Towards Compiling Graph Queries in Relational Engines

Ruby Y. Tahboub
Purdue University
West Lafayette, IN, USA
rtahboub@purdue.edu

Xilun Wu
Purdue University
West Lafayette, IN, USA
wu636@purdue.edu

Grégory M. Essertel
Purdue University
West Lafayette, IN, USA
gesserte@purdue.edu

Tiark Rompf
Purdue University
West Lafayette, IN, USA
tiark@purdue.edu

Abstract

The increasing demand for graph query processing has prompted the addition of support for graph workloads on top of standard relational database management systems (RDBMS). Although this appears like a good idea — after all, graphs are just relations — performance is typically suboptimal since graph workloads are naturally iterative and rely extensively on efficient traversal of adjacency structures that are not typically implemented in an RDBMS. Adding such specialized adjacency structures is not at all straightforward due to the complexity of typical RDBMS implementations. The iterative nature of graph queries also practically requires a form of runtime compilation and native code generation which adds another dimension of complexity to the RDBMS implementation and any potential extensions.

In this paper, we demonstrate how the idea of the first Futamura projection, which links interpreted query engines and compilers through specialization, can be applied to compile graph workloads in an efficient way that simplifies the construction of relational engines which also support graph workloads. We extend the LB2 main-memory query compiler with graph adjacency structures and operators. We implement a subset of the `Datalog` logical query language evaluation to enable processing graph and recursive queries efficiently. The graph extension matches, and sometimes outperforms, best-of-breed low-level graph engines.

DBPL '19, June 23, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6718-9/19/06…$15.00
https://doi.org/10.1145/3315507.3330200

CCS Concepts • **Information systems** → *Database management system engines*; • **Software and its engineering** → *Domain specific languages.*

Keywords Query Compilation; Graph Query Engines; Futamura Projections.

ACM Reference Format:
Ruby Y. Tahboub, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. 2019. Towards Compiling Graph Queries in Relational Engines. In *Proceedings of the 17th ACM SIGPLAN International Symposium on Database Programming Languages (DBPL '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3315507.3330200

1 Introduction

Graphs are Relations It is often desirable for graph and relational data to coexist and be processed together, which naturally suggests representing graphs as relations on top of an existing RDBMS. The unique advantage of such an approach is that the core data management operations are all provided by the RDBMS, and a graph extension would only need to provide a graph front-end (e.g., a declarative graph query language such as PGQL [72] or Cypher [1]) and graph operators. At runtime, the optimized graph query plan is mapped to the extended RDBMS operators and evaluated.

The execution pattern of graph workloads is typically dominated by long-running loops over the graph structure where each iteration performs computations on the adjacency of some vertices (e.g., set intersection during triangle counting operation). The performance of relational graph extensions is therefore stymied by the interpretive nature of typical relational engines and the lack of specialized data structures for adjacency relations. Instead, internal RDBMS data structures are often implemented in a generic form to unify various types of data layouts behind a common interface.

Stand-alone Graph Engines To tackle the challenges of relational graph extensions, several stand-alone graph engines have been developed (e.g., high-level Neo4j [3], low-level Snap-Ringo [5]) that process graphs in native adjacency

structures. While graph processing in stand-alone systems is more performant and also often more expressive in describing graph operations (e.g., shortest paths, centrality) than relational queries, there exists a high development cost in terms of core data management and loss of interoperability with front-end and back-end systems that integrate with RDBMSs.

The Complexity Dimension for Building Specialized Relational Graph Engines Realizing efficient relational graph processing requires combining relational evaluation with specialized graph structures and operations which is nontrivial due to the difficulty of modifying the internals of a mature RDBMS (typically several million lines of code). Aside from this, the key performance challenge in an RDBMS is the interpretive overhead associated with processing data in high-level form (e.g., generic libraries for hash maps) rather than generating optimized low-level code (as precisely described in Neumann's work [47]). Even supporting a minimally operational compiled path for graph queries entails writing thousands of lines of low-level code (e.g., programmatic LLVM API) that permeates large parts of the query engine code. *En masse*, the complexity of extending RDBMS with graph processing and compilation does not add up linearly, and is instead *multiplicative*.

Architecting a Lightweight Relational Graph Compiler In relational engines, compilation of query plans to native code is seeing a renaissance with significant speedups [38, 47, 61, 69]. The community is engaged in an active conversation on how to build such query compilers. Many proposed approaches incur a high degree of complexity through tedious low-level coding [47] or multiple intermediate languages and compiler passes [61]. In our recent work [69], we demonstrated that simple one-pass compilers can achieve the desired goals, and we give a recipe for constructing query compilers based on the first Futamura projection, which links interpreters and compilers via specialization. Our main-memory query compiler, LB2, is constructed in this fashion, and matches or outperforms the best existing query compilers on TPC-H-style workloads.

In the present paper, we demonstrate that the underlying idea of constructing simple but highly efficient query compilers not only applies to purely relational queries but carries over to graph queries. The key challenge is that graph processing relies on efficient traversal of adjacency structures. Given an optimized graph query plan, the key idea is to achieve optimal graph processing by facilitating the building of optimized data structures using *programmatic specialization*, the same technique LB2 already uses for generating efficient code and indexing structures for relational queries.

We extend the LB2 [69] main-memory query compiler with graph compilation, in particular, the compilation of graph data structures, high-level graph operators (e.g., PageRank, triangle count), and shared-memory parallelism. For low-level graph operators (e.g., pattern matching implemented in PGQL and Cypher), we believe that ultimately these operators could be efficiently supported using the correct graph adjacency structure and a general recursion capability. As a preliminary step towards that goal, we focus on high-level graph operators which can be either specified directly in QPlan DSL or extracted from a Datalog-style representation by a query optimizer.

This paper makes the following contributions:

- We discuss compiling optimized graph plans in LB2. We describe specializing graph adjacency structures and graph operators. Furthermore, we implement the semi-naive evaluation to support graph and recursive queries (Section 3).
- We discuss compiling parallel graph operators for shared-memory using OpenMP [4] and thread-aware data structures (Section 4).
- We compare the performance of the high-level graph operators implemented in LB2 on standard graph benchmarks with specialized low-level graph engines (SNAP [5], Ligra [64]), and the compiled graph relational engine EmptyHeaded [9]. The experiments validate that the performance of low-level engines can be achieved in LB2-Graph extension (Section 5).

