EmptyHeaded: A Relational Engine for Graph Processing

Christopher R. Aberger
Stanford University
caberger@stanford.edu

Susan Tu
Stanford University
sctu@stanford.edu

Kunle Olukotun
Stanford University
kunle@stanford.edu

Christopher Ré
Stanford University
chrismre@cs.stanford.edu

ABSTRACT

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 data layouts that leverage single-instruction multiple data (SIMD) parallelism. With this architecture, Empty-Headed 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 3x worse performance on SSSP.

Categories and Subject Descriptors

H.2 [**Information Systems**]: Database Management System Engines

Keywords

Worst-case optimal join; generalized hypertree decomposition; GHD; graph processing; single instruction multiple data; SIMD

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 [9, 22, 36, 43, 50]. 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 [18, 49] and (2) optimized data layouts that provide large constant factor runtime improvements [36]. We describe each point in detail:

1. Low-level graph engines [9, 22, 36, 43, 50] 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 [36]. High-level graph engines also use sparse layouts like tail-nested tables [24] 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 [21, 26], order of magnitude performance gaps remain between this approach and low-level engines [9, 24, 43]. Theoretically, traditional join engines face a losing battle, as all pairwise join engines are *provably suboptimal* on many common graph queries [18]. For example, low-level specialized engines execute the "triangle listing" query, which is common in graph workloads [31, 47], 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
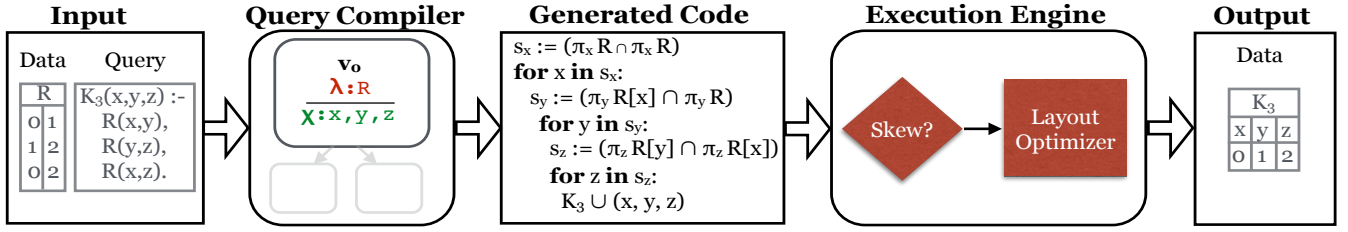
Figure 1: The EmptyHeaded engine works in three phases: (1) the query compiler translates a high-level datalog-like query into a logical query plan represented as a GHD (a hypertree with a single node here), replacing the traditional role of relational algebra; (2) code is generated for the execution 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.

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 [21, 24]. Recently, new multi-way join algorithms were discovered that obtain the correct asymptotic bound for any graph pattern or join [18].

These new multiway join algorithms are by themselves not enough to close the gap. LogicBlox [26] 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), which forms our main contribution. Our architecture includes a novel query compiler based on *generalized hypertree decompositions* (GHDs) [2, 14] 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 [51], we use *generalized hypertree decompositions* (GHDs) [14] 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 allow 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 dif-

ficult 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 [38, 44, 55], 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 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 [21] by 2-4x on PageRank and is at most 3x slower than the highly tuned implementation of Galois [9] 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 1000 lines.

Contribution Summary. This paper introduces the EmptyHeaded engine and demonstrates that a novel architecture can enable multi-way join engines to compete with specialized low-level graph processing engines. We demonstrate that EmptyHeaded outperforms specialized engines

---

[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 paper, 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.

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 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 2-60x and LogicBlox by three orders of magnitude. We demonstrate on PageRank and Single-Source Shortest Paths that our approach remains competitive, at most 3x off the highly tuned Galois engine (Section 5).

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 paper.

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. [19] 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 [14]. 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 [3] (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

