower than

EmptyHeaded on this query. For our comparison, we selected the highest degree node in the undirected version of the graph as the start node. EmptyHeaded consistently outperforms PowerGraph (low-level) and SociaLite (high-level) by an order of magnitude and LogicBlox by three orders of magnitude on this query. More sophisticated implementations of SSSP than what EmptyHeaded generates exist [11]. For example, Galois, which implements such an algorithm, observes a 2–30× performance improvement over EmptyHeaded on this application (Table 11). Still, EmptyHeaded is competitive with Galois (172 lines of code) compared to the other approaches while expressing the query in two lines of code (Table 1).

5.3 Micro-Benchmarking Results

We detail the effect of our contributions on query performance. We introduce two new queries and revisit the Barbell query (introduced in Section 3) in this section: (1) K_4 is a 4-clique query representing a more complex graph pattern, (2) $L_{3,1}$ is the Lollipop query that finds all 3-cliques (triangles) with a path of length one off of one vertex, and (3) $B_{3,1}$ the Barbell query that finds all 3-cliques (triangles) connected by a path of length one. We demonstrate how using GHDs in the query compiler and the set layouts in the execution engine can have a three orders of magnitude performance impact on the K_4, $L_{3,1}$, and $B_{3,1}$ queries.

Experimental Setup. These queries represent pattern queries that would require significant effort to implement in low-level graph analytics engines. For example, the simpler triangle counting implementation is 138 lines of code in Snap-R and 402 lines of code in PowerGraph. In contrast, each query is one line of code in EmptyHeaded. As such, we do not benchmark the low-level engines on these complex pattern queries. We run `COUNT(*)` aggregate queries in this section to test the full effect of GHDs on queries with the potential for early aggregation. The K_4 query is symmetric and therefore runs on the same pruned datasets as those used in the triangle counting query in Section 5.2.1. The $B_{3,1}$ and $L_{3,1}$ queries run on the undirected versions of these datasets.

5.3.1 Query Compiler Optimizations. GHDs enable complex queries to run efficiently in EmptyHeaded . Table 12 demonstrates that when the GHD optimizations are disabled ("-GHD"), meaning a single node GHD query plan is run, we observe up to an 8× slowdown on the $L_{3,1}$ query and over a three orders of magnitude performance improvement on the $B_{3,1}$ query. Interestingly, density skew matters again here, and for the dataset with the largest amount of density skew, Google+, EmptyHeaded observes the largest performance gain. GHDs enable early aggregation here and thus eliminate a large amount of computation on the datasets with large output cardinalities (high density skew). LogicBlox, which currently uses only the generic worst-case optimal join algorithm (no GHD optimizations) in their query compiler, is unable to complete the Lollipop or Barbell queries across the datasets that we tested. GHD optimizations do not matter on the K_4 query as the optimal query plan is a single node GHD.

5.3.2 Execution Engine Optimizations. Table 12 shows the relative time to complete graph queries with features of our engine disabled. The "-R" column represents EmptyHeaded without SIMD set layout optimizations and therefore density skew optimizations. This most closely resembles the implementation of the low-level engines in Table 9, who do not consider mixing SIMD friendly layouts. Table 12 shows that our set layout optimizations consistently have a two orders of magnitude performance impact on advanced graph queries. The "-RA" column shows EmptyHeaded without density skew (SIMD layout choices) and cardinality skew (SIMD set intersection algorithm choices). Our layout and algorithm optimizations provide the largest performance advantage (>20×) on extremely dense (`bitset`) and extremely sparse (`uint`) set intersections , which is what happens on the datasets with low density skew here. Our contribution here is the mixing

Table 12. 4-Clique (K_4), Lollipop ($L_{3,1}$), and Barbell ($B_{3,1}$) runtime in seconds for EmptyHeaded (EH) and relative runtime for SociaLite, LogicBlox, and EmptyHeaded while disabling features. "t/o" indicates the engine ran for over 30 minutes. "-R" is EmptyHeaded without layout representation optimizations. "-RA" is EmptyHeaded without both layout representation (density skew) and intersection algorithm (cardinality skew) optimizations. "-GHD" is EmptyHeaded without GHD optimizations (single-node GHD).

			EHw/o Optimizations			Other Engines	
Dataset	Query	EH	-R	-RA	-GHD	SociaLite	LogicBlox
Google+	K_4	4.12	10.01×	10.01×	-	t/o	t/o
	$L_{3,1}$	3.11	1.05×	1.10×	8.93×	t/o	t/o
	$B_{3,1}$	3.17	1.05×	1.14×	t/o	t/o	t/o
Higgs	K_4	0.66	3.10×	10.69×	-	666×	50.88×
	$L_{3,1}$	0.93	1.97×	7.78×	1.28×	t/o	t/o
	$B_{3,1}$	0.95	2.53×	11.79×	t/o	t/o	t/o
LiveJournal	K_4	2.40	36.94×	183.15×	-	t/o	141.13×
	$L_{3,1}$	1.64	45.30×	176.14×	1.26×	t/o	t/o
	$B_{3,1}$	1.67	88.03×	344.90×	t/o	t/o	t/o
Orkut	K_4	7.65	8.09×	162.13×	-	t/o	49.76×
	$L_{3,1}$	8.79	2.52×	24.67×	1.09×	t/o	t/o
	$B_{3,1}$	8.87	3.99×	47.81×	t/o	t/o	t/o
Patents	K_4	0.25	328.77×	1021.77×	-	20.05×	21.77×
	$L_{3,1}$	0.46	104.42×	575.83×	0.99×	318×	62.23×
	$B_{3,1}$	0.48	200.72×	1105.73×	t/o	t/o	t/o

Table 13. Relative Time When Disabling Features on the Triangle Counting Query

Dataset	-SIMD	-Representation	-SIMD & Representation
Google+	1.0×	3.0×	7.5×
Higgs	1.5×	3.9×	4.8×
LiveJournal	1.6×	1.0×	1.6×
Orkut	1.8×	1.1×	2.0×
Patents	1.3×	0.9×	1.1×

"-SIMD" is EmptyHeaded without SIMD. "-Representation" is EmptyHeaded using `uint` at the graph level.

of data representations ("-R") and set intersection algorithms ("-RA"), both of which are deeply intertwined with SIMD parallelism. In total, Table 12 and our discussion validate that the set layout and algorithmic features have merit and enable EmptyHeaded to compete with graph engines.