Finally, we review related work in Section 6, and Section 7 concludes the paper.

2 Background

We review relational query compilation in high-level languages using multi-stage generative programming, covering partial evaluation, programmatic specialization, and Lightweight Modular Staging (LMS) [56]. Furthermore, we give a brief background about the `Datalog` [20] logical query language, recursive queries, and semi-naive evaluation.

2.1 Programmatic Specialization with LMS

The core of compiling SQL to low-level code lies in specializing a given query plan with static pieces of information known at compile time (e.g., schema, constants), and also specializing the query engine with respect to a query plan which in turn removes the interpretive overhead associated with processing static parameters. In partial evaluation theory [21], the first Futamura projection [26] states that specializing an interpreter (e.g., query engine) with respect to a static input (e.g., query) is identical to a compiler (e.g., query compiler) as shown in Figure 1a-b. As discussed in our previous work on compiling query engines [69], fully automatic partial evaluation is largely intractable due to the difficulty of binding time analysis separation [35] (i.e., deciding which expressions to specialize and residualize). Therefore, implementing programmatic specialization to stage a query

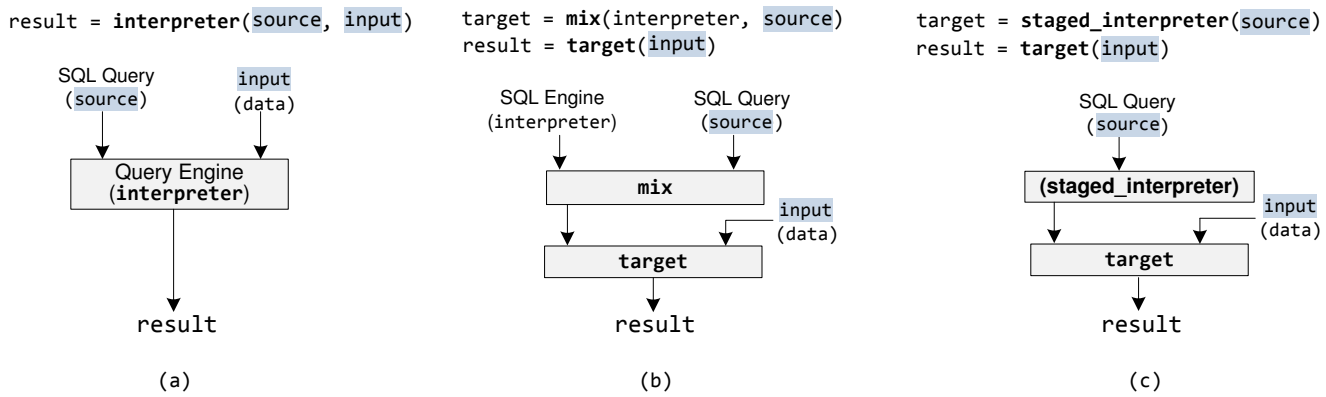


Figure 1. The practical realization of the first Futamura Projection through specialization (adapted from [69]).

interpreter with help from a programmer and generative programming frameworks is a more tractable approach in obtaining the desired compiled target.

The programmatic specialization approach for writing self-specializing code is summarized as first distinguishing the parameters to be specialized or residualized. Second, using a code generation framework such as Lightweight Modular Staging (LMS) to emit evaluation code for residual expressions. LMS is a library-based generative programming and compiler framework that uses operator overloading and other features in regular general-purpose languages to generate code. For instance, specializing the power function for a fixed `n` value is done as follows.

```
def power(x: Rep[Int], n:Int): Int =
  if (n == 0) 1  else x * power(x, n - 1)
```

The `Rep` constructor used to annotate the definition of the parameter `x` means that the staged expression `x` is a symbolic integer type that will emit code. The generated code for `power(x,4)` as follows.

```
int x0 = x * 1; int x1 = x * x0; int x2 = x * x1; int x3 = x * x2;
```

The practical realization of the first Futamura projection is illustrated in Figure 1c. The query engine (i.e., interpreter) is staged with `Rep` annotations and specialized with respect to a SQL query. The compiled target is used to evaluate dynamic data and produces the final query result.

2.2 The LB2 Query Compiler

LB2 is a high-level relational query engine that uses LMS with guidance from the first Futamura projection [27] (that links interpreters and compilers through specialization) to compile queries into optimized low-level code. LB2 implements the data-centric model with callbacks, where the produce-consume interface is refactored into a single `exec` method with a callback as shown in the implementation of the `select` operator, below.

```
class Select(op: Op)(pred: Record ⇒ Boolean) extends Op {
  def exec(cb: Record ⇒ Unit) = {
    Op.exec{ rec ⇒
      if (pred(rec))
        cb(rec)
}}}
```

Moreover, LB2 provides various data abstractions (e.g., records, buffers, data structures, etc.) that facilitate engine implementation and generate efficient code.

2.3 Datalog and Recursive Queries

Many graph operations (e.g., transitive closure, shortest paths) are simpler when expressed using recursion. The `Datalog` [20] logical query language enables expressing recursive and graph queries succinctly. `Datalog` is used to define rules where a rule consists of a head and body of the form *predicate(term1, term2, ...)*: Consider the following rule.

```
Colleagues(A,B) :- Employee(C,A), Employee(C,B)
```

This can be interpreted as follows: *A is a colleague of B if, for some C, C is the Department of A and C is the Department of B*. Furthermore, the previous rule is a schema that is used to define propositional implications, such as:

```
Colleagues(Joe,Sally) :- Employee(CS,Joe), Employee(CS,Sally)
```

Facts are rules without a condition (e.g., `Employee(Math,Ann)`). `Datalog` is widely used as a graph query language [45, 60, 63, 74] due to its expressiveness and the ability to write recursive queries succinctly. For example, transitive closure can be expressed as follows:

```
Path(x,y) :- Edge(x,y)
Path(x,y) :- Path(x,z),Path(z,y)
```

The semi-naive evaluation strategy [20] is an iterative bottom-up evaluation of recursive queries. In each iteration, the least fixpoint is computed by instantiating all subgoals of the existing rules until no new tuples are discovered. Thus, it avoids repeating computations and focuses on the derived deltas from the previous iterations.

