Mind the Gap: Bridging Multi-Domain Query Workloads with EmptyHeaded

Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré
Stanford University
{caberger,lamb,kunle,chrismre}@stanford.edu

ABSTRACT

Executing domain specific workloads from a relational data warehouse is an increasingly popular task. Unfortunately, classic relational database management systems (RDBMS) are suboptimal in many domains (e.g., graph and linear algebra queries), and it is challenging to transfer data from an RDBMS to a domain specific toolkit in an efficient manner. This demonstration showcases the EmptyHeaded engine: an interactive query processing engine that leverages a novel query architecture to support efficient execution in multiple domains. To enable a unified design, the EmptyHeaded architecture is built around recent theoretical advancements in join processing and automated in-query data transformations. This demonstration highlights the strengths and weaknesses of this novel type of query processing architecture while showcasing its flexibility in multiple domains. In particular, attendees will use EmptyHeaded's Jupyter notebook front-end to interactively learn the theoretical advantages of this new (and largely unknown) approach and directly observe its performance impact in multiple domains.

1. INTRODUCTION

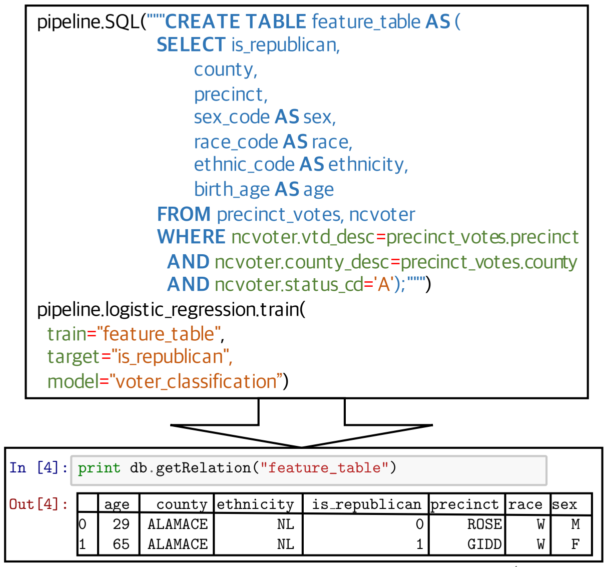
An increasing diversity of data has created the need for relational database management systems (RDBMS) that are efficient in multiple domains or integrate easily with domain specific analytics toolkits (like R or Scikit-learn). For example, an RDMBS might contain relations that represent a social network, a knowledge graph, a standard business intelligence data source, or feature vectors to be fed to a machine learning model. To analyze this data efficiently, conventional wisdom suggests that external toolkits should be used to augment an RDMBS. Unfortunately, it is not always clear which external toolkits should be used under what conditions. Even worse, integrating an external toolkit with a RDBMS adds complexity to the overall query subsystem and requires costly data transformations.

In this demonstration we showcase the EmptyHeaded engine: an interactive query processing engine that uses a single, novel query processing architecture to efficiently unify execution in multiple domains. EmptyHeaded provides a Jupyter notebook front-end that enables users to issue standard SQL queries or compose pipelines which mix SQL queries and machine learning algorithms. To process SQL queries, EmptyHeaded uses *generalized hypertree decompositions* (GHD) [3] to select a query plan [6] and a *worst-case optimal join algorithm* [4] to execute a query plan. Combined, these novel components of the EmptyHeaded architecture provide a stronger theoretical runtime guarantee than almost all other join processing engines [1]. To process mixed SQL and machine learning queries, EmptyHeaded combines its SQL (GHD-based) query plans and machine learning algorithms into a single pipeline—enabling EmptyHeaded to fuse computations and optimize feature engineering transformations across phases. EmptyHeaded enables users to interact with this novel query subsystem by inspecting both optimized and unoptimized query plans and by directly observing their runtime differences.