Table 13 shows the relative time to complete the triangle query features of our system disabled on unpruned data. The "-SR" column is the one that most closely resembles the implementation of the graph analytics competitors we compare to in Table 9. Our use of SIMD instructions can enable up to a 1.8× performance increase and our use of representations can enable up to a 3.9× performance increase. The amount of SIMD parallelism leveraged is highly intertwined with our representation decisions. Therefore, we notice this 3.9× performance decrease on datasets with high density skew when we solely use a `uint` representation ("-R") because the amount of available SIMD parallelism is decreased significantly. Finally, we also tested removing the decision between SIMD galloping and the SIMDShuffling algorithm for the `uint` intersections but found this

Table 14. Triangle Counting Runtime (in Seconds) for EmptyHeaded and Relative Slowdown for Other Engines Including LogicBlox, HyPer, MonetDB, and PostgreSQL

	Dataset					
Engine	Google+	Higgs	LiveJournal	Orkut	Patents	Twitter
EmptyHeaded	**0.31**	**0.15**	**0.48**	**2.36**	**0.14**	**56.81**
LogicBlox	83.74×	29.13×	23.53×	19.24×	27.82×	30.60×
HyPer	129.93×	12.61×	8.21×	8.67×	7.60×	13.56×
MonetDB	253.98×	154.67×	71.88×	49.58×	47.86×	t/o
PostgreSQL	t/o	8144×	t/o	t/o	2027×	t/o

48 threads used for all engines. "t/o" indicates the engine ran for over 30 minutes.

feature had only a 10% performance impact on overall query runtime. In total, Table 13 shows our vectorization and representation features have merit and are needed to attain optimal performance on graph queries over skewed data.

5.4 Pairwise Relational Comparison Results

Table 14 serves as a comparison of the worst-case optimal join algorithms in EmptyHeaded and the pairwise join algorithms present in popular and state-of-the-art database engines on the triangle counting query. We present a comparison to HyPer, MonetDB, and PostgreSQL on only the triangle counting query for two reasons: (1) expressing PageRank and SSSP in these engines is difficult and (2) when we benchmarked the other queries their performance difference with EmptyHeaded got even larger (several orders of magnitude slower). Interestingly, Table 14 shows that the worst-case optimal join algorithm on its own is not always enough to outperform existing state-of-the-art implementations. For example, on the triangle query HyPer outperforms LogicBlox on all datasets except the Google+ dataset which has high density skew. On the other hand, EmptyHeaded consistently outperforms HyPer on all datasets by 7.6×–129.9×, with the largest performance difference occurring on the Google+ dataset. MonetDB and PostgreSQL were at least an order of magnitude slower than EmptyHeaded across datasets and were often more than two orders of magnitude slower.

5.5 Selections

To test our implementation of selections in EmptyHeaded, we ran two graph pattern queries that contained selections. The first is a 4-clique selection query where we find all 4-cliques connected to a specified node. The second is a barbell selection query where we find all pairs of 3-cliques connected to a specified node. The syntax for each query in EmptyHeaded is shown in Table 1.

We run `COUNT(*)` versions of the queries here again as materializing the output for these queries is prohibitively expensive. We did materialize the output for these queries on a couple of datasets and noticed our performance gap with the competitors was still the same. We varied the selectivity for each query by changing the degree of the node we selected. We tested this on both high and low degree nodes.

The results of our experiments are in Table 15. Pushing down selections across GHDs can enable over a four order of magnitude performance improvement on these queries and is essential to enable peak performance. As shown in Table 15, the competitors are closer to EmptyHeaded when the output cardinality is low but EmptyHeaded still outperforms the competitors. For example, on the 4-clique selection query on the patents dataset the query contains no output but we still outperform LogicBlox by 3.66× and SociaLite by 5,754×. The data layouts we choose for the sets

Table 15. 4-Clique Selection (SK_4) and Barbell Selection ($SB_{3,1}$) Runtime in Seconds for EmptyHeaded and Relative Runtime for SociaLite, LogicBlox, and EmptyHeaded While Disabling Optimizations