SQL-99 added the `WITH RECURSIVE` clause to support recursive queries using Common Table Expressions (CTE). Recursive CTEs are incrementally evaluated and maintained,

which is equivalent to semi-naive evaluation. The following SQL query encodes the transitive closure operation from `src = 1`:

```
CREATE TABLE Edges (src Int, des Int);
WITH RECURSIVE TC as (
 SELECT src as dst FROM Edges WHERE src = 1
 UNION
 SELECT TC.dst FROM TC, Edges WHERE Edges.dst = TC.dst
)
SELECT * FROM TC;
```

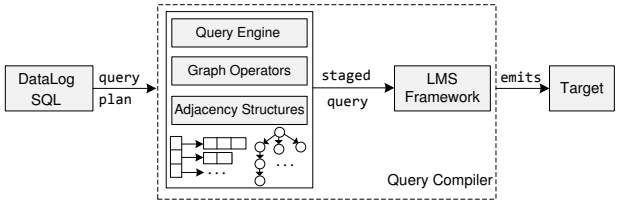


Figure 2. Extending LB2 with graph processing.

3 LB2 + Graph Queries

Existing relational approaches for graph processing fall into one of three categories. In the first, graph operations are translated into SQL procedures [25]. In the second, a specialized graph processing layer is added to extract and process graph data externally [77]. Finally, the last category involves extending the query engine with graph structures and operators [31] which enables processing pipelines that mix relational and graph operators. LB2 falls into this final category, though it is worth noting that queries implemented in any of the previous approaches can be compiled into low-level code.

The extended LB2 engine compiles optimized graph plans and recursive queries into optimized native code. Figure 2 shows a high-level architecture of the extended LB2 system. The front-end accepts SQL queries, `Datalog` rules, and QPlan (a domain-specific language DSL to compose query plans). LB2 is extended with three graph structures: adjacency lists, flattened adjacency lists, and tries. LB2 supports the following graph operations: PageRank, triangle counting, single source-destination shortest path, transitive closure, and all-pairs shortest paths. Furthermore, LB2 implements shared-memory parallel operators OpenMP PageRank and OpenMP triangle counting. As discussed previously (in Section 2 and in our previous work [69]), evaluating a query plan with respect to a staged query evaluator (e.g., using LMS `Rep` annotations) produces a staged query. The LMS framework builds an intermediate representation (IR) graph that encodes high-level constructs and operations. The result of executing the graph is a target program that implements the query evaluation without the interpretive overhead of processing static input (i.e., query pipeline).

Although the graph extension appears as if it is only wired to execute specialized queries, in reality connecting LB2 with an optimizer would allow for the execution of arbitrary graph and recursive queries by composing a plan that employs the available (i.e., recursive join, relational, and graph) operators. In other words, the query evaluator in a RDBMS is generic, and can process any query pipeline as long as it implements the evaluator's operator interface (e.g., the produce/ consume interface in data-centric models [47]).

3.1 Graph Data Loading

Data in LB2 is processed *in-situ* without an explicit preloading phase. The query plan contains the necessary information to identify key attributes used to create indexing and adjacency structures. At loading time, the data loader processes relational graph data and creates the specified adjacency structures. For the case of graph-only queries, LB2 provides a data loader that can process data in compressed row storage (CRS). This data loader is most beneficial for building flat adjacency structures, as it minimizes the memory allocated for preprocessing data during the graph creation phase.

3.2 Graph Processing

LB2 provides two front-ends for graph processing as illustrated in Figure 3a. Graph-only operations such as traversals and shortest paths are encoded as `Datalog` rules or QPlan, whereas queries that mix SQL and graph processing are written using QPlan DSL. In the case of graph-only operations, LB2 builds an adjacency structure to access the graph directly and avoid accessing the `Edge` relation while performing graph operations. In spirit, a graph structure resembles an index where the source vertex is the key and the data record (i.e., the edge that contains the destination vertex and relational attributes) is the value.

In general, main-memory query compilers implement the data-centric evaluation approach [47] since it can be efficiently specialized to remove operators' interpretive overhead, which leads to generating efficient code. Figure 3b-c shows two PageRank query execution plans (QEPs). QEP (b1) is a graph-only PageRank operation that prints the result of PageRank (i.e., vertices and scores). On the other hand, QEP (c1) mixes graph and SQL operators as follows. First, the query filters the graph edges before performing PageRank. After that, the PageRank intermediate result is joined with data from another table and the final result is printed. For both queries, the implementation of PageRank is similar, the difference is only in the operator interface. Figure 4a shows the `PageRankidx` operator; line 5 obtains the previously created graph. The `exec` method invokes the `PageRank` operator implemented inside the graph structure (see Section 3.3). Note that because `PageRankidx` is a leaf operator in the query plan, there is no `parent.exec` call. Figure 4b shows `PageRankOp` used in QEP (c) where the `exec` operator creates

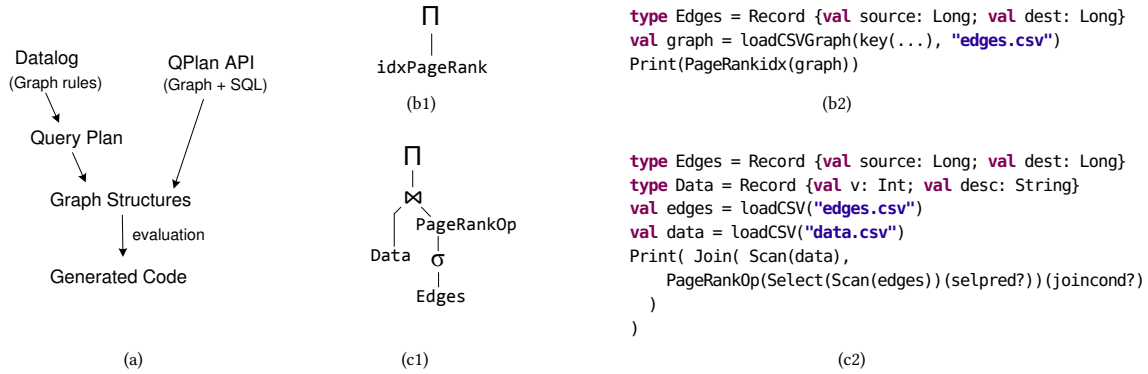


Figure 3. Graph front-end in LB2 and graph query evaluation pipelines.

a graph structure at evaluation time (due to the filter operation) and inserts the edges received from the interfacing filter operator. After that, `graph.PageRank` is invoked and the result is passed to the join operator through the callback function `cb`.