This work highlights the theoretical benefits and systems optimizations fundamental to the EmptyHeaded design, which are difficult to understand without a demonstration. In particular, attendees will interactively learn the novel components of the EmptyHeaded query architecture that enable it to be efficient on queries in the graph, business intelligence, and machine learning domains.

- **Graph Processing:** We have shown that the EmptyHeaded engine can translate theoretical advantages to runtime advantages on join queries in the graph and RDF domains [1, 2]. As such, we provide the first interactive tutorial of worst-case optimal join algorithms and GHDs in this part of the demonstration. We show first-hand that the theoretical advantage of our techniques translates to runtime advantages by providing attendees with the ability to select, visualize, and execute either (1) an optimized GHD-based query plan or (2) an unoptimized query plan which uses only the worst-case optimal join algorithm. In conjunction, we describe the theoretical advantages of optimized GHD-based query plans and show they provide up to a 3000x runtime improvement on graph pattern queries over real graph datasets.
- **Business Intelligence:** Unfortunately, worst-case optimal joins and GHDs do not always provide a theoretical advantage, and are not a direct match for most standard business intelligence queries. Even so, we show that EmptyHeaded can achieve competitive per-

Proceedings of the VLDB Endowment, Vol. 10, No. 12
Copyright 2017 VLDB Endowment 2150-8097/17/08.



```
pipeline.SQL("""CREATE TABLE feature_table AS (
                SELECT is_republican,
                    county,
                    precinct,
                    sex_code AS sex,
                    race_code AS race,
                    ethnic_code AS ethnicity,
                    birth_age AS age
                FROM precinct_votes, ncvoter
                WHERE ncvoter.vtd_desc=precinct_votes.precinct
                 AND ncvoter.county_desc=precinct_votes.county
                 AND ncvoter.status_cd='A');""")
pipeline.logistic_regression.train(
  train="feature_table",
  target="is_republican",
  model="voter_classification")
```

```
In [4]:  print db.getRelation("feature_table")

Out[4]:     age   county  ethnicity  is_republican  precinct  race  sex
         0   29   ALAMACE        NL              0      ROSE     W    M
         1   65   ALAMACE        NL              1      GIDD     W    F
```

Figure 1: Example Jupyter notebook interaction (input and output) with the EmptyHeaded engine. The input is a EmptyHeaded pipeline consisting of a SQL query followed by the training of a logistic regression model. The output of each stage (the SQL stage is shown here) can be returned as a Pandas DataFrame.

formance[1] on such workloads by adding two classic database optimizations to its GHD-based query compiler: (1) pushing down selections and (2) join ordering. In this part of the demonstration attendees will visually inspect these query optimizations in EmptyHeaded. They will also experience the effect of these optimization choices on the end-to-end query execution time. More precisely, the attendees will experience up to a 2x performance difference due to pushing down selections and up to a 5x performance difference for various join orders.

- **Machine Learning:** A common task for an analyst is to issue a query over relational data which *filters* relations via selections and *de-normalizes* relations via joins to produce a single feature set to train a machine learning model [7]. Unfortunately, most modern solutions require an analyst to transfer the data from the RDMBS, and perform some feature engineering tasks prior to training the model. EmptyHeaded is designed to eliminate some of the complexity in these pipelines by automatically fusing data transformations, such as the encoding of categorical variables, into the query. In this part of the demonstration attendees will experience the performance and complexity difference between running such a pipeline entirely inside EmptyHeaded, versus Pandas and Scikit-learn. Attendees will see how feature engineering steps are eliminated in EmptyHeaded and how an end-to-end workload can be an order of magnitude faster in EmptyHeaded than Pandas and Scikit-learn.

[1]EmptyHeaded has been benchmarked on TPC-H queries 1,3,5,6,8 at scale factors 1 and 100 and performed at worst 35% off the best-of-breed HyPer (v0.5.0) database engine on a single machine with a total of 56 cores on four Intel Xeon E7-4850 v3 CPUs.