Dataset	Query	$\|Out\|$	EmptyHeaded	-GHD	SociaLite	LogicBlox
Google+	SK_4	1.5E+11	154.24	6.09×	t/o	t/o
		5.5E+7	1.08	865.95×	t/o	50.91×
	$SB_{3,1}$	4.0E+17	0.92	3.22×	t/o	t/o
		2.5E+3	0.008	351.72×	t/o	t/o
Higgs	SK_4	2.2E+7	1.92	14.48×	t/o	58.10×
		2.7E+7	2.91	9.50×	t/o	52.44×
	$SB_{3,1}$	1.7E+12	0.060	17.36×	t/o	t/o
		2.4E+12	0.070	14.88×	t/o	t/o
LiveJournal	SK_4	1.7E+7	6.73	18.05×	t/o	14.83×
		5.1E+2	0.0095	13E3×	t/o	10.46×
	$SB_{3,1}$	1.6E+12	0.27	6.47×	t/o	t/o
		9.9E+4	0.0062	278.16×	t/o	70.23×
Orkut	SK_4	9.8E+8	208.20	1.26×	t/o	t/o
		2.8E+5	0.020	13E+3×	t/o	18.79×
	$SB_{3,1}$	1.1E+15	3.24	3.20×	t/o	t/o
		2.2E+8	0.0072	1,314×	21E+3×	23E+3×
Patents	SK_4	0	0.011	121.70×	5,754×	3.66×
		9.2E+3	0.011	117.56×	5,572×	10.72×
	$SB_{3,1}$	1.6E+1	0.0060	77.82×	223.29×	15.17x
		1.1E+7	0.0066	71.22×	1,073×	3,296×

"$|Out|$" indicates the output cardinality. "t/o" indicates the engine ran for over 30 minutes. "-GHD" is EmptyHeaded without pushing down selections across GHD nodes.

matter here as placing the selected attributes first in Algorithm 1, causes these attributes to appear in the first levels of the trie which are often dense and can be represented using a bitset (see Section 4.1). For equality selections this enables us to perform the actual selection in constant time versus a binary search in an unsigned integer array.

6 EXTENSIONS

In this section, we show that the generic EmptyHeaded design can easily be extended to accommodate RDF workloads, which are typically processed in specialized RDF processing engines. Our goal here is to reexamine the performance difference between specialized RDF engines and general-purpose relational engines with this our new multiway join engine. RDF queries are typically complex join queries and specialized RDF engines largely support a high-level query language called SPARQL. We validate that worst-case optimal join algorithms have merit here and can serve as an improvement over the traditional querying processing mechanisms for RDF workloads. We first begin with an overview of RDF queries. Next, we describe the pipelining optimization, which is a general optimization necessary for EmptyHeaded to compete with specialized RDF engines. Finally, we end with a comparison of EmptyHeaded to specialized RDF processing engines as well as MonetDB and LogicBlox.

6.1 RDF Overview

The volume of RDF data from the Semantic Web has grown exponentially in the past decade [48, 71]. RDF data is a collection of Subject-Predicate-Object triples that form a complex and massive

graph that traditional query mechanisms do not handle efficiently [33, 48]. As a result, there has been significant interest in designing specialized engines for RDF processing [8, 32, 48, 71]. These specialized engines accept the SPARQL query language and build several indexes (>10) over the Subject-Predicate-Object triples to process RDF workloads efficiently [48, 71]. In contrast, the natural way of storing RDF data in a traditional relational engine is to use triple tables [48] or vertically partitioned column stores [2], but these techniques can be three orders of magnitude slower than specialized RDF engines [48].

6.2 Pipelining Optimization

Pipelining is a classic query optimization used to reduce the size of materialized intermediate results in a query plan [27]. We add a simple rule such that the root node of a GHD can be pipelined with one child node for RDF workloads in EmptyHeaded :

Definition 6.1. Given a GHD T, we say $(t_0, t_1) \in V(T) \times V(T)$ are *pipelineable* if $t_0 \neq t_1$ and $\chi(t_0) \cap \chi(t_1)$ is a prefix of the trie orders for both t_0 and t_1.

Example 6.2. Consider the following query pattern from LUBM query 8 over relation `R` with attributes `(x,y)` and relation `S` with attributes `(x,z)`:

$$OUT(x, y, z) : -R(x, y), S(x, z).$$

The GHD EmptyHeaded produced for this query contains two nodes with respective ordered attributes `[x,y]` (root t_0) and `[x,z]` (child t_1). By definition this GHD is pipelinable as the nodes share the common prefix '`x`'.

6.3 RDF Experiments

We benchmark a standard relational engine, two worst-case optimal join engines, and two state-of-the-art specialized RDF engines on the LUBM benchmark. We select MonetDB as the classical relational data processing engine baseline, LogicBlox and EmptyHeaded as the worst-case optimal engine baselines, and RDF-3X and TripleBit as the specialized RDF engine baselines. Our comparison shows that EmptyHeaded and LogicBlox's designs outperform all other engines on cyclic queries, where, again, pairwise joins are suboptimal. On the remaining queries, we show how EmptyHeaded remains competitive with the specialized RDF engines due to the layout, pushing down selections, and pipelining optimizations presented in this article.

6.3.1 LUBM Benchmark. The LUBM benchmark is a standard RDF benchmark with a synthetic data generator [24]. The data generator produces RDF data representing a university system ontology. We generated 133 million triples for the comparisons in this section. The LUBM benchmark contains complex multiway star join patterns as well as two cyclic queries with triangle patterns. We run the complete LUBM benchmark while removing the inference step for each query. This is standard in benchmarking comparisons [8, 71]. We omit queries 6 and 10, since without the inference step, they correspond to other queries in the benchmark. The syntax to run each query in this section in EmptyHeaded is shown in Table 16.

6.3.2 Comparison Engines. We describe the specialized RDF engines (TripleBit and RDF-3X) and relational engines (MonetDB and LogicBlox) with which we compare.

