# A RELATIONAL ARCHITECTURE FOR GRAPH, LINEAR ALGEBRA, AND BUSINESS INTELLIGENCE QUERYING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Christopher R. Aberger
July 2018



This dissertation is online at: http://purl.stanford.edu/ft236yr6309

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

**Chris Re, Primary Adviser**

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

**Oyekunle Olukotun, Co-Adviser**

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

**Matei Zaharia**

Approved for the Stanford University Committee on Graduate Studies.

**Patricia J. Gumport, Vice Provost for Graduate Education**

*This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.*

# Abstract

The exponential growth of data being collected in nearly every sector has forced modern analytics workloads to become increasingly diverse [86]. This in turn has driven a need for data processing (and warehousing) tools that extend beyond the business intelligence queries for which relational database management systems (RDBMS) were originally designed. In fact, RDBMSs have been shown to be particularly inefficient, when compared to purpose-built (specialized) engines, on two important classes of emerging workloads: (1) the graph analysis queries needed for social and biological network algorithms [103] and (2) the linear algebra operations at the core of nearly all machine learning algorithms [51]. As a result, RDBMSs are often forsaken on such workloads in favor of specialized approaches. With that being said, RDBMSs have many inherent benefits (that specialized engines typically do not) such as high-level query languages, sophisticated query optimizers, and high-performance execution engines which have all been optimized, tuned, and deployed for decades. As a result, RDBMSs still *warehouse* much of the data today and therefore remain an integral part of modern analytics workflows.

Despite the benefits and prevalence of relational architectures, extending such systems to accommodate graph and linear algebra workloads is a challenging task. While some have argued that traditional RDBMSs can be modified in straightforward ways to accommodate these workloads [34, 65], order of magnitude performance gaps remain between such approaches and specialized engines [1, 75, 95]. Even worse, traditional RDMBSs theoretically face a losing battle: the query optimizers at the core of their architectures were recently shown to be *provably suboptimal* on many important classes of queries [73]. Shockingly, this asymptotic suboptimality is because of the pairwise join algorithms at the core of nearly every RDBMS design since System-R in 1976 [12]. Fortunately, researchers

have recently discovered new multiway join algorithms (or worst-case optimal join algorithms) that obtain the correct asymptotic bound for any join query [73].

In this dissertation we present an alternative approach for designing a relational architecture based on these new multiway (worst-case optimal) join algorithms. In particular, we study whether these multiway join algorithms [73] can be used to unify graph, linear algebra, and business intelligence query workloads. We argue that an engine based on multiway join algorithms can unify these seemingly disparate query workloads, but it requires a novel architecture. To do this, we present a new in-memory query processing engine called EmptyHeaded. At the core of EmptyHeaded is a novel query compiler based on *generalized hypertree decompositions* (GHDs) [22, 38] and an execution engine designed to exploit the low-level layouts necessary to increase single-instruction multiple data (SIMD) parallelism. We show that such a query compiler and execution engine is capable of capturing the optimizations necessary to achieve efficient performance in the graph, linear algebra, and business intelligence domains.

For the first time, EmptyHeaded demonstrates that such a relational architecture is not just theoretically superior, but that it also has merit in practice. To validate this, we compare EmptyHeaded's end-to-end query runtimes against popular pairwise relational engines (MonetDB and HyPeR), the only commercial worst-case optimal join engine (LogicBlox), and low-level specialized engines on standard benchmark queries in each application domain. In the graph domain, we show that EmptyHeaded can outperform relational (LogicBlox, MonetDB, and HyPeR) engines by up to three orders of magnitude, and specialized engines (PowerGraph, Galois, SociaLite, and Snap-Ringo) by up to 60x. In the linear algebra domain, we show that EmptyHeaded can outperform relational engines (LogicBlox, MonetDB, and HyPeR) orders of magnitude while competing on average within 31% of the performance of Intel MKL. In the business intelligence domain, we show that EmptyHeaded can outperform relational engines (MonetDB and LogicBlox) by close to two orders of magnitude while competing on average within 30% of the HyPeR database engine. This body of work demonstrates that multiway joins have merit in practice and can be used to unify popular analytics domains where pairwise joins fall flat.

# Acknowledgments

While at Stanford I have had the opportunity to work with some of the most brilliant, inspiring, and innovative people in the field of computer science. This dissertation would not be possible without any of them.

First and foremost I would like to thank Christopher Rè who has been my guiding light throughout graduate school. On a daily basis Chris inspires me to be bold and fearless, instilling in me the confidence to tackle problems that I would have otherwise considered too hard or infeasible. Most importantly Chris turned me into a more curious person, with a continual appetite to understand what I currently do not. In the past five years, Chris has not only made me the researcher and software engineer I am today, he has made me a better person.

I have also been very fortunate to work alongside Kunle Olukotun for the past five years. During my undergraduate studies (and still today) I regarded Kunle as a legend, and he was the sole reason why I came to Stanford. Kunle's patience, vision, and optimism was always there to lift me up when I felt the outlook was the bleakest or when I struggled to make progress throughout the course of my PhD. Kunle's advice has been invaluable to my growth at Stanford and I am forever grateful to have been mentored by such a great visionary.

I would also like to extend thanks to Matei Zaharia and Hector Garcia-Molina who I have been fortunate to interact with during my stay at Stanford. Matei's wisdom on how to structure, design, pitch, and create a successful systems projects has helped scope and guide my research over the past several years. I would also like to extend a thanks to Hector Garcia-Molina who served on my orals committee. It is rare opportunity to receive feedback from a titan of the database field such as Hector and I am fortunate to have been

able to interact with him.

Without my co-authors the work contained within this thesis would not have been possible. I would like to thank Andres Nötzli for helping me setup the foundations of this work and for his feedback, advice, and friendship throughout my stay at Stanford. I thank both Susan Tu and Andrew Lamb whose contributions to this project were invaluable. I am fortunate to have been able to work alongside such talented people as these two. Finally, I thank Rohan Puttagunta and Manas Joglekar for their theoretical contributions to this project, which would not have been possible without their brilliance.

I learned so much from my peers at Stanford and would like to thank them for their friendship which made my stay at Stanford so much fun. I thank Arvind Sujeeth, Kevin Brown, and HyoukJoong Lee for their mentorship during my first years at Stanford. I thank Christopher De Sa, Theodoros Rekatsinas, Stephen Bach, and Alex Ratner for their invaluable feedback on the work in this dissertation and for the opportunity to learn alongside such brilliant researchers. Finally, I thank Megan Leszczynski, Paroma Varma, Braden Hancock, Jian Zhang, and Sen Wu for their frequent interactions and thoughtful discussions. I am excited to see what each of you accomplishes throughout your careers.

Finally, I would like to thank my fiancèe for her continual support which provides me the fuel to fight through challenging and interesting problems on a daily basis. It takes a special person to not only understand, but also adapt their own schedule to the chaotic lifestyle of a PhD student; this dissertation is not possible without her. I would also like to thank my parents, brother, and sister whose support has helped me at every stage of my life. They are responsible for the best of me.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Modern analytics workloads extend far beyond the SQL-style business intelligence queries that relational database management systems (RDBMS) were designed to process efficiently. As a result, RDBMSs often incur orders of magnitude performance gaps with the best known implementations on modern analytics workloads like graph analysis and linear algebra queries. Therefore, the relational model is often forsaken on such workloads, resulting in a flurry of activity around designing specialized (low-level) graph and linear algebra packages [1, 18, 36, 43, 61, 75, 95]. In this dissertation we present a new type of relational query processing architecture that overcomes these shortcoming of traditional relational architectures. To do this we present a new in-memory query processing engine called EmptyHeaded. EmptyHeaded uses a new, worst-case optimal (multiway) join algorithm as its core execution mechanism which makes it fundamentally different from nearly every other relational architecture [11, 44, 50, 85]. With EmptyHeaded, we show how the crucial optimizations for graph analysis, linear algebra, and business intelligence workloads can be captured in a single relational architecture. The work presented in this document will show that, unlike traditional RDBMSs, a new relational query processing architecture is capable of delivering competitive performance in multiple application domains.

In the remainder of this section we, (1) motivate and describe the challenges of mapping the relational model to each of the aforementioned application domains, (2) describe the benefits of the relational model, (3) provide a high-level overview of join algorithms, (4) provide an overview of the relational architecture at the core of this dissertation, (5) outline

our contributions, and (6) provide an outline for the remainder of this dissertation.

## 1.1 The Challenge: Modern Workloads

The efficient processing of classic SQL-style workloads is no longer enough; both machine learning and graph algorithms are being adopted at an explosive rate. In fact, Intel projects that by 2020 the hardware cycles dedicated to machine learning tasks will grow by 12x, resulting in more servers running this than any other workload [20]. Furthermore, the volume of graph data collected from social and biological networks is growing at an exponential rate [36, 43, 61, 75, 95]. As a result, there is a booming need for query processing engines that are efficient on (1) graph queries at the core of network analysis workloads, (2) the ever-prevalent SQL-style queries at the core of business intelligence workloads, and (3) the linear algebra operations at the core of machine learning workloads. In this dissertation, we explore whether a new query processing architecture is capable of delivering competitive performance in each of these application domains.

However, designing a query processing architecture that is efficient in all of these application domains is a challenging task. While some have argued that traditional relational engines can be modified in straightforward ways to accommodate these workloads [11, 34], order of magnitude performance gaps remain between such approaches and specialized (domain-specific) engines [61, 75, 94]. This is because query and data characteristics vary drastically across domains, and RDBMSs were designed with only business intelligence workloads in-mind. In the remainder of this section we describe the key characteristics of workloads in each domain and explain why extending the relational model to cover all them efficiently is challenging.

**Graph Analysis** Graph algorithms contain highly irregular access patterns which leads to poor hardware utilization when compared to the regular access patterns that dominate business intelligence and dense linear algebra queries [89]. Because of this, designing

a high-level query processing engine for graph workloads is challenging, and such engines are typically orders of magnitude slower than hand-tuned code [94, 103]. As a result, there has been a flurry of activity around designing specialized graph analytics engines [36, 43, 61, 75, 95]. 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. Low-level graph engines [36, 43, 61, 75, 95] provide iterators and domain-specific primitives, with which users can write asymptotically faster algorithms than what traditional databases or other high-level approaches [34, 94] can provide. However, in low-level engines 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 low-level engines.

**RDF Processing** The volume of Resource Description Framework (RDF) data from the Semantic Web has grown exponentially in the past decade [71, 105]. RDF data is a collection of Subject-Predicate-Object triples that form a complex and massive graph that traditional query mechanisms do not handle efficiently [54, 71]. Because RDF data forms a complex graph, the challenges (and characteristics) of processing RDF data are nearly identical to the aforementioned ones on graph processing. Still, there has been significant interest in designing specialized engines for RDF processing [13, 53, 71, 105]. These specialized engines accept the SPARQL query language and build several indexes ($> 10$) over the Subject-Predicate-Object triples to process RDF workloads efficiently [71, 105]. In contrast, the natural way of storing RDF data in a traditional relational engine is to use triple tables [71] or vertically partitioned column stores [5], but these techniques can be three orders of magnitude slower than specialized RDF engines [71].

**Linear Algebra** Although it is well-known that linear algebra operations can be expressed using joins and aggregations, executing these queries via the join algorithms in standard RDBMSs is orders of magnitude slower than using a linear algebra package. Linear algebra data can be both sparse and dense. To compute linear algebra kernels using relational operators on sparse kernels both joins and aggregations are exercised heavily,

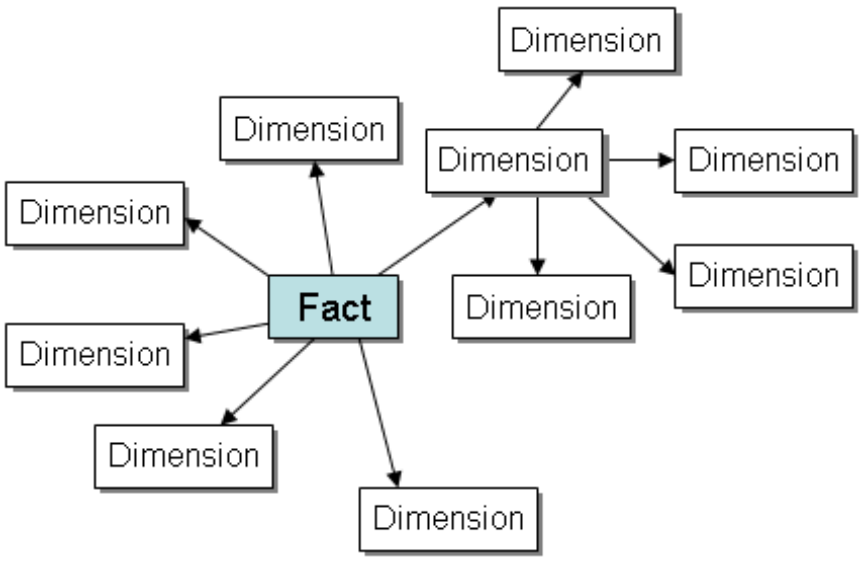



Figure 1.1: Snowflake schema. Source: Vertica Documentation [87]

whereas on dense data (because the access pattern is predictable) the computation is nearly all aggregations. To accommodate such patterns, others [65] have shown that an RDBMS must be modified to achieve reasonable performance. The most popular linear algebra packages (e.g. BLAS [18] or LAPACK [10]) provide high-performance through low-level procedural interfaces but therefore lack the ability for high-level querying. To address this, array databases with high-level querying, like SciDB [19], have been proposed. Unfortunately, array databases are highly specialized and are not designed for general business intelligence or graph querying.

**Business Intelligence Querying** Business intelligence queries have been the workhorse of RDBMS's for the past 40+ years and remain a vital component of the $40+ billion database market today [26]. Business intelligence queries most often operate over relational tables that compose a snowflake (sometimes star) schema where a few tables are centralized fact tables that are connected to multiple dimension tables (see Figure 1.1) [87]. As a result, most database schemas contain a large number of normalized[1] tables each with a large number of attributes. Queries over such schemas typically require de-normalizing or

[1]Normalization splits up data to avoid redundancy (duplication) by moving commonly repeating groups of data into new tables [35].

joining a few of these connected tables to produce a single output table (along with some other conditions to filter and aggregate data). This type of query pattern has received much attention from database researchers over the past several decades [30, 44, 50, 84, 97, 99], resulting in many proven optimizations for relational architectures to process such queries efficiently.

## 1.2 The Case for a Relational Architecture

Relational architectures are the most popular and widely used solutions for storing and maintaining data. Since their inception several decades ago, RDBMSs have experienced enormous success and account for a growing $40 billion market. This success is largely due to the simplicity (and therefore accessibility) of the relational model: data is stored in two-dimensional tables called relations where the columns are named attributes and the values in each row are called tuples. This model is simple for humans without even a basic understanding of computer science to understand. Even better, this model lends itself naturally to relational algebra which is the simple set of data manipulation operations that can be run over existing relations to produce new relations. There are many ways to expose relational algebra to end users in programming languages, but the most popular is called the the Structured Query Language (SQL)—the world's first widely successful domain specific language.

This simplicity of relational architectures lends itself to many advantages to that cannot be ignored. Some obvious ones are that much of the data today is already stored in a RDBMS and users are already trained to interact with relational systems (and if they are not, the test of time has proven that it is easy to learn). In the remainder of this section we present several inherent advantages to relational architectures that are sacrificed when specialized engines are favored instead.

**High-Level Query Languages** High-level query languages are useful because they are less powerful than standard programming languages like C or Java [35]. Although this seems counterintuitive, the benefit of limiting what can be done in a programming language makes it easier for non-experts to use and it leaves open the freedom for compilers to

produce highly optimized code. In high-level query languages, such as SQL or datalog, the user is not burdened with worrying about writing algorithms, but rather is tasked with expressing queries. However, in low-level languages it is the burden of the user to write the query properly through iterators and domain-specific primitives, which may require system-specific optimizations. Finally, the logical and physical independence (see next two points) provided by high-level query languages enables the query compiler to perform a wide range of sophisticated optimizations that remain opaque to the end user.

**Logical independence** The burden of changing the operations (e.g. join orders and query planing) of programs based on data characteristics should not be placed on the user. In RDBMSs the query subsystem automatically makes these decisions for the user based on query characteristics like skew, cardinality, sparsity, and density. In contrast, many specialized engines place this burden on the user because they are challenging to implement and these choices can improve query performance by over an order of magnitude.

**Physical independence** Building on the modern data warehouse, data processing architectures should support low-level optimizations, like indexes, that can opaquely speedup end-to-end query performance by orders of magnitude. User queries should be fast and should not break with the addition of new attributes. In addition, the next generation of data processing architectures should support complex queries that subselect a small number of attributes from a potentially huge fact table. RDBMSs accomplish all of this via a query subsystem that automatically makes decisions around how to layout data and operate over it based on characteristics like skew, cardinality, sparsity, and density. In specialized systems, one of these design points is selected and baked into the underlying architecture based the characteristics of the targeted domain. This makes it inflexible, slow, and often unusable in other domains.

**No Hairball Architectures** Provided that a substantial amount of the data today is stored in relational architectures, if one insists on using a specialized engines they are often forced to build an overlaying architecture that combines multiple systems. Unfortunately, it is inevitable that these systems will have impedance mismatches which means that unifying

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 6 | 7 | 8 |
| 9 | 7 | 8 |

(a) Relation R

| B | C | D |
|---|---|---|
| 2 | 3 | 4 |
| 2 | 3 | 5 |
| 7 | 8 | 10 |

(b) Relation S

| A | B | C | D |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 1 | 2 | 3 | 5 |
| 6 | 7 | 8 | 10 |
| 9 | 7 | 8 | 10 |

(c) Result R $\bowtie$ S

Figure 1.2: Example of a natural join over relations. Source: 'DATABASE SYSTEMS The Complete Book (2nd Edition)' Molina et al. [35].

them will require expensive data transformations (in addition to nasty glue code). Applications that span multiple domains should be efficient, and these data transformations, which can be more expensive than the actual computation [42], should be questioned. Even worse, this forces one to interact with several completely different systems, all with their own storage and computation models, which forms a massive headache for system architects to maintain. The result is a hairball architecture which, when contrasted with a single relational architecture, is much more difficult to manage and use.

## 1.3 Join Algorithms

The operator at the core of all relational architectures is the relational join operator. Interestingly, it was recently shown that traditional relational architectures are *provably suboptimal* on certain classes of join queries [73] due to their computation of joins in a pairwise fashion. In response, new multiway (worst-case optimal) join algorithms were recently proposed to address this suboptimality. Unlike pairwise join algorithms, these new multiway join algorithms were proven to obtain the correct asymptotic bound for any query [73]. Still, the practical benefit of these new join algorithms is highly questionable. Namely, worst-case optimal algorithms provide *no theoretical runtime advantage* over traditional join algorithms on most common workloads and are largely unexplored in practice.

**Natural Joins** Due to the structure of the aforementioned snowflake schemas, business intelligence queries often compute the product of multiple relations by pairing tuples where attributes common to multiple relations match in their values [35]. This operations is called the *natural join* operation (e.g. $R \bowtie S$ in Figure 1.2). For the remainder of this dissertation we will use the term natural join and join interchangeably. The join operation is the typically the most expensive operation of all relational join operators, and therefore has been a focus point of RDBMS designs since their inception. Interestingly, the natural join operator is also very expressive one; when combined with aggregations the natural join is capable of expressing common graph and linear algebra workloads.

**The Problem with Traditional Joins** Unfortunately, the query architectures found in relational architectures was recently shown to be suboptimal [74]. These architectures are suboptimal because they rely on pairwise join algorithms such as nested loop joins, hash joins, and sort-merge joins. Fundamental in each of these join algorithms operates is that relations are processed a pair at a time, with larger join queries being composed as a compositional series of these algorithms [35]. The reason pairwise join algorithms are suboptimal is because of the size of their worst-case output size is $O(N^2)$ which can be asymptotically suboptimal when this is the size of the intermediate output in more complex join queries. For example, the best known bound for the "triangle listing" join query, which is common in graph workloads [68, 72], is $O(N^{3/2})$ 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 what a specialized engines can achieve by a factor of $\sqrt{N}$.

**Multiway Joins** Recently, new multiway join algorithms were discovered that obtain the correct asymptotic bound for any graph pattern or join [73]—bringing hope to the idea that the relational model could be efficient in the domains where it has traditionally suffered. These new join algorithms are also called worst-case optimal join algorithms so we will use the terms worst-case optimal join and multiway join interchangeably throughout this dissertation. The multiway join algorithm we use is simple and works by compiling an arbitrary join query to a sequence of set intersection operations and for loops. This approach relies on 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. For cyclic queries that are common in graphs, any relational algebra based plan is provably slower by factors that depend on the size of the data. These factors can be large.

These new multiway join algorithms are by themselves not enough to close the gap. LogicBlox [11] 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 standard benchmarks. This leaves open the above question of whether these multiway joins are destined to be slower than specialized approaches and traditional relational architectures.

## 1.4 System Overview

We argue that an engine based on multiway join algorithms can compete with specialized engines and traditional RDBMSs, but it requires a novel architecture (see Figure 1.3). In this dissertation we present the EmptyHeaded engine which is a new multiway join engine. The EmptyHeaded architecture includes a novel query compiler based on *generalized hypertree decompositions* (GHDs) [22, 38], a code generator with the first cost-based optimizer for multiway join algorithms, and an execution engine designed to exploit the low-level layouts necessary to increase single-instruction multiple data (SIMD) parallelism. We

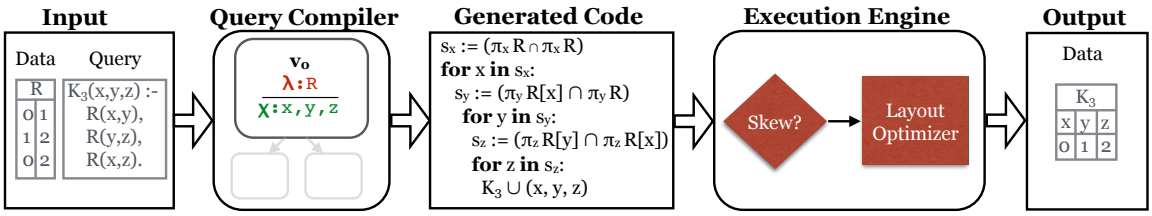



Figure 1.3: The EmptyHeaded engine works in three phases: (1) the query compiler translates a high-level datalog-like query into a logical query plan represented as a GHD (a hypertree with a single node here), replacing the traditional role of relational algebra; (2) code is generated for the execution 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.

argue that these techniques demonstrate that multiway join engines can compete with low-level graph engines, as our prototype is often faster than commodity pairwise join engines (in some cases by orders of magnitude) and competes with the best-of-breed specialized engines in each domain.

We design EmptyHeaded around tight theoretical guarantees, simple cost-based optimization techniques, 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 upon initial prototype developed in our group [101], we use *generalized hypertree decompositions* (GHDs) [38] for logical query plans in EmptyHeaded. GHDs allow one to apply classical query 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).[2] As these bounds depend on the data and the query it is difficult to expect users to write these algorithms in

[2]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.

a low-level framework. Our contribution is the design of a novel query optimizer and code generator, based on GHDs, that is able to provide the above guarantees with a high-level query language.

**New Database, Old Tricks** worst-case optimal join query optimizers need to select an attribute order [73] in a similar manner to how traditional query optimizers select a join order [35]. In EmptyHeaded's code generator, we present the first cost-based optimizer to select an attribute order for a worst-case optimal join algorithm. Because this optimizer is the first of its kind, its simplicity is crucial—our goal is to provide a simple but general foundation that can be easily applied, leveraged, and extended in any worst-case optimal join engine. Interestingly, we highlight that such an optimizer must follow heuristics that differ from what the conventional wisdom from pairwise join optimizers suggests (i.e. highest cardinality first) and describe how to leverage these heuristics to provide a simple but accurate cost-estimate. Using this cost-estimate, we validate that EmptyHeaded's cost-based optimizer selects attribute orders than can be up to 8815x faster than attribute orders that could be selected without such an optimizer.