3.3 Graph Data Structures

Adjacency lists are the most commonly used data structures to build and process graphs due to their optimized storage and intuitive interface. LB2 implements three graph data structures (shown in Figure 5): adjacency lists, flattened adjacency lists, and tries. For graph-only queries, the flattened adjacency list has the best runtime performance (see Section 5.1). However, the creation of such a graph structure is expensive, as it indirectly constructs an adjacency list to obtain the neighbor sets, as illustrated in Figure 5b-c (except when data is already stored in CRS form). The adjacency list performs well with queries that mix graph and SQL since the graph is built at runtime. Recently, tries have received attention as graph data structure [73] as they facilitate performing multi-way joins (in contrast to two binary joins). For instance, triangle counting can be implemented directly as a three-way join. Therefore, LB2 adds a trie adjacency structure to optimize operations with multi-way joins pattern. However, the performance of trie-based operations is sensitive to vertex ordering in the graph. A good vertex ordering assists the join operation to minimize the number of comparisons (the experiment in Section 5.1 gives insights about the performance of trie-based triangle count).

Internally, the specialized graph structures are implemented using LB2's generation-time abstractions (e.g., records, data buffers) that emit low-level code. In the following code, we show the extended graph adjacency list structure implemented in LB2 which is similar to any high-level graph implementation (for simplicity, we omit the vertex-to-list-index mapping and resizing steps):

```
1 abstract class Graph {
2  def insert(src: Rep[Long], dest: Rep[Long]): Unit
3  // graph operations e.g., PageRank, triangle count
4 }
5 class LB2Graph(val size: Rep[Long]) extends Graph {
6  val cap0 = defaultSize
7  val adjList = LB2BufferFlat2D[Long](verSize, cap0)
8  val adjListCount = NewArray[Long](verSize)
9
10  def insert(vs: Rep[Long], vd: Rep[Long]) ={
11       adjList.getRow(vs).update(adjListCount(vs), vd)
12       adjListCount(vs) = adjList(vd) + 1
13 }}
```

The `LB2BufferFlat2D` class is an abstraction over `Array[Array[Long]]` that is generated as `long** adjList`. Moreover, the method `getRow` that obtains a row from the adjacency list and is generated as `long* row = adjList[vs]`.

```
1 class PageRankidx(left: Op)
2  (gr: String) extends Op {
3
4 def exec(cb: Record ⇒ Unit) = {
5  val graph = left.getGraph(gr)
6   graph.PageRank {rec ⇒
7    cb(rec)
8   }
9  }
10 }
11
12
```

(a)

```
class PageRankOp(parent: Op)
  (srcKey: KeyFun)
  (destKey: KeyFun) extends Op {

 val size = defaultSize
 def exec(cb: Record ⇒ Unit) = {
  val graph = new GraphStruct(size)
  parent.exec { rec ⇒
    graph.insertEdge
        (srcKey(rec), desKey(rec))
    graph.PageRank{r ⇒
      cb(r)
}}}
```

(b)

Figure 4. PageRank operator as (a) graph-only and (b) graph + SQL

Auxiliary Data Structures Graph operators use auxiliary data structures while processing. For instance, Dijkstra's algorithm for finding the shortest path between two vertices in a graph uses a minimum heap for distance values. All auxiliary data structures in LB2 are implemented as high-level abstractions that emit low-level code.

3.4 Recursive Queries

In standard query engines, recursive queries are composed using CTEs and evaluated using a variant of the incremental

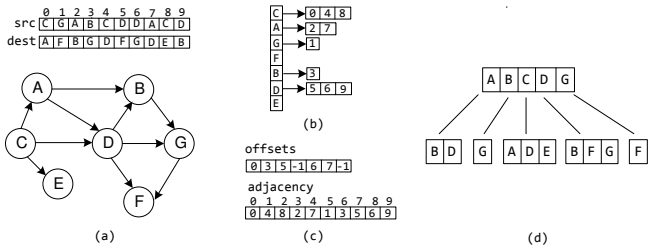


Figure 5. Graph structures in LB2.

view maintenance algorithms, which in essence improves on the basic semi-naive evaluation algorithm that performs a sequence of join, union, and set difference operations until a fixpoint is reached.

LB2 processes recursive graph queries encoded in `Datalog` as follows. Given a query optimizer that extracts the recursion from the rules and maps the result into either a specialized operator (if a particular pattern is recognized, such as triangle count which consists of two join conditions) or to an operator that unrolls recursion and performs the encoded join operation until no more new records are produced.

4 Parallelism

LB2 supports parallelism on shared memory systems using OpenMP [4]. The following code shows LB2's `parallelRegion` annotation that emits a `#pragma omp parallel` block:

```
def parallelRegion(worker: Rep[Long] ⇒ Unit): Unit = {
  parallel_region {
    val j = ompGetThreadId
        worker(j)
}}
```

Furthermore, the parallel evaluator structure and operators are internally modified to enable orchestrating parallel execution as follows. The scan operator is the point where threads are initiated (i.e., the loop that reads data from the source is parallelized) and the interfacing operators are invoked for data processing. Operators that materialize data (e.g., PageRankOp in Figure 3c1) are modified to use per-thread data structures that are merged after exiting the parallel section or use lock-free data structures. For graph-only operators QEP in Figure 3b1, the implementation of the graph operator is directly parallelized using `parallelRegion` annotation that emits a parallel block as illustrated in the following code that parallelizes the main loop in `PageRank`. In Line 1, the number of segments is computed based on the value of `nThreads`. Lines 2-13 shows the parallel region.

```
1 val segment = verSize / nThreads
2 parallelRegion { threadId ⇒
3   val start = threadId * segment
4   val end = if (threadId == nThreads - 1L) verSize else start + segment
5   for (i <- start until end){
6    val row = adjList_incoming.getRow(i)
7    var temp = unit[Double](0.0)
8    for(j<-0L until adjListCount_incoming(i)){
9     val  destVer = row(j)
10    temp = temp + inputVal(destVer) / adjListCount_outgoing(destVer)
11   }
12   rank(i) = temp * 0.85
13 }
```

Table 1. Graph datasets that are used in experiments.