RDF Engines. We compare against RDF-3X v0.3.8 and TripleBit, two high-performance shared-memory RDF engines. TripleBit [71] and RDF-3X [48] have been shown to consistently outperform traditional column and row store databases [48, 71]. RDF-3X is a popular and established RDF engine which performs well across a variety of SPARQL queries. RDF-3X builds a full set of permutations on all triples and uses selectivity estimates to choose the best join order. For fairness, it should

Table 16. Example LUBM RDF Query Patterns in EmptyHeaded

Name	Query Syntax
Q1	`out(x) :- takesCourse(x,`Univ0Course0'), type(x,`GraduateStudent').`
Q2	`out(x,y,z) :- memberOf(x,y),subOrganizationOf(y,z), undergraduateDegreeFrom(x,z), type(x,`GraduateStudent'),type(y,`Department'), type(z,`University').`
Q3	`out(x) :- type(x,`Publication'), publicationAuthor(x,`Univ0AsstProf0').`
Q4	`out(x,y,z,w) :- worksFor(x,`Univ0Dept0'), name(x,y),emailAddress(x,w),telephone(x,z), type(x,`AssociateProfessor').`
Q5	`out(x) :- type(x,`UndergraduateStudent'), memberOf(x,`University0').`
Q7	`out(x,y) :- teacherOf(`Univ0AsstProf0',x), takesCourse(y,x),type(x,`Course'), type(y,`UndergraduateStudent').`
Q8	`out(x,y,z) :- memberOf(x,y),emailAddress(x,z), type(x,`UndergraduateStudent'), subOrganizationOf(y,`University0'), type(y,`Department').`
Q9	`out(x,y,z) :- type(x,`UndergraduateStudent'),type(y,`Course'), type(z,`AssistantProfessor'), advisor(x,z),teacherOf(z,y),takesCourse(x,y).`
Q11	`out(x) :- type(x,`ResearchGroup'), subOrganizationOf(x,`University0').`
Q12	`out(x,y) :- worksFor(y,x),type(y,`FullProfessor'), subOrganizationOf(x,`University0'), type(x,`Department').`
Q13	`out(x) :- type(x,`GraduateStudent'), undergraduateDegreeFrom(x,`University567').`
Q14	`out(x) :- type(x,`UndergraduateStudent').`

Note that the equality constraint constants have been simplified here for readability (e.g., 'University567' is actually 'http://www.University567.edu').

be noted that RDF-3X uses data structures and operators optimized for disk-based databases. This can lead to a performance overhead when compared to in-memory engines like EmptyHeaded, even in the warm-cache scenarios we consider in this section. TripleBit [71] is a more recent RDF engine which uses a sophisticated matrix representation and has been shown to compete with and often outperform RDF-3X on a range of RDF queries on larger scale data. TripleBit reduces the

Table 17. Runtime in Milliseconds for Best Performing System and Relative Runtime for Each Engine on the LUBM Benchmark with 133 Million Triples

Query	Best	EmptyHeaded	TripleBit	RDF-3X	MonetDB	LogicBlox
Q1	4.00	1.51×	3.45×	**1.00**×	174.58×	8.62×
Q2	973.95	**1.00**×	2.38×	1.92×	8.79×	1.52×
Q3	0.47	**1.00**×	92.61×	8.44×	283.37×	83.41×
Q4	3.39	4.62×	**1.00**×	1.77×	2093.78×	116.32×
Q5	0.44	**1.00**×	99.21×	9.15×	303.11×	81.44×
Q7	6.00	3.18×	8.53×	**1.00**×	573.33×	6.52×
Q8	78.50	9.83×	**1.00**×	3.07×	206.62×	5.03×
Q9	581.37	**1.00**×	3.53×	6.63×	24.29×	1.35×
Q11	0.45	**1.00**×	6.07×	11.03×	58.63×	73.76×
Q12	3.05	2.22×	**1.00**×	7.86×	118.94×	50.23×
Q13	0.87	**1.00**×	48.90×	35.49×	86.18×	102.77×
Q14	3.00	1.90×	54.47×	**1.00**×	313.47×	325.02×

size of the data and indexes through two auxiliary data structures to minimize the cost of index selection during query evaluation. Both engines generate optimal join orderings.

Relational Engines. We also provide comparisons to MonetDB (Jul2015-SP1 release) and LogicBlox v4.3.4, which are two general-purpose relational engines. MonetDB is a popular open source column store database whose performance has been shown to outperform row store designs, such as PostgreSQL, by orders of magnitude on RDF workloads [48]. For all relational engines, including EmptyHeaded , we store and process the RDF data in a vertically partitioned manner as this has been shown to be superior to storing the data as triples [2, 48]. Vertical partitioning is the process of grouping the triples by their predicate name, with all triples sharing the same predicate name being stored under a table denoted by the predicate name [2].

6.3.3 End-to-End Comparison. LUBM queries 2 and 9 are the two cyclic queries that contain a triangle pattern. Unsurprisingly, here LogicBlox outperforms specialized engines by 3–5× and MonetDB by 17.96× (Table 17) due to the asymptotic advantage of worst-case optimal join algorithms. On these queries EmptyHeaded is 1.5× faster than LogicBlox due to our set layouts, which are designed for single-instruction multiple data parallelism. In general, our speedup over LogicBlox is more modest here than on the previously reported cyclic graph patterns due to the presence of selections.

On acyclic queries with high selectivity, EmptyHeaded also competes with the specialized RDF engines. On simple acyclic queries with selections (LUBM 1,3,5,11,13,14), EmptyHeaded is able to provide covering indexes, like the specialized engines, using only our trie data structure and the attribute order we described in Section 3.3.1. Therefore, EmptyHeaded maintains competitive performance with RDF-3X and TripleBit (Table 17). On more complex acyclic queries with selections (LUBM 7,8,12), RDF-3X and TripleBit observe a performance advantage over EmptyHeaded due to their sophisticated cost-based query optimizers which combine selectivity estimates and join order (Table 17). Our optimizations from Sections 3,4 and 6.2 can provide up to a 48× performance improvement here, but more sophisticated optimizations are needed to outperform the specialized engines. Finally, on LUBM query 8 we observe a performance slowdown when compared to LogicBlox. This is due to expensive reallocations that occur within the EmptyHeaded engine. When removing allocations, we observed that EmptyHeaded's performance for query 8 was equivalent to that of RDF-3X.