**Exploiting SIMD: The Battle With Skew** Optimizing relational databases for the SIMD hardware trend has become an increasingly hot research topic [63, 85, 107], as the available SIMD parallelism has been doubling consistently in each processor generation.[3] 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 data is often 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

[3]The Intel Ivy Bridge architecture, which we use in this dissertation, 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.

| Type | Component | Optimization | Foundation | Graph | RDF | BI | LA |
|---|---|---|---|---|---|---|---|
| Physical | Execution Engine | Attribute Elimination | - | | | ✓ | |
| | | SIMD Layouts | WCOJ | ✓ | ✓ | ✓ | ✓ |
| | | ∩ Algorithms | WCOJ | ✓ | ✓ | ✓ | ✓ |
| | | Group By Algorithms | - | | | ✓ | ✓ |
| Logical | Query Compiler | Hypergraph Translation | GHD | ✓ | ✓ | ✓ | ✓ |
| | | Attribute Elimination | - | | | ✓ | |
| | | GHD Choice | GHD | ✓ | ✓ | ✓ | ✓ |
| | | Push Down Selections | GHD | ✓ | ✓ | ✓ | ✓ |
| | | Elim. Redundant Work | GHD | ✓ | ✓ | ✓ | ✓ |
| | | Pipelining | GHD | | ✓ | | |
| | | Cost-Based Optimizer | WCOJ | ✓ | ✓ | ✓ | ✓ |

Table 1.1: Core logical and physical optimizations of the EmptyHeaded architecture. In the table BI stands for business intelligence, LA stands for linear algebra, and WCOJ stands for worst-case optimal join.

optimizer that can increase 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.

## 1.5 Contributions

This dissertation presents the EmptyHeaded engine and demonstrates that its novel query processing architecture can outperform commodity pairwise RDBMS engines while remaining competitive with specialized engines in their own domains. To validate our claims we compare EmptyHeaded's performance to that of standard baseline engines on popular benchmark queries in the graph, RDF, linear algebra, and business intelligence domains. Our results show that EmptyHeaded can often outperform traditional relational engines by orders of magnitude while achieving competitive (<2x off) with specialized engines in each domain.

A summary of our contributions is as follows:

- We describe the first worst-case optimal join processing engine to use GHDs for

logical query plans. First, we describe how GHDs enable EmptyHeaded to provide a tighter theoretical guarantee than previous worst-case optimal join engines. Next, we validate that the optimizations GHDs can provide more than a three orders of magnitude performance advantage over previous worst-case optimal query plans.

- We describe the core physical optimizations that are necessary for a worst-case optimal join query architecture to be efficient. In particular we describe 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. We demonstrate that our automatic optimizers can result in up to a three orders of magnitude performance improvement on common benchmark queries. A complete list of the physical optimizations we present are shown in Table 1.1.

- We describe how to unify important logical optimizations in our novel query architecture. In particular, we describe how to map classic query optimizations like pushing down selections to our GHD-based query plans and present the first-cost based optimizer to select join orders for a worst-case optimal join algorithm. A complete list of the logical optimizations we present are shown in Table 1.1. We are the first to unify these logical optimizations in such a query processing architecture and show that they add up to a three order of magnitude performance increase across benchmark queries in multiple domains.

- Finally, we validate that our general purpose engine can outperform pairwise relational engines and compete with specialized engines on standard benchmarks in the graph, RDF, business intelligence, and linear algebra domains. 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. We show that EmptyHeaded can outperform other relational engines by *an order of magnitude* on standard business intelligence and linear algebra benchmarks while remaining on average within 31% of a best-of-breed solution within each benchmark. For the first time, this evaluation validates that a

worst-case optimal join query architecture can compete with specialized approaches on business intelligence, graph analytics, and linear algebra workloads. We argue that the inherent benefits of such a unified (relational) design has the potential to outweigh its minor performance overhead when compared to specialized engines.

An exciting possibility that stems from our design of a general join processing engine is that the undesirable "hairball" architectures that combine a large number of data processing engines can potentially be avoided. Low-level specialized engines [36, 43, 61] lead to these "hairball" architectures, as they introduce new non-relational programming models that create an impedance mismatch with traditional data processing architectures. These results suggest that it may be possible to have the performance benefits of specialized engines without the impedance mismatch; more precisely, it may be possible to eliminate the need for specialized accelerators with a new style of join processing engine.

## 1.6 Outline

In Chapter 2 we present the theoretical foundations of the EmptyHeaded query architecture. In particular we describe the motivation and principles behind building a join processing engine around worst-case optimal joins and GHDs. In Chapter 3 we present the core components and optimizations present in the EmptyHeaded architecture. In Chapter 4 we describe how EmptyHeaded captures queries in the graph, RDF, linear algebra, and business intelligence domains while also validating EmptyHeaded's performance against other relational engines and specialized engines in each domain. In Chapter 5 we provide a comparison of our work to the current state-of-the-art in each domain. Finally, we conclude with some with some final thoughts in Chapter 6.

# Chapter 2

# Foundations

We briefly describe the query language, data model, worst-case optimal join algorithm, generalized hypertree decompositions (GHDs), and code generation process at the foundation of the EmptyHeaded architecture. The material presented here serves as the building blocks for the remainder of the dissertation.

## 2.1 Query Language

The EmptyHeaded engine accepts queries written in either a SQL or datalog variant. Our query languages support conjunctive queries with aggregations and simple recursion. We choose a datalog based query language (similar to LogicBlox [11]) because it captures complex graph (join) queries in a natural and succinct manner. We also choose to expose a SQL based query language due to SQL's popularity and ease of use, especially on business intelligence queries. In table 2.1 we provide EmptyHeaded's syntax for representative queries in multiple domains that we will use as running examples throughout this dissertation. The syntax we use for every query each domain is presented in their respective sections in Chapter 4. For simplicity, from here on out, we present each application domain along with the input query language that we believe is the most natural fit. In the graph, RDF, and linear algebra domains we present queries in our datalog query language and in the business intelligence domain we present queries in our SQL query language. Because it is standard, we omit a more in-depth description of our SQL query language and

| Domain | Query | Language | Syntax |
|---|---|---|---|
| Graph | Triangle | Datalog | `Triangle(x,y,z):-R(x,y),S(y,z),T(x,z).` |
| Graph | Triangle | SQL | `%Graph schema: <rel_name>(src, dst)`<br>`SELECT R.src, R.dst, S.dst`<br>`FROM R, S, T`<br>`WHERE R.dst = S.src AND S.dst = T.dst AND`<br>`T.src = R.src` |
| Linear Alg. | Mat. Mult. | Datalog | `M3(i,j;z:float):-M1(i,k),M2(k,j),`<br>`z=<<SUM(k*j)>>.` |
| Linear Alg. | Mat. Mult. | SQL | `%Matrix schema: <rel_name>(i, j, v)`<br>`SELECT M1.i, M2.j, SUM(M1.v*M2.v)`<br>`FROM M1, M2`<br>`WHERE M1.j = M2.i`<br>`GROUP BY R.i, S.j` |
| Graph | Barbell | Datalog | `Barbell(x,y,z,x',y',z'):-R(x,y),S(y,z),`<br>`T(x,z), U(x,x'), R'(x',y'),`<br>`S'(y',z'),T'(x',z').` |
| Busin. Intel. | TPC-H Q5 | SQL | `SELECT n_name,`<br>`sum(l_extendedprice *`<br>`(1 - l_discount)) as rev`<br>`FROM customer, orders, lineitem,`<br>`supplier, nation, region`<br>`WHERE c_custkey = o_custkey`<br>`AND l_orderkey = o_orderkey`<br>`AND l_suppkey = s_suppkey`<br>`AND c_nationkey = s_nationkey`<br>`AND s_nationkey = n_nationkey`<br>`AND n_regionkey = r_regionkey`<br>`AND r_name = 'ASIA'`<br>`AND o_orderdate >=`<br>`date '1994-01-01'`<br>`AND o_orderdate <`<br>`date '1995-01-01'`<br>`GROUP BY n_name` |
| RDF | LUBM Q1 | Datalog | `out(x) :- takesCourse(x,'Univ0Course0'),`<br>`type(x,'GraduateStudent').` |
| Graph | PageRank | Datalog | `N(;w:int):-Edge(x,y); w=<<COUNT(x)>>.`<br>`PageRank(x;y:float):-Edge(x,z); y= 1/N.`<br>`PageRank(x;y:float)*[i=5]:-Edge(x,z),`<br>`PageRank(z),InvDeg(z);`<br>`y=0.15+0.85*<<SUM(z)>>.` |

Table 2.1: Example Queries in EmptyHeaded

spend the remainder of this section explaining our datalog based query language in more detail.

### 2.1.1 EmptyHeaded Datalog

We describe the core syntax for our datalog inspired language, which is sufficient to express the standard benchmarks we run in Chapter 4. Our datalog language has two non-standard extensions: aggregations and a limited form of recursion. We overview both extensions next and provide an in-depth example on the PageRank query.

**Conjunctive Queries: Joins, Projections, Selections** Equality joins are expressed in EmptyHeaded as simple conjunctive queries. We show EmptyHeaded's syntax for three simple conjunctive queries in Table 2.1: the 3-clique query (also known as triangle or $K_3$), the Barbell query (two 3-cliques connected by a path of length 1), and a simple RDF benchmark query. Table 2.1 shows that EmptyHeaded easily enables both selections and projections in its query language. 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. 'Univ0Course0' for the RDF query in Table 2.1).

**Aggregation** Following Green et al. [39], 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 (see Table 2.1). 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.

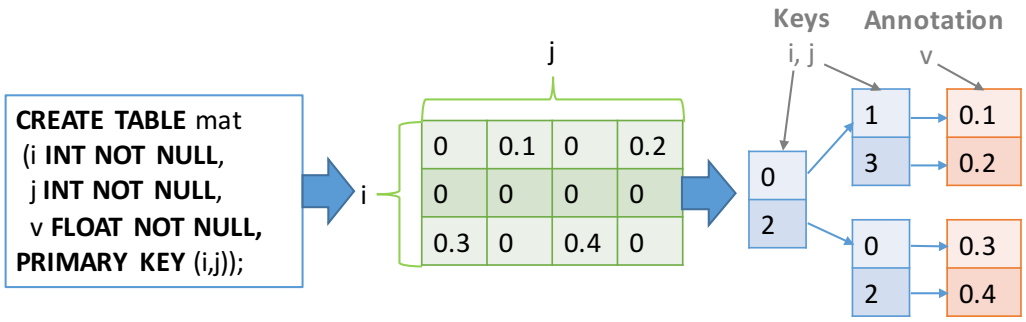
**Recursion** EmptyHeaded supports a simplified form of recursion similar to Kleene-star or transitive closure. Given an intensional or extensional relation $R$, one can write a Kleene-star rule like:

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

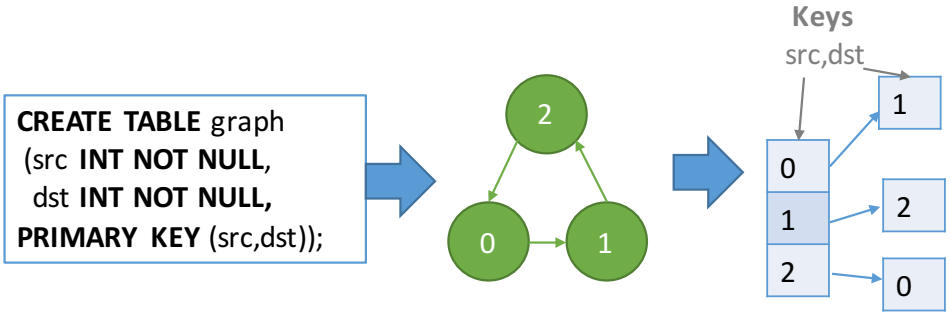
The rule $R*$ iteratively applies $q$ to the current instantiation of $R$ to generate new tuples which are added to $R$. It performs this iteration until (a) the relation doesn't change (a fixpoint semantic) or (b) a user-defined convergence criterion is satisfied (e.g. a number of iterations, `i=5`). An example that captures the familiar PageRank query is shown in Table 2.1.

To provide more insight into the non-standard aspects of our datalog query language, we illustrate how it works by example on the PageRank query:

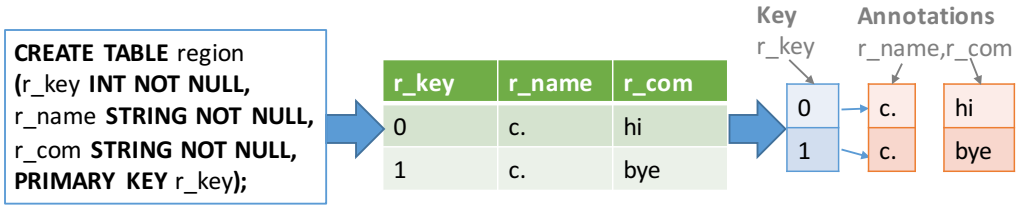
**Example 2.1.1.** Table 2.1 shows an example of the syntax used to express the PageRank query in EmptyHeded. The first line specifies that we aggregate over all the edges in the graph and count the number of source nodes assuming our $Edge$ relation is two-attribute relation filled with $(src, dst)$ 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 [47]. 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.




(a) Sparse matrix to a trie.




(b) Graph to a trie.




(c) TPC-H table to a trie.

Figure 2.1: Transformations of various input data types to a EmptyHeaded trie.

## 2.2 Data Model

The EmptyHeaded data model is relational with some minor restrictions. A core aspect of EmptyHeaded's data model is that attributes are classified as either *keys* or *annotations* via a user-defined schema. Keys in EmptyHeaded correspond to primary or foreign keys and are the only attributes which can partake in a join. Keys cannot be aggregated. Annotations are all other attributes and can be aggregated. Both keys and annotations support filter predicates and `GROUP BY` operations. In its current implementation, EmptyHeaded supports equality (=) filters on keys and range (>,<, =) filters on annotations. This represents the existing implementation, not fundamental restrictions. EmptyHeaded's current implementation supports attributes with types of `int`, `long`, `float`, `double`, and `string`. In many regards EmptyHeaded is similar to a key-value store combined with the relational model. EmptyHeaded is not unique in this regard, and commercial databases like Google's Mesa [41] and Spanner [28] follow a similar (if not identical) model.

EmptyHeaded's data model is tightly coupled with how it stores relations. All key attributes from a relation are stored in a trie, which serves as the only physical index in EmptyHeaded. Tries are multi-level data structures that are common in column stores and graph engines [43, 99]. In EmptyHeaded's trie, each level is composed of sets of dictionary encoded (unsigned integer) values. As is standard [6], EmptyHeaded stores dense sets using a bitset and sparse sets using unsigned integers. Annotations are stored in separate buffers attached to the trie. EmptyHeaded supports multiple annotations, and each can be reached from any level of the trie. An example EmptyHeaded trie is shown in Figure 2.1 where the dimensions `i` and `j` are keys and `v` is an annotation. EmptyHeaded tries can be thought of as an index on key attributes or a materialized view of the input table. We describe important details on EmptyHeaded's tries next.

**Trie Keys** The key attributes form the multi-level trie structure that serves as an index on our relation. As is standard [6, 36, 85], EmptyHeaded dictionary encodes all key data values to enable fast set intersections (the bottleneck operation in Algorithm 1) by leveraging single instruction multiple data parallelism and multiple data representations [6]. Additionally, the order of keys within a EmptyHeaded trie can affect query performance by over an

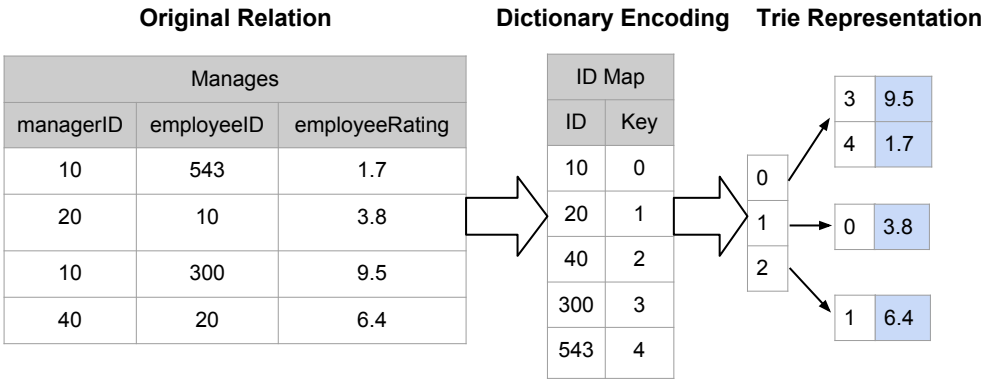



Figure 2.2: EmptyHeaded transformations from a table to trie representation using attribute order (*managerID,employerID*) and *employerID* attribute annotated with *employeeRating*.

order of magnitude. We describe a cost-based optimizer to select the order of the keys in a trie in Section 3.2.7.

**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* [39]. For example, a two-level trie annotated with a float value represents a sparse matrix or graph with edge properties (see Figure 2.1).

**Dictionary Encoding** The tries in EmptyHeaded currently support sets containing 32-bit values. As is standard [36, 85], 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. We explore the effect of various orderings on graph queries in Section 4.1.1.