---

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
```

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:

$$|\text{OUT}| \leq \prod_{e \in E} |R_e|^{x_e} \qquad (1)$$

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 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 [18]. 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 $\text{AGM}(Q)$. 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 multi-level data structures common in column stores and graph engines [29, 36].

---

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

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	`CountTriangle(;w:long) :- R(x,y),S(x,z),T(x,z); w=<<COUNT(*)>>.`

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.`

Table 1: Example Queries in EmptyHeaded

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* [37]. 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 [22,38], we use the popular database technique of dictionary encoding to build a 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 [49]. This creates up to 3x 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 Appendix A.1.1.

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.

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
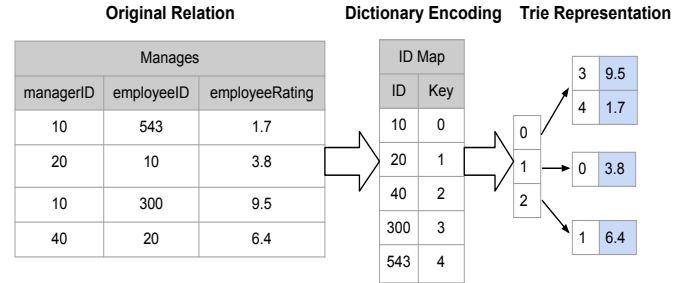

Figure 2: EmptyHeaded transformations from a table to trie representation using attribute order (*managerID,employerID*) and *employerID* attribute annotated with *employeeRating*.

express the standard benchmarks we run in Section 5. Table 1 shows the example queries used in this paper. Above the first horizontal line are conjunctive queries that express joins, projections, and selections in the standard way [52]. Our language has two non-standard extensions: aggregations and a limited form of recursion. We overview both extensions next and provide an example in Appendix A.2.

Aggregation. Following Green et al. [37], 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

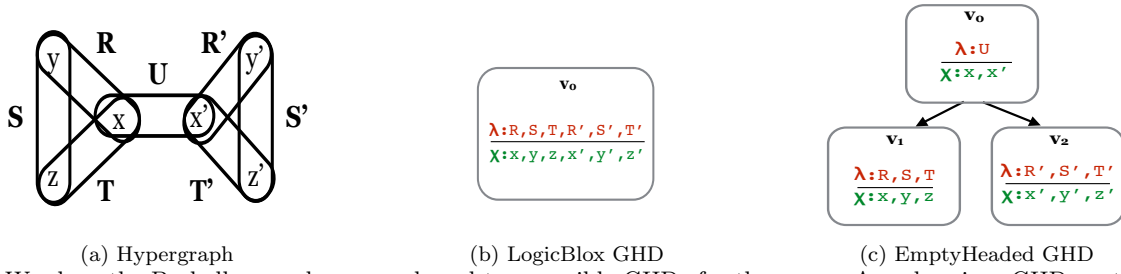(a) Hypergraph   (b) LogicBlox GHD   (c) EmptyHeaded GHD

Figure 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 $\chi(v)$.

Kleene-star rule like:

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

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 doesn't 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 Single-Source Shortest Paths are below the second horizontal line in table 1.

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). Finally, we show how EmptyHeaded translates a GHD into a series of loops, aggregations, and set intersections using the generic worst-case optimal join algorithm [18]. 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 3a 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* 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* 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 3b. 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, OUT *is N^3 in the worst-case for the single node GHD.*

Consider the alternative GHD shown in Figure 3c. 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 OUT *is smaller than N^3. For example, in an aggregation query* OUT *is just a single scalar, and so the difference in runtime between the two GHDs can be quadratic in the size of the database. We describe how we execute this query plan in Section 3.3. This type of optimization is currently not available in the LogicBlox engine.*

3.1.2 Formal Description

We describe GHDs and their advantages formally next.

Definition 1. Let H be a hypergraph. A ***generalized hypertree decomposition (GHD)*** of H is a triple $D = (T, \chi, \lambda)$, where:

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

435

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"* [32] 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 D is $\mathsf{AGM}(Q_v)$, 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 [54], which for acyclic queries enables us to compute the output in linear time in the input size ($O(N^w)$) plus the output size (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 Appendix B.1, we provide detail on how classic database optimizations, such as pushing down selections, can be captured using GHDs.

GHD Optimizer. The EmptyHeaded query compiler selects an optimal GHD to ensure tighter theoretical run time 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.

Aggregations over GHDs. Previous work has investigated aggregations over hypertree decompositions [14,48]. EmptyHeaded adopts this previous work in a straightforward way.

	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.

Table 2: Execution Engine Operations

To do this, we add a single attribute with "semiring annotations" following Green et al. [37]. 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 paper, 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 [9, 36]. 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 paper, EmptyHeaded simply performs a pre-order traversal over the GHD, adding the attributes from each visited GHD node into a queue.

3.3 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 as it is has been shown to be an efficient technique to translate high-level query plans into code optimized for modern hardware [46].

3.3.1 Code Generation API

We first describe the storage-engine operations which serve as the basic high-level 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.

3.3.2 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 3.2. *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:*

