LevelHeaded: A Unified Engine for Business Intelligence and Linear Algebra Querying

Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, Christopher Ré

Stanford University
{caberger,lamb,kunle,chrismre}@stanford.edu

***Abstract*—Pipelines combining SQL-style business intelligence (BI) queries and linear algebra (LA) are becoming increasingly common in industry. As a result, there is a growing need to unify these workloads in a single framework. Unfortunately, existing solutions either sacrifice the inherent benefits of exclusively using a relational database (e.g. logical and physical independence) or incur orders of magnitude performance gaps compared to specialized engines (or both). In this work, we study applying a new type of query processing architecture to standard BI and LA benchmarks. To do this, we present a new in-memory query processing engine called LevelHeaded. LevelHeaded uses worst-case optimal joins as its core execution mechanism for both BI and LA queries. With LevelHeaded, we show how crucial optimizations for BI and LA queries can be captured in a worst-case optimal query architecture. Using these optimizations, LevelHeaded outperforms other relational database engines (LogicBlox, MonetDB, and HyPer) by orders of magnitude on standard LA benchmarks, while performing on average within 31% of the best-of-breed BI (HyPer) and LA (Intel MKL) solutions on their own benchmarks. Our results show that such a *single* query processing architecture can be efficient on *both* BI and LA queries.**

I. Introduction

The efficient processing of classic SQL-style workloads is no longer enough; machine learning 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 [1]. As a result, there is a growing need for query processing engines that are efficient on (1) the SQL-style queries at the core of most business intelligence workloads and (2) the linear algebra operations at the core of most machine learning algorithms. In this work, we explore whether a new query processing architecture can be efficient in both cases.

An increasingly popular workflow combines business intelligence (BI) and linear algebra (LA) queries by executing SQL queries in a relational warehouse as a means to extract feature sets for machine learning models. Unsurprisingly, these SQL queries are similar to standard BI workloads: the data is de-normalized (via joins), filtered, and aggregated to form a single feature set [2]. Still, BI queries are best processed in a relational database management system (RDBMS) and linear algebra queries are best processed in a LA package. As a result, there has been a flurry of activity around building systems capable of unifying both BI and LA querying [3]–[9]. At a high-level, existing approaches fall into one of three classes:

- *Exclusively using a relational engine.* There are many inherent advantages to exclusively using a RDBMS to process both BI and LA queries. Simplifying extract-transform-load (ETL), increasing usability, and leveraging well-known optimizations are just a few [5]. On the other hand, although LA queries can be expressed using joins and aggregations, executing these queries via the pairwise join algorithms in standard RDBMSs is orders of magnitude slower than using a LA package (see Section VI). Thus, others [5] have shown that a RDBMS must be modified to compete on LA queries.
- *Extending a linear algebra package.* Linear algebra packages, like Intel MKL [10] and OpenBLAS [11], provide high-performance through the low-level (and procedural) [10], [11] Basic Linear Algebra Subprograms (BLAS) [12] interface, and therefore lack the ability for high-level querying. To address this, array databases with high-level querying, like SciDB [4], have been proposed. Unfortunately, array databases are highly specialized and are not designed for general BI querying. As a result, support for SQL-style BI querying [6], [13] has recently been combined with the LA support found in popular packages like Spark's MLlib [14] and Scikit-learn [15]. Still, these solutions lack many of the inherent benefits of a RDBMS, like a sophisticated (shared-memory) query optimizer or efficient data structures (e.g. indexes) for query execution, and therefore achieve suboptimal performance on BI workloads (see Section VII).
- *Combining a relational engine with a linear algebra package.* To preserve the benefits of using a RDBMS on BI queries, while also avoiding their pairwise join algorithms on LA queries, others (e.g. Oracle's UT_NLA [8] and MonetDB/NumPy [16]) have integrated a RDBMS with a LA package. Still, these approaches just tack on an external LA package to a RDBMS—they do not fully integrate it. Therefore, users are forced to write low-level code wrapping the LA package, and the LA computation is a black box to the query optimizer. Even worse, this integration, while straightforward on dense data, is complicated (and largely unsolved) on sparse data because RDBMSs and sparse LA packages use fundamentally different data layouts.

Therefore, existing approaches either (1) sacrifice the inherent benefits of using only a RDBMS to process both classes of queries, (2) are unable to process both classes of queries, or (3) incur orders of magnitude performance gaps relative to the best single-class approach.

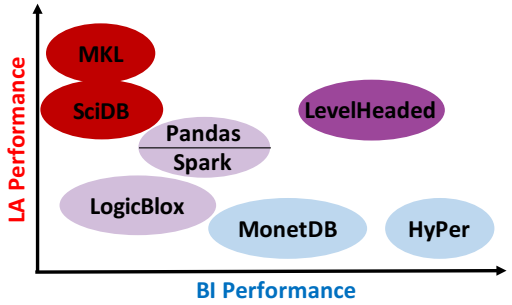


Fig. 1: The relative performance of popular engines and target performance for LevelHeaded on business intelligence (BI) and linear algebra (LA) queries.

In this work, we study an alternative approach to building a RDBMS for both BI and LA querying. In particular, we study using worst-case optimal join (WCOJ) algorithms [17] as the mechanism to unify these query workloads. To do this, we present a new in-memory query processing engine called LevelHeaded. LevelHeaded uses a novel WCOJ query architecture to execute both BI and LA queries. In contrast to previous WCOJ engines [18], LevelHeaded is designed for and evaluated on more than just graph queries. As such, LevelHeaded is the first WCOJ engine to present an evaluation on both BI and LA queries. In contrast to other query engines [4], [5], [19]–[22], LevelHeaded competes with both LA packages and RDBMSs on their own benchmarks (see Figure 1).

However, designing a new query processing engine that is efficient on both BI and LA queries is a challenging task. The recently proposed WCOJs at the core of this new query processing architecture are most effective on the graph queries where they have an asymptotic advantage over traditional pairwise join algorithms. In contrast, pairwise join algorithms are well-understood, and have the advantage of 45+ years of proven constant factor optimizations for BI workloads [20], [21], [23], [24]. Further, LA queries, which also have the benefit of decades of optimizations [12], [25], [26], are typically not a good match for the relational model. Therefore, it is unclear whether they are good match for this new type of relational query processing architecture.