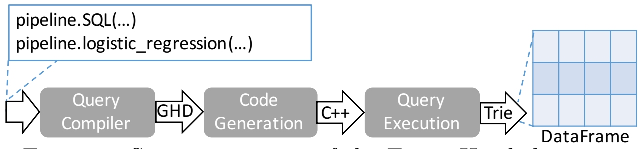


Figure 2: System overview of the EmptyHeaded engine.

By the end of this demonstration attendees will understand the tradeoffs, optimizations, and theoretical guarantees necessary to design a query processing architecture that uses worst-case optimal joins and GHDs. Throughout the demonstration attendees will interact directly with the EmptyHeaded engine to gain a deeper understanding its architecture and observe its flexibility in multiple domains.

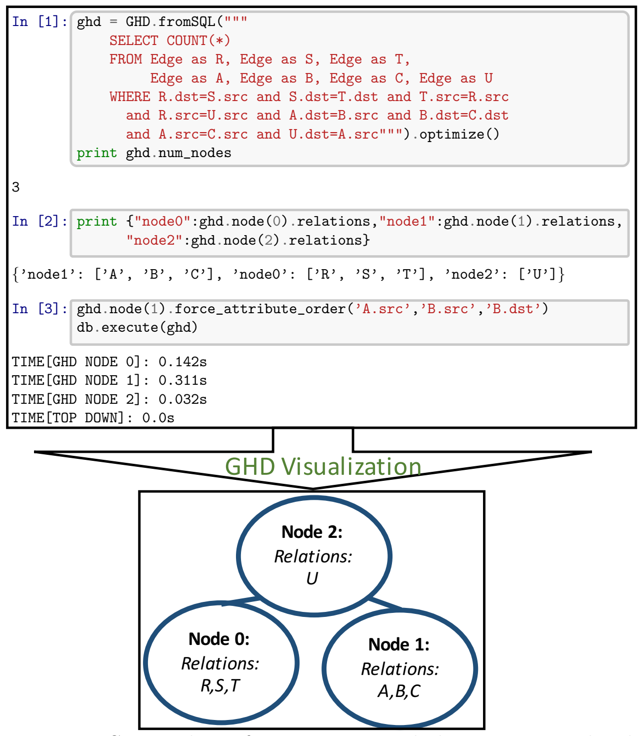
2. SYSTEM OVERVIEW

We briefly overview the input and output, query execution, and pipeline architecture of the EmptyHeaded engine.

System Input and Output. The EmptyHeaded engine accepts data from a Pandas DataFrame or from a comma separated value (CSV) file on disk. Queries are returned to the user in the form of a Pandas DataFrame. The EmptyHeaded engine accepts queries written in SQL and supports conjunctive queries with selections and aggregations. Additionally, EmptyHeaded enables users to compose pipelines that combine both SQL and machine learning stages (see Figure 1).

Query Execution. The EmptyHeaded engine translates each SQL query to a GHD [3], which is a directed acyclic graph (DAG) that represents a query plan for the worst-case optimal join algorithm. The theoretical benefits of GHDs extend beyond the benefits of the worst-case optimal join algorithm [6]. To select a GHD-based query plan (or GHD) EmptyHeaded uses a brute-force search to find the query plan (GHD) with the tightest theoretical guarantees. After a query plan (GHD) is selected, the generic worst-case optimal join algorithm [4] is used, at each node in the DAG, to generate C++ code for the query. As we show in this demonstration, EmptyHeaded uses simple heuristics to determine the order attributes are processed in the worst-case optimal join algorithm (and therefore the order in which attributes are emitted during code generation). Next, the emitted code is executed over the input relations (stored as tries) and it produces a single output relation (again stored as a trie). An overview of this process is shown in Figure 2.