Dataset	Nodes	Edges	Description
Amazon	334,863	925,872	Product network
YouTube	1,134,890	2,987,624	Social network
Skitter	1,696,415	11,095,298	Internet topology
Orkut	3,072,441	117,185,083	Social network
LiveJournal1	4,847,571	68,993,773	Social network

5 Evaluation

In this section, we evaluate the performance of the graph extension implemented in LB2. We compare LB2's performance with several low-level graph processing engines. EmptyHeaded [9] is regarded as a state-of-art graph relational engine that implements specialized graph structures, exploits SIMD parallelism, and generates optimized low-level code. We also selected two popular, low-level, shared-memory graph engines SNAP [5] and Ligra [64].

We conduct two experiments to evaluate the performance of the graph extension. In the first experiment, we evaluate the performance of graph pattern and graph analytical queries (triangle count and PageRank, respectively) in LB2, EmptyHeaded, SNAP, and Ligra using standard graph benchmarks in Table 1. Furthermore, we illustrate the effect of graph data structure choice (trie, adjacency list, and flat array) on the performance of graph queries. The second experiment compares the parallel scalability of the previous graph engines on triangle count and PageRank queries. We show the performance of LB2's graph extension is competitive with state-of-art low-level graph engines.

Environment All experiments are conducted on a single NUMA machine with 4 sockets, 24 cores in a Xeon(R) Platinum 8168 CPU per socket, and 750GB RAM per socket (3 TB total). The operating system is Ubuntu1 16.04.9. We use Scala 2.11, GCC 5.4 with optimization flag -O3. We use EmptyHeaded v.0.1 and SNAP 4.1.

Datasets and Configurations Table 1 shows the graph datasets used for our evaluation which span three application domains: social, web, and product networks. The two largest datasets are LiveJournal1 and Orkut (approximately 68M and 117M edges, respectively) which represent the size of practical graph workloads. For each query, we measure the runtime (excluding compilation time), time for data loading, and time taken for adjacency list (or index) creation. We run each query 5 times, and present the median measurement. To guarantee local execution, we run the queries in a single NUMA node using numactl `-m` and `-C` options.

Table 2. Runtime of single-core triangle count (in seconds) for LB2 (using flat adjacency list and trie), SNAP and EmptyHeaded.

Dataset	LB2-Flat	LB2-Trie	SNAP	EmptyHeaded
Amazon	0.1	0.9	0.19	0.5
YouTube	0.4	0.5	0.7	0.5
Skitter	1.4	1.7	2.2	1.7
Orkut	80.9	61	87.2	60.1
LiveJournal1	13.6	26.4	18.1	24.9

5.1 Single-core Graph Pattern and Analytics Queries

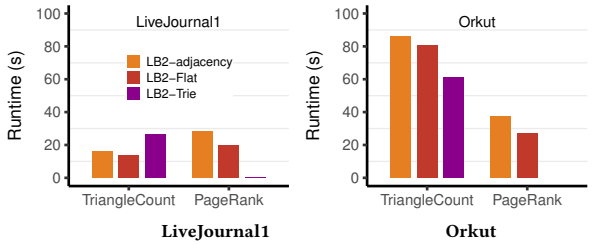
Triangle counting Triangle counting is an important subgraph pattern matching query largely used in graph structure mining and graph benchmarking in general. For this query, we follow SNAP's implementation that treats the graph as undirected when counting triangles. Table 2 gives the runtime of evaluating triangle count on LB2, SNAP[1], and EmptyHeaded on five graph datasets. For small to medium size datasets (i.e., Amazon, YouTube, Skitter, and LiveJournal1, where the number of edges is < 69M), LB2 outperforms SNAP by 7%-2.7× and EmptyHeaded by 80%-7×. On the Orkut dataset, EmptyHeaded is 35% faster than LB2-Flat and comparable to LB2-Trie. Also, SNAP is slower than both LB2 and EmptyHeaded due to the high-level implementation of its graph structure which is more expensive to access than the specialized structures in LB2 and EmptyHeaded. LB2's performance advantage over EmptyHeaded is attributed to its specialized data structures in addition to the fact that EmptyHeaded's SIMD parallel set operations work better with high-density skewed data. Thus, for datasets with small skew (as the ones in Table 1), EmptyHeaded is expected to perform better on larger datasets as Orkut.

PageRank Another graph analytical query is PageRank, which measures the importance of a webpage based on the link structure of the web [49]. The applicability of PageRank computations to any type of graph (e.g., road, social, biology) makes it an important graph workload. Similar to SNAP and Ligra, we implement the algorithm in Berkhin's survey [17]. Table 3 gives the runtime of running 25 iterations of PageRank on LB2, SNAP, Ligra and EmptyHeaded. For fairness across all systems, we commented out the convergence computations in LB2, SNAP, and Ligra, as the query is designed to run for a constant number of iterations without convergence testing that may incur additional runtime cost or cause an early exit. EmptyHeaded is the slowest among the benchmarked systems due to running naive recursion in this query, which is equivalent to a simple unrolling of the

[1]SNAP provides two implementations for triangle count. We picked `GetTriangleCnt` since it can be parallelized and it also outperformed `GetTriads`.

Table 3. The absolute runtime of single-core PageRank (in seconds) for LB2 (using flat adjacency structure), SNAP, Ligra and EmptyHeaded.

Dataset	LB2-Flat	SNAP	Ligra	EmptyHeaded
Amazon	0.2	0.4	0.3	2.6
YouTube	0.7	1.1	1.0	8.2
Skitter	2.5	2.8	3.2	16.6
Orkut	27.2	24.4	36.3	142.8
LiveJournal1	19.5	20.4	27.1	70.1

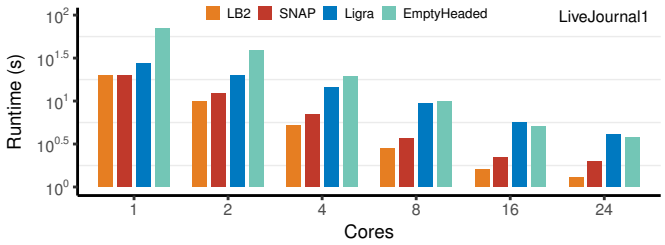


	LiveJournal1		Orkut	
	TriangleCount	**PageRank**	**TriangleCount**	**PageRank**
LB2-Adj	16.3	25	86	31
LB2-Flat	13.6	20	80.9	27
LB2-Trie	26.4	-	61	-