**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

**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
```

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 3.1.

The complete transformation process from a standard relational table to the trie representation in EmptyHeaded is pictured in Figure 2.2.

## 2.3 Worst-Case Optimal Joins

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. [73] 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 3.2.1.

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 can be expressed as hypergraphs [38]. 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. In Section 3.2.2 we describe how to map more complex (and general) query patterns to hypergraphs. For now, we will go freely back and forth between the query and the hypergraph that represents it.

A recent result of Atserias, Grohe, and Marx [14] (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 $x \in R^{|E|}$ be a vector indexed by edges, i.e., with one component for each edge, such that $x \geq 0$; $x$ is a *feasible cover* (or simply feasible) for $H$ if

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

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

$$|out| \leq \prod_{e \in E} |R_e|^{x_e} \tag{2.1}$$

For a query $Q$, we denote $\mathsf{AGM}(Q)$ as the smallest such right-hand side.[1]

**Example 2.3.1.** 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 2.1, we know that $|out| \leq N^2$. 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 2.1 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 [73]. 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

[1] One can find the best bound, $\mathsf{AGM}(Q)$, in polynomial time: take the $\log$ of Eq. 2.1 and solve the linear program.

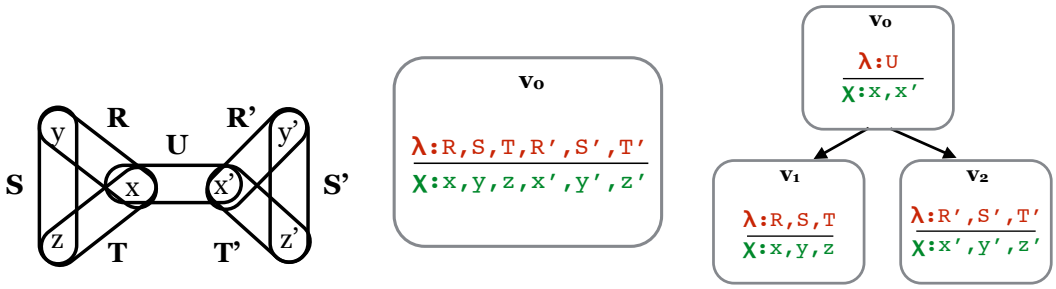
holds, a worst-case optimal running time for *any* join query is guaranteed. In fact, for *any* join query, its execution time can be upper bounded by $\mathsf{AGM}(Q)$. A simplified high-level description of the algorithm is presented in Algorithm 1. It was also shown that any pairwise join plan must be slower by asymptotic factors. However, we show in Section 2.4 that these optimality guarantees can be improved for non-worst-case data or more complex queries.

## 2.4 Generalized Hypertree Decompositions

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 multi-way 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. This section serves as foundation for Section 3.2 where we describe how to map arbitrary SQL queries to GHDs, select among GHDs that are equivalent based on the theoretical heuristics in this section, and optimize GHDs based on the characteristics of a given query. Therefore in this section we present the theoretical underpinnings of GHDs while in Sections 3.2.2 to 3.2.7 we describe the constant factor optimizations that are necessary to make them useful in practice.

### 2.4.1 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




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

Figure 2.3: 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 $\chi(v)$.

GHD is one goal of our optimizer.

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

**Example 2.4.1.** Figure 2.3a shows a hypergraph of the Barbell query introduced in Table 4.3. 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 2.3, 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 2.3b. 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 2.3c. 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 $O(N^{3/2}+|out|)$ 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 $N^{3/2}$ where $N$ is the size of the database. We describe how we execute this query plan in Section 2.5. This type of optimization is currently not available in the LogicBlox engine.

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

### 2.4.2 Formal Description

We describe GHDs and their advantages formally next.

**Definition 2.4.1.** 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 $T$;

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 2.3. 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"* [8] that says any attribute must form a connected subtree. The third property is redundant for us, as any GHD violating this condition is not considered (has infinite width which we describe next).

Using GHDs, we can define a non-trivial cardinality estimate based on the sizes of the relations. For a node $v$, define $Q_v$ as the query formed by joining the relations in $\lambda(v)$. The ***(fractional) width*** of a GHD is $\mathsf{AGM}(Q_v)$, 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 $O(N^w + |out|)$. 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 [104], which for acyclic queries enables EmptyHeaded to compute the output in time linear in the input size ($O(N^w)$) plus the output size ($|out|$).

## 2.5 Code Generation

Once a GHD is selected (via the process from Section 2.4 and Section 3.2), EmptyHeaded's code generator converts the chosen GHD into optimized C++ code that uses the operators in Table 2.2. 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 [70].

### 2.5.1 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

| | Operation | Description |
|---|---|---|
| Trie ($R$) | $R[t]$ | Returns the set matching the key tuple predicate $t$. The returned set is the matching key values at trie level $\|t\|$+1. |
| | $R(t)$ | Accessor to all annotations matching the key tuple predicate $t$. |
| | $R \leftarrow R \cup t$ | Appends key tuple $t$ to $R$. |
| 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 2.2: Core trie (and trie set) operations in EmptyHeaded.

a specified tuple predicate (see Table 2.2). 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 2.2.

The core operators supported by this trie-based storage model are shown in Table 2.2. These are the core operators needed for the worst-case optimal join algorithm which we present in the next section.

### 2.5.2 GHD Translation

The goal of code generation is to translate a GHD to the operations in Table 2.2. 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 2.5.1.** Consider the triangle query. The hypergraph is $V = \{X, Y, Z\}$ and $E = \{R, S, T\}$. In the first call, the loop body generates a loop with body Generic-Join( $\{Y, Z\}, E, t_X$). In turn, with two more calls this generates:

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

**Across Nodes** Recall Yannakakis' seminal algorithm [104]: we first perform a "bottom-up" pass, which is a reverse level-order traversal of $T$. For each $v \in V(T)$, the algorithm computes $Q_v$ and passes its results to the parent node. Between nodes $(v_0, v_1)$ we pass the relations projected onto the shared attributes $\chi(v_0) \cap \chi(v_1)$. Then, the result is constructed by walking the tree "top-down" and collecting each result.

**Recursion** EmptyHeaded supports both naive and semi-naive evaluation to handle recursion. For naive recursion, EmptyHeaded's optimizer produces a (potentially infinite) linear chain GHD with the output of one GHD node serving as the input to its parent GHD node. We run naive recursion for PageRank in Table 2.1. This boils down to a simple unrolling of the join algorithm. Naive recursion is not an acceptable solution in applications such as single source shortest paths (SSSP) where work is continually being eliminated. To detect when EmptyHeaded should run semi-naive recursion, we check if the aggregation is monotonically increasing or decreasing with a `MIN` or `MAX` operator. We use semi-naive recursion for SSSP.

**Example 2.5.2.** For the Barbell query (see Figure 2.3c), 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 $v_2$.

# Chapter 3

# Core Components

We describe the core components of the EmptyHeaded architecture which enable it to be efficient and general on workloads in multiple domains. In Section 3.1 we describe the core physical optimizations in the EmptyHeaded execution engine. These optimizations include a trie representation optimized for attribute elimination, an optimizer to select set layouts within the trie, an optimizer to select among SIMD set intersection algorithms, and an optimizer to select group by algorithms. In Section 3.2 we outline the core logical optimizations in the EmptyHeaded query compiler. These include EmptyHeaded's query language to hypergraph translation, GHD selection, attribute elimination, pushing down of selections, pipelining, and cost-based optimizer for worst-case optimal joins. All told these optimizations are the first of their kind for such a query architecture and enable over a three orders of magnitude performance improvement on queries in multiple domains.

## 3.1 Execution Engine

The EmptyHeaded execution engine runs code generated from the query compiler. The goal of the engine is to fully utilize SIMD parallelism, which is challenging because 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*

| Dataset | Nodes [M] | Dir. Edges [M] | Undir. Edges [M] | Density Skew | Card. Skew | Description |
|---|---|---|---|---|---|---|
| Google+ [59] | 0.11 | 13.7 | 12.2 | 1.17 | 1.17 | User network |
| Higgs [59] | 0.4 | 14.9 | 12.5 | 0.23 | 0.46 | Tweets about Higgs Boson |
| LiveJournal [60] | 4.8 | 68.5 | 43.4 | 0.09 | 0.97 | User network |
| Orkut [69] | 3.1 | 117.2 | 117.2 | 0.08 | 1.46 | User network |
| Patents [3] | 3.8 | 16.5 | 16.5 | 0.09 | 2.22 | Citation network |
| Twitter [56] | 41.7 | 1,468.4 | 757.8 | 0.12 | 0.07 | Follower network |

Table 3.1: Graph datasets that are used in the experiments.

*skew*.[1] Further, the cardinality of the data values is highly varied. We call this *cardinality skew*. A novel aspect of EmptyHeaded is that it automatically copes with both density and cardinality skew through optimizers that select among different data layouts, intersection algorithms, and `GROUP BY` algorithms to maximize SIMD parallelism. We use the graph datasets from Table 3.1 to validate our design decisions throughout this section because (as shown in Table 3.1) they nicely exhibit both types of skew.

Making these layout and algorithm choices is challenging, as the optimal choice depends both on characteristics of the data, such as density and cardinality, and characteristics of the query. In this section we describe how EmptyHeaded makes such decisions while providing validation for our decisions. We begin in Section 3.1.1 by describing how EmptyHeaded's trie representations are stored physically. The important optimization here is that our trie representation supports attribute elimination. Next, in Sections 3.1.2 and 3.1.3 we describe the set layouts (Section 3.1.2) and intersection algorithms (Section 3.1.3) that were tested in EmptyHeaded. This serves as necessary background for the tradeoff studies

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

and optimizers we present in Sections 3.1.4 and 3.1.5. In Section 3.1.4 we explore how to use different set intersection algorithms to cope with cardinality skew in the data. Based on this exploration, we present the simple optimizer EmptyHeaded uses to select among set intersect algorithms. In Section 3.1.5 we explore the proper granularity at which to make layout decisions to cope with density skew. Again, based on this exploration, we present the simple layout optimizer that EmptyHeaded uses to select among set layouts and show that it is close to an unachievable optimal. Finally, in Section 3.1.6 we present a simple optimizer to perform the `GROUP BY` operation in the present of skew on both key and annotation attributes.

### 3.1.1 A Trie Structure for Attribute Elimination

Attribute elimination is a crucial optimization from column store databases (or more generally struct-of-array transformations [32]) which is especially useful on queries where the majority of attributes are not used [27, 33]. Unsurprisingly, this is the case for many classic database queries, like those in the TPC-H benchmark. Therefore, in EmptyHeaded it was crucial to ensure that our trie representation was able to capture attribute elimination physically in order to preserve the benefits of a column store in a worst-case optimal design.

Although this elimination of unused attributes is obvious, ensuring this physically in the EmptyHeaded trie data structure was non-trivial. As a result, previous implementations of the worst-case optimal join algorithm [11] typically place either all or all but one attribute in a trie or trie-like representation. In EmptyHeaded we ensure that any number of levels from the trie can be used during query execution. This means that annotations can be reached individually from any level of the trie. Further, the annotations are all stored in individual data buffers (like a column store) to ensure that they can be loaded in isolation. Next, we provide more details on the trie data structure EmptyHeaded uses to physically eliminate unnecessary attributes from queries.

**Trie Data Structure** To support efficient attribute elimination, the EmptyHeaded storage engine needs the ability to dynamically process only certain attributes (from disk load to query execution). The EmptyHeaded engine accomplishes this by separating the storage

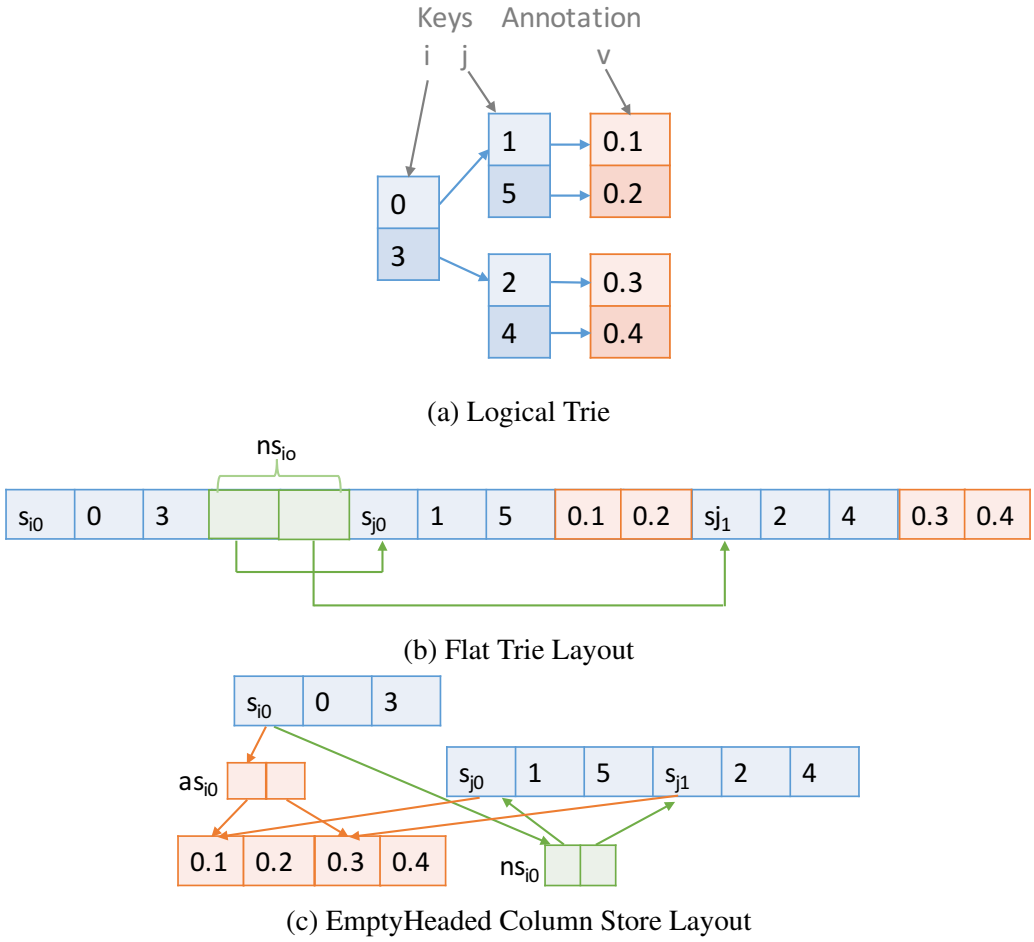



Figure 3.1: Two possible physical layouts of data in a trie. The flat layout contains one physical array and the column store layout contains 5 physical arrays. $s_*$ denotes the meta data for a set (e.g. cardinality), $ns_*$ denotes the buffer indexes for the next level of a trie, and $as_*$ denotes the meta data for annotations attached to all sets except the last level, where the mapping is 1-1 and explicit.

| | EmptyHeaded | Fixed Sized Trie | HyPer |
|---|---|---|---|
| TPC-H Q1 Memory | 25GB | 140GB | 161GB |
| TPC-H Q1 Loading | 1.35s | 140.96s | 490s |
| TPC-H Q1 Compute | 1.10s | 39.81s | 0.678s |
| PageRank Compute | 0.61s | 0.51s | 3.17s |

Table 3.2: Memory usage in gigabytes, loading time of database binaries from disk (with warm file system caches) in seconds, and compute time in seconds for the EmptyHeaded engine, EmptyHeaded with a fixed sized trie instead of variable sized trie, and HyPer database engine. TPC-H query 1 is run at scale factor 100 and the PageRank query is run over the LiveJournal dataset.

of every attribute—each is placed in different physical buffers (see Figure 3.1c). For annotations, this results in a column store being attached to the trie (in Figure 3.1c this is the lowest red buffer containing floating point values). For keys this means that each level of the trie is placed in separate buffers with references to the associated annotations for each level, except for the final level where the mapping is implicit (again shown in Figure 3.1c). These references allow EmptyHeaded to skip over unused levels of the trie to access annotations and are the $asi_0$ blocks in Figure 3.1c. Note, this not the optimal design (considering locality) for queries that touch every attribute, but can be useful on queries that touch a small number of keys.

**Empirical Advantages** This physical data layout enables EmptyHeaded to have variable sized tries that load and touch only the key and annotation attributes needed for a particular query. A nice side effect of such a design is that EmptyHeaded can process queries that do not touch join keys (no join) in the exact same manner as a column store (e.g. TPC-H Q1). We show in Table 3.2 that this optimization enables EmptyHeaded to load data an order of magnitude faster[2], consume 115GB less memory, and compute the query 18x faster than the EmptyHeaded design with a fixed sized trie on TPC-H Q1 at scale factor 100. In addition, we show that our approach has advantages over the HyPer design which requires

[2]Machine has a cached read bandwidth of 8227.42 MB/sec and buffered read bandwidth of 441.44 MB/sec measured from `hdparm`.

the entire database to be loaded from disk prior to executing a query. This optimization does not come at zero cost though. As alluded to, it is not the optimal choice for some queries which touch all attributes, like the PageRank query. As such, in Table 3.2 we show that this design can be 19% slower than the EmptyHeaded design on the PageRank query. Still, we believe the benefits of such a general design outweigh its slight cost.

### 3.1.2 Set Layouts

In addition to considering the physical layouts of the trie in EmptyHeaded it is crucial to consider the physical layouts of the sets within the tries. In this section we describe the set layouts that were tested in the EmptyHeaded engine and how their associated values are stored. These layouts are necessary to understand the SIMD set intersection algorithms presented in Section 3.1.3 and are at the core of our study in Section 3.1.5, where we use these layouts to exploit SIMD parallelism in the presence of density skew. In Section 3.1.5 we show that a combination of a simple 32-bit unsigned integer (`uint`) layout and a simple bit vector layout (`bitset`) yields the highest performance in our experiments. For dense data, the `bitset` layout makes it trivial to take advantage of SIMD parallelism but causes a quadratic blowup in memory usage for sparse data. The `uint` layout represents sparse data efficiently but makes extracting SIMD parallelism challenging.

In the following, we describe the `bitset` layout in EmptyHeaded and three additional set layouts that we tested: `pshort`, `varint`, and `bitpacked`. The `pshort` layout groups values with a common upper 16-bit prefix together and stores each prefix only once. The `varint` and `bitpacked` layouts use difference encoding[3] for compression and have been shown to both compress better and be up to an order of magnitude faster than compression tools such as LZO, Google Snappy, FastLZ, LZ4 or gzip [58]. As such, all layouts in this section (potentially) enable compression of the data, which is of interest as data compression has been shown to sometimes increase overall query performance by decreasing the memory bandwidth in applications [96]. Although interesting, we are

[3]Difference encoding encodes the difference between successive values in a sorted list of values $(x_1, \delta_2 = x_2 - x_1, \delta_3 = x_3 - x_2, \dots)$ instead of the original values $(x_1, x_2, x_3, \dots)$. The original array can be reconstructed by computing prefix sums $(x_i = x_1 + \sum_{n=2}^{i} x_n)$. The benefit of this approach is that the differences are always smaller than the original values, allowing for more aggressive compression.

not concerned with reducing memory usage—main memory sizes are consistently increasing and persistent storage is plentiful. Most importantly, some of these layouts (namely `bitset` and `pshort`) are well-suited for SIMD parallelism, enabling EmptyHeaded to potentially achieve higher computational performance in the generic-worst case optimal join algorithm. Note that we omit a description of the `uint` layout as it is just an array of sorted 32-bit unsigned integers. Finally, we conclude with a brief discussion of how associated data values are added to these set layouts.

**`bitset`** The `bitset` layout stores a set of pairs (offset, bit vector). Each offset stores the index of the smallest value in the corresponding bit vector. Thus, the layout is a compromise between sparse and dense layouts. We refer to the number of bits in the bitvector as the *block size*. EmptyHeaded supports block sizes that are powers of two with a default of 256.[4] As shown, we pack the offsets contiguously, which allows us to regard the offsets as a `uint` layout; in turn, this allows EmptyHeaded to use the same algorithm to intersect the offsets as it does for the `uint` layout. An example of the `bitset` layout that contains $n$ blocks and a sequence of offsets ($o_1$–$o_n$) and blocks ($b_1$–$b_n$) is shown below. The offsets store the start offset for values in the bitvector.

| $n$ | $o_1$ | ... | $o_n$ | $b_1$ | ... | $b_n$ |
|---|---|---|---|---|---|---|

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

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

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

[4] The width of an AVX register.

**varint** The `varint` layout uses variable byte encoding, which is a popular technique first proposed by Thiel and Heaps in 1972 [57]. The `varint` layout encodes the differences between data values into units of bytes where the lower 7 bits store the data and the 8th-bit indicates whether the data extends to another byte or not. The decoding procedure reads bytes sequentially. If the 8th bit is 0 it outputs the data value and if the 8th bit is 1 the decoder appends the data from this byte to the output data value and moves on to the next byte. This layout is simple to implement and reasonably efficient [57]. Below we show an example of the `varint` layout:

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

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

**bitpacked** The `bitpacked` layout partitions a set into blocks and compresses them individually. First, the layout determines the maximum bits of entropy of the values in each block $b$ and then encodes each value of the block using $b$ bits. Previous work in the literature [58] showed that this technique can be adapted to encode and decode values efficiently by packing and unpacking values at the granularity of SIMD registers rather than each value individually. Although Lemire et al. propose several variations of the layout, we chose to implement the `bitpacked` with the fastest encoding and decoding algorithms at the cost of a worse compression ratio. Thus, instead of computing and packing the deltas sequentially, the `bitpacked` layout computes deltas at the granularity of a SIMD register:

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

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

below.

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

| 0 31 | 32 39 | 40 42 | 43 45 | 46 48 |
|---|---|---|---|---|
| $\|S\|$ | bits/elem | $\delta_1$[2..0] | $\delta_2$[2..0] | $\delta_3$[2..0] |
| 3 | 3 | 0 | 2 | 6 |

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

### 3.1.3 Set Intersections

We present an overview of the intersection algorithms EmptyHeaded uses for each layout. This serves as the background for our cardinality skew study and set intersection algorithm optimizer in Section 3.1.4. We remind the reader that the *min property* presented in Section 2.3 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:

- **SIMDShuffling** iterates through both sets block-wise and compares blocks of values using SIMD shuffles and comparisons [49].
- **V1** iterates through the smaller set one-by-one and checks each value against a block of values in the larger set using SIMD comparisons [58].

- **V3** is similar to V1 but performs a binary search on four blocks of data in the larger set (each the size of a SIMD register) to identify potential matches [58].
- **SIMDGalloping** is similar to V1 but performs a scalar binary search in the larger set to find a block of data with a potential match and then uses SIMD comparisons [58].
- **BMiss** uses SIMD instructions to compare parts of blocks of values and filter potential matches then uses scalar comparisons to check the full values of the partial matches [45].

For `uint` intersections we found that the size of two sets being intersected may be drastically different. This is *cardinality skew*. So-called *galloping* algorithms [102] allow one to run in time proportional to the size of the smaller set, which copes with cardinality skew. However, for sets that are of similar size, galloping algorithms may have additional overhead. We empirically show this in Section 3.1.4.

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

**`uint` ∩ `bitset`** To compute the intersection between a `uint` and a `bitset`, we first intersect the `uint` values with the offsets in the `bitset`. We do this to check if it is possible that some value in a `bitset` block matches a `uint` value. As `bitset` block sizes are powers of two in EmptyHeaded, this can be accomplished by masking out the lower bits of each `uint` value in the comparison. This check may result in false positives, so, for each matching `uint` and `bitset` block we check whether the corresponding `bitset` blocks contain the `uint` value by probing the block. We store the result in a `uint` layout as the intersection of two sets can be at most as dense as the sparser set.[5] Notice that this

[5]Estimating data characteristics like output cardinality a priori is a hard problem [21] and we found it is too costly to reinspect the data after each operation.

algorithm satisfies the min property with a constant determined by the block size.

**pshort ∩ pshort** The pshort intersection uses a set intersection algorithm previously proposed in the literature [93]. This algorithm depends on the range of the data and therefore does not preserve the min property, but can process more elements per cycle than the SIMDShuffling algorithm. The pshort intersection uses the x86 STNII (String and Text processing New Instruction) comparison instruction allowing for a full comparison of 8 shorts, with a common 16-bit prefix, in one cycle. The pshort representation also enables jumps over chunks that do not share a common 16-bit prefix.

**uint ∩ pshort** For the uint and pshort set intersection we again take advantage of the STNII SIMD instruction. We compare the upper 16-bit prefixes of the values and shuffle the uint representation if there is a match. Next, we compare the lower 16-bits of each set, 8 elements at a time using the STNII instruction.

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

### 3.1.4 Cardinality Skew

The skew in the degree distribution of data causes set intersections to operate on sets with different cardinalities. The most interesting tradeoff space for cardinality skew is exploring the performance of various uint intersection algorithms on sets of different sizes. Therefore, in this section we compare the five different SIMD algorithms for uint set intersections from Section 3.1.3 and present a simple optimizer to select among them based on our results.xx

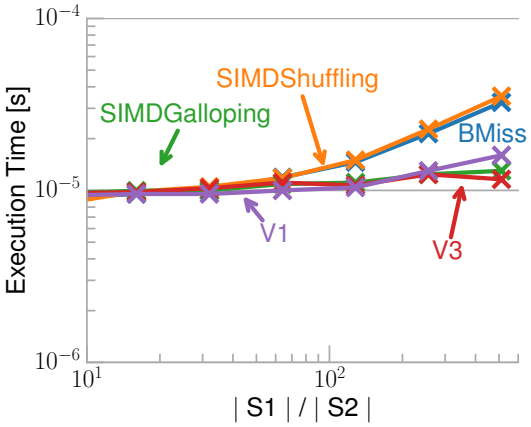



Figure 3.2: Intersection time of `uint` intersection algorithms for different ratios of set cardinalities.

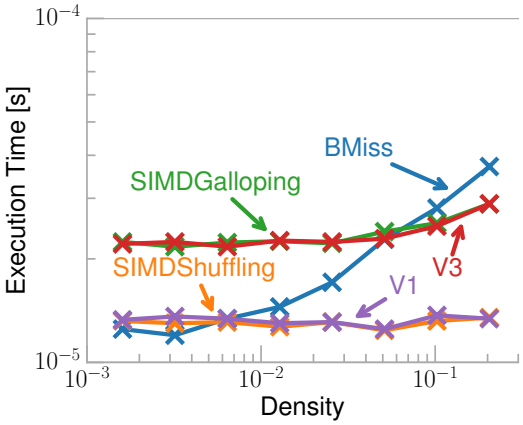



Figure 3.3: Intersection time of `uint` intersection algorithms for different densities.

### Tradeoffs

To test cardinality skew we set the range of the sets to 1M and set the cardinality of one set to 64 while changing the cardinality of the other set. Confirming the findings of others [45, 49, 58, 93], we find that SIMDGalloping and V3 algorithms outperform other intersection algorithms by more than 5x with a crossover point at a cardinality ratio of 1:32. Figure 3.2 shows that the SIMDGalloping and V3 algorithm outperform all other algorithms when the cardinality difference between the two sets becomes large. In contrast to the other algorithms, SIMDGalloping runs in time proportional to the size of the smaller set. Thus, SIMDGalloping is more efficient when the cardinalities of the sets are different.

We also vary the range of numbers that we place in a set from 10K-1.2M while fixing the cardinality at 2048. Figure 3.3 shows the execution time for sets of a fixed cardinality with varying ranges of numbers. BMiss is up to 5x slower when the sets have a small range and a high output cardinality. When the range of values is large and the output cardinality is small the algorithm outperforms all other algorithms by up to 20%. Figure 3.3 shows that the V1 and SIMDShuflling algorithms outperform all other algorithms, by over 2x, when the sets have a low density.

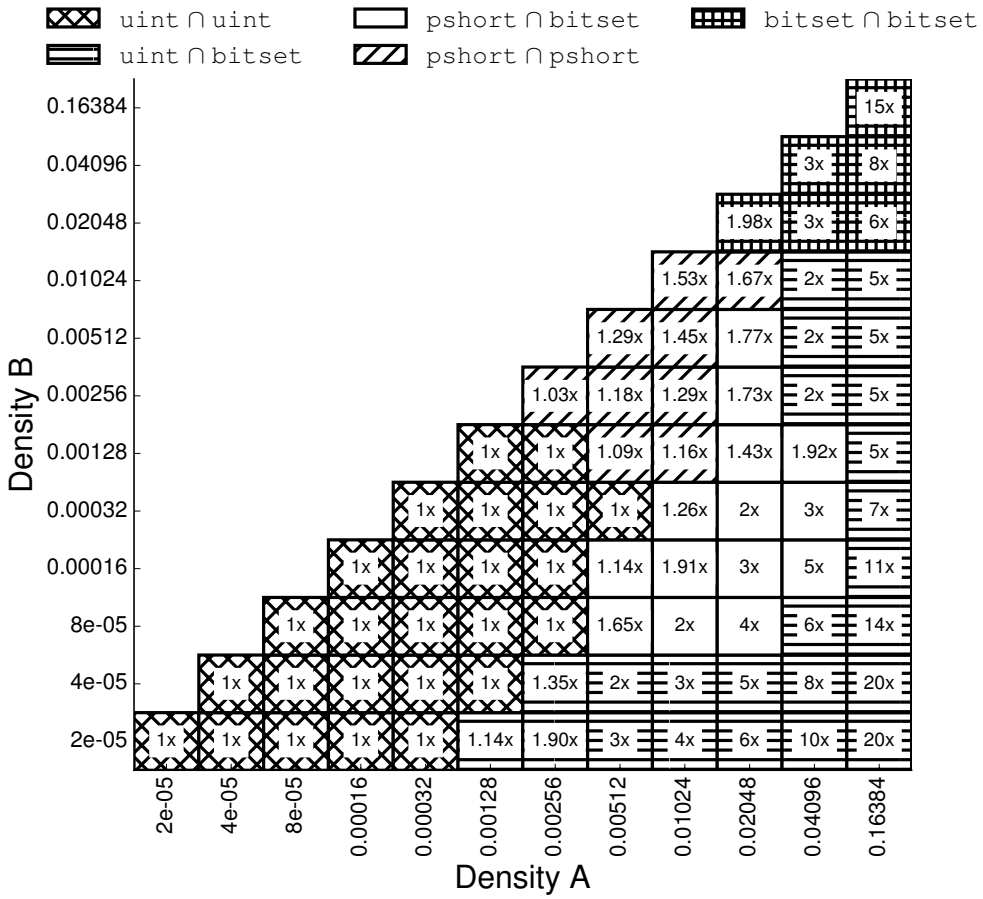



Figure 3.4: Best performing layouts for set intersections with relative performance over `uint`.