Pipeline Architecture. The EmptyHeaded engine supports a pipeline architecture which, similar to Spark ML Pipelines [5], enables users to mix SQL and machine learning workloads in a single pipeline. The goal of EmptyHeaded pipelines is to enable optimizations across workloads that combine SQL and machine learning operations. To simplify such optimizations, all pipeline stages in EmptyHeaded use a single trie-based data storage model. This storage model, although designed for the worst-case optimal join algorithm, is versatile enough to provide high-performance on other common tasks. EmptyHeaded pipelines currently support fusion of one-hot encoding in pipelines containing SQL queries and a logistic regression or collaborative filtering learning algo-



```
In [1]: ghd = GHD.fromSQL("""
            SELECT COUNT(*)
            FROM Edge as R, Edge as S, Edge as T,
                 Edge as A, Edge as B, Edge as C, Edge as U
            WHERE R.dst=S.src and S.dst=T.dst and T.src=R.src
              and R.src=U.src and A.dst=B.src and B.dst=C.dst
              and A.src=C.src and U.dst=A.src""").optimize()
        print ghd.num_nodes

3

In [2]: print {"node0":ghd.node(0).relations,"node1":ghd.node(1).relations,
               "node2":ghd.node(2).relations}

{'node1': ['A', 'B', 'C'], 'node0': ['R', 'S', 'T'], 'node2': ['U']}

In [3]: ghd.node(1).force_attribute_order('A.src','B.src','B.dst')
        db.execute(ghd)

TIME[GHD NODE 0]: 0.142s
TIME[GHD NODE 1]: 0.311s
TIME[GHD NODE 2]: 0.032s
TIME[TOP DOWN]: 0.0s
```

Figure 3: Screenshot of an EmptyHeaded Jupyter notebook which shows a user entering the Barbell query and inspecting its GHD. The screenshot also shows how users can force a specific attribute order for the worst-case optimal join algorithm. An image of the GHD for the Barbell query, which is embedded in our Jupyter notebook tutorial, is shown below the screenshot of the EmptyHeaded interface.

rithm. We are adding support for more machine learning algorithms and the fusion of other feature engineering tasks (such as bucketization and feature crosses).

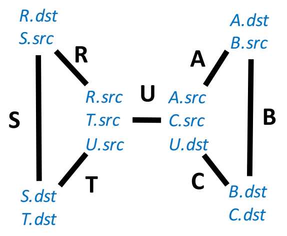
3. DEMONSTRATED FEATURES

Our demonstration of EmptyHeaded is composed of queries in three domains: (1) graph processing queries where the theoretical advantages of worst-case optimal joins and GHDs are exercised, (2) business intelligence queries where the classic database optimizations of join ordering and pushing down selections are exercised, and (3) mixed analytics queries which exercise the EmptyHeaded pipeline architecture. Our demonstration will focus on a single query in each domain (described next), but will also contain additional queries for attendees to experiment within each domain.

3.1 Graph Processing

Our first demonstration query is a graph pattern query that includes an interactive introduction to the theoretical advantages of worst-case optimal joins and GHDs.

EXAMPLE 3.1. *Attendees will run a query which finds all pairs of triangles connected by a path of length one. We call this the Barbell query. Attendees will experiment with this query on several real graph datasets, with the default being a Facebook social network graph. The Barbell query pattern is shown next.*



Warm Up: Worst-Case Optimal Joins. As an introduction to the EmptyHeaded engine, attendees will run the popular triangle counting query, whose pattern is contained in the Barbell query. Attendees will use a Jupyter notebook tutorial that highlights the asymptotic difference between pair-wise ($O(N^2)$) and worst-case optimal ($O(N^{3/2})$) join engines on this query. Thus, for the first time, attendees will have the opportunity to experience the effect of worst-case optimal joins in a practical analysis scenario.