Despite these challenges, we found that the unification of two new techniques in a WCOJ architecture could enable it to deliver efficient performance on both BI and LA queries. The techniques LevelHeaded leverages are (1) a new mechanism to translate general SQL queries to WCOJ query plans and (2) a new cost-based query optimizer for WCOJs. To the best of our knowledge, we are the first to identify and unify these techniques in a WCOJ framework. The two new techniques at the core of LevelHeaded in more detail are:

- *SQL to GHDs (Generalized Hypertree Decompositions): Attribute Elimination.* Neither the theoretical literature [27] nor the query compilation techniques for WCOJs [28] maps directly to all SQL features. In LevelHeaded we implement a practical extension of these techniques that captures more general query workloads as well as the classic query optimization of attribute elimination. Besides providing up to 4x speedup on BI queries, a core artifact of our attribute elimination implementation is that it enables LevelHeaded to target BLAS packages on dense LA queries at little to no execution cost. It is challenging to outperform BLAS packages, like Intel MKL [10] on sparse data and is usually not possible[1] on dense data. Therefore, LevelHeaded leverages attribute elimination to opaquely call Intel MKL on dense LA queries while executing sparse LA queries as pure aggregate-join queries (entirely in LevelHeaded).
- *Cost-Based Optimizer: Attribute Ordering.* WCOJ query optimizers need to select an attribute order [17] in a similar manner to how traditional query optimizers select a join order [29]. With LevelHeaded, we present the first cost-based optimizer to select an attribute order for a WCOJ algorithm. Because this optimizer is the first of its kind, its simplicity is crucial—our goal here is to provide a simple but general foundation for WCOJ engines. Interestingly, we show that such an optimizer must follow different heuristics from what conventional wisdom for pairwise join optimizers suggests (i.e. highest cardinality first). We describe how to leverage these heuristics to provide a simple but accurate cost-estimate that enables LevelHeaded to select attribute orders that can be up to 8815x faster than attribute orders that previous WCOJ engines [18] might select.

We evaluate LevelHeaded on standard BI and LA benchmarks: seven TPC-H queries[2] and four (two sparse, two dense) LA kernels. These benchmarks are de-facto standards for BI and LA engines. Thus, each engine we compare to is designed to process one of these benchmarks efficiently by using specific optimizations that enable high-performance on one type of benchmark, but not necessarily the other. Therefore, although these engines are the state-of-the-art solutions within a benchmark, they are unable to remain competitive across benchmarks. For example, HyPeR delivers high performance on BI queries, but is not competitive on most LA workloads; similarly, Intel MKL [10] delivers high performance on LA queries, but does not provide support for BI querying. In contrast, LevelHeaded is designed to be generic, maintaining efficiency across the queries in both benchmarks.

Contribution Summary : This paper introduces the LevelHeaded engine and demonstrates that its novel architecture can be efficient across standard BI and LA benchmarks. Our contributions and an outline are as follows.

- In Section II we describe the essential background necessary to understand the LevelHeaded architecture which we present in Section III. In particular, we describe how LevelHeaded's query and data model, which is different from that of previous WCOJ engines [18], [22], preserves the theoretical benefits of a WCOJ architecture *and* enables it to efficiently process BI and LA queries.
- In Sections IV and V we present the core (logical and physical) optimizations that we unify in a WCOJ query architecture for the first time. In more detail, we present

[1]Intel MKL often gets peak hardware efficiency on dense LA [10].
[2]The TPC-H queries are run without the `ORDER BY` clause.

the classic query optimization of attribute elimination in Section IV and a cost-based optimizer for selecting an attribute order for a WCOJ algorithm in Section V. We show that these optimizations provide up to a three orders of magnitude speedup on BI and LA queries.
- In Section VI we show that LevelHeaded can outperform other relational engines by one order of magnitude on standard BI and LA benchmarks while remaining on average[3] within 31% of a best-of-breed solution within each benchmark. For the first time, this evaluation validates that a WCOJ architecture can maintain efficiency on both BI and LA workloads. We argue that the inherent benefits of such a unified (relational) design has the potential to outweigh its minor performance overhead.

We believe LevelHeaded represents a first step in unifying relational algebra and machine learning in a single engine. As such, we extend LevelHeaded in Section VII to explore some of the potential benefits of this approach on a full application. We show here that LevelHeaded is up to one order of magnitude faster than the popular (unified) solutions of MonetDB/Scikit-learn, Pandas/Scikit-learn, and Spark on a workload that combines SQL-style querying and a machine learning algorithm. We hope LevelHeaded adds to the debate surrounding the design of unified querying systems.

A. Related Work

LevelHeaded extends previous work in worst-case optimal join processing (EmptyHeaded and LogicBlox), relational data processing, and linear algebra processing.

EmptyHeaded: The techniques presented in EmptyHeaded [18], [30] alone are not enough for a WCOJ query architecture to achieve competitive performance on general (non-graph) queries. As such, LevelHeaded represents an important advancement in WCOJ processing by presenting the crucial optimizations for such an engine to be efficient on BI and LA workloads. The core differences between LevelHeaded and EmptyHeaded are (1) how LevelHeaded translates general SQL queries to GHDs (Section IV) and (2) the cost-based optimizer (Section IV) LevelHeaded uses to select an attribute order for its WCOJ algorithm. Both of these new optimizations for a WCOJ query architecture are enabled by LevelHeaded's query model and data model (Section III-C), which is less restrictive than EmptyHeaded's (e.g. only a single annotation and limited operations); EmptyHeaded is simply unable to capture most of the queries run in this paper. Although LevelHeaded's optimizations (Sections III-A, III-B, IV and V) are simple, they are crucial on BI and LA workloads, and are the first of their kind in this new line query processing.

LogicBlox: LogicBlox [22] is a full featured commercial database engine built around similar worst-case optimal join [31] and query compilation [32] algorithms. Still, a systematic evaluation of the LogicBlox engine on BI and LA workloads is yet to be presented. From our conversations with LogicBlox, we learned that they often avoid using a WCOJ algorithm on these workloads in favor of more traditional approaches to join processing. In contrast, LevelHeaded *always* uses a WCOJ algorithm. Further, LogicBlox uses a query optimizer that has similar benefits to LevelHeaded's (see Section III-C), but does so using custom algorithms [32]. In contrast, LevelHeaded uses a generalization of these algorithms [28] that maps to well-known techniques [33].