### `uint` ∩ `uint` Algorithm Optimizer

We find that no one algorithm dominates the others, so EmptyHeaded switches dynamically between `uint` algorithms. Based on these results, we select the SIMDShuffling algorithm by default but when the ratio between the cardinality of the two sets was greater than 1:32, like others [45, 58], we select the SIMDGalloping algorithm. Because the sets in data are typically sparse, we found the impact of selecting SIMDGalloping on datasets to be minimal, often under a 5% total performance impact. Still we use this simple optimizer to select among these two `uint` intersection algorithms in EmptyHeaded.

### 3.1.5 Density Skew

In this section, we present our study that serves as the foundation for the layout optimizer in EmptyHeaded. To perform this study we considered all set intersection algorithm and layout combinations from Sections 3.1.2 and 3.1.3. We first show in Section 3.1.5 that using layouts and associated algorithm combinations other than those on the `uint` and `bitset` layouts resulted in no significant performance advantage. Next, in Section 3.1.5 we present our tradeoff study for making layout choices between the `uint` and `bitset` representations at three different granularities: the relation level, the set level, and the block level. We evaluate this tradeoff space and show that the set level is best choice. We finish in Section 3.1.5 by presenting the simple set layout optimizer that EmptyHeaded uses to cope with density skew.

#### Eliminating Complexity

Figure 3.4 displays the best performing layout combinations and their increased relative performance compared to the best performing `uint` algorithm while changing the density of the input sets in a fixed range of 1M. Unsurprisingly, the `varint` and `bitpacked` representations never achieve the best performance. In fact, on real data, we found the `varint` and `bitpacked` types typically perform the triangle counting query 2x slower due the decoding step not outweighing the slight memory bandwidth decrease.[6] Finally, our experiments on synthetic data show only moderate performance gains from using the `pshort` layout and on real data we found that it is rarely a good choice for a set in combination with other representations. Even worse these layouts are all expensive to build when compared with the simple `uint` layout (see Table 3.3). Based on these results, we focus on only exploiting the tradeoffs for the `uint` and `bitset` intersections and layouts for the remainder of this section.

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

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

Table 3.3: Construction times in seconds.

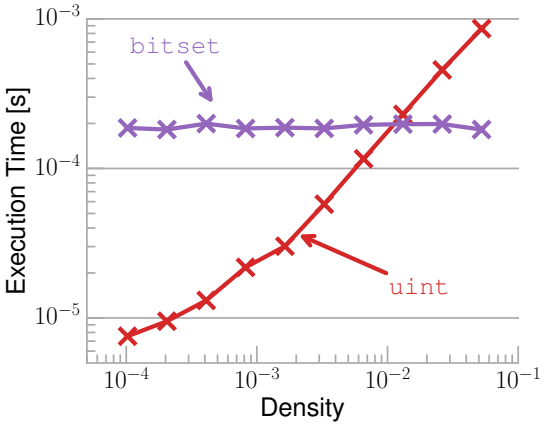



Figure 3.5: Intersection time of `uint` and `bitset` layouts for different densities.

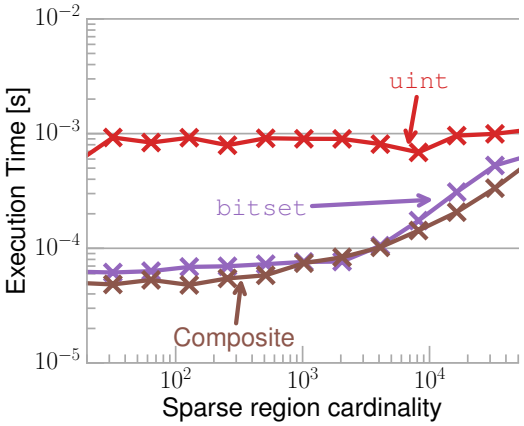



Figure 3.6: Intersection time of layouts for sets with different densities in a region.

### Tradeoffs

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

**Relation Level** Set layout decisions at the relation level force the data in all relations to be stored using the same layout and therefore do not address density skew. The simplest layout in memory is to store all sets in every trie using the `uint` layout. Unfortunately, it is difficult to fully exploit SIMD parallelism using this layout, as only four elements fit in a single SIMD register.[7] In contrast, the `bitset` layout can store up to 256 elements

[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.

in a single SIMD register. However, the `bitset` layout is inefficient on sparse data and can result in a quadratic blowup of memory usage. Therefore, one would expect `uint` to be well suited for sparse sets and `bitset` for dense sets. Figure 3.5 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 Chapter 4 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 in a single `uint` region and dense blocks using the `bitset` layout in a single `bitset` region. Conceptually this is similar to our `bitset`, layout but now when encoding a set using the composite set, the system checks the density of each block(again with a default size of 256) and decides in which region to store it. To benchmark the performance of our composite type, we generate sets with internal skew by having two regions: (1) a region with a fixed range of 5M. We change the density of this region by varying the cardinality from 16-131K, and (2) a dense region with 500K consecutive values. We show in Figure 3.6 that our composite layout outperforms the `bitset` by up to 2x when the sparse region is sparse. As the sparse region gets denser, the performance gap between the `bitset` and composite layouts increases. The `uint` type is not competitive in the range of data we present because the dense region is best represented using the `bitset` representation.

| 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 3.4: Relative time of the level optimizers on triangle counting compared to the oracle.

### Evaluation

We introduce the concept of an oracle optimizer to properly evaluate our representation decisions by providing a lower bound for our overall query runtime. We use a comparison to the oracle optimizer and a study of the overheads associated with making decisions at the set and block level to validate that EmptyHeaded should make decisions between the `uint` and `bitset` layouts at the set level. We finish by presenting our set level optimizer which is used for all the experiments in Chapter 4.

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

To determine if our system should make representation decisions at a relation, set, or block level we compare each approach on the triangle counting query to the time of the oracle optimizer. We found that on real data choosing representations at a set level provided the best overall performance. Table 3.4 demonstrates that choosing at the set level is at most 1.6x off of the optimal performance. Choosing at the relation and block levels can be up to 7.3x and 3.2x slower than the oracle, respectively. Representation decisions at the

relation level do not optimize at all for density skew and therefore are the least robust across datasets. Representation decisions at the block level are fine-grained and compensate for density skew but are too fine grained on the datasets we consider and do not outweigh their increased overhead. Real data often has a high density skew across sets making the middle level set optimizer perform the most robust across the datasets we consider.

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

To evaluate the overhead of optimizers at the block and set levels, we modify both optimizers to always pick the `uint` representation and compare the execution time for these optimizers to relation level selection of a `uint` (no overhead). Table 3.5 shows the relative overhead of both optimizers across different datasets on the triangle counting query. The overhead ranges from 1%-10% of the total runtime for our set level optimizer and from 5%-25% for our block level optimizer. The amount of overhead we pay for each dataset is linked to its size and density skew as these are the two factors that can amortize this overhead. For example, the small Patents dataset with a low density skew of 0.09 consistently has the highest overhead at each level. The block level optimizer overhead is more pronounced on data due to the fact that the majority of sets are either extremely sparse or extremely dense. Thus, the sets do not contain a high enough level of internal skew to outweigh the cost making such fine grained decisions when compared to the set level optimizer. Thus, set level representation comes at a lower cost than block level decisions and enable larger performance gains than both the relation level and block level decisions.

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

Table 3.5: Set level and block level optimizer overheads on triangle counting.

#### Set Layout Optimizer

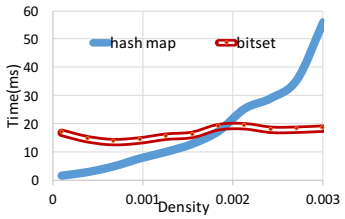
Based on the results of our tradeoff study, we present the simple optimizer used in the EmptyHeaded engine to automatically select between the `bitset` and `uint` at the set level. The set optimizer in EmptyHeaded selects the layout for a set in isolation based on its cardinality and range. It selects the `bitset` layout when each value in the set consumes at most as much space as a SIMD (AVX) register and the `uint` layout otherwise. The optimizer uses the `bitset` layout with a block size equal to the range of the data in the set. We find this to be more effective than a fixed block size since it lacks the overhead of storing multiple offsets which is not necessary when the entire set is stored as a `bitset`. Our simple optimizer is shown in Algorithm 2.

**Algorithm 2** Set layout optimizer

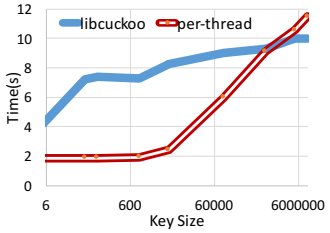
```
def get_layout_type(S):
  inverse_density = S.range/|S|
  if inverse_density < SIMD_register_size:
    return bitset
  else:
    return uint
```

### 3.1.6 Group By Tradeoffs

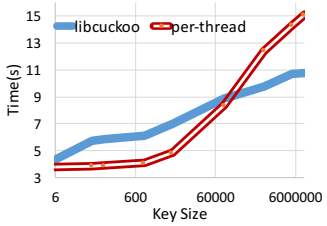
Designing a skew-resistant `GROUP BY` operator requires careful consideration of the tradeoffs associated with its implementation. In this section we present these tradeoffs in the




(a) The performance of EmptyHeaded's hash map and bitset `GROUP BY` operators for key values for various output set densities. The range of output values is fixed to [0,1e6].




(b) The performance of 1e7 parallel upsert operations for libcuckoo and a collection of per-thread hash maps with different output hash map key sizes. The `GROUP BY` is on a single `uint` attribute. 56-threads are used.




(c) The performance of 1e7 parallel upsert operations for libcuckoo and a collection of per-thread hash maps with different output hash map key sizes. The `GROUP BY` is on 2 `uint` and 4 `double` attributes. 56-threads are used.

Figure 3.7: The performance of EmptyHeaded's `GROUP BY` implementations on key and annotation attributes.

context of EmptyHeaded. Although many `GROUP BY`s are automatically captured in EmptyHeaded's tries, two classes of `GROUP BY`s require an additional implementation in EmptyHeaded: (1) a `GROUP BY` to union keys in the attribute orders from Section 3.2.7 and (2) a `GROUP BY` on annotations. In this section we present the tradeoffs around two implementations for each type of `GROUP BY` and describe a simple optimizer that automatically exploits these tradeoffs to select the implementation used during execution. In Sections 4.3 and 4.4, we show that this `GROUP BY` optimizer provides up to a 875x and 185x speedup over using a single `GROUP BY` implementation on business intelligence and linear algebra queries respectively.

**GROUP BY Key** In Section 3.2.7 we will show that sometimes EmptyHeaded selects key attributes orders where one projected away key appears before the materialized keys. In such cases a `GROUP BY` on a key attribute is needed to union the result. EmptyHeaded selects between two implementations for this: (1) a hash map that upserts (key, value) pairs as they are encountered and (2) a bitset that unions (OR's) keys and a dense array to hold the values. Figure 3.7a shows that the performance of these implementations depends greatly on the density of the output key attribute set. The bitset performs better when the output

density is high and the hash map performs better when the output density is low. To predict the output set's density EmptyHeaded leverages a simple observation: the output density is correlated in a 1-1 manner with the density of projected away attribute (set being looped over). As such, EmptyHeaded uses the density of the projected away attribute to predict the output set's density and select the implementation used.

**GROUP BY Annotation** EmptyHeaded supports SQL queries with `GROUP BY` operations on annotations or on both annotations and keys. To support this in parallel, EmptyHeaded selects between two implementations of a parallel hash map: (1) libcuckoo [62] and (2) a per-thread instance of the C++ standard library unordered map. In the context of a `GROUP BY`, the crucial operation for these hash maps is an upsert and we found that each implementation worked best under different conditions. Figures 3.7b and 3.7c show that when the output key size is small (many collisions), the per-thread implementation outperforms libcuckoo by up to 4x. In contrast, libuckoo outperforms the per-thread approach by an order of magnitude when the output size is large (few collisions). Unfortunately, predicting output cardinalities is a difficult problem [92], so EmptyHeaded leverages a general observation to avoid this when selecting between these two approaches: libcuckoo is at worst close ($<$2x) to the performance of the per-thread approach when the hash-map key tuple is wide ($>$3 values), and the per-thread approach is at worst close ($<$2x) to the performance of libcuckoo when the hash map key is small ($<$3 values). This trend is shown in Figures 3.7b and 3.7c. Thus, the EmptyHeaded optimizer selects a per-thread hash map when the hash map key is a tuple of three elements or less and libcuckoo otherwise.

## 3.2 Query Compiler

The query compilation techniques presented in Section 2.4 are able to capture a wide range of domains, including linear algebra, message passing, and graph queries [6, 7, 48]. However, most of this work has been theoretical and none of the current literature demonstrates how to capture general SQL-style queries in such a framework. In this section we show how to extend this theoretical work to more complex queries. In particular we describe how EmptyHeaded translates general query patterns to hypergraphs in Section 3.2.2. Using

this translation, we describe the importance of this process logically capturing well-known optimization of attribute elimination. In Section 3.2.3 we describe the manner in which EmptyHeaded selects a GHD from theoretically equivalent GHDs and we describe how EmptyHeaded optimizes it by pushing down selections (Section 3.2.4), eliminating redundant work (Section 3.2.5), and pipelining work between nodes (Section 3.2.6). Finally, in Section 3.2.7 we describe the first cost-based optimizer for a worst-case optimal join algorithm. We argue that with these extensions EmptyHeaded represents the first practical implementation of these techniques capable of capturing general query workloads.

### 3.2.1 Capturing Aggregate-Join Queries

Aggregate-join queries are common in many workloads. As such, previous work has investigated aggregations over hypertree decompositions [38, 78]. EmptyHeaded adopts this previous work in a straightforward way. To do this, we add attributes with "semiring annotations" following Green et al. [39]. EmptyHeaded simply manipulates these values as they are 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.

In more detail, to capture aggregate-join queries EmptyHeaded uses the AJAR framework [48]. Despite that others [52] have argued that custom algorithms are necessary to capture such queries, AJAR extends the theoretical results of GHDs to queries with aggregations. AJAR does this by associating each tuple in a one-to-one mapping with an *annotation*. Aggregated annotations are members of a *commutative semiring*, which is equipped with product and sum operators that satisfy a set of properties (identity and annihilation, associativity, commutativity, and distributivity). Therefore, when relations are joined, the annotations on each relation are multiplied together to form the annotation on the result. Aggregations are expressed by an *aggregation ordering* $\alpha = (\alpha_1, \oplus_1), (\alpha_2, \oplus_2), \ldots$ of attributes and operators.

Using AJAR, EmptyHeaded picks a query plan with the best worst-case guarantee by going through three phases.

1. It breaks the input query into characteristic hypergraphs, which are subgraphs of

the input that can be decomposed to optimal GHDs. This process is the focus of Section 3.2.2.

2. A list of optimal GHDs are computed. The optimal GHD ensures tighter theoretical run time guarantees. As such, it is key that the EmptyHeaded optimizer selects a GHD with the smallest FHW (see Section 2.4) 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 FHW. 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, we simply brute force search GHDs of all possible widths. The algorithm EmptyHeaded uses for performing this brute force search is shown in Algorithm 3. The algorithm computes a list of GHDs by recursively computing the GHDs for all subsets of the edge set. To avoid unnecessary intermediate results, EmptyHeaded compresses all final GHDs with a FHW of 1 into a single node, as the query plans here are always equivalent to running just the generic worst-case optimal join algorithm.

3. A GHD and a global attribute order is selected from the list of optimal GHDs. The heuristics to select a GHD from the optimal GHDs are described in Section 3.2.3

4. The selected GHD undergoes a series of constant-factor optimizations and transformations. The constant-factor optimizations that EmptyHeaded applies to the selected GHD are described in Sections 3.2.4, 3.2.6 and 3.2.7.

The core stages of EmptyHeaded's query compilation process are shown in Figure 3.8 on TPC-H query 5 and a matrix multiplication query.

### 3.2.2 Query Language to Hypergraph Translation

The process described in Section 3.2.1 does not describe how to translate arbitrary queries to the query hypergraphs introduced in Section 2.3. In this section, we describe how certain general query features, such as complex expressions inside aggregation functions, can be

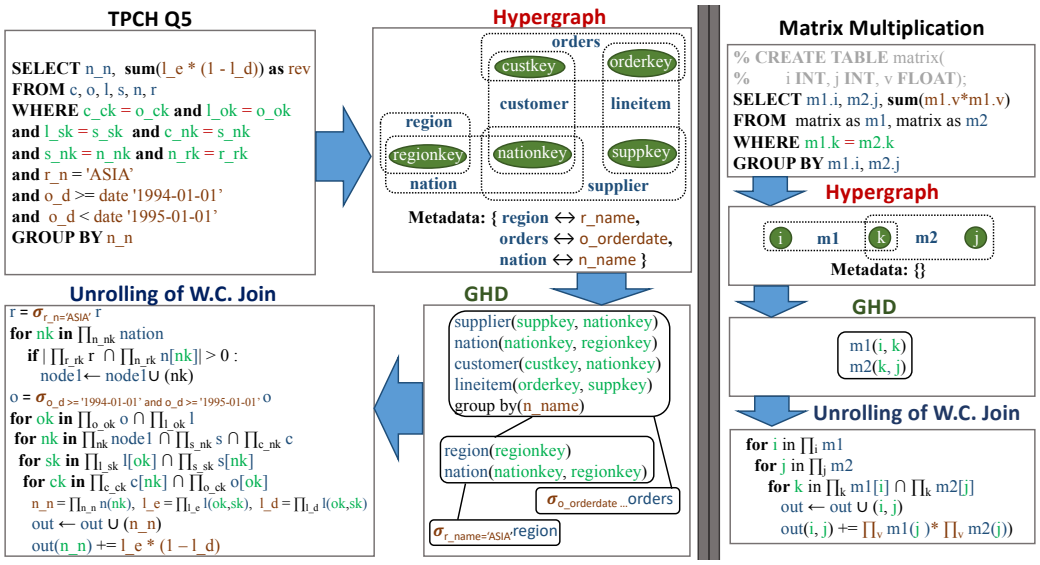



Figure 3.8: Illustration of the core stages in EmptyHeaded's query compiler on TPC-H query 5 and matrix multiplication. The queries are expressed in SQL, translated to a hypergraph, the hypergraph is used to generate an optimal GHD-based query plan, and code instantiating the worst-case optimal algorithm is generated from the resulting query plan. Relation names and attribute names are abbreviated (e.g. 'nk' = 'nationkey', 'n' = 'nation', ...) in the SQL and generated code for readability.

translated to operations on annotated relations using a series of simple rules to construct query hypergraphs.

The rules EmptyHeaded uses to translate a general queries to a hypergraph $H = (V, E)$ (see Section 2.3) and an aggregation ordering $\alpha$ are as follows:

1. The set $V$ of vertices in the hypergraph contains all of the key columns in the query. The set of hyperedges $E$ is each relation in the query. All attributes that appear in an equi-join condition are mapped to the same attribute in $V$.

2. All key attributes that do not appear in the output of the query must be in the aggregation ordering $\alpha$.

3. If only the columns of a single relation appear inside of an aggregation function, the expression inside of the aggregation function is the annotation of that relation. If none of the relation's columns appear inside an aggregation function, the relation's

**Algorithm 3** Enumerating all GHDs via Brute Force Search

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

annotation is the identity element. If the inner expression of an aggregation function touches multiple relations, those relations are constrained to be in the same node of the GHD where the expression is the output annotation.

4. The rules above do not capture annotations which are not aggregated, so these annotations are added to a meta data container $M$ that associates them to the hyperedge (relation) from which they originate.

**Example 3.2.1.** Consider how these rules capture TPC-H query 5 from Figure 3.8 in a EmptyHeaded query hypergraph.

By Rule 1, the equality join in this query is captured in the set of vertices ($V$) and hyperedges ($E$) shown in the hypergraph in Figure 3.8. The columns `c_custkey` and `o_custkey` must be mapped to the same vertex in "$custkey$" $\in V$. Similarly, the columns `l_orderkey` and `o_orderkey` are mapped to the vertex "$orderkey$", the columns `l_suppkey` and `s_suppkey` are mapped to the vertex "$suppkey$", the columns `c_nationkey`, `s_nationkey`, and `n_nationkey` are mapped to the vertex "$nationkey$", the columns `n_regionkey` and `r_regionkey` are mapped to the vertex "$regionkey$".

By Rule 2, a valid aggregation ordering is:

$$\alpha = [regionkey, nationkey, suppkey, custkey, orderkey]$$

with the aggregation operator $\Sigma$ (the order is irrelevant here).

To apply Rule 3, consider the expression inside the aggregation function, `l_extendedprice * (1 - l_discount)`. Only columns on the `lineitem` table are involved in this expression, so the annotations on the `lineitem` table will be this expression for each tuple. The `orders` and `customer` tables do not have any columns in aggregation expressions, so they are annotated with the identity element.

By Rule 4, the hypergraph does not capture the attributes `n_name`, `o_orderdate`, or `r_name` but our metadata container $M$ does. $M$ here is the following: {`n_name` $\leftrightarrow$ `nation`, `r_name` $\leftrightarrow$ `region`, `o_orderdate` $\leftrightarrow$ `orders`}.

Thus, the final aggregate-join query is:

$$\sum_{regionkey,nationkey,suppkey,custkey,orderkey} ( supplier(custkey, suppkey) \bowtie orders(custkey, orderkey) \bowtie nation(nationkey, regionkey) \bowtie region(regionkey) \bowtie lineitem(orderkey, suppkey) )$$

Each $(orderkey, suppkey)$ tuple on relation $lineitem$ is annotated with the value of `SUM(l_extendedprice*(1-l_discount))`, and the tuples on all other relations are annotated with the identity element.

**Attribute Elimination** An important artifact of the previous rules is that only the attributes that are used in the query are added to the hypergraph. Eliminating unnecessary attributes can simplify the worst-case optimal join algorithm without changing the runtime bounds. This is because without attribute elimination the query compiler has an exponential number of choices to consider during compilation. The optimization is simple: we remove all attributes which are projected away in the query result and do not participate in the query. Because we eliminate keys in addition to annotations, the number of possible GHDs is much smaller. For example, attribute elimination, in combination with our storage model presented in Section 2.2, enables the EmptyHeaded design to only consider $120$ possible query plans for TPC-H Q5. If EmptyHeaded included unused key attributes there would be $6.23e9$ possible query plans. Even worse, a design that considers both unused key and annotation attributes is forced to consider $4.27e69$ possible query plans.

### 3.2.3 Choosing Among GHDs with the same FHW

After applying the rules in Section 3.2.2, a GHD is selected using the process described in Section 3.2.1. Still, EmptyHeaded needs a way to select among multiple GHDs that the theory cannot distinguish. In this section we explain how EmptyHeaded adapts the theoretical definition of GHDs to both select and produce practical query plans.

For many queries, multiple GHDs have the same FHW. Therefore, a practical implementation must also include principled methods to choose between query plans with the same worst-case guarantee. Fortunately, there are three intuitive characteristics of GHD-based query plans that makes this choice relatively simple (and cheap). The first is that the smaller a GHD is (in terms of number of nodes and height), the quicker it can be executed (less generated code). The second is that fewer intermediate results (shared vertices between nodes) results in faster execution time. Finally, the lower the selection constraints appear in a query plan corresponds to how early work is eliminated in the query plan. Therefore, EmptyHeaded uses the following order of heuristics to choose between GHDs with the same FHW:

1. Minimize $|V_T|$ (number of nodes in the tree).
2. Minimize the depth (longest path from root to leaf).
3. Minimize the number of shared vertices between nodes.
4. Maximize the depth of selections.

Although most of the queries in this dissertation are single-node GHDs, on the two node TPC-H query 5, using these rules to select a GHD results in a 3x performance advantage over a GHD (with the same FHW) that violates the rules above. We explain how selections are pushed down below joins in EmptyHeaded GHDs next.

### 3.2.4 Pushing Down Selections Below Joins

A classic database optimization is to force high selectivity operations to be processed as early as possible in a query plan [46]. In EmptyHeaded we can do this at two different

granularities in our query plans: within GHD nodes and across GHD nodes.[8]

### Within a Node

In EmptyHeaded pushing down selections within a GHD node corresponds to rearranging the attribute order for the generic worst-case optimal join algorithm that we described in Section 2.3. Recall, the attribute order determines both the order that attributes are processed in Algorithm 1 and the order in which the attributes will appear in the trie.

**Example 3.2.2.** Consider a query where the relation `R` has one selected attribute and another ($x$) to be materialized in the output:

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

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

### Across Nodes

Our mechanism of selecting a GHD from Section 3.2.3 ensures that selections appear as low as possible in a GHD, but we still need a mechanism to push selections down below joins in our chosen GHD. Our goal here 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. EmptyHeaded accomplishes this with two additional steps to modify the chosen GHD:

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

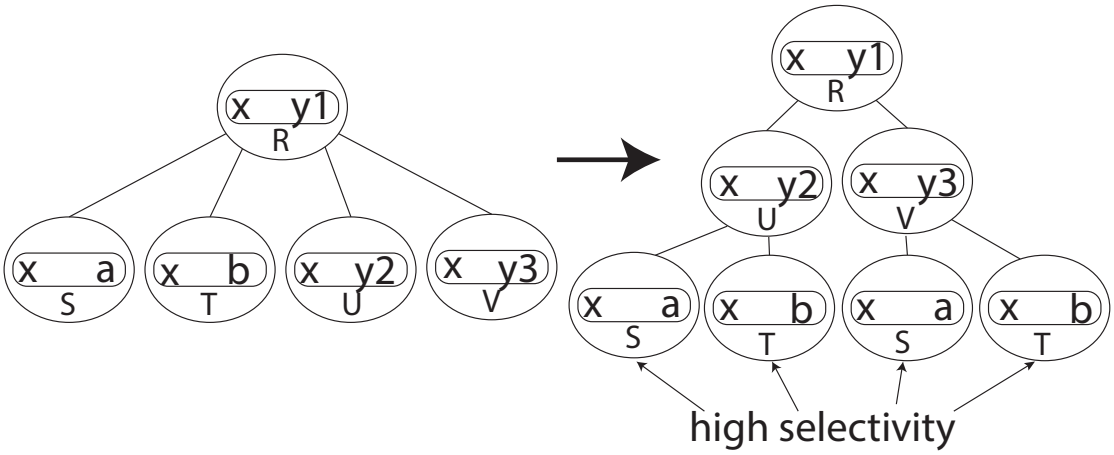



Figure 3.9: "Across nodes" transformation example query (LUBM query 4) where the $a$ and $b$ attributes are high selectivity.

1. Taking as input an optimal (with respect to FHW) GHD $T = (V_T, E_T)$ and a set of selections $\sigma_{a_i}$ applied to attributes $a_i$.

2. For each $\sigma_{a_i}$: Let $e_i$ be the edge that contains the vertex derived from $a_i$ or that has the meta data $M$ associated with $a_i$. Let $t_i$ be the GHD node in $V_T$ associated with $e_i$. If $t_i$ contains more than one hyperedge, create a new GHD node $t'_i$ that contains only $e_i$, and make $t'_i$ a child of $t_i$.

On TPC-H query 5 this results in the GHD shown in Figure 3.8 which executes 1.8x faster than the GHD without this optimization. To illustrate this process in more detail we provide more detail on an example benchmark query where this optimization provides up to 69.94x performance advantage:

**Example 3.2.3.** Consider the acyclic join pattern (LUBM query 4) containing high selectivity attributes `(a,b)`:

$$\texttt{R(x,y1)} \bowtie \texttt{S(x,a)} \bowtie \texttt{T(x,b)} \bowtie \texttt{U(x,y2)} \bowtie \texttt{V(x,y3)}$$

Figure 3.9 shows two possible GHDs for this query. The GHD on the left is the one produced without using the three steps above. This GHD does not filter out any intermediate results across potentially high selectivity nodes when results are first passed up the GHD.

The GHD on the right uses the three steps above. Here the nodes with attributes `'a'` and `'b'` are below all other nodes in the GHD, ensuring that high selectivity attributes are processed early in the query plan.

This does mean that EmptyHeaded pushes down selections by creating new GHD nodes with only the selection constraints under the original GHD nodes. We describe how to eliminate the redundant work that can be introduced by this process next.

### 3.2.5 Eliminating Redundant Work

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