Query Plans: GHDs. Next, attendees will experiment with the Barbell query using the interface presented in Figure 3. Attendees be able to execute two different query plans for the Barbell query: (1) a query plan which uses only the worst-case optimal join algorithm and (2) a query plan which uses the GHD-based query plan shown in Figure 3. Attendees will learn that the GHD-based query plan has a worst-case running time of $O(N^{3/2} + \text{OUT})$ where OUT is the size of the output, whereas the query plan that uses only the worst-case optimal join algorithm has a bound of $\Omega(N^3)$. Attendees will experience this asymptotic difference translating to a large empirical difference (>3000x) by running the aggregation version of Barbell query shown in Figure 3 on both query plans.

3.2 Business Intelligence

The second part of the demonstration uses a standard business intelligence query, where worst-case optimal joins and GHDs are not theoretically superior[2] to traditional pair-wise approaches. As such, to process these queries efficiently it is necessary to add classic database optimizations, like pushing down selections and different join orders, to the EmptyHeaded query architecture. In this part of the demonstration attendees will learn how these optimizations are added to the GHD-based query plans in EmptyHeaded, and again experience their impact on overall query runtime.

EXAMPLE 3.2. *Attendees will run TPC-H query 5 at scale factor 1, which is a standard business intelligence benchmark query. A GHD for this query is shown in Figure 4.*

Pushing Down Selections. Processing selection constraints as early as possible in the query plan is a classic optimization for business intelligence queries. While it is standard that traditional query optimizers push these selection constraints down as far as possible in the query plan, it is not obvious how to add such optimizations to a GHD-based query compiler. In this part of the demonstration, attendees will learn how EmptyHeaded pushes down selections by adding additional nodes to its GHD-based query

[2]Most business intelligence queries are acyclic joins and/or have selection constraints that restrict the size of the output.

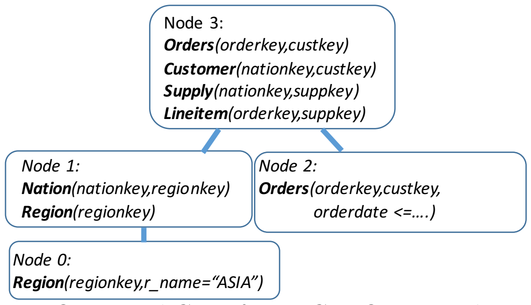

Node 3:
Orders(orderkey,custkey)
Customer(nationkey,custkey)
Supply(nationkey,suppkey)
Lineitem(orderkey,suppkey)
Node 1:
Nation(nationkey,regionkey)
Region(regionkey)
Node 2:
Orders(orderkey,custkey, orderdate <=....)
Node 0:
Region(regionkey,r_name="ASIA")

Figure 4: Optimized GHD for TPC-H Query 5 where selections are pushed down. The unoptimized GHD, w/o pushing down selections, contains only nodes 1 and 3.

plans. In particular the GHD for TPC-H query 5 (see Figure 4) will be investigated and discussed. The attendees will inspect the optimized and unoptimized query plans for this query and observe up to a 2x performance difference.

Attribute Order. An important component of the worst-case optimal join algorithm is the order in which attributes are processed. In this part of the demonstration users will experience the effect of different attribute orders when executing only GHD node 3 of TPC-H query 5 (see Figure 4), which is where over 98% of the total query runtime occurs. The attendees will force different attribute orders here and learn the heuristics (loop independent intersections and parallelization hints) EmptyHeaded uses to select its attribute order. While forcing different attribute orders, attendees will be able to experience up to a 5x runtime difference between their best and worst possible choices.

3.3 Machine Learning

The final portion of this demonstration uses a workload that is composed of two phases: (1) a SQL phase and (2) a machine learning phase. Attendees will learn how to use EmptyHeaded pipelines in this part of the demonstration and will directly compare the performance and verbosity of completing this pipeline in EmptyHeaded versus the popular Pandas and Scikit-learn toolkits.