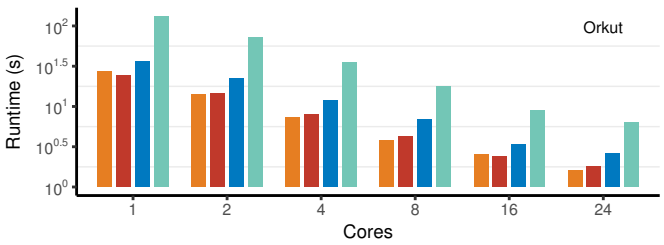
Figure 6. Runtime of single-core triangle counting and PageRank using LB2 with adjacency list and flat array in LiveJournal1 and Orkut datasets.

join algorithm [9]. The authors pointed out that PageRank performance can be further improved with double buffering and redundant join elimination. For the large datasets LiveJournal1 and Orkut, the performance of LB2 and SNAP is comparable (LB2 is 5% faster in LiveJournal1, whereas SNAP is 10% faster in Orkut). Furthermore, LB2 is 30%-50% faster than Ligra across all datasets.

The Effect of Adjacency List Specialization LB2 implements three graph adjacency structures: adjacency lists, flattened adjacency lists (using arrays that maintain the offset of vertices and neighboring edges), and tries. In this experiment, we evaluate the performance of triangle counting and PageRank using the previous three adjacency structures as illustrated in Figure 6. For the triangle counting query in LiveJournal1, the flat implementation is faster than the adjacency list and trie implementations by 20% and 95%, respectively, whereas the trie is faster than the adjacency and flat list in Orkut by 40% and 30%, respectively. For the PageRank query, the adjacency list is slower than the flat array version by 28% and 14%, respectively. For adjacency lists, the difference in performance is a result of generating adjacency lists as an array of arrays (`long** adj`). Hence, the first array access to obtain a row from the adjacency list (`long* nbr_lst = adj[index]`) is slightly more expensive than two simple array accesses (i.e., first access to obtain



	1	2	4	8	16	24
LB2-Flat	20	9.9	5.2	2.8	1.6	1.3
SNAP	20	12.2	7	3.7	2.2	2
Ligra	27	20	14.5	9.3	5.7	4.1
EmptyHeaded	70	39	19.5	10	5.1	3.8



	1	2	4	8	16	24
LB2-Flat	27	14	7.2	3.8	2.5	1.6
SNAP	24	14.3	7.9	4.2	2.4	1.8
Ligra	36	22	12	6.8	3.4	2.6
EmptyHeaded	132	71	35.5	17.9	8.9	6.4

Figure 7. The absolute runtime in seconds (s) for parallel scaling up LB2, SNAP, Ligra and EmptyHeaded in PageRank on 2, 4, 8, 16 and 24 cores for LiveJournal1 and Orkut datasets.

the vertex offset in the edges array and second to start iterating over the vertex neighbors). Similarly, a trie similar to the one in Figure 5d incurs two accesses to obtain an edge. Moreover, the performance of trie-based operations is sensitive to vertex ordering in the graph which justifies the variance in performance on different datasets. Finally, the key insight gained from comparing various implementations of data structures is that optimized memory access matters in processing large graphs.

5.2 Parallel Graph Pattern and Analytics Queries

In this experiment, we compare the scalability of LB2 with SNAP, Ligra, and EmptyHeaded. Figure 7 gives the absolute runtime for scaling up LB2, SNAP, Ligra, and EmptyHeaded in PageRank on 2, 4, 8, 16, and 24 cores for the LiveJournal1 and Orkut datasets. We scale the number of cores up to 24 in order to keep the execution local within a single socket.

Overall, the speedup in all systems linearly increases with the number of cores. The systems with the fastest single-core runtime (SNAP and LB2) achieves speedups of 10× and 15×, respectively, in LiveJournal1, and speedup of 13× and 16×, respectively, in Orkut. Furthermore, LB2 and SNAP outperform Ligra and EmptyHeaded in this query. At a closer look, the scaling up of LB2 is 2×-4× faster than EmptyHeaded in LiveJournal1 and 5×-6× times faster in Orkut. Similarly, LB2 is, on average, 5×-6× times faster than Ligra in LiveJournal1.

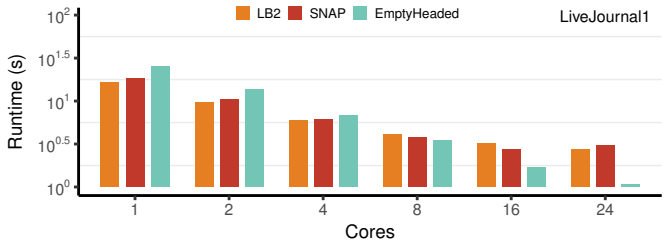
We observed that the implementation of SNAP PageRank spends time in creating a vertex lookup index that maps the vertex id into a vector index which explains, in part, the small difference in runtime numbers. As discussed in the single-core experiment, EmptyHeaded implements the naive evaluation (for this operator) which is not aggressive enough in duplicate elimination, which causes the operator to perform extraneous work at each iteration. It is likely that EmptyHeaded would match the performance of LB2 if the implementation is improved. Although Ligra is implemented in C++, our observation is that the high-level data structure adds runtime interpretive overhead in contrast to LB2's specialized data structure.

Figure 8 gives the absolute runtime for scaling up LB2, SNAP, and EmptyHeaded in triangle count on 2, 4, 8, 16 and 24 cores for LiveJournal1 and Orkut datasets. Overall, EmptyHeaded's scaling up in LiveJournal1 outperforms LB2 and SNAP by an average of 8%-2.8× from 8 cores and above. Similarly, EmptyHeaded outperforms LB2 and SNAP in Orkut by an average of 40%-2×. At closer look, the performance of LB2 and SNAP is comparable. For instance, the scaling of LB2 is 10% faster than SNAP in LiveJournal1 at core 24 whereas SNAP is faster than LB2 by 25% for the same core value. Moreover, EmptyHeaded results are attributed, in part, to the data structure, data partitioning, query optimizer that picks the proper layout based on data skew and SIMD parallelism.