Our query compiler performs a simple analysis to determine if two GHD nodes are identical and therefore if redundant work can be eliminated. For each GHD node in the "bottom-up" pass of Yannakakis' algorithm, we scan a list of the previously computed GHD nodes to determine if the result of the current node has already been computed. We use the conditions below to determine if two GHD nodes are equivalent in the Barbell query. Recognizing this provides a 2x performance increase on the Barbell query from Table 2.1.

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.

### 3.2.6 Pipelining

Pipelining is a classic query optimization used to reduce the size of materialized intermediate results in a query plan [46]. We add a simple rule such that the root node of a GHD can be pipelined with one child node in EmptyHeaded:

**Definition 3.2.1.** Given a GHD $T$, we say $(t_0, t_1) \in V(T) \times V(T)$ are *pipelineable* if $t_0 \neq t_1$ and $\chi(t_0) \cap \chi(t_1)$ is a prefix of the trie orders for both $t_0$ and $t_1$.

**Example 3.2.4.** Consider the following query pattern from LUBM query 8 over relation `R` with attributes `(x,y)` and relation `S` with attributes `(x,z)`:

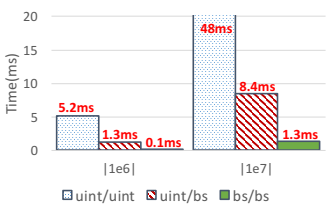
$$OUT(x, y, z) : -R(x, y), S(x, z)$$

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

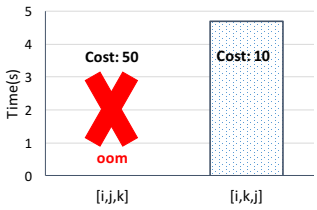
On LUBM query 8 we found that pipelining results between nodes with materialized attributes provides up to a 4.67x performance advantage as shown in the +Pipelining column of Table I. Unfortunately, the impact of pipelining is negligible on most LUBM queries as the number of output attributes is often small and so are the intermediate cardinalities.

### 3.2.7 Cost-Based Optimizer

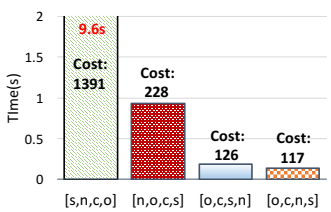
After a GHD-based query plan is produced using the process described in Sections 3.2.1 to 3.2.4, EmptyHeaded needs to select an attribute order for the worst-case optimal join algorithm. The attribute ordering determines the order in which EmptyHeaded code generates the generic worst-case optimal algorithm (Algorithm 1) and the index structure of our tries (Section 2.2). Therefore, selecting a global attribute ordering is analogous to selecting




(a) The performance of `uint ∩ uint`, `uint ∩ bs`, and `bs ∩ bs` intersections with cardinalities of 1e6 and 1e7. This is used to derive each intersection cost (`icost`) estimate.




(b) The performance and cost of two attribute orders for sparse matrix multiplication on the nlp240 matrix. The cost 50 order runs out of memory (oom) on a machine with 1TB of RAM.




(c) The performance and cost of four attribute orders for the expensive GHD node on TPC-H query 5 at SF 10. Attributes are: o = orderkey, c = custkey, s = suppkey, and n = nationkey.

Figure 3.10: Cost estimation experiments.

a join and index order in a traditional pairwise relational engine. And, similar to the classic query optimization problem of selecting a join order [35], the attribute ordering in a worst-case optimal join algorithm can result in orders of magnitude performance differences on the same query. Unfortunately, the known techniques for estimating the cost of join orders are designed for Selinger-style [12] query optimizers using pairwise join algorithms—not a GHD-based query optimizer with a worst-case optimal join algorithm. In this section we present the first cost-based optimizer for the generic worst-case optimal algorithm and validate the crucial observations it uses to derive its cost estimate. 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 and the data. Still, using only this we are able to build an optimizer that selects attribute orders that provide up to a three orders of magnitude speedup over other possible orders.

**Optimizer Overview** EmptyHeaded's cost-based optimizer selects a key attribute order for each node in a GHD-based query plan. As is standard [6], EmptyHeaded requires that materialized key attributes appear before those that are projected away (with one important exception described in Section 3.2.7) and that materialized attributes always adhere to some global ordering (e.g. if attribute 'a' is before attribute 'b' in one GHD node order, then 'a' must be before 'b' in all GHD orders). To assign an attribute order to each GHD node,

EmptyHeaded's cost-based optimizer: traverses the GHD in a top-down fashion, considers all attribute orders[9] adhering to the previously described criteria at each node, and selects the attribute order with the lowest cost estimate.

For each order, a cost estimate is computed based on two characteristics of the generic WCOJ algorithm: (1) the algorithm processes one attribute at a time and (2) set intersection is the bottleneck operator. As such, EmptyHeaded assigns a set intersection cost (`icost`) and a cardinality weight (Section 3.2.7) to each key attribute (or vertex) in the hypergraph of a GHD node. Using this, the cost estimate for a given a key attribute (or hypergraph vertex) order [$v_0$,$v_1$,...,$v_{|V|}$] is:

$$\text{cost} = \sum_{i=0}^{|V|} (\texttt{icost}(v_i) \times \text{weight}(v_i))$$

The remainder of this section discusses how the `icost`s (Section 3.2.7) and weights (Section 3.2.7) are derived.

### Intersection Cost

The bottleneck of the generic worst-case optimal join algorithm is set intersection operations. In this section, we describe how to derive a simple cost estimate, called `icost`, for the set intersections in the generic worst-case optimal join algorithm.

**Cost Metric** Recall that the sets in EmptyHeaded tries are stored using an unsigned integer layout (`uint`) if they are sparse and a bitset layout (`bs`) if they are dense, a design inherited from EmptyHeaded. Thus, the intersection algorithm used is different depending on the data layout of the sets. These different layouts have a large impact on the set intersection performance, even with similar cardinality sets. For example, Figure 3.10a, shows that a `bs` $\cap$ `bs` is roughly 50x faster than a `uint` $\cap$ `uint` with the same cardinality sets. Therefore, EmptyHeaded uses the results from Figure 3.10a to assign the following `icost`s:

[9]We remind the reader that the number of attributes considered here is only the number of key (joined) attributes inside of a GHD node which is typically small.

$$\texttt{icost}(\texttt{bs} \cap \texttt{bs}) = 1,\ \texttt{icost}(\texttt{bs} \cap \texttt{uint}) = 10,$$
$$\texttt{icost}(\texttt{uint} \cap \texttt{uint}) = 50$$

Unfortunately, it is too expensive for the query compiler to check (or track) the layout of each set during query compilation—set layouts are chosen dynamically during data ingestion and a single trie can have millions of sets. To address this EmptyHeaded uses the following observation.

**Observation 3.2.1.** The sets in the first level of a trie are typically dense and therefore represented as a bitset. The sets of any other level of a trie are typically sparse (unless the relation is completely dense) and therefore represented using the unsigned integer layout.
**Empirical Validation:** Consider a trie for the TPC-H `lineitem` relation where the trie levels correspond to the key attributes [`l_orderkey`,`l_suppkey`,`l_partkey`,`l_linenumber`] in that order. At scale factor 10, each level of this trie has the following number of `uint` and `bs` sets:

- 1st level(`l_orderkey`) = {0 `uint` sets, 1 `bs` set}
- 2nd level(`l_suppkey`) = {14999914 `uint` sets, 86 `bs` sets}
- 3rd level(`l_partkey`) = {59984817 `uint` sets, 0 `bs` sets}
- 4th level(`l_linenumber`) = {59986042 `uint` sets, 0 `bs` sets}

Thus, given a key attribute order [$v_0$,...,$v_{|V|}$] (where each $v_i \in V$), the EmptyHeaded optimizer assigns an `icost` to each $v_i$, in order of appearance, using the following method which leverages this observation:

- For each edge $e_j$ with node $v_i$, assign $l(e_j)$ (where $l$=layout), to either `uint` or `bs`. As a reminder, edges are relations and vertices are attributes. Thus, for each relation this assignment guesses one data layout for all of the relation's $v_i$ sets. If $e_j$ has been assigned with a previous vertex $v_k$ where k $<$ i, $l(e_j) =$ `uint` (not the first trie level), otherwise $l(e_j) =$ `bs`.
- Compute the cost of intersecting the $v_i$ attribute from each edge (relation) $e_j$. For a vertex with two edges, the pairwise `icost` is used. For a vertex with $N$ edges,

where $N > 2$, the `icost` is the sum of pairwise `icost`s where the `bs` sets are *always* processed first. For example, when $N = 3$ and $\text{l}(e_0) \leq \text{l}(e_1) \leq \text{l}(e_2)$ where `bs` $<$ `uint`, `icost` $= \texttt{icost}(\text{l}(e_0) \cap \text{l}(e_1)) + \texttt{icost}(\text{l}(\text{l}(e_0) \cap \text{l}(e_1)) \cap \text{l}(e_2))$. Note, `uint` $= \text{l}(\texttt{bs} \cap \texttt{uint})$.

**Example 3.2.5.** Consider the attribute order [`orderkey`,`custkey`,`nationkey`,`suppkey`] for one of the GHD nodes in TPC-H query 5 (see Figure 3.8). The `orderkey` vertex is assigned an `icost` of 1 as it is classified with `[bs ∩ bs]` intersections. The `custkey` vertex is assigned an `icost` of 10, classified with `[uint ∩ uint]` intersections. The `nationkey` vertex is assigned an `icost` of 11, classified with `[bs ∩ bs ∩ uint]` intersections. Finally, the `suppkey` vertex is assigned an `icost` of 50, classified with `[uint ∩ uint]` intersections.

Finally, in the special case of a completely dense relation, the LevelHeaded optimizer assigns an `icost` of 0 because no set intersection is necessary in this case. This is essential to estimate the cost of linear algebra queries properly.

**Relaxing the Materialized Attributes First Rule** An interesting aspect of the intersection cost metric is that the cheapest key attribute order could have materialized key attributes come after those which are projected away.[10] To support such key attribute orders, the execution engine must be able to combine children (in the trie) of projected away key attributes using a set union or `GROUP BY` (to materialize the result sets). Unfortunately, it is difficult to design an efficient data structure to union more than one level of a trie (materialized key attribute) in parallel (e.g. use a massive 2-dimensional buffer or incur expensive merging costs). In EmptyHeaded we relax this rule by allowing 1-attribute unions (see Section 3.1.6) on keys when it can lower the `icost`.

Within a GHD node, EmptyHeaded relaxes the materialized attributes first rule under the following conditions:

1. The last attribute is projected away.

2. The second to last attribute is materialized.

[10] We remind the reader that later (or after) in the attribute order corresponds to a lower level of the tries.

3. The `icost` is improved by swapping the two attributes.

These conditions ensure that 1-attribute union will only be introduced when the `icost` can be lowered.

**Example 3.2.6.** Consider the sparse matrix multiplication query and its unrolling of the generic WCOJ algorithm for an attribute order of `[i,j,k]` shown in Figure 3.8. This attribute order has a cost 50 `[uint ∩ uint]` assigned to the `k` attribute.

Now consider an attribute order `[i,k,j]`. Here a cost 10 `[uint ∩ bs]` is assigned to `k` and the unrolling of the generic worst-case optimal join algorithm (where () denotes accesses to keys and [] denotes access to annotations) is the following:

for $i \in \pi_i M1$ do
  $s_j \leftarrow \emptyset$
  for $k \in \pi_k M1(i) \cap \pi_k M2$ do
    for $j \in \pi_j M2(k)$ do
      $s_j[j] += \pi_{v1} M1[i,k] * \pi_{v2} M2[k,j]$
  $out(i) \leftarrow s_j$

This lower cost attribute order recovers the same loop ordering as Intel MKL on sparse matrix multiplication [80] and its bottleneck is the += operation that unions `j` values and sums `v1*v2` values. This is in contrast to the standard bottleneck operation of a set intersection as in the `[i,j,k]` order. In Section 3.1.6 we discuss the tradeoffs around implementing this union add or `GROUP BY` operation. Figure 3.10b shows that this order is essential to run sparse matrix multiplication as a join query without running out of memory.

### Weights

Like classic query optimizers, EmptyHeaded also tracks the cardinality of each relation as this influences the `icosts` for the generic worst-case optimal join algorithm. Figure 3.10 shows the unsurprising fact that larger cardinality sets result in longer intersection times. To take this into account when computing a cost estimate EmptyHeaded assigns weights to each vertex using the observation below, which directly contradicts conventional wisdom from pairwise optimizers:

**Observation 3.2.2.** The highest cardinality attributes should be processed first in the generic worst-case optimal join algorithm. This enables these attributes to partake in fewer intersections (outermost loops) and ensures that they are a higher trie levels (more likely `bs`'s with lower `icosts`).

**Empirical Validation:** We show in Figure 3.10c that the generic worst-case optimal join can run over 70x faster on TPC-H query 5 when the high cardinality `orderkey` attribute is first in the attribute order instead of last.

EmptyHeaded's goal when assigning weights is to follow the aforementioned observation by assigning high cardinality attributes heavier weights so that they appear earlier in the attribute order. To do this, EmptyHeaded assigns a cardinality score to each queried relation and uses this to weight to each attribute. We describe this in more detail next.

**Score** EmptyHeaded maintains a cardinality score for each relation in a query which is just the relation's cardinality relative to the highest cardinality relation in the query. The score (out of 100) for a relation $r_i$ is:

$$\text{score} = \text{ceiling}\left(\frac{|r_i|}{|r_{heavy}|} \times 100\right)$$

where $r_{heavy}$ is the highest cardinality relation in the query.

**Weight** To assign a weight to each vertex EmptyHeaded uses the highest score edge (or relation) with the vertex when a high selectivity (equality) constraint is present, otherwise EmptyHeaded takes the lowest score edge (or relation). The intuition for using the highest score edge (or relation) with a high selectivity constraint is that this relation represents the amount of work that could be filtered (or eliminated) at this vertex (or attribute). The intuition for otherwise taking the lowest score edge (or relation) is that the output cardinality of an intersection is at most the size of the smallest set.

**Example 3.2.7.** Consider TPC-H Q5 at scale factor 10. The cardinality score for each relation here is:

score(lineitem) = 100, score(orders) = 26, score(customer)= 3,

score(region) = 1, score(supplier) = 1, score(nation) = 1

The weight for each vertex is (region is equality selected):

weight(orderkey) = min(26,100), weight(custkey) = min(3,26)
weight(suppkey) = min(1,100), weight(nationkey) = min(1,1,3)
weight(regionkey) = max(1,1)

These weights are then used to derive the cost estimates shown in Figure 3.10c.

# Chapter 4

# Application Domains

We provide an evaluation of EmptyHeaded on popular benchmark queries in the graph, RDF, linear algebra, and business intelligence domains. We show that EmptyHeaded can compete with specialized engines in each domain while outperforming traditional relational architectures, sometimes by several orders of magnitude. In each domain, we perform a series of micro-benchmark evaluations that validate the importance of the optimizations presented in Chapter 3. Finally, we conclude by presenting a few experiments on query pipelines that span multiple domains. We show here that the unified approach of EmptyHeaded can outperform commodity solutions for such query pipelines by up to an order of magnitude.

## 4.1 Graph Evaluation

We compare EmptyHeaded against state-of-the-art high- and low-level specialized graph engines on standard graph benchmarks. Additionally, we compare EmptyHeaded to several state-of-the-art relational database engines which use a pairwise join algorithm on a standard graph benchmark. We show that by using our optimizations from Section 3.1 and Section 3.1, EmptyHeaded is able to compete with specialized graph engines while outperforming pairwise relational database engines on standard graph benchmarks. Prior to this, in Section 4.1.1 we discuss how different node orderings can change the density skew

(and potentially the cardinality skew[1]) of the graph. Although this is well-known, we go on to validate that, with the optimizations presented in this section, EmptyHeaded is able to mitigate the effects of these orderings by achieving robust performance regardless of the ordering.

### 4.1.1 The Impact of Node Ordering

As is standard, EmptyHeaded encodes the nodes of a graph as unique unsigned integers using the dictionary encoding technique presented in Section 2.2. Interestingly, a side effect of this technique is that the order in which nodes are encoded can affect both the cardinality skew and density skew of the processed data. Therefore, it is important to consider the node ordering used during dictionary encoding, as this can change the heuristics used by the optimizers in Sections 3.1.4 and 3.1.5. We provide an overview of dictionary encoding and node ordering in Section 4.1.1, and an evaluation of its impact in Section 4.1.1.

#### Overview

Dictionary encoding is a common technique used in online analytical processing (OLAP) and graph engines to compress the columns of a relation. Graph analytic engines use the same technique on edge relations and refer to the assignment of entries in the dictionary as node ordering. Formally, there is a mapping $\pi(v)$ that assigns each node $v$ a unique integer, which implicitly orders the nodes. The choice of $\pi(v)$ affects the internal skew of the sets in a graph. Node ordering may impact the performance of a class of symmetrical queries like triangle without selections [23, 90]. The key idea of this method is that given an undirected graph, one creates a new directed graph that preserves the number of triangles by defining a unique orientation of the edges. This new directed graph has the property that no node has a degree more than $\sqrt{N}$ where N is the number of edges in the original graph. Node ordering in combination with this symmetry technique affects the density and cardinality skew of the data. Because EmptyHeaded maps each node to an integer value, it is natural to consider the performance implications of these mappings.

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

### Evaluation

We explore the impact of node ordering on query performance using triangle counting query on synthetically generated power law graphs with different power law exponents. We generate the data using the Snap Random Power-Law graph generator [61] and vary the Power-Law degree exponents from 1 to 3. We find that the best ordering can achieve over an order of magnitude better performance than the worst ordering on symmetrical queries such as triangle counting.

We consider the following orderings:

- **Random:** random ordering of vertices. We use this as a baseline to measure the impact of the different orderings.
- **BFS:** labels the nodes in breadth-first order.
- **Strong-Runs** first sorts the node by degree and then starting from the highest degree node, the algorithm assigns continuous numbers to the neighbors of each node. This ordering can be seen as an approximation of 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 [24].

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 4.1 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 4.1 shows the performance of `hybrid` ordering and how it tracks the performance of BFS or degree where each is optimal. We find this ordering to be the most robust ordering for symmetrical queries.

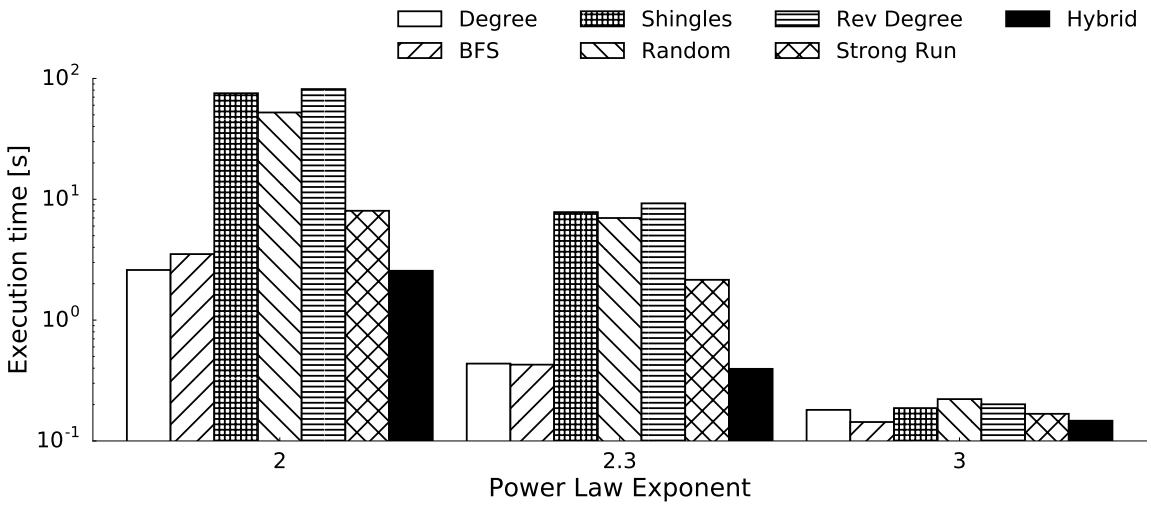



Figure 4.1: Effect of data ordering on triangle counting with synthetic data.

| 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 4.1: Node ordering times in seconds on two popular graph datasets.

Each ordering incurs the cost of performing the actual ordering of the data. Table 4.1 shows examples of node ordering times in EmptyHeaded. The execution time of the BFS ordering grows linearly with the number of edges, while sorting by degree and reverse degree depend on the number of nodes. The cost of the `hybrid` ordering is the sum of the costs of the BFS ordering and ordering by degree.

In total, we found that, although considering node orderings is an interesting problem, the optimizers to select among intersections and layouts presented in this section often mitigated the effects of node ordering. For example, Table 4.2 shows that a random ordering in EmptyHeaded often outperforms a compressed sparse row (CSR) layout that is sorted by degree. As such, we found that considering node orderings is not as crucial of a problem

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

Table 4.2: Slow down of a random ordering to a CSR layout sorted by degree for the triangle counting query.

in EmptyHeaded as it is in other graph engines which do not implement our optimizations.

### 4.1.2 Experimental Setup

We describe the datasets, comparison engines, metrics, and experimental setting used to validate EmptyHeaded's performance on graph queries in Sections 4.1.3 and 4.1.4.

#### Datasets

Table 3.1 provides a list of the six popular datasets that we use in our comparison to other graph analytics engines. LiveJournal, Orkut, and Patents are graphs with a low amount of density skew, and Patents is 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.

#### Comparison Engines

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

**Low-Level Engines** We benchmark several graph analytic engines and compare their performance. The engines that we compare to are PowerGraph v2.2 [36], the latest release of commercial graph tool (CGT-X), and Snap-R [61]. Each system provides highly optimized shared memory implementations of the triangle counting query. Other shared memory graph engines such as Ligra [95] and Galois [75] do not provide optimized implementations of the triangle query and requires one to write the query 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 [89] on these queries.

**High-Level Engines** We compare to LogicBlox v4.3.4 on all queries since LogicBlox is the first general purpose commercial engine to provide similar worst-case optimal join guarantees. LogicBlox also provides a relational model that makes complex queries easy and succinct to express. It is important to note that LogicBlox is a full-featured commercial system (supports transactions, updates, etc.) and therefore incurs inefficiencies that EmptyHeaded does not. Regardless, we demonstrate that using GHDs as the intermediate representation in EmptyHeaded's query compiler not only provides tighter theoretical guarantees, but provides more than a three orders of magnitude performance improvement over LogicBlox. We further demonstrate that our set layouts account for over a one order of magnitude performance advantage over the LogicBlox design. We also compare to SociaLite [94] 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 [76, 89, 94].

**Pairwise Join Engines** We compare to HyPer v0.5.0 [50] as HyPer is a state-of-the-art in-memory RDBMS. HyPeR is a high-performance engine that outperforms most other

popular database engines on commodity hardware, like that which is used in this dissertation. We also compare to version 9.3.5 of PostgreSQL and MonetDB (Jul2015-SP1 release). We configured the sizes of PostgreSQL and MonetDB's buffers to be more than an order of magnitude larger than the input size in uncompressed text form. Before running the queries, we created indices and used `ANALYZE` to collect statistics (both excluded from the running time). We stored the edge relation on a RAM disk using `tablespaces` for PostgreSQL and `tmpfs` for MonetDB. These relational engines are full-featured commercial strength systems (support transactions, etc.) and therefore incur inefficiencies that EmptyHeaded does not.

**Omitted Comparisons** We compared EmptyHeaded to GraphX [37] which is a graph engine designed for scale-out performance and Neo4j which is a commercial graph database engine. Both GraphX and Neo4j were consistently several orders of magnitude slower than EmptyHeaded's performance in a shared-memory setting. We also compared to a commercial column store database engine but they were consistently over three orders of magnitude off of EmptyHeaded's performance. We exclude a comparison to the Grail method [34] as this approach in a SQL Server has been shown to be comparable to or sometimes worse than PowerGraph [36] when the entire dataset can easily fit in-memory (like we consider in this dissertation). It should be noted that the Grail approach, in a persistent database, has been shown to be more robust than in-memory engines, such as EmptyHeaded and PowerGraph, when the entire dataset does not fit easily in-memory [34].

### 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 dissertation) is often faster than other high-level engines' (Socialite or LogicBlox).

#### 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, PowerGraph, TripleBit) with g++ 4.9.3 (-O3) and ran the Java-based engines (CGT-X, LogicBlox, SociaLite) on OpenJDK 7u65 on Ubuntu 12.04 LTS. For all engines, we chose buffer and heap sizes that were at least an order of magnitude larger than the dataset itself to avoid garbage collection.

### 4.1.3 End-to-End Comparisons

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.

#### 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 [68, 72]. 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

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

Table 4.3: EmptyHeaded's Syntax for Benchmark Graph Queries

| | | Low-Level | | | High-Level | |
|---|---|---|---|---|---|---|
| Dataset | EmptyHeaded | PowerGraph | CGT-X | Snap-Ringo | SociaLite | LogicBlox |
| 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 4.4: Triangle counting runtime (in seconds) for EmptyHeaded and relative slowdown for other engines including PowerGraph, a commercial graph tool (CGT-X), Snap-Ringo, SociaLite, and LogicBlox. 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.

degree of the node. This process (describe more in Section 4.1.1) is standard as it limits the size of the intersected sets and has been shown to empirically work well [90]. Nearly every graph engine implements pruning in this fashion for the triangle query.

**Takeaways** The results from this experiment are in Table 4.4. 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`. This 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 [102] 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 4.1.4 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 3.1 and EmptyHeaded's triangle counting numbers are comparable to Intel's hand-coded numbers which are slightly (10-20%) faster than the Galois implementation [89]. We provide a comparison to Galois on SSSP and PageRank in Section 4.1.3.

### 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

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

Table 4.5: Runtime for 5 iterations of PageRank (in seconds) using 48 threads for all engines. The other engines include Galois, PowerGraph, a commercial graph tool (CGT-X), Snap-Ringo, SociaLite, and LogicBlox. "-" indicates the engine does not process over 70 million edges.

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

Table 4.6: SSSP runtime (in seconds) using 48 threads for all engines. The other engines include Galois, PowerGraph, a commercial graph tool (CGT-X), SociaLite, and LogicBlox. "x" indicates the engine did not compute the query properly. "-" indicates the engine does not process over 70 million edges.

relational engines typically do not process efficiently: (1) linear algebra operations (in PageRank) and (2) transitive closure (in SSSP). We run each query on undirected versions of the graph datasets and demonstrate competitive performance compared to specialized graph engines. Our results suggest that our approach is competitive outside of classic join workloads.