Table 18. Relative Speedup of Each Optimization on Selected LUBM Queries with 133 Million Triples

Query	+Layout	+Attribute	+GHD	+Pipelining
Q1	2.10×	129.85×	-	-
Q2	8.22×	1.03×	-	-
Q4	2.02×	12.88×	69.94×	-
Q7	4.35×	95.01×	-	-
Q8	2.24×	1.99×	1.5×	4.67×
Q14	7.92×	234.49×	-	-

`+Layout` refers to EmptyHeaded when using multiple layouts versus solely an unsigned integer array (index layout). `+Attribute` refers to reordering attributes with selections within a GHD node. `+GHD` refers to pushing down selections across GHD nodes in our query plan. `+Pipelining` refers to pipelining intermediate results in a given query plan. "-" means the optimization has no impact on the query.

6.3.4 Micro-Benchmarking Results. The layout optimizations presented in Section 4.1, the pushing down selections optimization presented in Section 3.3, and the pipelining optimization presented in Section 6.2 can enable up to a 234× performance advantage on the LUBM benchmark (see Table 18). We explain the impact of these optimizations on RDF queries next.

Pushing Down Selections. Recall from Section 3.3 that EmptyHeaded pushes down selections in a GHD query plan in two phases: (1) by rearranging the attribute order inside of each node in the GHD and (2) by rearranging the nodes in the GHD such that high-selectivity or low-cardinality nodes appear as far away from the root as possible (so that they are executed earlier in our bottom-up pass). We explain the empirical impact for each phase by walking through an example RDF query for each next.

— Within a Node: Consider LUBM query 14 from Table 16. For this trivial query, we produce a single node GHD containing attributes `{x,'UndergraduateStudent'}`.[12] An attribute ordering of `[x,'UndergraduateStudent']` means that `'x'` is the first level of the trie and `'UndergraduateStudent'` is the second level of the trie. Thus, EmptyHeaded would execute this query by probing the second level of the trie for each `'x'` attribute to determine if there was a corresponding value of `'UndergraduateStudent'`. This is much less efficient than selecting the attribute ordering of `['UndergraduateStudent',x]`, where EmptyHeaded can perform a lookup in the first level of the trie (to find if a value of `'UndergraduateStudent'` exists) and, if successful, return the corresponding second level as the result. This same optimization holds for more complex queries, such as LUBM query 2 (see Table 16), where EmptyHeaded selects the attribute ordering of `['GraduateStudent', 'Department', 'University', x, y, z]`. We show in the `+Attribute` column of Table 18 that forcing the attributes with selections or small cardinalities to come first can enable an up to a 234.49× performance increase.
— Across Nodes: Consider the acyclic join pattern for LUBM query 4 from Table 16. Figure 11 shows two possible GHDs for this query. The GHD on the left is the one produced without the pushing out selection optimization. This GHD does not filter out any intermediate results across potentially high-selectivity nodes when results are first passed up the GHD. The GHD on the right uses the pushing down selections across nodes optimization from

[12] Here `{}` denotes an unordered set of attributes while `[]` denotes an ordered list of attributes.

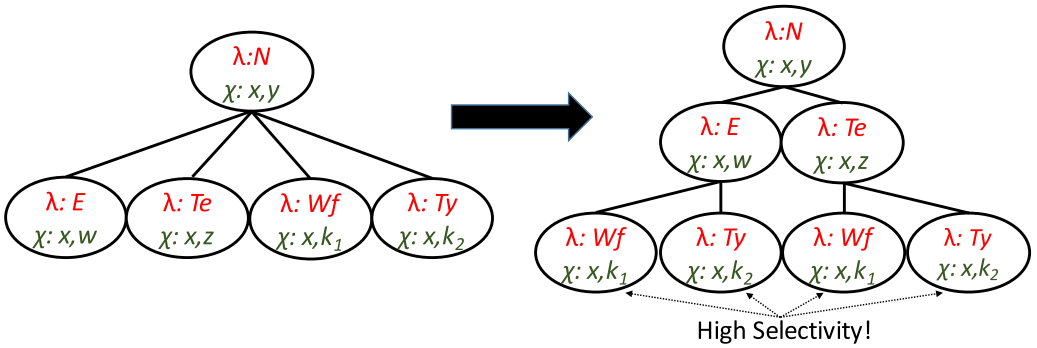


Fig. 11. Pushing down selections across nodes GHD transformation for LUBM query 4. Abbreviations in the GHD nodes are 'N'='name', 'E'='emailAddress', 'Te'='telephone', 'Ty'='type', 'Wf'='worksFor', 'k_1'='Univ0Dept0', and 'k_2'='AssociateProfessor'.

Section 3.3.2. Here the nodes with attributes `'Univ0Dept0'` and `'AssociateProfessor'` are below all other nodes in the GHD, ensuring that these high-selectivity attributes are processed early in the query plan. Although it appears that the optimized GHD has redundant work, EmptyHeaded uses the optimization in Section 3.5 to eliminate this redundant computation. Pushing down selections across nodes applies to only two queries in the LUBM benchmark, but provides up to a 69.94× speedup as we show in the `+GHD` column of Table 18.