Relational Processing: Most relational database engines [20], [21], [24] since System-R [34] have used the pairwise (over relation) join algorithms that work with Sellinger-style [34] query optimizers. This is fundamentally different from the WCOJ (multiway over attribute) algorithm and GHD-based query optimizer in LevelHeaded. Still, a significant amount of work has focused on bringing LA to these pairwise relational data processing engines. Some have suggested treating LA objects as first class citizens in a column store [7]. Others, such as Oracle's UTL_NLA [8] and MonetDB with embedded Python [16], allow users to call LA packages through user defined functions. Still, the relational optimizers in these approaches do not see, and therefore are incapable of optimizing, the LA routines. Even worse, these packages place significant burden on the user to make low-level library calls. Finally, the SimSQL project [5] suggests that relational engines can be modified in straightforward ways to accommodate LA workloads. Our goals are similar to SimSQL, but explored with different mechanics. SimSQL studied the necessary modifications for a classic database architecture to support LA queries and was only evaluated on distributed LA 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.

Linear Algebra Processing: Researchers have long studied how to implement high-performance LA kernels. Intel MKL [10] represents the culmination of this work on Intel CPUs. Unsurprisingly, researchers have shown [35] 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 VI). In response, researchers have released array databases, like SciDB [4] and TileDB [36], which provide high-level LA 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.

II. Background

We briefly summarize the WCOJ algorithm [27] LevelHeaded uses as its core computational kernel, the *generalized hypertree decompositions (GHDs)* [33] LevelHeaded uses to represent its query plans, and the *Aggregations and Joins over Annotated Relations (AJAR)* framework [28] LevelHeaded uses to capture aggregate join queries. We present the essential details informally and refer the reader to Aberger et al. [18] for a complete survey.

[3]The arithmetic mean of the difference to the best competitor from Table II.

Algorithm 1 Generic Worst-Case Optimal Join Algorithm

```
1   // Input: Hypergraph H = (V, E), and a tuple t.
2   //
3   // The order of each v ∈ V corresponds to the order
4   // that attributes are processed. R_e[t] returns all
5   // values in relation R_e that match tuple t.
6   Generic−Join(V, E, t):
7     // Base case: for a single vertex, return the
8     // intersection of all relations matching t.
9     if |V| = 1 then return ∩_{e∈E} R_e[t].
10    Let I = {v_1} // the first attribute.
11    Q ← ∅ // the return value
12    // Intersect all relations that contain v_1 and
13    // contain tuples that agree with t.
14    for every t_v ∈ ∩_{e∈E:e∋v_1} π_I(R_e[t]) do
15      Q_t ← Generic−Join(V − I, E, t :: t_v )
16      Q ← Q ∪ {t_v} × Q_t
17    return Q
```

A. Worst-Case Optimal Joins (WCOJs)

The generic WCOJ algorithm [27] shown in Algorithm 1 is the core computational kernel for all queries in the LevelHeaded engine. The generic WCOJ algorithm can be asymptotically better than any pairwise join algorithm [27]. For *any* join query, the generic WCOJ algorithm's execution time can be upper bounded by the Atserias, Grohe, and Marx (AGM) bound [37]. This can be easily computed when the query is represented as a hypergraph. 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). Note that vertices correspond to attributes and hyperedges correspond to relations. Now, fix a hypergraph H and let $x \in R^{|E|}$ be a vector indexed by edges. The AGM bound tells us that the output size is upper bounded by $\prod_{e \in E} |R_e|^{x_e}$ where $\forall v \in V, \sum_{e \in E: e \ni v} x_e \geq 1$ and $\forall e \in E, x_e \geq 0$. To get the tightest bound, we minimize $\prod_{e \in E} |R_e|^{x_e}$ subject to these constraints.

B. Generalized Hypertree Decompositions (GHDs)

The LevelHeaded query compiler uses **generalized hypertree decompositions (GHDs)** [33] to represent query plans. It is useful to think of GHDs as an analog of relational algebra for a WCOJ algorithm. Given a hypergraph $H = (V_H, E_H)$, a GHD is a tree $T = (V_T, E_T)$ and a mapping $\chi : V_T \to 2^{V_H}$ that associates each node of the tree with a subset of vertices in the hypergraph. The following properties must hold for a GHD to be valid:

- Each edge of the hypergraph $e \in E_H$ must be a subset of the vertices in one of the nodes of the tree, i.e. there exists a tree node $t \in V_T$ such that $e \subseteq \chi(t)$.
- For each attribute $v \in V_H$, the tree nodes to which it belongs, $\{t \mid v \in \chi(t)\}$, must form a connected subtree. This is called the *running intersection property*.

GHDs are designed to capture the degree of cyclicity in a query and therefore can be used to define non-trivial cardinality estimates based on the sizes of the relations. We formalize this idea by defining the **fractional hypertree width (FHW)** of a GHD. Each node t of a GHD defines a subgraph (V', E'). That is, V' is $\chi(t)$ and E' is the set of edges that are subsets of $\chi(t)$. To calculate the width of this node, we define a linear program with one variable x_e for each hyperedge $e \in E'$, with the constraints $\sum_{e \in E': e \ni v} x_e \geq 1, \forall v \in V'$ and objective function $\sum_{e \in E'} x_e$. The FHW of a GHD is then defined as the maximum width of all of the nodes V_T. The worst-case runtime of a GHD is bound by its FHW. Therefore, LevelHeaded chooses a plan with the best worst-case guarantee by choosing a GHD with the lowest FHW.

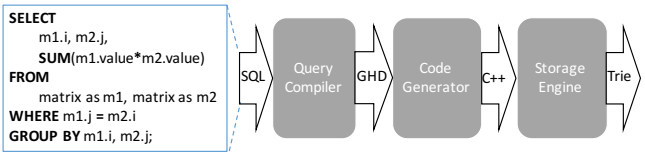


Fig. 2: System overview with matrix multiplication input.

C. Capturing Aggregate Join Queries (AJAR)

Aggregate-join queries are common in both BI and LA workloads. To capture aggregate-join queries, LevelHeaded uses the AJAR framework [28]. AJAR extends the theoretical results of GHDs to queries with aggregations 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, one can pick a query plan with the best worst-case guarantee by going through three phases: (1) Break the input query into characteristic hypergraphs, which are subgraphs of the input that can be decomposed to optimal GHDs. (2) For each characteristic hypergraph all possible decompositions are enumerated, and a decomposition that minimizes the FHW is chosen. (3) The chosen GHDs are combined to form an optimal GHD. To avoid unnecessary intermediate results, LevelHeaded also compresses all final GHDs with a FHW of one into a single node, as the query plans here are always equivalent to running just a WCOJ algorithm.