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

Across Nodes. Recall Yannakakis' seminal algorithm [54]: 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 seminaive 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 to 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 seminaive recursion, we check if the aggregation is monotonically increasing or decreasing with a `MIN` or `MAX` operator. We use seminaive recursion for SSSP.

EXAMPLE 3.3. *For the Barbell query (see Figure 3c), 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.*

4. EXECUTION ENGINE OPTIMIZER

The EmptyHeaded execution engine runs code generated from the query compiler. The goal of the EmptyHeaded execution engine is to fully utilize SIMD parallelism, but extracting SIMD parallelism is challenging as 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*.[4]

---

[4] 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.

Figure 4: Example of the `bitset` layout that contains n blocks and a sequence of offsets (o_1-o_n) and blocks (b_1-b_n). The offsets store the start offset for values in the bitvector.

A novel aspect of EmptyHeaded is that it automatically copes with density skew through an optimizer that selects among different data layouts. We implemented and tested five different set layouts previously proposed in the literature [6,8,16,40]. We found that the simple `uint` and `bitset` layouts yield the highest performance in our experiments [7]. Thus, we focus on selecting between (1) a 32-bit unsigned integer (`uint`) layout for sparse data and (2) a `bitset` layout for dense data. For dense data, the `bitset` layout makes it trivial to take advantage of SIMD parallelism. But for sparse data, the `bitset` layout causes a quadratic blowup in memory usage while `uint` sets make extracting SIMD parallelism challenging.

Making these layout choices is challenging, as the optimal choice depends both on the characteristics of the data, such as density, and the characteristics of the query. We first describe layouts and intersection algorithms in Sections 4.1 and 4.2. This serves as background for the tradeoff study we perform in Section 4.3, where we explore the proper granularity at which to make layout decisions. Finally, we present our automatic optimizer and show that it is close to an unachievable lower-bound optimal in Section 4.4. This study serves as the basis for our automatic layout optimizer that we use inside of the EmptyHeaded storage engine.

4.1 Layouts

In the following, we describe the `bitset` layout in EmptyHeaded. We omit a description of the `uint` layout as it is just an array of 32-bit unsigned integers. We also detail how both layouts support associated data values.

BITSET. The `bitset` layout stores a set of pairs (offset, bitvector), as shown in Figure 4. The offset is the index of the smallest value in the bitvector. Thus, the layout is a compromise between sparse and dense layouts. We refer to the number of bits in the bitvector as the *block size*. EmptyHeaded supports block sizes that are powers of two with a default of 256.[5] 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.

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 the `uint` layout 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 briefly present an overview of the intersection algorithms EmptyHeaded uses for each layout. This serves as

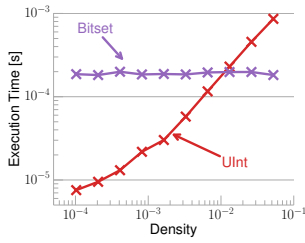---

[5] The width of an AVX register.

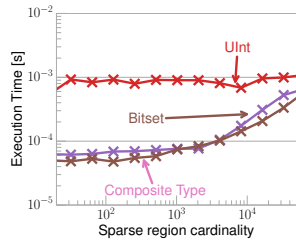Figure 5: Intersection time of `uint` and `bitset` layouts for different densities.

Figure 6: Intersection time of layouts for sets with different sizes of dense regions.

Dataset	Nodes [M]	Dir. Edges [M]	Undir. Edges [M]	Density Skew	Description
Google+ [42]	0.11	13.7	12.2	1.17	User network
Higgs [42]	0.4	14.9	12.5	0.23	Tweets about Higgs Boson
LiveJournal [23]	4.8	68.5	43.4	0.09	User network
Orkut [4]	3.1	117.2	117.2	0.08	User network
Patents [1]	3.8	16.5	16.5	0.09	Citation network
Twitter [17]	41.7	1,468.4	757.8	0.12	Follower network

Table 3: Graph datasets presented in Section 5.1.1 that are used in the experiments.

the background for our tradeoff study 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 ∩ UINT. For the `uint` layout, we implemented and tested five state-of-the-art SIMD set intersections [6, 8, 16, 40]. For `uint` intersections we found that the size of two sets being intersected may be drastically different. This is another type of skew, which we call *cardinality skew*. So-called *galloping* algorithms [53] 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. Therefore, like others [8, 16], EmptyHeaded uses a simple hybrid algorithm that selects a SIMD galloping algorithm when the ratio of cardinalities is greater than 32:1, and a SIMD shuffling algorithm otherwise.

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 as a `uint` as the intersection of two sets can be at most as dense as the sparser set.[6] Notice that this algorithm satisfies the min property with a constant determined by the block size.

4.3 Tradeoffs