**PageRank** As shown in Table 4.5, 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 4.5, while expressing the query in three lines of code (Table 4.3). 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 Section 4.1.3 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 [16]. For example, Galois, which implements such an algorithm, observes a 2-30x performance improvement over EmptyHeaded on this application (Section 4.1.3). 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 4.3).

### Pairwise Relational Comparison Results

Table 4.7 serves as a comparison of the worst-case optimal join algorithms in EmptyHeaded and the pairwise join algorithms present in popular and state-of-the-art database engines on the triangle counting query. We present a comparison to HyPer, MonetDB, and PostgreSQL on only the triangle counting query for two reasons: (1) expressing PageRank and SSSP in these engines is difficult and (2) when we benchmarked the other queries their performance difference with EmptyHeaded got even larger (several orders of magnitude slower). Interestingly, Table 4.7 shows that the worst-case optimal join algorithm on its own is not always enough to outperform existing state-of-the-art implementations. For example, on

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

Table 4.7: Triangle counting runtime (in seconds) for EmptyHeaded and relative slowdown for other engines including LogicBlox, HyPer, MonetDB, and PostgreSQL. 48 threads used for all engines. "t/o" indicates the engine ran for over 30 minutes.

the triangle query HyPer outperforms LogicBlox on all datasets except the Google+ dataset which has high density skew. On the other hand, EmptyHeaded consistently outperforms HyPer on all datasets by 7.6x-129.9x, with the largest performance difference occurring on the Google+ dataset. MonetDB and PostgreSQL were at least an order of magnitude slower than EmptyHeaded across datasets and were often more than two-orders of magnitude slower.

### Selections

To test our implementation of selections in EmptyHeaded we ran two graph pattern queries that contained selections. The first is a 4-clique selection query where we find all 4-cliques connected to a specified node. The second is a barbell selection query where we find all pairs of 3-cliques connected to a specified node. The syntax for each query in EmptyHeaded is shown in Table 4.3.

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 4.8. Pushing down selections across GHDs

| Dataset | Query | $\|Out\|$ | EmptyHeaded | -GHD | SociaLite | LogicBlox |
|---|---|---|---|---|---|---|
| Google+ | $SK_4$ | 1.5E+11 | 154.24 | 6.09x | t/o | t/o |
| | | 5.5E+7 | 1.08 | 865.95x | t/o | 50.91x |
| | $SB_{3,1}$ | 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 4.8: 4-Clique Selection ($SK_4$) and Barbell Selection ($SB_{3,1}$) runtime in seconds for EmptyHeaded and relative runtime for SociaLite, LogicBlox, and EmptyHeaded while disabling optimizations. "$|Out|$" indicates the output cardinality. "t/o" indicates the engine ran for over 30 minutes. "-GHD" is EmptyHeaded without pushing down selections across GHD nodes.

can enable over a four order of magnitude performance improvement on these queries and is essential to enable peak performance. As shown in Table 4.8 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. The data layouts we choose for the sets matter here as placing the selected attributes first in Algorithm 1, causes these attributes to appear in the first levels of the trie which are often

| | | | EH w/o Optimizations | | | Other Engines | |
|---|---|---|---|---|---|---|---|
| Dataset | Query | EH | -R | -RA | -GHD | SociaLite | LogicBlox |
| 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 | 2.53x | 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 4.9: 4-Clique ($K_4$), Lollipop ($L_{3,1}$), and Barbell ($B_{3,1}$) runtime in seconds for EmptyHeaded (EH) and relative runtime for SociaLite, LogicBlox, and EmptyHeaded while disabling features. "t/o" indicates the engine ran for over 30 minutes. "-R" is EmptyHeaded without layout representation optimizations. "-RA" is EmptyHeaded without both layout representation (density skew) and intersection algorithm (cardinality skew) optimizations. "-GHD" is EmptyHeaded without GHD optimizations (single-node GHD).

dense and can be represented using a bitset (see Section 3.1.2). For equality selections this is enables us to perform the actual selection in constant time versus a binary search in an unsigned integer array.

### 4.1.4 Micro-Benchmarking Results

We detail the effect of our contributions on query performance. We introduce two new queries and revisit the Barbell query (shown in Table 4.3) 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 triangle counting implementation is 138 lines of code in Snap-R and 402 lines of code in PowerGraph. In contrast, each query is one line of code in EmptyHeaded. As such, we do not benchmark the low-level engines on these complex pattern queries. We run `COUNT(*)` aggregate queries in this section to test the full effect of GHDs on queries with the potential for early aggregation. The $K_4$ query is symmetric and therefore runs on the same pruned datasets as those used in the triangle counting query in Section 4.1.3. The $B_{3,1}$ and $L_{3,1}$ queries run on the undirected versions of these datasets.

### Query Compiler Optimizations

GHDs enable complex queries to run efficiently in EmptyHeaded. Table 4.9 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.

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

Table 4.10: Relative time when disabling features on the triangle counting query. "-SIMD" is EmptyHeaded without SIMD. "-Representation" is EmptyHeaded using `uint` at the graph level.

**Execution Engine Optimizations**

Table 4.9 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 4.4, which do not consider mixing SIMD friendly layouts. Table 4.9 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 , which is what happens on the datasets with low density skew. 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 4.9 and our discussion validate that the set layout and algorithmic features have merit and enable EmptyHeaded to compete with graph engines.

Table 4.10 shows the relative time to complete the triangle query features of our system disabled on unpruned data. The "-SR" column is the one that most closely resembles the implementation of the graph analytics competitors we compare to in Table 4.4. Our use of SIMD instructions can enable up to a 1.8x performance increase and our use of representations can enable up to a 3.9x performance increase. The amount of SIMD parallelism leveraged is highly intertwined with our representation decisions. Therefore we notice this 3.9x performance decrease on datasets with high density skew when we solely use a `uint`

representation ("-R") because the amount of available SIMD parallelism is decreased significantly. Finally, we also tested removing the decision between SIMD galloping and the SIMDShuffling algorithm for the `uint` intersections but found this feature had only a 10% performance impact on overall query runtime. In total, Table 4.10 shows our vectorization and representation features have merit and are needed to attain optimal performance on graph queries over skewed data.

## 4.2 RDF Evaluation

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

### 4.2.1 RDF Overview

The volume of Resource Description Framework (RDF) data from the Semantic Web has grown exponentially in the past decade [71, 105]. RDF data is a collection of Subject-Predicate-Object triples that form a complex and massive graph that traditional query mechanisms do not handle efficiently [54, 71]. As a result, there has been significant interest in designing specialized engines for RDF processing [13, 53, 71, 105]. These specialized engines accept the SPARQL query language and build several indexes ($> 10$) over the Subject-Predicate-Object triples to process RDF workloads efficiently [71, 105]. In contrast, the natural way of storing RDF data in a traditional relational engine is to use triple tables [71] or vertically partitioned column stores [5], but these techniques can be three orders of magnitude slower than specialized RDF engines [71].

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

Table 4.11: Example LUBM RDF query patterns in EmptyHeaded . Note that the equality constraint constants have been simplified here for readability (e.g. 'University567' is actually 'http://www.University567.edu').

### 4.2.2 Experimental Setup

We describe the details of our experimental setting.

#### LUBM Benchmark

The LUBM benchmark is a standard RDF benchmark with a synthetic data generator [40]. The data generator produces RDF data representing a university system ontology. We generated 133 million triples for the comparisons in this section. The LUBM benchmark contains complex multiway star join patterns as well as two cyclic queries with triangle patterns. We run the complete LUBM benchmark while removing the inference step for each query. This is standard in benchmarking comparisons [13, 105]. We omit queries 6 and 10, since without the inference step, they correspond to other queries in the benchmark.

#### Comparison Engines

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

**RDF Engines** We compare against RDF-3X v0.3.8 and TripleBit, two high performance shared memory RDF engines. TripleBit [105] and RDF-3X [71] have been shown to consistently outperform traditional column and row store databases [71, 105]. RDF-3X is a popular and established RDF engine which performs well across a variety of SPARQL queries. RDF-3X builds a full set of permutations on all triples and uses selectivity estimates to choose the best join order. TripleBit [105] is a more recent RDF engine which uses a sophisticated matrix representation and has been shown to compete with and often outperform RDF-3X on a range of RDF queries on larger scale data. TripleBit reduces the size of the data and indexes through two auxiliary data structures to minimize the cost of index selection during query evaluation. Both engines use optimizers that generate optimal join orderings.

**Relational Engines** We also provide comparisons to MonetDB (Jul2015-SP1 release) and LogicBlox v4.3.4 which are two general purpose relational engines. MonetDB is a popular open source column store database whose performance has been shown to outperform row store designs, such as PostgreSQL, by orders of magnitude on RDF workloads [71]. LogicBlox is a commercial database engine that uses a worst-case optimal join algorithm similar to that inside of the EmptyHeaded engine. For all relational engines, including EmptyHeaded, we store and process the RDF data in a vertically partitioned manner as this has been shown to be superior to storing the data as triples [5, 71]. Vertical partitioning is the process of grouping the triples by their predicate name, with all triples sharing the same predicate name being stored under a table denoted by the predicate name [5].

### Experimental Setting

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 (RDF-3X, TripleBit, EmptyHeaded) with g++ 4.9.3 (-O3). For all engines, we chose buffer sizes and heap sizes that were at least an order of magnitude larger than the dataset itself to avoid garbage collection. For all engines, we put the database files in *tmpfs*, a RAM disk, which is a standard resource to use when comparing in-memory engines to databases [53]. For MonetDB, we explicitly built indexes on each column and each pair of columns as well as histograms over each relation (with the `ANALYZE` command).

### Metrics

For each query we measure the wall clock time of the engine to complete the query. We do not measure the time for data loading, index creation, or query output for each engine. For TripleBit and RDF-3X, we do not include the time for lookups from ID to String (or output time) at the end of the query. We run each query seven times, discarding the worst and best runtimes while reporting the average of the remaining times. We do not measure compilation time for EmptyHeaded. Since we run queries back-to-back, often only the first execution incurs compilation costs, and this longest run is discarded for all engines.

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

Table 4.12: Runtime in milliseconds for best performing system and relative runtime for each engine on the LUBM benchmark with 133 million triples.

### 4.2.3 End-to-End Comparison

LUBM queries 2 and 9 are the two cyclic queries that contain a triangle pattern. Unsurprisingly, here LogicBlox outperforms specialized engines by 3-5x and MonetDB by 17.96x (Table 4.12) due to the asymptotic advantage of worst-case optimal join algorithms. On these queries EmptyHeaded is 1.5x faster than LogicBlox due to our set layouts, which are designed for single-instruction multiple data parallelism. In general, our speedup over LogicBlox is more modest here than on the previously reported cyclic graph patterns due to the presence of selections.

On acyclic queries with high selectivity, EmptyHeaded also competes with the specialized RDF engines. On simple acyclic queries with selections (LUBM 1,3,5,11,13,14), EmptyHeaded is able to provide covering indexes, like the specialized engines, using only our trie data structure and the attribute order we described in Section 3.2.4. Therefore EmptyHeaded maintains competitive performance with RDF-3X and TripleBit (Table 4.12). On more complex acyclic queries with selections (LUBM 7,8,12), RDF-3X and TripleBit observe a performance advantage over EmptyHeaded due to their sophisticated cost-based query optimizers which combine selectivity estimates and join order (Table 4.12). Our optimizations from Sections 3.1 and 3.2 can provide up to a 48x performance improvement

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

Table 4.13: Relative speedup of each optimization on selected LUBM queries with 133 million triples. `+Layout` refers to EmptyHeaded when using multiple layouts versus solely a unsigned integer array (index layout). "+Attribute" refers to reordering attributes with selections within a GHD node. "+GHD" refers to pushing down selections across GHD nodes in our query plan. "+Pipelining" refers to pipelining intermediate results in a given query plan. "-" means the optimization has no impact on the query.

here, but more sophisticated optimizations are needed to outperform the specialized engines. Finally, on LUBM query 8 we observe a performance slowdown when compared to LogicBlox. This is due to expensive reallocations that occur within the EmptyHeaded engine. When removing allocations, we observed that EmptyHeaded's performance for query 8 was equivalent to that of RDF-3X.

### 4.2.4 Micro-Benchmarking Results

The layout optimizations presented in Section 3.1.2, the pushing down selections optimization presented in Section 3.2.4, and the pipelining optimization presented in Section 3.2.6 can enable up to a 234x performance advantage on the LUBM benchmark (see Section 4.2.3). We explain the impact of these optimizations on RDF queries next.

**Pushing Down Selections** Recall from Section 3.2.4 that EmptyHeaded pushes down selections in a GHD query plan in two phases: (1) by rearranging the attribute order inside of each node in the GHD and (2) by rearranging the nodes in the GHD such that high selectivity or low-cardinality nodes appear as far away from the root as possible (so that they are executed earlier in our bottom-up pass). We explain the empirical impact for each

phase by walking through an example RDF query for each next.

- Within a Node: Consider LUBM query 14 from Table 4.11. For this trivial query we produce a single node GHD containing attributes {x,'UndergraduateStudent'}.[2] An attribute ordering of [x,'UndergraduateStudent'] means that 'x' is the first level of the trie and 'UndergraduateStudent' is the second level of the trie. EmptyHeaded would execute this query by probing the second level of the trie for each 'x' attribute to determine if there was a corresponding value of 'UndergraduateStudent'. This is much less efficient than selecting the attribute ordering of ['UndergraduateStudent',x], where EmptyHeaded can perform a lookup in the first level of the trie (to find if a value of 'UndergraduateStudent' exists) and, if successful, return the corresponding second level as the result. This same optimization holds for more complex queries, such as LUBM query 2 (see Table 4.11), where EmptyHeaded selects the attribute ordering of ['GraduateStudent', 'Department', 'University', x, y, z]. We show in the +Attribute column of Section 4.2.3 that forcing the attributes with selections or small cardinalities to come first can enable an up to a 234.49x performance increase.

- Across Nodes: Consider the acyclic join pattern for LUBM query 4 from Table 4.11. Figure 3.9 shows two possible GHDs for this query. The GHD on the left is the one produced without the pushing out selection optimization. This GHD does not filter out any intermediate results across potentially high selectivity nodes when results are first passed up the GHD. The GHD on the right uses the pushing down selections across nodes optimization from Section 3.2.4. Here the nodes with attributes 'Univ0Dept0' and 'AssociateProfessor' are below all other nodes in the GHD, ensuring that these high selectivity attributes are processed early in the query plan. Although it appears that the optimized GHD has redundant work, EmptyHeaded uses the optimization in Section 3.2.5 to eliminate this redundant computation. Pushing down selections across nodes applies to only two queries in the LUBM benchmark, but provides up to a 69.94x speedup as we show in the +GHD column of Section 4.2.3.

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

**Set Layout Choices** Recall that in EmptyHeaded a single trie is analogous to a single index in a standard database. Therefore, EmptyHeaded performs an equality selection by checking whether a set in the trie contains a value. The set layouts we choose from Section 3.1.2 can have a large impact on the runtime performance when performing this set contains operation. For example, the `bitset` layout can perform this operation in constant time whereas the `uint` layout performs it in $O(\log n)$ with a binary search. Because of this, the `+Layout` column in Section 4.2.3 shows that the set layout optimizer presented in Section 3.1.2 provides up to a 8.22x performance increase when compared to executing solely over the `uint` layout. As such, it is preferred to have equality selected attributes in the `bitset` layout whenever possible. Interestingly, our pushing down selections optimization will often ensure this. Pushing down selections forces the equality selected attributes to appear in the first levels of the trie, which are more likely to be dense, and therefore best represented using the `bitset` layout.

**Pipelining** On LUBM query 8 we found that pipelining the results between nodes with materialized attributes provided up to a 4.67x performance advantage as shown in the `+Pipelining` column of Section 4.2. This is due to the optimization presented in Section 3.2.6 which enables EmptyHeaded to materialize fewer intermediate results. Unfortunately, the impact of pipelining is negligible on the other LUBM queries as the output cardinality is often small and so are the intermediate cardinalities.

## 4.3 Linear Algebra Evaluation

We compare EmptyHeaded to state-of-the-art relational database management engines and linear packages on standard linear algebra benchmark queries. We show that EmptyHeaded is able to outperform relational engines by orders of magnitude while competing within 2.44x of Intel MKL.

### 4.3.1 Experimental Setup

We describe the experimental setup common to all linear algebra experiments.

**Environment** EmptyHeaded is a shared 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 56 cores on four Intel Xeon E7-4850 v3 CPUs and 1 TB of RAM. For all engines, we chose buffer and heap sizes that were at least an order of magnitude larger than the dataset to avoid garbage collection and swapping data out to disk.

**Relational Comparisons** We compare to HyPer, MonetDB, and LogicBlox on all linear algebra queries to highlight the performance of other relational databases. Unlike EmptyHeaded, these engines are unable to compete within an order of magnitude of the best approaches on linear algebra queries. We compare to HyPer v0.5.0 [50] as HyPer is a state-of-the-art in-memory RDBMS design. We also compare to the MonetDB Dec2016-SP5 release. MonetDB is a popular columnar store database engine and is a widely used baseline [44]. Finally, we compare to LogicBlox v4.4.5 as LogicBlox is the first general purpose commercial engine to provide similar worst-case optimal join guarantees [11]. Our setup of LogicBlox was aided by a LogicBlox engineer. HyPer, MonetDB, and LogicBlox are full-featured commercial strength systems (support transactions, etc.) and therefore incur inefficiencies that EmptyHeaded does not.

**Linear Algebra Package Comparison** We use Intel MKL v121.3.0.109 as the specialized linear algebra baseline. This is the best baseline for linear algebra performance on Intel CPUs (as we use in this dissertation). Others [98] have shown that it takes considerable effort and tedious low-level optimizations to approach the performance of such libraries on linear algebra queries.

**Omitted Comparisons** We omit a comparison to array databases like SciDB [19] as these engines call BLAS or LAPACK libraries (like Intel MKL) on the linear algebra queries that we present in this dissertation. Therefore, Intel MKL was the proper baseline for our linear algebra comparisons. Additionally, SciDB is not designed to process TPC-H queries.

**Metrics** For end-to-end performance, we measure the wall-clock time for each system to complete each query. We repeat each measurement seven times, eliminate the lowest and

the highest value, and report the average. This measurement excludes the time used for outputting the result, data statistics collection, and index creation for all engines. We omit the data loading and query compilation for all systems except for LogicBlox and MonetDB where this is not possible. The query compilation time for these queries, especially at large scale factors, is negligible to the execution time. Still, we found that EmptyHeaded's unoptimized query compilation process performed within 2x of HyPer's. To minimize unavoidable differences with disk-based engines (LogicBlox and MonetDB) we place each database in the *tmpfs* in-memory file system and collect hot runs back-to-back. Between measurements for the in-memory engines (HyPer and Intel MKL), we wipe the caches and re-load the data to avoid the use of intermediate results.

**Datasets** We evaluate linear algebra queries on three dense matrices and three sparse matrices. The first sparse matrix dataset we use is the Harbor dataset, which is a 3D CFD model of the Charleston Harbor [29]. The Harbor dataset is a sparse matrix [80] that contains 46,835 rows and columns and 2,329,092 nonzeros. The second sparse matrix dataset we use is the HV15R dataset, which is a CFD matrix of a 3D engine fan [29]. The HV15R matrix contains 2,017,169 rows and columns and 283,073,458 nonzeros and is a large, non-graph, sparse matrix. The final sparse matrix dataset we use is the nlpkkt240 dataset with 27,993,600 rows and columns and 401,232,976 nonzeros [91]. The nlpkkt240 dataset is a symmetric indefinite KKT matrix. For dense matrices, we use synthetic matrices with dimensions of 8192x8192 (8192), 12288x12288 (12288), and 16384x16384 (16384).

**Queries** We run matrix dense vector multiplication and matrix multiplication queries on both sparse (SMV,SMM) and dense (DMV,DMM) matrices. These queries were chosen because they are simple to express using joins and aggregations in SQL and are the core operations for most machine learning algorithms. Further, Intel MKL is specifically designed to process these queries and, as a result, achieves the largest speedups over using a RDBMS here. Thus, these queries represent the most challenging linear algebra baseline queries possible. For both SMM and DMM we multiply the matrix by itself, as is standard for benchmarking [80]. The syntax we use to run each of these queries in EmptyHeaded is shown in Table 4.14.

| Query | Syntax |
|---|---|
| Matrix-Vector Multiplication | `%Matrix schema: <rel_name>(i, j, v)`<br>`%Vector schema: <rel_name>(i, v)`<br>`SELECT`<br>`  M.i as i,`<br>`  sum(M.v*V.v) as v`<br>`FROM`<br>`  M, V`<br>`WHERE`<br>`  M.j = V.i`<br>`GROUP BY`<br>`  M.i` |
| Matrix Multiplication | `%Matrix schema: <rel_name>(i, j, v)`<br>`SELECT`<br>`  M1.i, M2.j, SUM(M1.v*M2.v)`<br>`FROM`<br>`  M1, M2`<br>`WHERE`<br>`  M1.j = M2.i`<br>`GROUP BY`<br>`  R.i, S.j` |

Table 4.14: EmptyHeaded's Syntax for Benchmark Linear Algebra Queries

| Query | Data | Baseline | EmptyHeaded | Intel MKL | HyPer | MonetDB | LogicBlox |
|---|---|---|---|---|---|---|---|
| SMV | Harbor | 2.66ms | **1x** | 2.89x | 10.81x | 30.80x | 89.74x |
| | HV15R | 68.01ms | 2.43x | **1x** | 25.82x | 26.47x | 40.72x |
| | NLP240 | 114.97ms | 1.49x | **1x** | 17.23x | 53.93x | 113x |
| SMM | Harbor | 110ms | 1.63x | **1x** | 13.10x | 27.27x | 112x |
| | HV15R | 18.79s | 1.35x | **1x** | oom | t/o | 48.11x |
| | NLP240 | 1.92s | 2.44x | **1x** | 4.91x | t/o | 78.70x |
| DMV | 8192 | 7.96ms | **1x** | **1x** | 4.34x | 55.14x | 121x |
| | 12288 | 13.5ms | **1x** | **1x** | 5.78x | 88.89x | 330x |
| | 16384 | 23.45ms | **1x** | **1x** | 18.13x | 51.18x | 587x |
| DMM | 8192 | 2.76s | 1.02x | **1x** | oom | t/o | t/o |
| | 12288 | 4.43s | 1.01x | **1x** | oom | t/o | t/o |
| | 16384 | 9.29s | 1.01x | **1x** | oom | t/o | t/o |

Table 4.15: Runtime for the best performing engine ("Baseline") and relative runtime for comparison engines. '-' indicates that the engine did not provide support for the query. 't/o' indicates the system timed out and ran for over 30 minutes. 'oom' indicates the system ran out of memory.

| Dataset | Query | EmptyHeaded | Attr. Elim. | -Attr. Order | -Group By |
|---|---|---|---|---|---|
| hv15r | SMV | 165ms | - | - | - |
| hv15r | SMM | 25.44s | - | oom | 3.02x |
| nlp240 | SMV | 171ms | - | - | - |
| nlp240 | SMM | 4.69s | - | oom | 185x |
| 16384 | DMV | 13.50ms | 1.96x | - | - |
| 16384 | DMM | 4.43s | 500x | - | - |

Table 4.16: Runtime for EmptyHeaded and relative performance without optimizations on linear algebra queries. '-' indicates that the optimization had no effect on the query.

### 4.3.2 End-to-End Comparisons

Table 4.15 shows that EmptyHeaded is able to compete within 2.44x of Intel MKL on both sparse and dense linear algebra queries. On dense data, EmptyHeaded uses the attribute elimination optimization from Section 3.1.1 to store dense annotations in single buffers that are BLAS compatible and code generates to Intel MKL. Still, MKL produces only the output annotation, not the key values, so EmptyHeaded incurs a minor performance penalty ($<2\%$) for producing the key values. On sparse data, EmptyHeaded is able to compete with MKL when executing these linear algebra queries as pure aggregate-join queries. To do this, the attribute order and `GROUP BY` optimizations from Sections 3.1.6 and 3.2.7 were essential. Although we tested more sophisticated optimizations, like cache blocking (by adding additional levels to the trie), we found that the performance benefit of these optimizations did not consistently outweigh their added complexity. In contrast, other relational designs fall flat on these linear algebra queries. Namely, HyPer usually runs out of memory on the matrix multiplication query, and on the queries which it does complete, is often an order of magnitude slower than Intel MKL. Similarly, LogicBlox and MonetDB are at least an order of magnitude slower than Intel MKL on these queries.