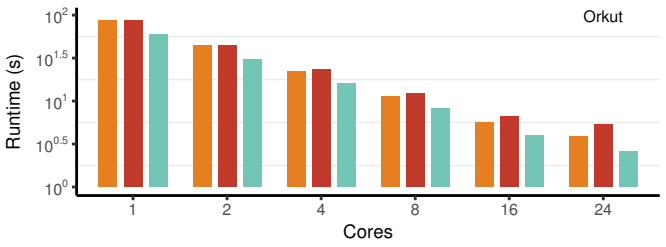
Discussions In conclusion, The outcomes of the parallel scaling experiment are consistent with the single-core experiment. The performance of LB2 is attributed, in part, to optimized evaluation and data structures specialization. Altogether, our experiments show that LB2 can compete against state-of-the-art graph engines. However, LB2's design is simpler as it is derived from a straightforward query interpreter design.

6 Related Work

Relational Graph Processing In SQLGraph [68] and Grail [25], graphs are stored using relational tables. For instance, a syntactic layer translates Gremlin queries to procedural SQL. Integrating graph processing inside an RDBMS greatly benefits relational-graph workloads. Graphite [51], SAP HANA Graph [58], and GRFusion integrate graph processing inside an RDBMS. GRFusion processes graphs natively as views.



	1	2	4	8	16	24
LB2-Adj	16.3	9.6	6	4.1	3.2	2.7
SNAP	18.1	10.3	6.1	3.8	2.7	3
EmptyHeaded	24.9	13.7	6.7	3.5	1.7	1.07



	1	2	4	8	16	24
LB2-Adj	86	44.3	22.3	11.2	5.7	3.9
SNAP	87	44.7	23	12.1	6.6	5.4
EmptyHeaded	60	30.5	15.9	8.1	4	2.6

Figure 8. The absolute runtime in seconds (s) for parallel scaling up LB2, SNAP, Ligra and EmptyHeaded in triangle count on 2,4,8,16 and 24 cores for LiveJournal1 and Orkut datasets.

In contrast to SQLGraph, LB2-Graph generates native code instead of SQL. Moreover, LB2-Graph shares similarity with main-memory SAP HANA Graph and GRFusion in implementing specialized adjacency data structures. However, GRFusion is an interpreter graph engine, whereas SAP HANA Graph and LB2-Graph compile graph queries into optimized code.

Single-machine Shared-memory Ligra [64] and Galois [48] are shared-memory graph frameworks. Ligra switches between different implementations of operators, whereas Galois utilizes a task scheduler. Examples of semi-external memory and SSD engines that apply techniques such as the sliding window algorithm to minimize disk I/O are GraphChi [41], X-Stream [57], and Grid-Graph [79]. EmptyHeaded [9] compiles graph join queries with strong theoretical guarantees. Lux [34] leverages multi-GPUs for efficient graph processing. The graph extension in this work falls into this category. Similar to the previous systems, LB2-Graph runs on a single-machine and supports shared-memory parallelism.

Specialized and Distributed Specialized graph engines implement a native graph model with a rich set of graph operations. Titan [7] and Neo4j [3] are stand-alone graph engines that provide rich graph queries (e.g., paths, subgraphs). Ringo [52] and GraphGen [77] convert graph relational data to a graph-native representation. On scaled-out graph analytics, Teradata Aster [65] provides an SQL front-end where graph functions are executed using a specialized graph engine. GraphBuilder [76] is a Hadoop-based graph construction library that extracts graphs from unstructured data. Giraph [44] is a Hadoop-based iterative graph processing system. GraphLab [43] is an asynchronous parallel framework where evaluation is based on a Gather-Apply-Scatter model and supports operations on both distributed and shared-memory platforms. Powergraph [42] is a distributed GraphLab. Naiad [46] is a main-memory distributed data-flow computation system that supports graph streaming algorithms. GraphX [2] is a Spark-based graph system that provides a graph abstraction layer for graph computations using relational operators such as join and group-by.

Graph Query Languages Graph query languages provide declarative constructs to encode graph operations, as surveyed in [13]. Recent examples include Oracle's PGQL [72], Cypher [1], and GCore [12]. Green-Marl[32] is a DSL for expressing graphs algorithms intuitively. The Green-Marl compiler generates efficient graph operations code. While both LB2-Graph and Green-Marl generate optimized, low-level code, LB2-Graph uses `Datalog` or QPlan as front-end. The following are examples of `Datalog` graph engines: SociaLite [60] is developed for social network analysis; LogicBlox [14]; Soufflé [6] implements program specialization techniques to compile `Datalog` programs; BigDatalog [63] is a Spark-based distributed Datalog engine; Datalography [45] is built on the top of Giraph; and Myria [74]. G-SPARQL [59] is a SPARQL language for querying RDF graphs.

Graph Mining Numerous graph systems focus on graph mining tasks (e.g., Motif Counting, Frequent Subgraph Mining (FSM)). Arabesque [71] provides a graph exploration model referred to as an *embedding*. In each exploration round, the existing embeddings are expanded, and are refined using a filter operation. ScaleMine [8] is a parallel FSM system. DistGraph [28] is distributed mining system that optimizes communication costs. RStream [75] is a single machine graph mining system that extends the GAS evaluation model. G-thinker [78] offers an API for graph mining algorithms.

Query Compilation Compiling SQL queries into native code goes back to IBM's System R [16]. Daytona [30] is a propriety system developed by AT&T that generates query code which enables running large queries efficiently. Rao et. al. [54] compiled queries to Java bytecode by removing virtual functions from iterator evaluation. HIQUE [40] performs

query plan level optimizations and uses templates to generate code. The HyPer [47] query compiler is a pivotal work that presented the data-centric evaluation model that enables generating efficient code for main-memory databases. Another important line of work employs the Lightweight Modular Staging platform (LMS) [56] to compile domain-specific languages (e.g., SQL, machine learning, linear algebra, etc.) to low-level code using a high-level language as in Delite [18, 66] and its DSLs OptiQL, OptiML [67] and OptiMesh. Similarly, Legobase [38], the "SQL to C in 500 lines" compiler [55], Flare [24], a native compiler back-end for SparkSQL, LB2-Spatial [70], and the LB2 [69] single-pass query compiler all generate code using LMS. Moreover, DBLAB [61] uses multi-pass DSL transformations, and DBToaster [10] supports compiling recursive delta view queries. Tupleware [22] compiles user-defined functions. Voodoo [53] compiles portable query plans that can run on CPUs and GPUs. Weld [50] provides a common runtime for diverse libraries (e.g., SQL and machine learning). Butterstein et al. [19] compile query subexpressions into machine code in PostgreSQL. Similarly, the work in [62] uses program specialization and LLVM to generate query code in Postgres. To address the issue of processing raw heterogeneous data, RAW [37] uses compilation to generate scanning code based on the type of raw data which it is loading, H2O [11] generates code for layout-aware access operators, and ViDa [36] compiles data structures and query operators. Tungsten in SparkSQL [15] generates Java code. Commercial examples of query engines that implement a form of query compilation are Hekaton [23], DryadLINQ [33], Impala [39], and Spade[29].