We explore three different levels of granularity to decide between `uint` and `bitset` layouts in our trie data structure: 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.[7] In contrast, bitvectors can store 256 elements in a single SIMD register. However, bitvectors are inefficient on sparse data and can result in a quadratic blowup of memory usage. Therefore, one would expect unsigned integer arrays to be well suited for sparse sets and bitvectors for dense sets. Figure 5 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 6x performance increase over the best homogeneous `uint` intersection and a 132x increase over a homogeneous `bitset` intersection. We show in Sections 4.4 and 5.3 that the impact of mixing layouts at the set level on real data can increase overall query performance by over an order of magnitude.

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 and dense blocks using the `bitset` layout. We show in Figure 6 that on synthetic data the composite layout can outperform the `uint` and `bitset` layouts by 2x.

---

[6] Estimating data characteristics like output cardinality a priori is a hard problem [34] and we found it is too costly to reinspect the data after each operation.

[7]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.

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

Table 4: Relative time of the level optimizers on triangle counting compared to the oracle.

4.4 Layout Optimizer

Our synthetic experiments in Section 4.3 show there is no clear winner, as the right granularity at which to make a layout decision depends on the data characteristics and the query. To determine if our system should make layout decisions at a relation, set, or block level on real data, we compare each approach to the time of a lower-bound oracle optimizer. We found that while running on the real graph datasets shown in Table 3, choosing layouts at the set level provided the best overall performance (see Table 4).

Oracle Comparison. The oracle optimizer we compare to provides a lower bound as it is able to freely select amongst all layouts per set operation. Thus, it is allowed to choose any layout and intersection combination while assuming perfect knowledge of the cost of each intersection. We implement the oracle optimizer by brute-force, running all possible layout and algorithm combinations for every set intersection in a given query. The oracle optimizer then counts only the cost of the best-performing combination (from all possible combinations), therefore providing a lower bound for the EmptyHeaded optimizer. On the triangle counting query, the set level optimizer was at most 1.6x off the optimal oracle performance, while choosing at the relation and block levels can be up to 7.3x and 3.2x slower respectively than the oracle. Although more sophisticated optimizers exist, and were tested in the EmptyHeaded engine, we found that this simple set level optimizer performed within 10%-40% of the oracle optimizer on real graph data. Because of this we use the set optimizer by default inside of EmptyHeaded (and for the remainder of this paper).

Set Optimizer. 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.

5. EXPERIMENTS

We compare EmptyHeaded against state-of-the-art high- and low-level specialized graph engines on standard graph benchmarks. We show that by using our optimizations from Section 3 and Section 4, EmptyHeaded is able to compete with specialized graph engines.

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 and 5.3.

5.1.1 Datasets

Table 3 provides a list of the 6 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 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, as it is the first commercial database with a worst-case optimal join optimizer.

Low-Level Engines. We benchmark several graph analytic engines and compare their performance. The engines that we compare to are PowerGraph v2.2 [22], the latest release of commercial graph tool (CGT-X), and Snap-R [43]. Each system provides highly optimized shared memory implementations of the triangle counting query. Other shared memory graph engines such as Ligra [50] and Galois [9] 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 [30] 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 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 [24] 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 [10, 24, 30].

Omitted Comparisons. We compared EmptyHeaded to GraphX [20] which is a graph engine designed for scale-out performance. GraphX was consistently several orders of magnitude slower than EmptyHeaded's performance in a shared-memory setting. We also compared to a commercial database and PostgreSQL but they were consistently over three orders of magnitude off of EmptyHeaded's performance. We exclude a comparison to Grail [21] as its per-

formance has been shown to be comparable to or substantially worse than PowerGraph [22], to which we provide a comparison.

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 paper) 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 1 TB of RAM. We compiled the C++ engines (EmptyHeaded, Snap-R, Power-Graph, 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.

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 2-60x 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 [31, 47]. 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 2-60x 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 is standard as it limits the size of the intersected sets and has been shown to empirically work well [49]. Nearly every graph engine implements pruning in this fashion for the triangle query.

		Low-Level			High-Level	
Dataset	EH	PG	CGT-X	SR	SL	LB
Google+	**0.31**	8.40x	62.19x	4.18x	1390.75x	83.74x
Higgs	**0.15**	3.25x	57.96x	5.84x	387.41x	29.13x
LiveJournal	**0.48**	5.17x	3.85x	10.72x	225.97x	23.53x