III. LevelHeaded Engine

In this section we overview the data model, storage engine, query compiler, and join algorithm at the core of the LevelHeaded architecture. This overview presents the preliminaries necessary to understand the optimizations presented in Sections IV and V. LevelHeaded ingests structured data from delimited files on disk and has a Python front-end that accepts Pandas dataframes. The query language is a subset of the SQL 2008 syntax with some standard limitations that we detail in Section III-A. The input and output of each query is a LevelHeaded table which we describe in Section III-B. The SQL queries are translated to a GHD which we describe in Section III-C. Finally, code is generated from the selected GHD using a WCOJ algorithm which we describe in Section III-C. The entire process is shown in Figure 2.

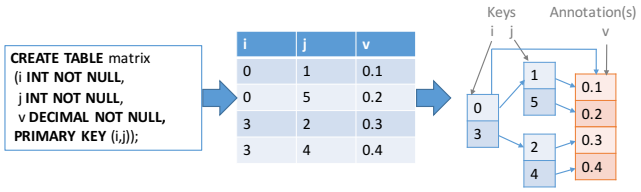


Fig. 3: Storage of a matrix in a LevelHeaded trie.

	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 I: Core trie (and trie set) operations in LevelHeaded.

A. Data Model

The LevelHeaded data model is relational with some minor restrictions. A core aspect of LevelHeaded's data model is that attributes are classified as either *keys* or *annotations* via a user-defined schema. Keys in LevelHeaded 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. LevelHeaded's current implementation supports attributes with types of `int`, `long`, `float`, `double`, and `string`. In many ways LevelHeaded's data model is similar to Google Mesa's [38] where a table schema specifies both a key and value space.

B. Storage Engine

LevelHeaded's data model is tightly coupled with how it stores relations. In LevelHeaded, keys are stored in a trie while annotations are stored in flat columnar buffer (see Figure 3).

Key Attributes: All key attributes from a relation are stored in a trie, which serves as the only physical index in LevelHeaded. In the trie, each level is composed of sets of order-preserved dictionary encoded (unsigned integer) values. Like EmptyHeaded [18], LevelHeaded stores dense sets using a bitset and sparse sets using unsigned integers. Each trie level corresponds to one attribute and each attribute is stored in its own buffer. As others have shown [31], a trie is a natural data structure choice for a WCOJ algorithm. This is because a trie implicity stores the required tuple matching operations ($R_e[t]$ from Algorithm 1), which would have to be computed on the fly if a row or column store was used. Therefore, when using a trie, Algorithm 1 consists of nothing more than a series of trie traversals and set intersections.

Annotation Attributes: Annotations are the associated data values attached to the last level of a trie (1-1 mapping). In LevelHeaded, each annotation is stored in its own buffer that is attached to the trie. LevelHeaded supports multiple annotations, and each can be reached from any level of the trie (core differences from EmptyHeaded).

C. Query Compiler

LevelHeaded's query compiler leverages the techniques presented in Section II to convert an input SQL query to executable code using the three step process shown in Figure 4:

1) *Translate the input query to a query hypergraph.* Using the definition of a query hypergraph from Section II-A, LevelHeaded's process for translating generic SQL queries is described in Section IV-A.

2) *Translate the query hypergraph to a GHD.* LevelHeaded uses the three phase process described in Section II-C to produce the GHDs with the best worst-case guarantee (lowest FHW). Unfortunately, this often produces many theoretically equivalent GHDs to choose from. To address this, LevelHeaded presents a simple series of heuristics to select among GHDs with the best worst-case guarantee in Section IV-B.

3) *Translate the GHD to executable code using the generic WCOJ algorithm.* Like EmptyHeaded, the generic WCOJ algorithm is used to compute each node of the GHD-based plan, and Yannakakis' [39] algorithm is used to communicate results between GHD nodes (a GHD is an acyclic plan). The code generation for each of these phases uses the storage engine operations presented in Table I. Note that the bottleneck operation in Algorithm 1 is a set intersection.

D. Implementation Details

BLAS Integration: On dense LA kernels LevelHeaded calls Intel MKL [10] for its processing of the annotation values. This is done because (1) Intel MKL is highly optimized to get peak hardware efficiency on dense kernels and (2) LevelHeaded requires no data transformation to integrate with dense BLAS libraries (like Intel MKL). LevelHeaded can seamlessly integrate with any package supporting the BLAS interface (like OpenBLAS [11]), but calls Intel MKL in this paper because it is highly optimized for the Intel CPUs that we use. LevelHeaded does not integrate with a BLAS library on sparse kernels because (1) sparse BLAS support is not yet standard (e.g. no support in OpenBLAS) and (2) the (normally) accepted compressed sparse row format (CSR) would require an expensive data transformation (see Table IV).

Parallelization: LevelHeaded utilizes multiple cores by naively parallelizing the outermost for loop from Algorithm 1. To do this LevelHeaded provides a `parfor` operator in addition to the `for` operator from Table I to iterate over set values.

IV. SQL to AJAR GHDs

The AJAR query compilation techniques [28] presented in Section III-C are able to capture a wide range of domains, including linear algebra, message passing, and graph queries [18], [28], [30]. 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.[4] In this section we show how to extend this work to more complex queries. In particular we describe how LevelHeaded translates

[4] Although others [40] have presented ad-hoc techniques to translate SQL to hypertrees for pairwise join optimizers, this work does not encompass the recent theoretical advances [28], [32] for capturing aggregate-join queries.