Set Layout Choices. Recall that in EmptyHeaded a single trie is analogous to a single index in a standard database. Therefore, EmptyHeaded performs an equality selection by checking whether a set in the trie contains a value. The set layouts we choose from Section 4.1 can have a large impact on the runtime performance when performing this set contains operation. For example, the `bitset` layout can perform this operation in constant time, whereas the `uint` layout performs it in $O(\log n)$ with a binary search. Because of this, the `+Layout` column in Table 18 shows that the set layout optimizer presented in Section 4.1 provides up to a 8.22× performance increase when compared to executing solely over the `uint` layout. As such, it is preferred to have equality selected attributes in the `bitset` layout whenever possible. Interestingly, our pushing down selections optimization will often ensure this. Pushing down selections forces the equality selected attributes to appear in the first levels of the trie, which are liklier dense, and therefore best represented using the `bitset` layout.

Pipelining. On LUBM query 8 we found that pipelining the results between nodes with materialized attributes provided up to a 4.67× performance advantage as shown in the `+Pipelining` column of Section 6.3. This is due to the optimization presented in Section 6.2, which enables EmptyHeaded to materialize fewer intermediate results. Unfortunately, the impact of pipelining is negligible on the other LUBM queries as the output cardinality is often small and so are the intermediate cardinalities.

7 RELATED WORK

Our work extends previous work in five main areas: our prior work, join processing, graph processing, SIMD processing, and set intersection processing. We cover each of these in detail as well as prior published by us in this area.

Prior Work. We briefly describe the main differences between this article and previous work [3] with which there is significant overlap. The first difference comes in our expanded discussion of the

query language in Section 2.3, which now contains more concrete examples. Next, we expanded and presented new content during our discussion of the query compiler in Section 3. Namely, we now discuss how selections are supported in our GHD-based query compiler (Section 3.3), and how we eliminate redundant work (Section 3.5). Moving on, in Section 4 we performed a more detailed and complete experimental study surrounding our optimizer design for various set intersection algorithms and layouts. This included presenting and testing more layouts (Section 4.1) and intersections (Section 4.2). We also added an experimental study to test the impact of node ordering in our design, which we discuss in Section 4.5. Additionally, our new section on implementing selections in Section 3.3 motivated new experiments in Section 5.5 to measure the effectiveness of our design. Finally, we merged previous work on RDF workloads [4] in the EmptyHeaded engine to this article, presenting the relevant work in Section 6.

Join Processing. The first worst-case optimal join algorithm was recently derived [51]. The LogicBlox (LB) engine [68] is the first commercial database engine to use a worst-case optimal algorithm. Researchers have also investigated worst-case optimal joins in distributed settings [17] and have looked at minimizing communication costs [6] or processing on compressed representations [54]. Recent theoretical advances [28, 31] have suggested worst-case optimal join processing is applicable beyond standard join pattern queries. We continue in this line of work. The algorithm in EmptyHeaded is a derived from the worst-case optimal join algorithm [51] and uses set intersection operations optimized for SIMD parallelism, an approach we exploit for the first time. Additionally, our algorithm satisfies a stronger optimality property that we describe in Section 3.

Graph Processing. Due to the increase in main memory sizes, there is a trend toward developing shared-memory graph analytics engines. Researchers have released high-performance shared-memory graph processing engines, most notably SociaLite [61], Green-Marl [25], Ligra [62], and Galois [52]. With the exception of SociaLite, each of these engines proposes a new domain-specific language for graph analytics. SociaLite, based on datalog, presents an engine that more closely resembles a relational model. Other engines such as PowerGraph [20], Graph-X [21], and Pregel [43] are aimed at scale-out performance. Additionally, there have been several recent systems that focus on using graph mining as the basis for graph computation [10, 12, 18, 55, 65]. For example, Arabesque [65] and NScale [55] are two such systems designed for scale-out performance on core graph mining problems such as finding subgraphs or motif counting. The merit of these specialized approaches against traditional OLAP engines is a source of much debate [69], as some researchers believe general approaches can compete with and outperform these specialized designs [21, 44]. Recent products, such as SAP HANA, integrate graph accelerators as part of a OLAP engine [57]. Others [19] have shown that relational engines can compete with distributed engines [20, 43] in the graph domain, but have not targeted shared-memory baselines. We hope our work contributes to the debate about which portions of the workload can be accelerated.

SIMD Processing. Recent research has focused on taking advantage of the hardware trend toward increasing SIMD parallelism. DB2 Blu integrated an accelerator supporting specialized heterogeneous layouts designed for SIMD parallelism on predicate filters and aggregates [56]. Our approach is similar in spirit to DB2 Blu, but applied specifically to join processing. Other approaches such as WideTable [42] and BitWeaving [41] investigated and proposed several novel ways to leverage SIMD parallelism to speed up scans in OLAP engines. Furthermore, researchers have looked at optimizing popular database structures, such as the trie [72], and classic database operations [73] to leverage SIMD parallelism. Our work is the first to consider heterogeneous layouts to leverage SIMD parallelism as a means to improve worst-case optimal join processing.

Set Intersection Processing. In recent years, there has been interest in SIMD sorted set intersection techniques [26, 29, 37, 60]. Techniques such as the SIMDShuffling algorithm [29] break the min property of set intersection but often work well on graph data, while techniques such as SIMD-Galloping [37] that preserve the min property rarely work well on graph data. We experiment with these techniques and slightly modify our use of them to ensure min property of the set intersection operation in our engine. We use this as a means to speed up set intersection—the core operation in our join algorithm.