Orkut	**2.36**	2.94x	-	4.09x	191.84x	19.24x
Patents	**0.14**	10.20x	7.45x	22.14x	49.12x	27.82x
Twitter	**56.81**	4.40x	-	2.22x	t/o	30.60x

Table 5: Triangle counting runtime (in seconds) for Empty-Headed (EH) and relative slowdown for other engines including PowerGraph (PG), a commercial graph tool (CGT-X), Snap-Ringo (SR), SociaLite (SL) and LogicBlox (LB). 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.

Takeaways. The results from this experiment are in Table 5. 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 [53] 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 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 [30]. 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.

		Low-Level				High-Level	
Dataset	EH	G	PG	CGT-X	SR	SL	LB
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

Table 6: Runtime for 5 iterations of PageRank (in seconds) using 48 threads for all engines. "-" indicates the engine does not process over 70 million edges. EH denotes EmptyHeaded and the other engines include Galois (G), Power-Graph (PG), a commercial graph tool (CGT-X), Snap-Ringo (SR), SociaLite (SL), and LogicBlox (LB).

		Low-Level			High-Level	
Dataset	EH	G	PG	CGT-X	SL	LB
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

Table 7: SSSP runtime (in seconds) using 48 threads for all engines. "-" indicates the engine does not process over 70 million edges. EH denotes EmptyHeaded and the other engines include Galois (G), PowerGraph (PG), a commercial graph tool (CGT-X), and SociaLite (SL). "x" indicates the engine did not compute the query properly.

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

Table 8: 4-Clique (K_4), Lollipop ($L_{3,1}$), and Barbell ($B_{3,1}$) runtime in seconds for EmptyHeaded (EH) and relative run-time for SociaLite (SL), LogicBlox (LB) and EmptyHeaded while disabling features. "t/o" indicates the engine ran for over 30 minutes. "-R" is EH without layout optimizations. "-RA" is EH without both layout (density skew) and inter-section algorithm (cardinality skew) optimizations. "-GHD" is EH without GHD optimizations (single-node GHD).

PageRank. As shown in Table 6, we are consistently 2-4x 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 6, 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 7 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 [33]. For example, Galois, which implements such an algorithm, observes a 2-30x performance improvement over EmptyHeaded on this application (Table 7). 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 represents pattern queries that would require significant effort to implement in low-level graph analytics engines. For example, the simpler tri-angle 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 8 demonstrates that when the GHD optimizations are disabled ("-GHD"), meaning a single node GHD query plan is run, we observe up to an 8x 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 8 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 5, who do not consider mixing SIMD friendly layouts. Table 8 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 (>20x) on extremely dense (`bitset`) and extremely sparse (`uint`) set intersections [7] , which is what happens on the datasets with low density skew here. Like others [41], we found that explicitly disabling SIMD vectorization, in addition to our layout and algorithm choices, decreases our performance by another 2x (see Appendix A.1.2). Our contribution here is the mixing of data representations ("-R") and set intersection algorithms ("-RA"), both of which are deeply intertwined with SIMD parallelism. In total, Table 8 and our discussion validate that the set layout and algorithmic features have merit and enable EmptyHeaded to compete with graph engines.

6. RELATED WORK

Our work extends previous work in four main areas: join processing, graph processing, SIMD processing, and set intersection processing.

Join Processing. The first worst-case optimal join algorithm was recently derived [19]. The LogicBlox (LB) engine [53] is the first commercial database engine to use a worst-case optimal algorithm. Researchers have also investigated worst-case optimal joins in distributed settings [35] and have looked at minimizing communication costs [11] or processing on compressed representations [48]. Recent theoretical advances [25, 27] 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 [19] 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 [24], Green-Marl [36], Ligra [50], and Ga-

lois [9]. With the exception of SociaLite, each of these engines proposes a new domain-specific language for graph analytics. SociaLite, based on datalog, presents a engine that more closely resembles a relational model. Other engines such as PowerGraph [22], Graph-X [20], and Pregel [15] are aimed at scale-out performance. The merit of these specialized approaches against traditional online analytical processing (OLAP) engines is a source of much debate [5], as some researchers believe general approaches can compete with and outperform these specialized designs [13, 20]. Recent products, such as SAP HANA, integrate graph accelerators as part of a OLAP engine [28]. Others [21] have shown that relational engines can compete with distributed engines [15,22] 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 [38]. Our approach is similar in spirit to DB2 Blu, but applied specifically to join processing. Other approaches such as WideTable [45] and BitWeaving [44] 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 [39], and classic database operations [55] 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 [6, 8, 16, 40]. Techniques such as the SIMDShuffling algorithm [40] break the min property of set intersection but often work well on graph data, while techniques such as SIMDGalloping [8] 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, which is the core operation in our approach to join processing.