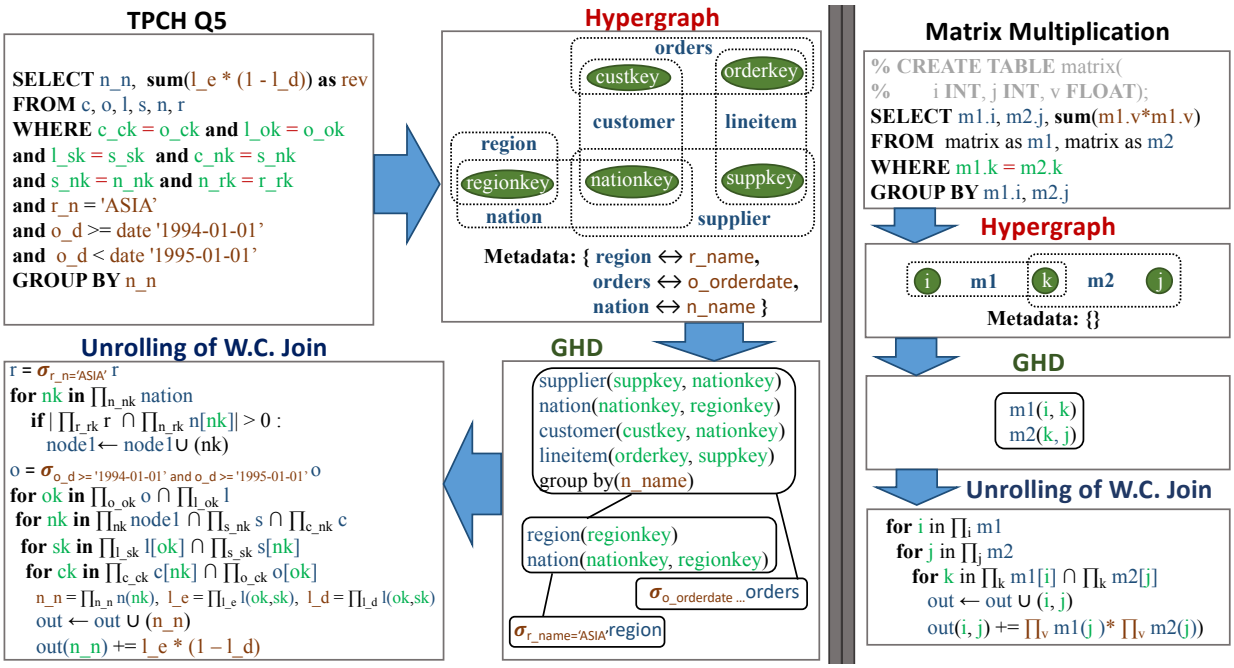


Fig. 4: Illustration of the core components of LevelHeaded on TPC-H query 5 and matrix multiplication. The queries are expressed in SQL, translated to a hypergraph, the hypergraph is used to generate an optimal generalized hypertree decomposition (GHD) query plan, and code instantiating the WCOJ algorithm 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.

generic SQL queries to hypergraphs in Section IV-A. Using this translation, we show how the well-known optimization of attribute elimination is captured both logically and physically in LevelHeaded. In Section IV-B we describe the manner in which LevelHeaded selects a GHD. We argue that with this LevelHeaded represents the first practical implementation of these techniques capable of capturing general SQL workloads.

A. SQL to Hypergraph Translation

We demonstrate how certain features of SQL, such as complex expressions inside aggregation functions, can be translated to operations on annotated relations using a series of simple rules to construct query hypergraphs. The rules LevelHeaded uses to translate a SQL query to a hypergraph $H = (V, E)$ and an aggregation ordering α are as follows:

1) The set of vertices V in the hypergraph contains all of the keys in the SQL query. The set of hyperedges E is each relation in the SQL query. All attributes in a 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 α as the annotations associated with these keys correspond to the aggregated values.

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 annotation is the identity element. If the inner expression of an aggregation function touches multiple relations, those relations must be in the same GHD node 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 metadata container M that associates them to the hyperedge (relation) from which they originate.

Example 4.1: Consider how these rules capture TPC-H query 5 from Figure 4 in a LevelHeaded 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 4. 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 Σ (the order is irrelevant here).

To apply Rule 3, consider the expression inside the `SUM` *aggregation function, on TPC-H query 5. Only columns on the* `lineitem` *table are involved in this expression, so the annotations on the* `lineitem` *table are this expression for each tuple. The* `orders` *and* `customer` *tables are annotated with the identity element (no columns in aggregation expressions).*

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` ↔ `nation`*,* `r_name` ↔ `region`*,* `o_orderdate` ↔ `orders`*.*

Attribute Elimination: The rules above only add the attributes that are used in the query to the hypergraph. Although this elimination of unused attributes is obvious, ensuring this physically in LevelHeaded required a complete redesign from the tries used in EmptyHeaded. To do this we ensured 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. These fundamental differences with EmptyHeaded enable LevelHeaded to support attribute elimination both logically and physically. This is essential on dense LA kernels because it enables LevelHeaded to store each dense annotation in a BLAS compatible buffer.

B. GHD Optimization

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

Choosing Among GHDs with the same FHW: 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): (1) the smaller a GHD is (in terms of number of nodes and height), the quicker it can be executed (less generated code), (2) fewer intermediate results (shared vertices between nodes) results in faster execution time, and (3) the lower selection constraints appear in a query plan corresponds to how early work is eliminated in the query plan. Therefore, LevelHeaded 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 paper 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.

V. Cost-Based Optimizer

After a GHD-based query plan is produced using the process described in Sections III-C and IV, LevelHeaded needs to select an attribute order for the WCOJ algorithm. Similar to the classic query optimization problem of selecting a join order [29], WCOJ attribute ordering 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 [34] query optimizers using pairwise join algorithms—not a GHD-based query optimizer with a WCOJ algorithm. In this section we present the first cost-based optimizer for the generic WCOJ algorithm and show that this it selects attribute orders that can provide up to a three orders of magnitude speedup over attribute orders that EmptyHeaded could select.

Optimizer Overview: LevelHeaded's cost-based optimizer selects a key attribute order for each node in a GHD-based query plan. Like EmptyHeaded [18], LevelHeaded requires that materialized key attributes appear before those that are projected away (with one important exception described in Section V-A2) 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, LevelHeaded's cost-based optimizer: (1) traverses the GHD in a top-down fashion, (2) considers all attribute orders adhering to the previously described criteria at each node, and (3) 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, LevelHeaded assigns a set intersection cost (`icost`) and a cardinality weight (Section V-B) to each key attribute (or vertex) in the hypergraph of a GHD node. Using this, the cost estimate for a given 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 V-A) and weights (Section V-B) are derived.

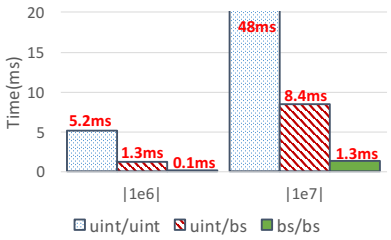
A. Intersection Cost

The bottleneck of the generic WCOJ 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 WCOJ algorithm.