Example 3.3. *Attendees will be presented with a real dataset of voter information from North Carolina which contains two relations: (1) one containing information about individual voters (7,503,555 rows) and (2) one containing information about voting precincts in North Carolina (2,751 rows). Attendees will use logistic regression to train a machine learning model that predicts voters' preferences based on features in these relations. To accomplish this, the relations must be filtered for valid training data (active voters), and joined to create a single feature set to train the model. The high-level flow of this pipeline is illustrated in Figure 5.*

Attendees will experience the following optimizations in EmptyHeaded when experimenting with this pipeline:

- **Feature Engineering:** By compiling and executing this pipeline in a single engine, EmptyHeaded can remove verbose, but trivial, feature engineering tasks from the user—in this case one-hot encoding. Attendees will compose this pipeline in EmptyHeaded using only two statements and compare this to the 20+ lines of Python needed to complete this same pipeline in Pandas and Scikit-learn.
- **Data Transformations:** Having knowledge of the entire pipeline, EmptyHeaded can skip materializing the table in the workflow shown in Figure 5 and instead only materialize the final feature matrix for the machine learning model. Here attendees will learn how EmptyHeaded fuses data transformations to optimize materializations across pipeline stages.

In a side-by-side comparison, attendees will observe up to an order of magnitude better performance (see Figure 6) when executing this pipeline in EmptyHeaded versus Pandas and Scikit-learn.

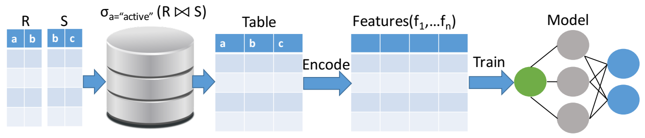


Figure 5: Example data flow for a pipeline containing a SQL stage followed by a machine learning stage. One-hot encoding is needed for featurization between stages.

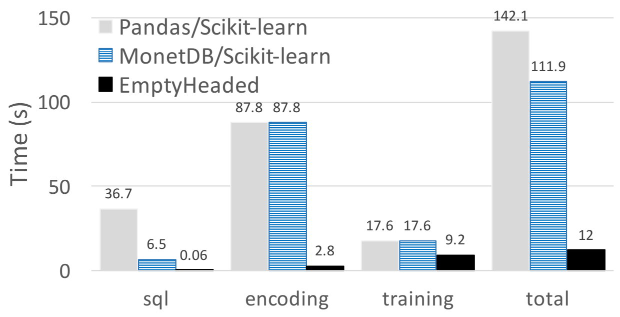


Figure 6: Performance of engines on the mixed SQL and machine learning workload used in the final part of this demonstration. MonetDB Jun2016-SP2 release is benchmarked, and logistic regression is trained for 5 fixed iterations in all engines. Run on a single machine with a total of 56 cores on four Intel Xeon E7-4850 v3 CPUs.

Acknowledgments: We gratefully acknowledge the support of the Defense Advanced Research Projects Agency (DARPA) XDATA Program under No. FA8750-12-2-0335, the National Science Foundation (NSF) CAREER Award under No. IIS-1353606, the Office of Naval Research (ONR) under awards No. N000141210041 and No. N000141310129, the Sloan Research Fellowship, and the Moore Foundation.

4. REFERENCES

[1] C. Aberger et al. Emptyheaded: A relational engine for graph processing. *SIGMOD '16*, pages 431–446.
[2] C. Aberger et al. Old techniques for new join algorithms: A case study in rdf processing. *ICDE Workshops '16*, pages 97–102.
[3] G. Gottlob et al. Hypertree decompositions: Structure, algorithms, and applications. *Graph-theoretic concepts in computer science*, pages 1–15, 2005.
[4] H. Q. Ngo et al. Worst-case optimal join algorithms. *PODS '12*, pages 37–48.
[5] M. Armbrust et al. Spark sql: Relational data processing in spark. *SIGMOD '15*, pages 1383–1394.
[6] M. Joglekar et al. Aggregations over generalized hypertree decompositions. *PODS '16.*
[7] A. Kumar. To join or not to join?: Thinking twice about joins before feature selection. *SIGMOD '16*, pages 19–34.