7. 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. Finally, we show that on popular graph queries our prototype engine can outperform specialized graph analytics engines by 4-60x and LogicBlox by over three orders of magnitude. Our study suggests that this type of engine is a first step toward unifying standard SQL and graph processing engines.

8. REFERENCES

[1] U.S. patents network dataset – KONECT, 2014.

[2] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. In *ICDT '97*, pages 56–70. Springer.

[3] A. Atserias et al. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.

[4] A. Mislove et al. Measurement and analysis of online social networks. In *Proc. Internet Measurement Conf.*, 2007.

[5] A. Welc et al. Graph analysis: Do we have to reinvent the wheel? GRADES '13, pages 7:1–7:6.

[6] B. Schlegel et al. Fast sorted-set intersection using simd instructions. In *ADMS Workshop*, 2011.

[7] C. Aberger et al. Emptyheaded: A relational engine for graph processing. *Technical Report*, abs/1503.02368, 2016.

[8] D. Lemire et al. SIMD compression and the intersection of sorted integers. Software: Practice and Experience, 2015.

[9] D. Nguyen et al. A lightweight infrastructure for graph analytics. SOSP '13, pages 456–471.

[10] D. Nguyen et al. Join processing for graph patterns: An old dog with new tricks. *arXiv preprint arXiv:1503.04169*, 2015.

[11] F. Afrati et al. Gym: A multiround join algorithm in mapreduce. Technical report, Stanford University.

[12] F. Chierichetti et al. On compressing social networks. KDD '09, pages 219–228.

[13] F. McSherry et al. Scalability! but at what cost? HOTOS '15, pages 14–14, Berkeley, CA, USA, 2015. USENIX Association.

[14] G. Gottlob et al. Hypertree decompositions: Structure, algorithms, and applications. In *Graph-theoretic concepts in computer science*, pages 1–15. Springer, 2005.

[15] G. Malewicz et al. Pregel: A system for large-scale graph processing. SIGMOD '10, pages 135–146.

[16] H. Inoue et al. Faster set intersection with simd instructions by reducing branch mispredictions. *VLDB '14*, 8(3).

[17] H. Kwak et al. What is Twitter, a social network or a news media? In *WWW*, 2010.

[18] H. Q. Ngo et al. Worst-case optimal join algorithms: [extended abstract]. In *PODS '12*, pages 37–48. ACM.

[19] H. Q. Ngo et al. Skew strikes back: New developments in the theory of join algorithms. *CoRR*, abs/1310.3314, 2013.

[20] J. E. Gonzalez et al. Graphx: Graph processing in a distributed dataflow framework. In *OSDI '14*, pages 599–613. USENIX Association, October 2014.

[21] J. Fan et al. The case against specialized graph analytics engines. In *CIDR '15*.

[22] J. Gonzalez et al. Powergraph: Distributed graph-parallel computation on natural graphs. OSDI '12, pages 17–30.

[23] J. Leskovec et al. Statistical properties of community structure in large social and information networks. In *Proc. Int. World Wide Web Conf.*, pages 695–704, 2008.

[24] J. Seo et al. Socialite: Datalog extensions for efficient social network analysis. In *ICDE '13*, pages 278–289. IEEE, 2013.

[25] M. A. Khamis et al. FAQ: Questions asked frequently. *arXiv preprint arXiv:1504.04044*, 2015.

[26] M. Aref et al. Design and implementation of the logicblox system. In *SIGMOD '15*, pages 1371–1382. ACM, 2015.

[27] M. Joglekar et al. Aggregations over generalized hypertree decompositions. *arXiv preprint arXiv:1508.07532*, 2015.

[28] M. Rudolf et al. The graph story of the SAP HANA database. In *BTW*, volume 13, pages 403–420. Citeseer, 2013.

[29] M. Stonebraker et al. C-store: a column-oriented dbms. In *VLDB '05*, pages 553–564.

[30] N. Satish et al. Navigating the maze of graph analytics frameworks using massive graph datasets. SIGMOD '14, pages 979–990.

[31] R. Milo et al. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[32] S. Abiteboul et al. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[33] S. Beamer et al. Direction-optimizing breadth-first search. SC '12, pages 12:1–12:10.

[34] S. Chaudhuri et al. On random sampling over joins. volume 28 of *SIGMOD '99*, pages 263–274.

[35] S. Chu et al. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD '15*, pages 63–78. ACM, 2015.

[36] S. Hong et al. Green-marl: A dsl for easy and efficient graph analysis. ASPLOS XVII, pages 349–362, 2012.

[37] T. J. Green et al. Provenance semirings. In *SIGMOD '07*, pages 31–40. ACM.

[38] V. Raman et al. DB2 with BLU acceleration: So much more than just a column store. *VLDB.*, 6(11):1080–1091, 2013.

[39] S. Z. F. Huber and J. Freytag. Adapting tree structures for processing with simd instructions.

[40] I. Katsov. Fast intersection of sorted lists using sse instructions. https://highlyscalable.wordpress.com/2012/06/05/fast-intersection-sorted-lists-sse/, 2012.

[41] Christian Lemke, Kai-Uwe Sattler, Franz Faerber, and Alexander Zeier. Speeding up queries in column stores: A case for compression. DaWaK '10, pages 117–129.

[42] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[43] J. Leskovec and R. Sosič. SNAP: A general purpose network analysis and graph mining library in C++. http://snap.stanford.edu/snap, 2014.