### 4.3.3 Micro-Benchmarking Results

We break down the performance impact of the core optimizations presented in Sections 3.1 and 3.2.

**Attribute Elimination** Table 4.16 shows that attribute elimination provides up to a 500x performance advantage on dense linear algebra queries. Attribute elimination is crucial on dense linear algebra queries because it allows EmptyHeaded to call Intel MKL with little overhead. This is because attribute elimination enables us to store each dense annotation in a BLAS acceptable buffer. As Table 4.16 shows, this yields up to a 500x speedup over processing these queries purely in EmptyHeaded, due to the sophisticated cache blocking and parallelization techniques present in BLAS packages [18].

**Attribute Order** As shown in Table 4.16, the cost-based attribute ordering optimizer presented in Section 3.2.7 enables EmptyHeaded to run sparse matrix multiplication as a join query without running out of memory. Table 4.16 shows the difference between the best-cost and the worst-cost attribute orders. The cost-based attribute ordering optimizer is crucial on sparse matrix multiplication. Here it is essential that the lower cost attribute order, with a projected away attribute before one that is materialized, is selected. This order not only prevents a high cost intersection, but eliminates the computation and materialization of annotation values do not participate in the output (due to sparsity).

**GROUP BY** Finally, we show in Table 4.16 that EmptyHeaded's `GROUP BY` optimizers provide up to a 185x performance advantage on linear algebra queries. The `GROUP BY` key optimizer provides a 185x speedup on SMM with the nlp240 dataset because it correctly predicts that the many of the output key attribute sets are sparse. As such EmptyHeaded's optimizer chooses to use a standard hash map to produce these sparse sets because using a bitset is highly inefficient (due to the amount of memory it wastes). This optimizer making the opposite choice provides a 3x performance advantage on SMM with the HV15R dataset.

## 4.4 Business Intelligence Evaluation

We compare EmptyHeaded to state-of-the-art relational database management engines on standard business intelligence benchmark queries. We show that EmptyHeaded is able to compete within 1.88x of these engines, while sometimes outperforming them, and that the techniques from in Sections 3.1 and 3.2 can provide up to a three orders of magnitude

speedup. This validates that a worst-case optimal join architecture is a practical solution for business intelligence queries.

### 4.4.1 Experimental Setup

We describe the experimental setup common to all business intelligence experiments.

**Environment** EmptyHeaded is a shared 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 56 cores on four Intel Xeon E7-4850 v3 CPUs and 1 TB of RAM. For all engines, we chose buffer and heap sizes that were at least an order of magnitude larger than the dataset to avoid garbage collection and swapping data out to disk.

**Relational Comparisons** We compare to HyPer, MonetDB, and LogicBlox on all business intelligence queries to highlight the performance of other relational databases. We compare to HyPer v0.5.0 [50] as HyPer is a state-of-the-art in-memory RDBMS design. We also compare to the MonetDB Dec2016-SP5 release. MonetDB is a popular columnar store database engine and is a widely used baseline [44]. Finally, we compare to LogicBlox v4.4.5 as LogicBlox is the first general purpose commercial engine to provide similar worst-case optimal join guarantees [11]. Our setup of LogicBlox was aided by a LogicBlox engineer. Because of this, we know that LogicBlox does not use a WCOJ algorithm for many of the join queries in the TPC-H benchmark. HyPer, MonetDB, and LogicBlox are full-featured commercial strength systems (support transactions, etc.) and therefore incur inefficiencies that EmptyHeaded does not.

**Datasets** We run the TPC-H queries at scale factors 1, 10, and 100. We stopped at TPC-H 100 as in-memory engines, such as HyPer, often use 2-3x more memory than the size of the input database during loading—therefore approaching the memory limit of our machine. For reference, on TPC-H query 1 HyPer uses 161GB of memory whereas EmptyHeaded uses 25GB (only loading the data it needs from disk). Additionally, scale factor 100 was the largest one where we could guarantee that the buffer space for disk-based engines was one order of magnitude large than the data.

| Query | Data | Baseline | EmptyHeaded | HyPer | MonetDB | LogicBlox |
|---|---|---|---|---|---|---|
| Q1 | SF 1 | 12ms | 1.79x | **1x** | 30.59x | 74.17x |
| | SF 10 | 84ms | 1.73x | **1x** | 17.86x | 23.45x |
| | SF100 | 608ms | 1.78x | **1x** | 80.43x | 26.12x |
| Q3 | SF 1 | 29ms | 1.11x | **1x** | 5.56x | 48.28x |
| | SF 10 | 111ms | **1x** | 1.45x | 9.88x | 32.59x |
| | SF100 | 963ms | 1.01x | **1x** | 9.76x | 10.99x |
| Q5 | SF 1 | 19ms | 1.49x | **1x** | 6.54x | 109x |
| | SF 10 | 92ms | 1.40x | **1x** | 4.84x | 55.33x |
| | SF100 | 867ms | 1.21x | **1x** | 4.04x | 21.33x |
| Q6 | SF 1 | 5ms | 1.73x | **1x** | 12.27x | 270x |
| | SF 10 | 34ms | 1.50x | **1x** | 6.65x | 101x |
| | SF100 | 283ms | 1.61x | **1x** | 7.42x | 73.43x |
| Q8 | SF 1 | 16ms | **1x** | 2.78x | 7.96x | 72.77x |
| | SF 10 | 45ms | 1.74x | **1x** | 15.16x | 73.78x |
| | SF100 | 1.06ms | 1.88x | **1x** | 21.55x | 25.02x |
| Q9 | SF 1 | 27ms | **1x** | 1.84x | 4.23x | 97.62x |
| | SF 10 | 115ms | **1x** | 4.05x | 4.14x | 57.84x |
| | SF100 | 1020ms | **1x** | 5.71x | 5.19x | 21.78x |
| Q10 | SF 1 | 32ms | 1.36x | **1x** | 5.88x | 31.56x |
| | SF 10 | 196ms | 1.26x | **1x** | 6.12x | 18.06x |
| | SF100 | 869ms | 1.78x | **1x** | 9.9x | 7.79x |

Table 4.17: Runtime for the best performing engine ("Baseline") and relative runtime for comparison engines.

**Queries** We choose TPC-H queries 1, 3, 5, 6, 8, 9 and 10 to benchmark, as these queries exercise the core operations of business intelligence querying and also contain interesting join patterns (except 1 and 6). The TPC-H queries are run without the `ORDER BY` clause. TPC-H queries 1 and 6 do not contain a join and demonstrate that although EmptyHeaded is designed for join queries, it can also compete on scan queries. The syntax we use to run each query in EmptyHeaded is shown in Appendix A.

### 4.4.2 End-to-End Comparisons

In Table 4.17 we show that EmptyHeaded can outperform MonetDB by up to 80x and LogicBlox by up to 270x while remaining within 1.88x of the highly optimized HyPer database. Unsurprisingly, the queries where EmptyHeaded is the farthest off the performance of the HyPer engine are TPC-H queries 1 and 8 where the output cardinality is

| Query | EmptyHeaded | -Attr. Elim. | -Sel. | -Attr. Ord. | -Group By |
|---|---|---|---|---|---|
| Q1 | 145ms | 2.54x | - | - | 875x |
| Q3 | 111ms | 2.46x | 1.55x | 1.17x | 1.37x |
| Q5 | 128ms | 1.46x | 1.80x | 72.68x | 1.12x |
| Q6 | 51ms | 4.82x | - | - | - |
| Q8 | 78ms | - | 2.67x | 8815x | 13.49x |
| Q9 | 115ms | - | 2.54x | 18.96x | 1.13x |
| Q10 | 246ms | 1.70x | 0.88x | 8.32x | 2.44x |

Table 4.18: Runtime for EmptyHeaded and relative performance without optimizations on TPC-H queries. Run at scale factor 10. '-' indicates there is no effect on the query.

small and the runtime is dominated by the `GROUP BY` operation. On queries 3 and 9, where the output cardinality is larger (and closer to worst-case), EmptyHeaded is able to compete within 11% of HyPer and sometimes outperform it. It should be noted that on TPC-H query 9, where EmptyHeaded has the largest performance advantage compared to HyPer, HyPer runs 2.91x faster when the `ORDER BY` clause is used in the query—making its performance within 39% of EmptyHeaded. Two optimizations that could further enhance EmptyHeaded's performance on these queries are placing the selection annotations in the trie such that selections are indexed and implementing a more sophisticated group by operation. We note that at scale factor 1, LogicBlox's relative numbers are worse than scale factor 100 as the runtime here is more dominated by query compilation.

### 4.4.3 Micro-Benchmarking Results

We break down the performance impact of the optimizations presented in Sections 3.1 and 3.2.

**Attribute Elimination** Table 4.18 shows that attribute elimination can enable up to a 4.82x performance advantage on the TPC-H queries. Attribute elimination is crucial on most on TPC-H queries, as these queries typically touch a small number of attributes from schemas with many attributes. Unsurprisingly, Table 4.18 shows that attribute elimination

provides the largest benefit on the scan TPC-H queries (1 and 6) because it allows EmptyHeaded to scan less data.

**Selections** As shown in Table 4.18, pushing down selections can provide up to 2.67x performance advantage on the TPC-H queries we benchmark. Although pushing down selections is generally useful, on TPC-H Q10 it actually hurts execution time. This is because this optimization adds additional GHD nodes (intermediate results) to our query plans. Still, the performance impact is small on this query ($<$12%) and on average this optimization provides a 90% performance gain across TPC-H queries.

**Attribute Order** As shown in Table 4.18, the cost-based attribute ordering optimizer presented in Section 3.2.7 can enable up to a 8815x performance advantage on TPC-H queries. Table 4.18 shows the difference between the best-cost and the worst-cost attribute orders. The most interesting queries here are TPC-H query 5 and TPC-H query 8. On TPC-H query 5, it is essential that the high cardinality `orderkey` attribute appears first. On TPC-H query 8, it is essential that the `partkey` attribute, which was connected to an equality selection, appears first. The process of assigning weights to the intersection costs in Section 3.2.7 ensures that orders satisfying these constraints are chosen.

**GROUP BY** Finally, we show in Table 4.18 that EmptyHeaded's `GROUP BY` optimizers provide up to a 875x performance advantage on TPC-H queries. The `GROUP BY` annotation optimizer provides a 875x speedup on TPC-H query 1 because `GROUP BY` is the bottleneck operation in this query and our optimizer properly selects to use a per-thread hash map instead of libcuckoo here.

## 4.5 Query Pipelines

Finally, we extend EmptyHeaded to show that such a unified query processing architecture could enable faster end-to-end applications. To do this, we add the ability for EmptyHeaded to process workloads that combine SQL queries and full machine learning algorithms. We evaluate EmptyHeaded on two modern workloads which span the domains of querying and

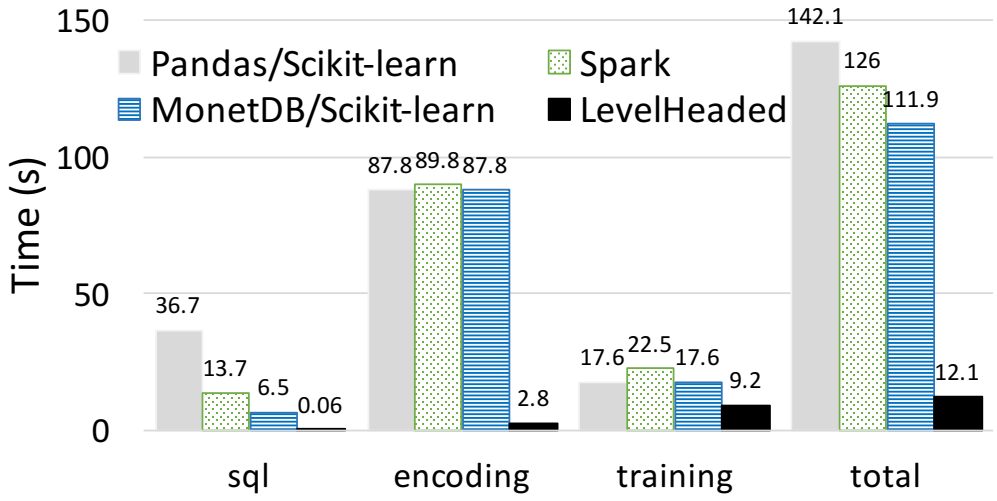



Figure 4.2: Performance of engines on the voter classification application which combines a SQL query, feature encoding, and the training of a machine learning model.

machine learning. The first is a classification task that predicts the party preferences of voters from the 2012 election using logistic regression. The second is a collaborative filtering task. Both have selection constraints on joined tables that serve as inputs to the model. We show on these full-fledged applications that EmptyHeaded can be an order of magnitude faster than the popular solutions of Spark, MonetDB/Scikit-learn, and Pandas/Scikit-learn.

### 4.5.1 Experimental Setup

We evaluate EmptyHeaded on two modern workloads which span the domains of querying and machine learning. The first is a classification task that predicts the party preferences of voters from the 2012 election using logistic regression or random forests. The second is a collaborative filtering task. Both have selection constraints on joined tables that serve as inputs to the model.

**Environment** EmptyHeaded is a shared 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 56 cores on four Intel Xeon E7-4850 v3 CPUs and 1 TB of RAM. For all engines, we chose buffer and heap sizes that were at least an order of magnitude larger than the dataset to

avoid garbage collection and swapping data out to disk.

**Metrics** For end-to-end performance, we measure the wall-clock time for each system to complete each query. We repeat each measurement seven times, eliminate the lowest and the highest value, and report the average. This measurement excludes the time used for outputting the result, data statistics collection, and index creation for all engines.

**Baselines** We compare to Spark v2.0.0 MonetDB Dec2016[3]/Scikit-learn v0.17.1, and Pandas v0.18.1/Scikit-learn v0.17.1, who all provide the ability to combine querying and learning in a single engine. Spark provides a SQL interface and machine learning algorithms through its SparkSQL and MLlib modules. MonetDB allows users to embed Python functions, and therefore take advantage of popular machine learning packages like scikit-learn. Both engines therefore provide physical independence, logical independence, and compositionality, but MonetDB places the burden of compositionality solely on the user, whereas Spark provides a seamless pipeline at the cost of sacrificing performance in each stage.

**Pipeline Breakdown** Our pipeline comparison is composed out of at most three phases: SQL, data transfer/preparation, and machine learning. Each engine runs the SQL phases using its corresponding SQL interface. The data transfer/preparation phase is the time that Spark and MonetDB spend dictionary encoding their string values from the voter classification dataset. This is necessary data preparation for the machine learning packages that each system uses, but EmptyHeaded avoids this by always dictionary encoding non-numeric values. For each machine learning algorithm in Spark, we used the provided libraries in MLlib (random forests, logistic regression, and collaborative filtering). In MonetDB we used the scikit-learn random forests and logistic regression libraries. For MonetDB collaborative filtering we implemented SGD using NumPy as no scikit-learn implementation exists. Note that EmptyHeaded splits train and test data in the SQL phase, whereas MonetDB and Spark perform this operation in the machine learning phase.

[3]Development build with embedded Python [82].

### Voter Classification

We show that EmptyHeaded can outperform Spark and MonetDB by close to an order of magnitude, on a task that runs joins and selections on input tables and then passes the result to a binary classification algorithm.

**Queries** This workload joins voter and precinct tables, filters out voters that did not cast a vote, encodes categorical features, splits the dataset into train and test sets, and runs a logistic regression or random forest classifier. For MonetDB, we used a provided implementation that was shown to achieve superior performance when compared to other RDBMS solutions on this exact workload and dataset [83].

**Dataset** The voter classification dataset consists of two tables: the first is a set of 7,503,555 voters, with information such as gender and county for each voter. The second is the 2,751 precincts in which the voters were registered.

**Logistic Regression Discussion** EmptyHeaded provides a library implemented directly in the EmptyHeaded backend which allows users to run a logistic regression classifier. Despite its lack of application-level optimizations (this is currently single-threaded), the logistic regression implementation is able to run faster than both Spark and MonetDB (see Figure 4.2). As shown in Figure 4.2, EmptyHeaded performs the SQL preprocessing step orders of magnitude faster than MonetDB and Spark. Additionally, EmptyHeaded has no costly data preparation and transformation phase, which is necessary to encode strings for the machine learning libraries in Spark and MonetDB. EmptyHeaded avoids expensive data transformations (in the encoding phase) by using its trie-based data structure for all phases. This demonstrates that non-relational algorithms can be implemented efficiently using EmptyHeaded's backend API.

**Eliminate Transformations** Our performance advantage on this query pipelines is largely due to EmptyHeaded's optimized shared-memory SQL processing and ability to minimize data transformations between the SQL and training phase. To motivate the impact of data transformations a bit further, in Table 4.19 we show the cost of converting from a column

| Dataset | Conversion | SMV | Ratio |
|---|---|---|---|
| Harbor | 0.039s | 0.0026s | 15.00 |
| HV15R | 5.76s | 0.17s | 33.88 |
| nlp240 | 7.11s | 0.17s | 41.82 |

Table 4.19: Runtime for dataset conversion, SMV query time in EmptyHeaded, and corresponding ratio (conversion/query). The conversion time measures Intel MKL's `mkl_scsrcoo` library call which is the (optimistic) time it takes to convert a column store to a acceptable sparse BLAS format. The ratio is the number of times EmptyHeaded could run the query while a column store is converting the data.

store to the compressed sparse row format used by most sparse library packages. This transformation is not necessary in EmptyHeaded as it always uses a single, trie-based data structure. As a result, Table 4.19 shows that up to 41 SMV queries can be run in EmptyHeaded in the time that it takes for a column store to convert the data to a BLAS compatible format.

### Collaborative Filtering

On a task that combines SQL queries with collaborative filtering, EmptyHeaded runs 13.11x faster than Spark and 287.63x faster than MonetDB.

**Query** Collaborative filtering is a technique used by recommender systems that aggregates the preferences of many users to predict the preferences of a single user—for example, how a certain user would rate a movie [55]. We used each engine to run a query that joined a table of movie ratings with tables of age and genre information, and selected all the ratings made by users of a certain age range on movies of certain genres. The collaborative filtering algorithm was then run on the resulting ratings.

**Dataset** The dataset consists of three tables: movie ratings, age data for each user, and genre data for each movie. For the ratings, we use the popular MovieLens dataset [4], which is a collection of 10 million ratings. The age and genre data were synthetically generated and annotated to the original MovieLens dataset.

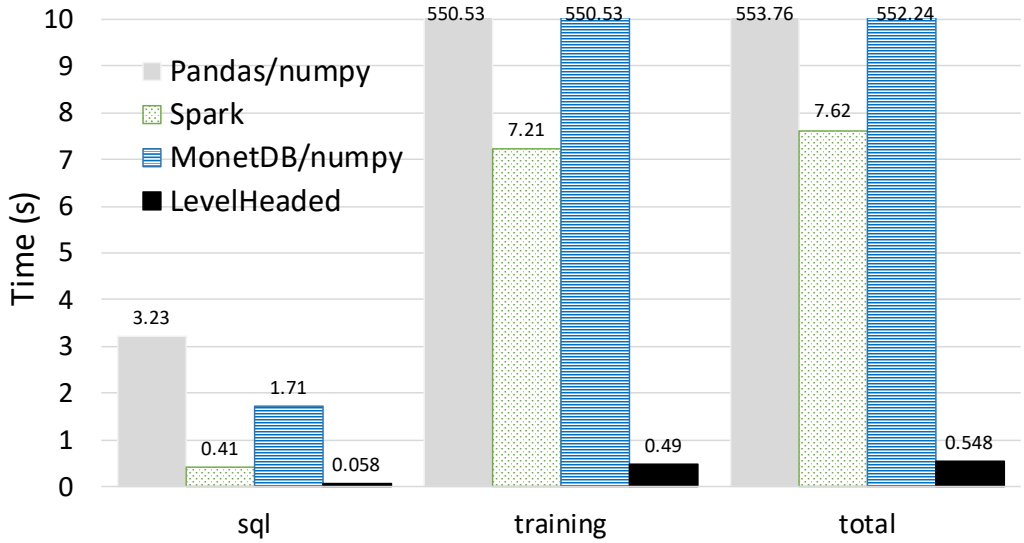



Figure 4.3: Performance of engines on the colloborative filtering application which combines a SQL query and the training of a machine learning model.

**Discussion** Figure 4.3 shows that the machine learning phase greatly outweighs the SQL phase in this workload, even though EmptyHeaded outperforms the other engines by close to an order of magnitude in the SQL phase. EmptyHeaded and MonetDB run collaborative filtering using Stochastic Gradient Descent (SGD) to learn the low-rank representations of users and movies, while Spark uses Alternating Least Squares. To provide a fair comparison in the machine learning phase, all engines are run until the root-mean-square error on the training dataset is the same. The implementation of collaborative filtering in EmptyHeaded is done with EmptyHeaded's backend API and is presented as a library to the user. When run using standard single-threaded SGD, EmptyHeaded is able to converge in 1.92s, over 3x faster than Spark and over 200x faster than our single-threaded NumPy implementation. EmptyHeaded also provides the option to run SGD with a parallelized update scheme [77], and that enables EmptyHeaded to outperform Spark and MonetDB by over an order of magnitude (see Figure 4.3). There is no transformation cost for any engine in this workload as all input data is numeric.

# Chapter 5

# Related Work

Our work extends previous work in seven main areas: join processing, relational processing, SIMD processing, set intersection processing, graph processing, RDF processing, and linear algebra processing. We cover each of these in detail.

**Join Processing** The first worst-case optimal join algorithm was recently derived [73, 74]. The LogicBlox (LB) engine [102] is the first commercial database engine to use a worst-case optimal algorithm. Researchers have also investigated worst-case optimal joins in distributed settings [25] and have looked at minimizing communication costs [9] or processing on compressed representations [78]. Recent theoretical advances [47, 52] 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 derived from the worst-case optimal join algorithm [73, 74] 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 2.4.

**Relational Processing** Most relational database engines [30, 44, 50] since System-R [12] have used the pairwise (over relation) join algorithms that work with Sellinger-style [12] query optimizers. This is fundamentally different from the worst-case optimal join (multi-way over attribute) algorithm and GHD-based query optimizer in EmptyHeaded.

**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 [85]. Our approach is similar in spirit to DB2 Blu, but applied specifically to join processing. Other approaches such as WideTable [64] and BitWeaving [63] 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 [106], and classic database operations [107] 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 [45, 49, 58, 93]. Techniques such as the SIMDShuffling algorithm [49] break the min property of set intersection but often work well on graph data, while techniques such as SIMDGalloping [58] that preserve the min property rarely work well on graph data. We experiment with these techniques and slightly modify our use of them to ensure min property of the set intersection operation in our engine. We use this as a means to speed up set intersection—the core operation in our join algorithm.

**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 [94], Green-Marl [43], Ligra [95], and Galois [75]. With the exception of SociaLite, each of these engines proposes a new domain-specific language for graph analytics. SociaLite, based on datalog, presents an engine that more closely resembles a relational model. Other engines such as PowerGraph [36], Graph-X [37], and Pregel [66] are aimed at scale-out performance. Additionally, there have been several recent systems that focus on using graph mining as the basis for graph computation [15, 17, 31, 81, 100]. For example, Arabesque [100] and NScale [81] are two such systems designed for scale-out performance on core graph mining problems such as finding subgraphs or motif counting. The merit of these specialized approaches against traditional OLAP engines is a source of much debate [103],

as some researchers believe general approaches can compete with and outperform these specialized designs [37, 67]. Recent products, such as SAP HANA, integrate graph accelerators as part of a OLAP engine [88]. Others [34] have shown that relational engines can compete with distributed engines [36,66] 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.

**RDF Engines** Two of the most popular specialized RDF engines are RDF-3X and TripleBit. Both accept queries in the SPARQL query language and have been shown to significantly outperform traditional relational engines. RDF-3X creates a full set of subject-predicate-object indexes by building clustering B+ trees on all six permutations of the triples [71]. RDF-3X also maintains nine aggregate indexes, which include all six binary and all three unary projections. Each index provides some selectivity estimates and the aggregate indexes are used to select the fastest index for a given query. In the TripleBit engine, RDF triples are represented using a compact matrix representation [105]. TripleBit stores two auxiliary index structures and two binary aggregate indexes which enable selectivity estimates for query patterns. This enables TripleBit to select the most effective indexes, minimize the number of indexes needed, and determine the query plan. Like EmptyHeaded, both RDF-3X and TripleBit use dictionary encoding.

**Linear Algebra Processing** Researchers have long studied how to implement high-performance linear algebra kernels. Intel MKL [1] represents the culmination of this work on Intel CPUs. Unsurprisingly, researchers have shown [98] that it requires tedious low-level code optimizations to come near the performance of a BLAS package like Intel MKL. As a result, processing these queries in a traditional RDBMS (using relational operators) is at least one order of magnitude slower than using such packages (see Section 4.3). In response, researchers have released array databases, like SciDB [19] and TileDB [79], which provide high-level linear algebra querying, often by wrapping BLAS libraries. In contrast, our goal is not to design an entirely different and specialized engine for these workloads, but rather to design a single (relational) engine that processes multiple classes of queries efficiently.

A significant amount of work has focused on bringing linear algebra to these pairwise relational data processing engines. Some have suggested treating linear algebra objects as first class citizens in a column store [51]. Others, such as Oracle's UTL_NLA [2] and MonetDB with embedded Python [82], allow users to call linear algebra packages through user defined functions. Still, the relational optimizers in these approaches do not see, and therefore are incapable of optimizing, the linear algebra routines. Even worse, these packages place significant burden on the user to make low-level library calls. Finally, the SimSQL project [65] suggests that relational engines can be modified in straightforward ways to accommodate linear algebra workloads. Our goals are similar to SimSQL, but explored with different mechanics. SimSQL studied the necessary modifications for a classic database architecture to support linear algebra queries and was only evaluated on distributed linear algebra queries. Other high performance in-memory databases, like HyPer, focus on classic OLTP and OLAP workloads and were not designed with other workloads in mind.

# Chapter 6

# Conclusions

In this dissertation we present the first, general-purpose worst-case optimal join processing engine that competes with low-level specialized engines on their own benchmarks in the graph, RDF, business intelligence, and linear algebra domains. In the query compiler, our use of GHDs provides strong worst-case running times and can lead to over a three orders of magnitude performance gain over other relational designs. We detail several, first of their kind, constant factor optimizations for a GHD-based query compiler that are necessary for such a query architecture to be efficient, contributing to over three orders of magnitude performance gains. In the execution engine, we perform a detailed study of the set layouts and algorithms necessary to exploit SIMD parallelism on modern hardware. We show that over a three orders of magnitude performance gain can be achieved by selecting among algorithmic choices for set intersection and set layouts at different granularities of the data. Combining all of these optimizations, this dissertation demonstrates that a query architecture built around worst-case optimal joins and GHDs can be efficient on standard workloads in multiple application domains. We believe our results are promising and suggest that such a query architecture could serve as the foundation for future unified querying engines.

# Appendix A

# TPC-H Query Syntax

### Query 1:

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_quantity) as sum_qty,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
    avg(l_quantity) as avg_qty,
    avg(l_extendedprice) as avg_price,
    avg(l_discount) as avg_disc,
    count(*) as count_order
FROM
    lineitem
WHERE
    lineitem.l_shipdate <= date '1998-12-01'
GROUP BY
    l_returnflag,
    l_linestatus);
```