RDF Engines. Two of the most popular specialized RDF engines are RDF-3X and TripleBit. Both accept queries in the SPARQL query language and have been shown to significantly outperform traditional relational engines. RDF-3X creates a full set of subject-predicate-object indexes by building clustering B+ trees on all six permutations of the triples [48]. RDF-3X also maintains nine aggregate indexes, which include all six binary and all three unary projections. Each index provides some selectivity estimates and the aggregate indexes are used to select the fastest index for a given query. In the TripleBit engine, RDF triples are represented using a compact matrix representation [71]. TripleBit stores two auxiliary index structures and two binary aggregate indexes which enable selectivity estimates for query patterns. This enables TripleBit to select the most effective indexes, minimize the number of indexes needed, and determine the query plan. Like EmptyHeaded, both RDF-3X and TripleBit use dictionary encoding.

8 CONCLUSION

We demonstrate the first general-purpose worst-case optimal join processing engine that competes with low-level specialized engines on standard graph workloads. Our approach provides strong worst-case running times and can lead to over a three orders of magnitude performance gain over standard approaches due to our use of GHDs. We perform a detailed study of set layouts to exploit SIMD parallelism on modern hardware and show that over a three orders of magnitude performance gain can be achieved through selecting among algorithmic choices for set intersection and set layouts at different granularities of the data. We show that on popular graph queries our prototype engine can outperform specialized graph analytics engines by 4–60× and LogicBlox by over three orders of magnitude. Finally, we demonstrate that our new design can easily be extended to accommodate RDF workloads and achieve competitive performance with specialized RDF engines. Our study suggests that this type of engine is a first step toward unifying standard SQL and graph processing engines.

ACKNOWLEDGMENTS

We thank LogicBlox and SociaLite for their helpful conversations and verification of our comparisons, and Rohan Puttagunta and Manas Joglekar for their theoretical underpinnings.

REFERENCES

[1] 2014. U.S. patents network dataset – KONECT. Retrieved from http://konect.uni-koblenz.de/networks/patentcite.
[2] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. 2007. Scalable semantic web data management using vertical partitioning. In *VLDB*. ACM, 411–422.
[3] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A relational engine for graph processing. In *SIGMOD Conference*. ACM, 431–446.
[4] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. Old techniques for new join algorithms: A case study in RDF processing. *DESWEB: ICDE Workshop (2016)*.
[5] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Vol. 8. Addison-Wesley. Chapters 3–4.
[6] Foto N. Afrati, Manas Joglekar, Christopher Ré, Semih Salihoglu, and Jeffrey D. Ullman. 2014. GYM: A multiround join algorithm in MapReduce. *CoRR abs/1410.4156 (2014)*. http://arxiv.org/abs/1410.4156.

[7] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and implementation of the logicblox system. In *SIGMOD Conference*. ACM, 1371–1382.
[8] Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. 2010. Matrix bit loaded: A scalable lightweight join query processor for RDF data. In *Proceedings of the 19th International Conference on World Wide Web*. ACM, 41–50.
[9] Albert Atserias, Martin Grohe, and Dániel Marx. 2013. Size bounds and query plans for relational joins. *SIAM J. Comput.* 42, 4 (2013), 1737–1767.
[10] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Densest subgraph in streaming and mapreduce. *Proceedings of the VLDB Endowment* 5, 5 (2012), 454–465.
[11] Scott Beamer, Krste Asanovic, and David A. Patterson. 2012. Direction-optimizing breadth-first search. In *SC*. IEEE/ACM, 12.
[12] Mansurul A. Bhuiyan and Mohammad Al Hasan. 2015. An iterative MapReduce based frequent subgraph mining algorithm. *IEEE Transactions on Knowledge and Data Engineering* 27, 3 (2015), 608–620.
[13] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. 1999. On random sampling over joins. In *SIGMOD Conference*. ACM, 263–274.
[14] Chandra Chekuri and Anand Rajaraman. 1997. Conjunctive query containment revisited. In *ICDT*. Lecture Notes in Computer Science, Vol. 1186. Springer, 56–70.
[15] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and subgraph listing algorithms. *SIAM J. Comput.* 14, 1 (1985), 210–223.
[16] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. 2009. On compressing social networks. In *KDD*. ACM, 219–228.
[17] Shumo Chu, Magdalena Balazinska, and Dan Suciu. 2015. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD Conference*. ACM, 63–78.
[18] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. 2014. Grami: Frequent subgraph and pattern mining in a single large graph. *VLDB* 7, 7 (2014), 517–528.
[19] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. 2015. The case against specialized graph analytics engines. In *CIDR*. www.cidrdb.org.
[20] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*. USENIX Association, 17–30.
[21] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*. USENIX Association, 599–613.
[22] Georg Gottlob, Martin Grohe, Nysret Musliu, Marko Samer, and Francesco Scarcello. 2005. Hypertree decompositions: Structure, algorithms, and applications. In *WG*, Lecture Notes in Computer Science, Vol. 3787. Springer, 1–15.
[23] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *PODS*. ACM, 31–40.
[24] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.* 3, 2–3 (2005), 158–182.
[25] Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. 2012. Green-Marl: A DSL for easy and efficient graph analysis. In *ASPLOS*. ACM, 349–362.
[26] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. 2014. Faster set intersection with SIMD instructions by reducing branch mispredictions. *PVLDB* 8, 3 (2014), 293–304.
[27] Matthias Jarke and Jurgen Koch. 1984. Query optimization in database systems. *ACM Computing Surveys (CsUR)* 16, 2 (1984), 111–152.
[28] Manas Joglekar, Rohan Puttagunta, and Christopher Ré. 2015. Aggregations over generalized hypertree decompositions. *CoRR* abs/1508.07532.
[29] Ilya Katsov. 2012. Fast Intersection of Sorted Lists Using SSE Instructions. https://highlyscalable.wordpress.com/2012/06/05/fast-intersection-sorted-lists-sse/.
[30] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. IEEE, 195–206.
[31] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions asked frequently. In *PODS*. ACM, 13–28.
[32] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. 2015. Taming subgraph isomorphism for RDF query processing. *CoRR* abs/1506.01973.
[33] Graham Klyne and Jeremy J. Carroll. 2006. Resource description framework (RDF): Concepts and abstract syntax.
[34] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon. 2010. What is Twitter, a social network or a news media?. In *WWW*. ACM, 591–600.
[35] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. IEEE, 38–49.