[44] Y. Li and J. M. Patel. Bitweaving: Fast scans for main memory data processing. SIGMOD '13, pages 289–300.

[45] Y. Li and J. M. Patel. Widetable: An accelerator for analytical data processing. *VLDB*, 7(10), 2014.

[46] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *VLDB '11*, 4(9):539–550.

[47] M. E. J. Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, 2003.

[48] D. Olteanu and J. Závodný. Size bounds for factorised representations of query results. *TODS '15*, 40(1):2.

[49] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. WEA '05.

[50] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, 2013.

[51] S. Tu and C. Ré. Duncecap: Query plans using generalized hypertree decompositions. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 2077–2078. ACM, 2015.

[52] J. Ullman. Conjunctive queries. http://infolab.stanford.edu/~ullman/cs345notes/slides01-6.pdf.

[53] T. L. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481*, 2012.

[54] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.

[55] J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *SIGMOD '02*.

APPENDIX

A. APPENDIX FOR SECTION 2

A.1 Dictionary Encoding and Node Ordering

A.1.1 Node Ordering

Because EmptyHeaded maps each node to an integer value, it is natural to consider the performance implications of these mappings. Node ordering can affect the performance

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

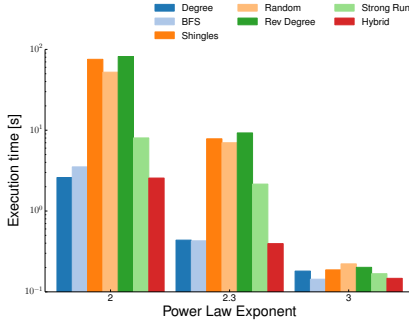Table 9: Node ordering times in seconds on two popular graph datasets.

Figure 7: Effect of data ordering on triangle counting with synthetic data.

	Default		Symmetrically Filtered	
Dataset	uint	EmptyHeaded	uint	EmptyHeaded
Google+	1.0x	1.4x	1.8x	4.7x
Higgs	0.9x	1.2x	3.0x	1.9x
LiveJournal	1.2x	1.1x	1.7x	1.6x
Orkut	1.1x	1.1x	1.4x	1.5x
Patents	1.2x	1.1x	1.9x	1.3x

Table 10: Relative time of random ordering compared to ordering by degree.

	Default			Symmetrically Filtered		
Dataset	-S	-R	-SR	-S	-R	-SR
Google+	1.0x	3.0x	7.5x	1.0x	4.9x	13.4x
Higgs	1.5x	3.9x	4.8x	1.2x	0.9x	1.7x
LiveJournal	1.6x	1.0x	1.6x	1.2x	0.9x	1.2x
Orkut	1.8x	1.1x	2.0x	1.4x	1.0x	1.6x
Patents	1.3x	0.9x	1.1x	1.0x	0.7x	0.8x

Table 11: Relative time when disabling features on the triangle counting query. Symmetrically filtered refers to the data preprocessing step which is specific to symmetric queries. "-S" is EmptyHeaded without SIMD. "-R" is EmptyHeaded using uint at the graph level.

in two ways: It changes the ranges of the neighborhoods and, for queries that use symmetry breaking, it affects the number of comparisons needed to answer the query. In the following, we discuss the impact of node ordering on triangle counting with and without symmetry breaking.

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 and vary the Power-Law degree exponents from 1 to 3. 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 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 [12].

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 7 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 7 shows the performance of hybrid ordering and how it tracks the performance of BFS or degree where each is optimal.

Each ordering incurs the cost of performing the actual ordering of the data. Table 9 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 or reverse degree depends 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.

A.1.2 Pruning Symmetric Queries

We explore the effect of node ordering on query performance with and without the data pruning that symmetrical queries enable. Symmetric queries such as the triangle query or the 4-clique query on undirected graphs produce equivalent results for graphs where each src, dst pair occurs only once and datasets where each src, dst has a corresponding dst, src pair (the latter producing a result that is a multiple of the former). Specialized engines take advantage of restricted optimization that only holds for symmetric patterns. For this experiment, we measure the effect of the node orderings introduced in Appendix A.1.1 on five datasets with different set layouts. We show that node ordering only has a substantial impact on queries that enable symmetry breaking and that our layout optimizations typically have a larger impact on the queries which do not enable symmetry breaking, which is the more general case.

We use the relative triangle counting performance on 5 datasets with a random ordering and ordering by degree as a proxy for the impact of node ordering. For each dataset, we measure the triangle counting performance with random ordering and ordering by degree (the default standard), with and without pruning, and with the EmptyHeaded set level optimizer and with a homogeneous uint layout. We call pruned data on symmetrical queries *symmetrically filtered*. We report the relative performance of the random ordering compared to ordering by degree. Table 10 shows that ordering does not have a large impact on queries that do not enable symmetry breaking. In addition, Table 10 shows

that our optimizer is more robust to various orderings in the special cases where symmetry filtering is allowed. Table 11 shows that our optimizations typically have a larger impact on data which is not symmetrically filtered. This is important as symmetrical queries are infrequent and their symmetrical property breaks with even a simple selection.

A.2 Extended Query Language Discussion