### Query 3:

```
SELECT
    l_orderkey,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    o_orderdate,
    o_shippriority
FROM
    customer,
    orders,
```

```
  lineitem
WHERE
  c_mktsegment = 'BUILDING'
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < date '1995-03-15'
  and l_shipdate > date '1995-03-15'
GROUP BY
  l_orderkey,
  o_orderdate,
  o_shippriority);
```

### Query 5:

```
SELECT
  l_orderkey,
  sum(l_extendedprice * (1 - l_discount)) as revenue,
  o_orderdate,
  o_shippriority
FROM
  customer,
  orders,
  lineitem
WHERE
  c_mktsegment = 'BUILDING'
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < date '1995-03-15'
  and l_shipdate > date '1995-03-15'
GROUP BY
  l_orderkey,
  o_orderdate,
  o_shippriority);
```

### Query 6:

```
SELECT
  sum(l_extendedprice * l_discount) as revenue
FROM
  lineitem
WHERE
  l_shipdate >= date '1994-01-01'
  and l_shipdate < date '1995-01-01'
  and l_discount >= 0.05 and l_discount <= 0.07
  and l_quantity < 24
```

### Query 8:

```
SELECT
  o_year,
  sum(case
    when nation = 'BRAZIL' then volume
    else 0
  end) / sum(volume) as mkt_share
FROM
  (
    SELECT
      extract(year from o_orderdate) as o_year,
      l_extendedprice * (1 - l_discount) as volume,
      n2.n_name as nation
    FROM
      part,
      supplier,
      lineitem,
      orders,
      customer,
      nation as n1,
      nation as n2,
      region
    WHERE
      p_partkey = l_partkey
      and s_suppkey = l_suppkey
      and l_orderkey = o_orderkey
      and o_custkey = c_custkey
      and c_nationkey = n1.n_nationkey
      and n1.n_regionkey = r_regionkey
      and r_name = 'AMERICA'
      and s_nationkey = n2.n_nationkey
      and o_orderdate >= date '1995-01-01'
      and o_orderdate <= date '1996-12-31'
      and p_type = 'ECONOMY ANODIZED STEEL'
  ) as all_nations
GROUP BY
  o_year
```

### Query 9:

```
SELECT
  nation,
  o_year,
  sum(amount) as sum_profit
FROM
  (
    SELECT
      n_name as nation,
```

```
      extract(year from o_orderdate) as o_year,
      l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount
    FROM
      part,
      supplier,
      lineitem,
      partsupp,
      orders,
      nation
    WHERE
      s_suppkey = l_suppkey
      and ps_suppkey = l_suppkey
      and ps_partkey = l_partkey
      and p_partkey = l_partkey
      and o_orderkey = l_orderkey
      and s_nationkey = n_nationkey
      and p_name like '%green%'
  ) as profit
GROUP BY
  nation,
  o_year
```

# Bibliography


[1] Intel Math Kernel Library. https://software.intel.com/en-us/mkl.

[2] Oracle corporation. https://docs.oracle.com/cd/B1930-6_01/index.html.

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

[4] Movielens 10m network dataset – KONECT, October 2016.

[5] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422. ACM, 2007.

[6] Christopher R Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. In *Proceedings of the 2016 International Conference on Management of Data*, pages 431–446. ACM, 2016.

[7] Christopher R Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. Old techniques for new join algorithms: A case study in rdf processing. *DESWEB: ICDE Workshop*, 2016.

[8] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*, volume 8. Addison-Wesley, 1995.

[9] Foto N. Afrati, Manas Joglekar, Christopher Ré, Semih Salihoglu, and Jeffrey D. Ullman. GYM: A multiround join algorithm in mapreduce. *CoRR*, abs/1410.4156, 2014.

[10] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. *LAPACK Users' guide*. SIAM, 1999.

[11] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the logicblox system. In *SIGMOD Conference*, pages 1371–1382. ACM, 2015.

[12] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. System R: relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.

[13] Medha Atre, Vineet Chaoji, Mohammed J Zaki, and James A Hendler. Matrix bit loaded: a scalable lightweight join query processor for rdf data. In *Proceedings of the 19th international conference on World wide web*, pages 41–50. ACM, 2010.

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

[15] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. Densest subgraph in streaming and mapreduce. *Proceedings of the VLDB Endowment*, 5(5):454–465, 2012.

[16] Scott Beamer, Krste Asanovic, and David A. Patterson. Direction-optimizing breadth-first search. In *SC*, page 12. IEEE/ACM, 2012.

[17] Mansurul A Bhuiyan and Mohammad Al Hasan. An iterative mapreduce based frequent subgraph mining algorithm. *IEEE transactions on knowledge and data engineering*, 27(3):608–620, 2015.

[18] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al.

An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[19] Paul G Brown. Overview of scidb: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 963–968. ACM, 2010.

[20] Diane Bryant. Live from intel ai day 2016.

[21] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. On random sampling over joins. In *SIGMOD Conference*, pages 263–274. ACM Press, 1999.

[22] Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. In *ICDT*, volume 1186 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 1997.

[23] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, 1985.

[24] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *KDD*, pages 219–228. ACM, 2009.

[25] Shumo Chu, Magdalena Balazinska, and Dan Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD Conference*, pages 63–78. ACM, 2015.

[26] Peter Cohan. Mongodb taking share from oracle in $40 billion market. 2017.

[27] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. *SIGMOD Rec.*, 14(4):268–279, May 1985.

[28] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[29] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[30] Kalen Delaney. *Inside Microsoft SQL Server 2000*. Microsoft Press, 2000.

[31] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *VLDB*, 7(7):517–528, 2014.

[32] A. Aho et al. *Compilers, Principles, Techniques*. Addison wesley, 1986.

[33] D. Abadi et al. Column-stores vs. row-stores: How different are they really? *SIGMOD '08*, pages 967–980.

[34] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. The case against specialized graph analytics engines. In *CIDR*. www.cidrdb.org, 2015.

[35] Hector Garcia-Molina. *Database systems: the complete book*. Pearson Education India, 2008.

[36] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30. USENIX Association, 2012.

[37] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613. USENIX Association, 2014.

[38] Georg Gottlob, Martin Grohe, Nysret Musliu, Marko Samer, and Francesco Scarcello. Hypertree decompositions: Structure, algorithms, and applications. In *WG*, volume 3787 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2005.

[39] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, pages 31–40. ACM, 2007.

[40] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[41] Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan, Kevin Lai, Shuo Wu, Sandeep Govind Dhoot, Abhilash Rajesh Kumar, Ankur Agiwal, et al. Mesa: Geo-replicated, near real-time, scalable data warehousing. *Proceedings of the VLDB Endowment*, 7(12):1259–1270, 2014.

[42] Brandon Haynes, Alvin Cheung, and Magdalena Balazinska. Pipegen: Data pipe generator for hybrid analytics. *arXiv preprint arXiv:1605.01664*, 2016.

[43] Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. Green-marl: a DSL for easy and efficient graph analysis. In *ASPLOS*, pages 349–362. ACM, 2012.

[44] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, Martin Kersten, et al. Monetdb: Two decades of research in column-oriented database architectures. *A Quarterly Bulletin of the IEEE Computer Society Technical Committee on Database Engineering*, 35(1):40–45, 2012.

[45] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. Faster set intersection with SIMD instructions by reducing branch mispredictions. *PVLDB*, 8(3):293–304, 2014.

[46] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Computing surveys (CsUR)*, 16(2):111–152, 1984.

[47] Manas Joglekar, Rohan Puttagunta, and Christopher Ré. Aggregations over generalized hypertree decompositions. *CoRR*, abs/1508.07532, 2015.

[48] Manas R Joglekar, Rohan Puttagunta, and Christopher Ré. Ajar: Aggregations and joins over annotated relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 91–106. ACM, 2016.

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

[50] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206. IEEE, 2011.

[51] David Kernert, Frank Köhler, and Wolfgang Lehner. Bringing linear algebra objects to life in a column-oriented in-memory database. In *In Memory Data Management and Analysis*, pages 44–55. Springer, 2015.

[52] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: questions asked frequently. In *PODS*, pages 13–28. ACM, 2016.

[53] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. Taming subgraph isomorphism for RDF query processing. *CoRR*, abs/1506.01973, 2015.

[54] Graham Klyne and Jeremy J Carroll. Resource description framework (rdf): Concepts and abstract syntax. 2006.

[55] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, Aug 2009.

[56] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600. ACM, 2010.

[57] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.*, 45(1):1–29, 2015.

[58] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. SIMD compression and the intersection of sorted integers. *CoRR*, abs/1401.6399, 2014.

[59] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[60] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Statistical properties of community structure in large social and information networks. In *WWW*, pages 695–704. ACM, 2008.

[61] Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

[62] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, page 27. ACM, 2014.

[63] Yinan Li and Jignesh M. Patel. Bitweaving: fast scans for main memory data processing. In *SIGMOD Conference*, pages 289–300. ACM, 2013.

[64] Yinan Li and Jignesh M. Patel. Widetable: An accelerator for analytical data processing. *PVLDB*, 7(10):907–918, 2014.

[65] Shangyu Luo, Zekai J Gao, Michael Gubanov, Luis L Perez, and Christopher Jermaine. Scalable linear algebra on a relational database system. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 523–534. IEEE, 2017.

[66] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146. ACM, 2010.

[67] Frank McSherry, Michael Isard, and Derek Gordon Murray. Scalability! but at what cost? In *HotOS*. USENIX Association, 2015.

[68] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[69] Alan Mislove, Massimiliano Marcon, P. Krishna Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Internet Measurement Comference*, pages 29–42. ACM, 2007.

[70] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[71] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.

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

[73] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In *PODS*, pages 37–48. ACM, 2012.

[74] Hung Q Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record*, 42(4):5–16, 2014.

[75] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471. ACM, 2013.

[76] Dung T. Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Join processing for graph patterns: An old dog with new tricks. In *GRADES@SIGMOD/PODS*, pages 2:1–2:8. ACM, 2015.

[77] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, pages 693–701, USA, 2011. Curran Associates Inc.

[78] Dan Olteanu and Jakub Závodný. Size bounds for factorised representations of query results. *ACM Trans. Database Syst.*, 40(1):2, 2015.

[79] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. The tiledb array data storage manager. *Proc. VLDB Endow.*, 10(4):349–360, November 2016.

[80] Md Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Jongsoo Park, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, Sergey G Pudov, Vadim O Pirogov, and Pradeep Dubey. Parallel efficient sparse matrix-matrix multiplication on multicore platforms. In *International Conference on High Performance Computing*, pages 48–57. Springer, 2015.

[81] Abdul Quamar, Amol Deshpande, and Jimmy Lin. Nscale: neighborhood-centric large-scale graph analytics in the cloud. *VLDB*, 25(2):125–150, 2016.

[82] Mark Raasveldt. Embedded python/numpy in monetdb. `https://www.monetdb.org/blog/embedded-pythonnumpy-monetdb`, 2016.

[83] Mark Raasveldt. Voter classification using monetdb/python. `https://www.monetdb.org/blog/voter-classification-using-monetdbpython`, 2016.

[84] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. Db2 with blu acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment*, 6(11):1080–1091, 2013.

[85] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[86] Lauro Rizzatti. Digital data storage is undergoing mind-boggling growth. 2016.

[87] Barry Rogoff. Logical schema design.

[88] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The graph story of the SAP HANA database. In *BTW*, volume 214 of *LNI*, pages 403–420. GI, 2013.

[89] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *SIGMOD Conference*, pages 979–990. ACM, 2014.

[90] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *WEA*, volume 3503 of *Lecture Notes in Computer Science*, pages 606–609. Springer, 2005.

[91] Olaf Schenk, Andreas Wächter, and Martin Weiser. Inertia-revealing preconditioning for large-scale nonconvex constrained optimization. *SIAM Journal on Scientific Computing*, 31(2):939–960, 2008.

[92] B. Schiefer. Method for estimating cardinalities for query processing in a relational database management system, June 2 1998. US Patent 5,761,653.

[93] Benjamin Schlegel, Thomas Willhalm, and Wolfgang Lehner. Fast sorted-set intersection using SIMD instructions. In *ADMS@VLDB*, pages 1–8, 2011.

[94] Jiwon Seo, Stephen Guo, and Monica S. Lam. Socialite: Datalog extensions for efficient social network analysis. In *ICDE*, pages 278–289. IEEE Computer Society, 2013.

[95] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPOPP*, pages 135–146. ACM, 2013.

[96] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *2015 Data Compression Conference*, pages 403–412. IEEE, 2015.

[97] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, et al. F1: A distributed sql database that scales. *Proceedings of the VLDB Endowment*, 6(11):1068–1079, 2013.

[98] Tyler M Smith, Robert Van De Geijn, Mikhail Smelyanskiy, Jeff R Hammond, and Field G Van Zee. Anatomy of high-performance many-threaded matrix multiplication. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1049–1059. IEEE, 2014.

[99] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented DBMS. In *VLDB*, pages 553–564. ACM, 2005.

[100] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: a system for distributed graph mining. In *SOSP*, pages 425–440. ACM, 2015.

[101] Susan Tu and Christopher Ré. Duncecap: Query plans using generalized hypertree decompositions. In *SIGMOD Conference*, pages 2077–2078. ACM, 2015.

[102] Todd L. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *CoRR*, abs/1210.0481, 2012.

[103] Adam Welc, Raghavan Raman, Zhe Wu, Sungpack Hong, Hassan Chafi, and Jay Banerjee. Graph analysis: do we have to reinvent the wheel? In *GRADES*, page 7. CWI/ACM, 2013.

[104] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94. IEEE Computer Society, 1981.

[105] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. Triplebit: a fast and compact system for large scale RDF data. *PVLDB*, 6(7):517–528, 2013.

[106] Steffen Zeuch, Johann-Christoph Freytag, and Frank Huber. Adapting tree structures for processing with SIMD instructions. In *EDBT*, pages 97–108. OpenProceedings.org, 2014.

[107] Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD Conference*, pages 145–156. ACM, 2002.