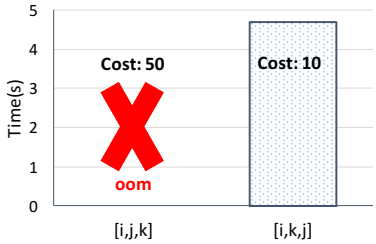
1) Cost Metric: Recall that the sets in LevelHeaded tries are stored using an unsigned integer layout (`uint`) if they are sparse and a bitset layout (`bs`) if they are dense, a design inherited from EmptyHeaded. Thus, the intersection algorithm used is different depending on the data layout of the sets. These different layouts have a large impact on the set intersection performance, even with similar cardinality sets. For example, Figure 5a, shows that a `bs` ∩ `bs` is roughly 50x faster than a `uint` ∩ `uint` with the same cardinality sets. Therefore, LevelHeaded uses the results from Figure 5a to assign the following `icost`s:

`icost`(`bs` ∩ `bs`) = 1, `icost`(`bs` ∩ `uint`) = 10,
`icost`(`uint` ∩ `uint`) = 50

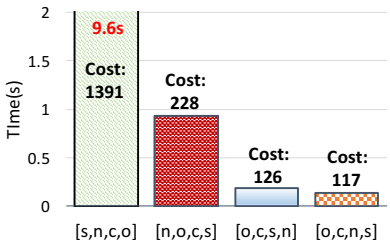
Unfortunately, it is too expensive for the query compiler to check (or track) the layout of each set during query compilation—set layouts are chosen dynamically during data ingestion and a single trie can have millions of sets. On the other hand, it is intuitive that the first level of the trie is likely a dense set while the sets at the remaining levels are increasingly sparse. This is because the set in first trie level holds an entire



(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.

Fig. 5: Cost estimation experiments.

column of values, whereas the sets in the lower levels contain only the values that match the previous columns values. This held true in practice leading to Observation 5.1.

Observation 5.1: The set in the first level of a trie is likely a `bs` while the sets in the remaining levels are likely `uints`.

Thus, given a key attribute order $[v_0,...,v_{|V|}]$ (where each $v_i \in V$), the LevelHeaded optimizer assigns an `icost` to each v_i, in order of appearance, using the following method which leverages Observation 5.1:

- 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 `icosts` where the `bs` sets are *always* processed first. For example, when $N = 3$ and $l(e_0) \leq l(e_1) \leq l(e_2)$ where `bs` < `uint`, `icost` = `icost`$(l(e_0) \cap l(e_1)) +$ `icost`$(l(l(e_0) \cap l(e_1)) \cap l(e_2))$. Note, `uint` $= l($`bs` $\cap$ `uint`$)$.

Example 5.1: Consider the attribute order [`orderkey`, `custkey`, `nationkey`, `suppkey`] for one of the GHD nodes in TPC-H query 5 (see Figure 4). 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 LA queries properly.

2) 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.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). Therefore, EmptyHeaded kept its design simple and never considered relaxing the rule that materialized attributes must appear before those which are projected away. In LevelHeaded we relax this rule by allowing 1-attribute unions on keys when it can lower the `icost`.

Within a GHD node, LevelHeaded 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.
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 5.2: Consider the matrix multiplication query and its unrolling of the generic WCOJ algorithm for an attribute order of `[i,j,k]` shown in Figure 4. 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 WCOJ algorithm 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*
      $s_j \leftarrow s_j \cup (j, \pi_v m1(i,k) * \pi_v m2(k,j))$
  $\forall (j,v) \in s_j : out \leftarrow out \cup (i,j)$
  $\forall (j,v) \in s_j : out(i,j) += v$

This lower cost attribute order recovers the same loop ordering as Intel MKL on sparse matrix multiplication [41], which is essential to run sparse matrix multiplication without running out of memory (see Figure 5b).

B. Weights

Like classic query optimizers, LevelHeaded also tracks the cardinality of each relation as this influences the `icosts`

for the generic WCOJ algorithm. Figure 5 shows the unsurprising fact that larger cardinality sets result in longer intersection times. To take this into account when computing a cost estimate LevelHeaded assigns weights to each vertex using Observation 5.2, which directly contradicts conventional wisdom from pairwise optimizers:

Observation 5.2: The highest cardinality attributes should be processed first in the generic WCOJ algorithm. This enables these attributes to partake in fewer intersections (outermost loops) and ensures that they are a higher trie levels (more likely `bss` *with lower* `icosts`*).*

LevelHeaded's goal when assigning weights is to follow Observation 5.2 by assigning high cardinality attributes heavier weights so that they appear earlier in the attribute order. To do this, LevelHeaded assigns a cardinality score to each queried relation and uses this to weight to each attribute.

Score: LevelHeaded 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, LevelHeaded uses the highest score edge (or relation) with the vertex when a high selectivity (equality) constraint is present, otherwise LevelHeaded 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 5.3: 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 get costs shown in Figure 5c.

VI. Experiments

We compare LevelHeaded to state-of-the-art relational database management engines and LA packages on standard BI and LA benchmark queries. We show that LevelHeaded is able to compete within 2.5x of these engines, while sometimes outperforming them, and that the techniques from Sections IV and V can provide up to a three orders of magnitude speedup. This section validates that a WCOJ architecture can serve as a practical solution for both BI and LA queries.

A. Setup

We describe the experimental setup for all experiments.

Environment: LevelHeaded 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 one 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 queries to highlight the performance of other relational databases. Unlike LevelHeaded, these engines are unable to compete within one order of magnitude of the best approaches on BI and LA queries. We compare to HyPer v0.5.0 [21] 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 [20]. 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 [22]. 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 LevelHeaded 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 LA performance on the Intel CPUs we use in this paper.

Metrics: For end-to-end performance, we measure the wall-clock time for each system to execute 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 data loading, data statistics collection, and index creation for all engines. 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.

B. Experimental Results

We show that LevelHeaded can compete within 2x of HyPer on seven TPC-H queries and within 2.5x of Intel MKL on four LA queries, while outperforming MonetDB and LogicBlox by up to two orders of magnitude.

1) Business Intelligence: On seven queries from the TPC-H benchmark we show that LevelHeaded can compete within 2x of HyPer while outperforming MonetDB by up to one order of magnitude and LogicBlox by up to two orders of magnitude.

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.

Queries: We choose TPC-H queries 1, 3, 5, 6, 8, 9 and 10 to benchmark, as these queries exercise the core operations of

	Query	Data	Baseline	LevelHeaded	Intel MKL	HyPer	MonetDB	LogicBlox
TPC-H	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
Linear Algebra	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 II: 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.

BI 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 LevelHeaded is designed for join queries, it can also compete on scan queries.

Discussion: In Table II we show that LevelHeaded 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 LevelHeaded is the farthest off the performance of the HyPer engine are TPC-H queries 1 and 8 where the output cardinality is 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), LevelHeaded is able to compete within 11% of HyPer and sometimes outperforms it. It should be noted that on TPC-H query 9, where LevelHeaded 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 LevelHeaded.

2) Linear Algebra: We show that LevelHeaded can compete within 2.5x of Intel MKL while outperforming HyPer by more than 18x on LA benchmarks.