[36] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29.
[37] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. 2014. SIMD compression and the intersection of sorted integers. *CoRR abs/1401.6399*.
[38] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[39] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2008. Statistical properties of community structure in large social and information networks. In *WWW*. ACM, 695–704.
[40] Jure Leskovec and Rok Sosič. 2016. SNAP: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1.
[41] Yinan Li and Jignesh M. Patel. 2013. BitWeaving: Fast scans for main memory data processing. In *SIGMOD Conference*. ACM, 289–300.
[42] Yinan Li and Jignesh M. Patel. 2014. WideTable: An accelerator for analytical data processing. *PVLDB* 7, 10 (2014), 907–918.
[43] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *SIGMOD Conference*. ACM, 135–146.
[44] Frank McSherry, Michael Isard, and Derek Gordon Murray. 2015. Scalability! But at what COST?. In *HotOS*. USENIX Association.
[45] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. 2002. Network motifs: Simple building blocks of complex networks. *Science* 298, 5594 (2002), 824–827.
[46] Alan Mislove, Massimiliano Marcon, P. Krishna Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and analysis of online social networks. In *Internet Measurement Comference*. ACM, 29–42.
[47] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *PVLDB* 4, 9 (2011), 539–550.
[48] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *VLDB Journal* 19, 1 (2010), 91–113.
[49] Mark E. J. Newman. 2003. The structure and function of complex networks. *SIAM Rev.* 45, 2 (2003), 167–256.
[50] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case optimal join algorithms: [Extended abstract]. In *PODS*. ACM, 37–48.
[51] Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2013. Skew strikes back: New developments in the theory of join algorithms. *SIGMOD Record* 42, 4 (2013), 5–16.
[52] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *SOSP*. ACM, 456–471.
[53] Dung T. Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2015. Join processing for graph patterns: An old dog with new tricks. In *GRADES@SIGMOD/PODS*. ACM, 2:1–2:8.
[54] Dan Olteanu and Jakub Závodný. 2015. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems* 40, 1 (2015), 2.
[55] Abdul Quamar, Amol Deshpande, and Jimmy Lin. 2016. NScale: Neighborhood-centric large-scale graph analytics in the cloud. *VLDB* 25, 2 (2016), 125–150.
[56] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. 2013. DB2 with BLU acceleration: So much more than just a column store. *PVLDB* 6, 11 (2013), 1080–1091.
[57] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. 2013. The graph story of the SAP HANA database. In *BTW (LNI)*, Vol. 214. GI, 403–420.
[58] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. 2014. Navigating the maze of graph analytics frameworks using massive graph datasets. In *SIGMOD Conference*. ACM, 979–990.
[59] Thomas Schank and Dorothea Wagner. 2005. Finding, counting and listing all triangles in large graphs, an experimental study. In *WEA*, Lecture Notes in Computer Science, Vol. 3503. Springer, 606–609.
[60] Benjamin Schlegel, Thomas Willhalm, and Wolfgang Lehner. 2011. Fast sorted-set intersection using SIMD instructions. In *ADMS@VLDB*. 1–8.
[61] Jiwon Seo, Stephen Guo, and Monica S. Lam. 2013. SociaLite: Datalog extensions for efficient social network analysis. In *ICDE*. IEEE Computer Society, 278–289.
[62] Julian Shun and Guy E. Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. In *PPOPP*. ACM, 135–146.
[63] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. 2015. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *2015 Data Compression Conference*. IEEE, 403–412.

[64] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A column-oriented DBMS. In *VLDB*. ACM, 553–564.
[65] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: A system for distributed graph mining. In *SOSP*. ACM, 425–440.
[66] Susan Tu and Christopher Ré. 2015. DunceCap: Query plans using generalized hypertree decompositions. In *SIGMOD Conference*. ACM, 2077–2078.
[67] Jeffrey D. Ullman. 2001. Conjunctive Queries. Retrieved from http://infolab.stanford.edu/ullman/cs345notes/slides01-6.pdf.
[68] Todd L. Veldhuizen. 2012. Leapfrog triejoin: A worst-case optimal join algorithm. *CoRR* abs/1210.0481.
[69] Adam Welc, Raghavan Raman, Zhe Wu, Sungpack Hong, Hassan Chafi, and Jay Banerjee. 2013. Graph analysis: Do we have to reinvent the wheel? In *GRADES*. CWI/ACM, 7.
[70] Mihalis Yannakakis. 1981. Algorithms for acyclic database schemes. In *VLDB*. IEEE Computer Society, 82–94.
[71] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. 2013. TripleBit: A fast and compact system for large scale RDF data. *PVLDB* 6, 7 (2013), 517–528.
[72] Steffen Zeuch, Johann-Christoph Freytag, and Frank Huber. 2014. Adapting tree structures for processing with SIMD instructions. In *EDBT*. OpenProceedings.org, 97–108.
[73] Jingren Zhou and Kenneth A. Ross. 2002. Implementing database operations using SIMD instructions. In *SIGMOD Conference*. ACM, 145–156.

Received December 2016; revised May 2017; accepted July 2017