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. b = 'Chris').

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

EXAMPLE A.1. *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 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 [27]. 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 (5) iterations as specified in the head.*

B. APPENDIX FOR SECTION 3

B.1 Selections

Implementing high performance selections in EmptyHeaded requires three additional optimizations that significantly effect performance: (1) pushing down selections within the worst-case optimal join algorithm, (2) index layout trade-offs, and (3) pushing down selections across GHD nodes. The first two points are trivial so we briefly overview them next while providing a detailed description and experiment for pushing down selections across GHDs in Appendix B.1.1. We narrow our scope in this section to only equality selections, but our techniques are general and can be applied to general selection constraints.
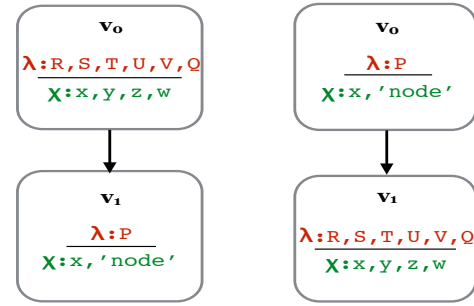
(a) GHD without pushing down (b) GHD with pushing down

Figure 8: We show two possible GHDs for the 4-clique selection query.

Within a Node. Pushing down selections within a GHD node is akin to rearranging the attribute ordering for the generic worst-case optimal algorithm. Simply put, the attributes with selections should come first in the attribute ordering forcing the attributes with selections to be processed first in Algorithm 1.

Index Layouts. The data layouts matter again 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 therefore best represented as a bitset. For equality selections this is enables us to perform the actual selection in constant time versus a binary search in an unsigned integer array.

B.1.1 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.

B.1.2 Queries

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 12.

Consider the 4-clique selection query:

Name	Query Syntax
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').`

Table 12: Selection Queries in EmptyHeaded

EXAMPLE B.1. *Figure 8 shows two possible GHDs for this query. The GHD on the left is the one produced without using the three steps above to push down selections across GHD nodes. This GHD does not filter out any intermediate results across the potentially high selectivity node containing the selection when results are first passed up the GHD. The GHD on the right uses the three steps above. Here the node with the selection is below all other nodes in the GHD, ensuring that high selectivities are processed early in the query plan.*

B.1.3 Discussion

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 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 13. 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 13 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 3.66x and SociaLite by 5754x.

B.2 Eliminating Redundant Work

Our compiler is the first worst-case optimal join optimizer to eliminate 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. 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 2x 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.

Dataset	Query	|Out|	EH	-GHD	SL	LB
Google+	SK_4	1.5E+11	154.24	6.09x	t/o	t/o
		5.5E+7	1.08	865.95x	t/o	50.91
	$SB_{3,1}$	4.0E+17	0.92	3.22x	t/o	t/o
		2.5E+3	0.008	351.72x	t/o	t/o
Higgs	SK_4	2.2E+7	1.92	14.48x	t/o	58.10x
		2.7E+7	2.91	9.50x	t/o	52.44x
	$SB_{3,1}$	1.7E+12	0.060	17.36x	t/o	t/o
		2.4E+12	0.070	14.88x	t/o	t/o
LiveJournal	SK_4	1.7E+7	6.73	18.05x	t/o	14.83x
		5.1E+2	0.0095	13E3x	t/o	10.46x
	$SB_{3,1}$	1.6E+12	0.27	6.47x	t/o	t/o
		9.9E+4	0.0062	278.16x	t/o	70.23x
Orkut	SK_4	9.8E+8	208.20	1.26x	t/o	t/o
		2.8E+5	0.020	13E+3x	t/o	18.79x
	$SB_{3,1}$	1.1E+15	3.24	3.20x	t/o	t/o
		2.2E+8	0.0072	1314x	21E+3X	23E+3x
Patents	SK_4	0	0.011	121.70x	5754x	3.66x
		9.2E+3	0.011	117.56x	5572x	10.72x
	$SB_{3,1}$	1.6E+1	0.0060	77.82x	223.29x	15.17x
		1.1E+7	0.0066	71.22x	1073x	3296x

Table 13: 4-Clique Selection (SK_4) and Barbell Selection ($SB_{3,1}$) runtime in seconds for EmptyHeaded (EH) and relative runtime for SociaLite (SL), LogicBlox (LB) and EmptyHeaded while disabling optimizations. "|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.

C. APPENDIX FOR SECTION 5

C.1 Extended Triangle Counting Discussion

PowerGraph represents each neighborhood using a hash set (with a cuckoo hash) if the degree is larger than 64 and otherwise represents the neighborhood as a vector of sorted node ID's. PowerGraph incurs additional overhead due to its programming model and parallelization infrastructure in a shared memory setting. CGT-X uses a CSR layout and runs Java code for queries which might not be as efficient as native code. Snap-R prunes each neighborhood on the fly using a simple merge sort algorithm and then intersects each neighborhood using a custom scalar intersection over the sets. We note that the runtimes in Table 5 do not reflect the cost of pruning the graph in our system, PowerGraph, SociaLite, or LogicBlox, while CGT-X and Snap-R include this time in their overall runtime. In Snap-R we found, depending on the skew in the graph, the pruning time accounts for 2%-46% of the runtime on the triangle counting.

C.2 Memory Usage

We utilize a small amount of the available memory (1TB RAM) for the datasets run in this paper. For example, when running the PageRank query on the LiveJournal dataset our engine uses at most 8362MB of memory. For comparison, Galois uses 7915MB and PowerGraph uses 8620MB.