Datasets: We evaluate LA 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 [42] containing 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 describing a 3D engine fan [42] containing 2,017,169 rows and columns and 283,073,458 nonzeros. The final sparse matrix dataset we use is the nlpkkt240 dataset [43], which is a symmetric indefinite KKT matrix containing 27,993,600 rows and columns and 401,232,976 nonzeros. For dense matrices, we use synthetic matrices with dimensions of 8192x8192 (8192), 12288x12288 (12288), and 16384x16384 (16384).

Queries: We run matrix-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. Like others [41], for both SMM and DMM we multiply the matrix by itself.

	Dataset	Query	LH	-Attr. Elim.	-Attr. Ord.
TPC-H	SF 10	Q1	145ms	2.54x	-
	SF 10	Q3	111ms	2.46x	1.17x
	SF 10	Q5	128ms	1.46x	72.68x
	SF 10	Q6	51ms	4.82x	-
	SF 10	Q8	78ms	-	8815x
	SF 10	Q9	115ms	-	18.96x
	SF 10	Q10	246ms	1.70x	8.32x
LA	hv15r	SMV	165ms	-	-
	hv15r	SMM	25.44s	-	oom
	nlp240	SMV	171ms	-	-
	nlp240	SMM	4.69s	-	oom
	16384	DMV	13.50ms	1.96x	-
	16384	DMM	4.43s	500x	-

TABLE III: Runtime for LevelHeaded (LH) and relative performance without optimizations on TPC-H and LA queries. 'SF 10' indicates scale factor 10. '-' indicates no effect on the query. 'oom' indicates the system ran out of memory.

Discussion: Table II shows that LevelHeaded is able to compete within 2.44x of Intel MKL on both sparse and dense LA queries. On dense data, LevelHeaded uses the attribute elimination optimization from Section IV 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 LevelHeaded incurs a minor performance penalty (<2%) for producing the key values. On sparse data, LevelHeaded is able to compete with MKL when executing these LA queries as pure aggregate-join queries. To do this, the attribute order optimization from Section IV was essential. In contrast, other relational designs fall flat on these LA queries. Namely, HyPer usually runs out of memory on the matrix multiplication query and, on the queries which finish, is often one order of magnitude slower than Intel MKL. Similarly, LogicBlox and MonetDB are at least one order of magnitude slower than Intel MKL.

C. Micro-Benchmarking Results

We break down the performance impact of each optimization presented in Sections IV and V.

Attribute Elimination: Table III shows that attribute elimination can enable up to a 4.82x speedup on the TPC-H queries and up to a 500x speedup on dense LA queries. Attribute elimination is crucial on most TPC-H queries, as these queries typically touch a small number of attributes from schemas with many attributes. Unsurprisingly, Table III shows that attribute elimination provides the largest benefit on the scan TPC-H queries (1 and 6) because it allows us to scan less data. On dense LA queries, LevelHeaded calls Intel MKL with little overhead because attribute elimination enables us to store each dense annotation in a BLAS acceptable buffer. As Table II shows, this yields up to a 500x speedup over processing these queries purely in LevelHeaded.

Attribute Order: As shown in Table III, the cost-based attribute ordering optimizer presented in Section V can enable up to a 8815x performance advantage on TPC-H queries and enables LevelHeaded to run sparse matrix multiplication as a join query without running out of memory. Table III 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 V-B ensures that orders satisfying these constraints are chosen. Finally, the cost-based attribute ordering optimizer is also 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 materialization of annotation values that are not in the output.

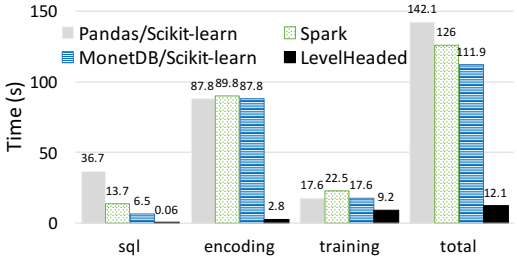


Fig. 6: Performance of engines on the voter classification application which combines a SQL query, feature encoding, and the training of a machine learning model.

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 IV: Runtime for dataset conversion, SMV query time in LevelHeaded, 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 an acceptable sparse BLAS format. The ratio is the number of times LevelHeaded could run SMV while a column store is converting the data.

VII. Extension

We extend LevelHeaded to show that such a unified query processing architecture could enable faster end-to-end applications. To do this, we add the ability for LevelHeaded to process workloads that combine SQL queries and full machine learning algorithms (as described in [44]). We show on a full-fledged application that LevelHeaded can be one order of magnitude faster than the popular solutions of Spark v2.0.0, MonetDB Dec2016[5]/Scikit-learn v0.17.1, and Pandas v0.18.1/Scikit-learn v0.17.1 (using the setup from Section VI).

Application: We run a voter classification application [45] that joins and filters two tables to create a single feature set which is then used to train a logistic regression model

[5]Development build with embedded Python [16].

for five iterations. This application is a pipeline workload that consists of three pipeline phases: (1) a SQL-processing phase, (2) a feature engineering phase where categorical variables are encoded, and (3) a machine learning phase. The dataset [45] consists of two tables: (1) one with information such as gender and age for 7,503,555 voters and (2) one with information about the 2,751 precincts that the voters were registered in.

Performance: Figure 6 shows that LevelHeaded outperforms Spark, MonetDB, and Pandas on the voter classification application by up to one order of magnitude. This is largely due to LevelHeaded's optimized shared-memory SQL processing and ability to minimize data transformations between the SQL and training phase. To expand on the cost of data transformations a bit further, in Table IV we show the cost of converting from a column store to the CSR format used by most sparse library packages. This transformation is not necessary in LevelHeaded as it always uses a single, trie-based data structure. As a result, Table IV shows that up to 41 SMV queries can be run in LevelHeaded in the time that it takes for a column store to convert the data to a BLAS compatible format. Similarly, on the voter classification application LevelHeaded avoids expensive data transformations (in the encoding phase) by using its trie-based data structure for all phases.

VIII. Conclusions

This paper introduced the LevelHeaded engine and demonstrated that a query architecture built around WCOJs can be efficient on both standard BI and LA benchmarks. We showed that LevelHeaded outperforms other relational engines by at least one order of magnitude on LA queries, while remaining on average within 31% of best-of-the-breed solutions on BI and LA benchmark queries. Our results are promising and suggest that such a query architecture could serve as the foundation for future unified query engines.

Acknowledgments: We thank LogicBlox for their helpful conversations and assistance in setting up our comparisons. We gratefully acknowledge the support of the Defense Advanced Research Projects Agency (DARPA) XDATA Program under No. FA8750-12-2-0335 and DEFT Program under No. FA8750-13-2-0039, DARPA's MEMEX program and SIMPLEX program, the National Science Foundation (NSF) CAREER Award under No. IIS-1353606, the Office of Naval Research (ONR) under awards No. N000141210041 and No. N000141310129, the Sloan Research Fellowship, the Moore Foundation, American Family Insurance, Google, and Toshiba. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA, AFRL, NSF, ONR, NIH, or the U.S. government.

References

[1] D. Bryant, "Live from intel ai day 2016." [Online]. Available: https://www.servethehome.com/live-intel-ai-day-2016/
[2] A. Kumar et al., "To join or not to join?: Thinking twice about joins before feature selection," in *SIGMOD*, 2016, pp. 19–34.
[3] J. M. Hellerstein et al., "The madlib analytics library: or mad skills, the sql," *VLDB*, vol. 5, no. 12, pp. 1700–1711, 2012.
[4] P. G. Brown, "Overview of scidb: large scale array storage, processing and analysis," in *SIGMOD*, 2010, pp. 963–968.
[5] S. Luo et al., "Scalable linear algebra on a relational database system," in *ICDE*, 2017, pp. 523–534.
[6] M. Armbrust et al., "Spark sql: Relational data processing in spark," in *SIGMOD*, 2015, pp. 1383–1394.
[7] D. Kernert et al., "Bringing linear algebra objects to life in a column-oriented in-memory database," in *IMDM*, 2015, pp. 44–55.
[8] "Oracle corporation," https://docs.oracle.com/cd/B1930-6_01/index.html.
[9] J. Duggan et al., "The bigdawg polystore system," *SIGMOD*, vol. 44, no. 2, pp. 11–16, 2015.
[10] "Intel Math Kernel Library," https://software.intel.com/en-us/mkl.
[11] Z. Xianyi et al., "Openblas," 2012. [Online]. Available: http://www.openblas.net/
[12] L. S. Blackford et al., "An updated set of basic linear algebra subprograms (blas)," *ACM TOMS*, vol. 28, no. 2, pp. 135–151, 2002.
[13] W. McKinney, "pandas: a foundational python library for data analysis and statistics," *PyHPC*, pp. 1–9, 2011.
[14] X. Meng et al., "Mllib: Machine learning in apache spark," *JMLR*, vol. 17, no. 1, pp. 1235–1241, 2016.
[15] P. et al., "Scikit-learn: Machine learning in python," *JMLR*, vol. 12, no. Oct, pp. 2825–2830, 2011.
[16] M. Raasveldt, "Embedded python/numpy in monetdb," https://www.monetdb.org/blog/embedded-pythonnumpy-monetdb, 2016.
[17] H. Q. Ngo et al., "Worst-case optimal join algorithms," in *PODS*, 2012, pp. 37–48.
[18] C. R. Aberger et al., "Emptyheaded: A relational engine for graph processing," in *SIGMOD*, 2016, pp. 431–446.
[19] M. Zaharia et al., "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.
[20] S. Idreos et al., "Monetdb: Two decades of research in column-oriented database architectures," *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 40–45, 2012.
[21] A. Kemper et al., "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," in *ICDE*. IEEE, 2011, pp. 195–206.
[22] M. Aref et al., "Design and implementation of the logicblox system," in *SIGMOD*, 2015, pp. 1371–1382.
[23] M. Stonebraker et al., "C-store: a column-oriented dbms," in *VLDB*, 2005, pp. 553–564.
[24] K. Delaney, *Inside Microsoft SQL Server 2000*. Microsoft Press, 2000.
[25] R. C. Whaley et al., "Automatically tuned linear algebra software," in *SC*, 1998, pp. 1–27.
[26] E. Anderson et al., *LAPACK Users' guide*. SIAM, 1999.
[27] H. Q. Ngo et al., "Skew strikes back: New developments in the theory of join algorithms," *SIGMOD*, vol. 42, no. 4, pp. 5–16, 2014.
[28] M. R. Joglekar et al., "Ajar: Aggregations and joins over annotated relations," in *PODS*, 2016, pp. 91–106.
[29] H. Garcia-Molina, *Database systems: the complete book*. Pearson Education India, 2008.
[30] C. R. Aberger et al., "Old techniques for new join algorithms: A case study in rdf processing," in *ICDEW*, 2016, pp. 97–102.
[31] T. L. Veldhuizen, "Leapfrog triejoin: A simple, worst-case optimal join algorithm," *arXiv preprint arXiv:1210.0481*, 2012.
[32] M. Abo Khamis et al., "Faq: questions asked frequently," in *PODS*. ACM, 2016, pp. 13–28.
[33] G. Gottlob et al., "Hypertree decompositions: Structure, algorithms, and applications," *WG*, vol. 5, pp. 1–15, 2005.
[34] M. M. Astrahan et al., "System r: relational approach to database management," *TODS*, vol. 1, no. 2, pp. 97–137, 1976.
[35] T. M. Smith et al., "Anatomy of high-performance many-threaded matrix multiplication," in *IDPPS*, 2014, pp. 1049–1059.
[36] S. Papadopoulos et al., "The tiledb array data storage manager," in *VLDB*, vol. 10, no. 4, 2016, pp. 349–360.
[37] A. A. et al., "Size bounds and query plans for relational joins," *SIAM Journal on Computing*, vol. 42, no. 4, pp. 1737–1767, 2013.
[38] A. Gupta et al., "Mesa: Geo-replicated, near real-time, scalable data warehousing," *VLDB*, vol. 7, no. 12, pp. 1259–1270, 2014.
[39] M. Yannakakis, "Algorithms for acyclic database schemes," in *VLDB*, vol. 81, 1981, pp. 82–94.
[40] L. Ghionna et al., "Hypertree decompositions for query optimization," in *ICDE*, 2007, pp. 36–45.
[41] M. M. A. Patwary et al., "Parallel efficient sparse matrix-matrix multiplication on multicore platforms," in *HiPC*, 2015, pp. 48–57.
[42] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *TOMS*, vol. 38, no. 1, p. 1, 2011.
[43] O. Schenk, A. Wächter, and M. Weiser, "Inertia-revealing preconditioning for large-scale nonconvex constrained optimization," *SISC*, vol. 31, no. 2, pp. 939–960, 2008.
[44] C. Aberger et al., "Mind the gap: Bridging multi-domain workloads with emptyheaded," *VLDB*, vol. 10, no. 12, 2017.
[45] M. Raasveldt, "Voter classification using monetdb/python," https://www.monetdb.org/blog/voter-classification-using-monetdbpython, 2016.