7 Concluding Remarks

The increasing demand for processing graph data from various applications has renewed interest in supporting graph processing inside RDBMS engines due to the fact that graphs are relations. Although graph processing inside an RDBMS is a sensible choice at first glance, the performance of query workloads is often suboptimal due to the interpretive overhead of RDBMS evaluation and generic data structures that are not optimized for fast graph traversal. Similarly, extending relational engines with specialized graph adjacency structures and architecting graph compilers are two hard tasks due to the complexity of the RDBMS architecture and the need to write thousands of lines of code in low-level code generation frameworks when using standard techniques.

Compiling Graph Workloads In this work, we address the system-based challenges for supporting relational query processing and we show that the idea of the first Futamura projection that links interpreters and compilers through specialization can be applied to compile graph queries into low-level code, largely eliminating the complexities that so far have prevented integrating specialized graph structures with relational engines in an efficient manner. As proof of concept, we add graph query compilation inside the LB2 main-memory query compiler. We support parallelism for shared memory using OpenMP and thread-aware data structures. We implement a subset of the `Datalog` evaluation algorithm to enable graph and recursive queries. The graph extension matches, and sometimes outperforms, specialized low-level graph engines. Finally, the graph specialization and code generation ideas presented in this work are not strictly tied to LMS and Scala. In fact, the same abstractions can be implemented in any object-oriented language that supports operator overloading. Researchers interested in architecting query compilers for graph workloads may find building specialized data structures discussed in this paper useful.

Acknowledgments

We would like to thank our shepherd, Guido Wachsmuth, and the anonymous reviewers for their valuable feedback. This work was supported in part by NSF awards 1553471 and 1564207, DOE award DE-SC0018050, and a Google Faculty Research Award.

References

[1] [n.d.]. Cypher Graph Query Language. https://neo4j.com/developer/cypher-query-language/.
[2] [n.d.]. Graph-X. https://spark.apache.org/graphx/.
[3] [n.d.]. Neo4j. https://neo4j.com/.
[4] [n.d.]. OpenMP. http://openmp.org/.
[5] [n.d.]. SNAP: Stanford Network Analysis Project. https://snap.stanford.edu/index.html.
[6] [n.d.]. Soufflé: A Datalog Synthesis Tool for Static Analysis. https://souffle-lang.github.io/.
[7] [n.d.]. Titan. http://titan.thinkaurelius.com/.
[8] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. 2016. Scalemine: scalable parallel frequent subgraph mining in a single large graph. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 61.
[9] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Emptyheaded: A relational engine for graph processing. *TODS* 42, 4 (2017), 20.
[10] Yanif Ahmad and Christoph Koch. 2009. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *VLDB* 2, 2 (2009), 1566–1569.
[11] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H2O: a hands-free adaptive store. In *SIGMOD*. 1103–1114.
[12] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, et al. 2018. G-CORE: A core for future graph query languages. In *SIGMOD*. 1421–1432.
[13] Renzo Angles and Claudio Gutierrez. 2008. Survey of graph database models. *ACM Computing Surveys (CSUR)* 40, 1 (2008), 1.
[14] Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. 2015. Design and implementation of the LogicBlox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1371–1382.
[15] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin,

Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *SIGMOD*. ACM, 1383–1394.

[16] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. 1976. System R: relational approach to database management. *TODS* 1, 2 (1976), 97–137.

[17] Pavel Berkhin. 2005. A survey on pagerank computing. *Internet Mathematics* 2, 1 (2005), 73–120.

[18] Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. 2016. Have Abstraction and Eat Performance, Too: Optimized Heterogeneous Computing with Parallel Patterns. In *CGO*. ACM, 194–205.

[19] Dennis Butterstein and Torsten Grust. 2016. Precision performance surgery for CostgreSQL: LLVM—based Expression Compilation, Just in Time. *VLDB* 9, 13 (2016), 1517–1520.

[20] Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What you always wanted to know about Datalog (and never dared to ask). *ICDE* 1, 1 (1989), 146–166.

[21] Charles Consel and Olivier Danvy. 1993. Partial evaluation: Principles and perspectives. In *Journees Francophones des Langages Applicatifs*. 493–501.

[22] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. 2015. An architecture for compiling udf-centric workflows. *PVLDB* 8, 12 (2015), 1466–1477.

[23] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*. ACM, 1243–1254.

[24] Gregory Essertel, Ruby Tahboub, James Decker, Kevin Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data. In *OSDI*. 799–815.

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

[26] Yoshihiko Futamura. 1971. Partial Evaluation of Computation Process — An approach to a Compiler-Compiler. *Transactions of the Institute of Electronics and Communication Engineers of Japan* 54-C, 8 (1971), 721–728.

[27] Yoshihiko Futamura. 1999. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation* 12, 4 (1999), 377–380.

[28] Wensheng Gan, Jerry Chun-Wei Lin, Han-Chieh Chao, and Justin Zhan. 2017. Data mining in distributed environment: a survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 7, 6 (2017), e1216.

[29] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S Yu, and Myungcheol Doo. 2008. SPADE: the system s declarative stream processing engine. In *SIGMOD*. ACM, 1123–1134.

[30] Rick Greer. 1999. Daytona and the fourth-generation language Cymbal. In *ACM SIGMOD Record*, Vol. 28. ACM, 525–526.

[31] Mohamed S. Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G. Aref, and Mohammad Sadoghi. 2018. Extending In-Memory Relational Database Engines with Native Graph Support. In *EDBT*. 25–36.

[32] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: a DSL for easy and efficient graph analysis. In *SIGARCH*, Vol. 40. 349–362.

[33] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *SIGOPS/EuroSys (EuroSys)*. 59–72.

[34] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A distributed multi-GPU system for fast graph processing. *PVLDB* 11, 3 (2017), 297–310.

[35] Neil D Jones. 1996. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)* 28, 3 (1996), 480–503.

[36] Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. 2015. Just-In-Time Data Virtualization: Lightweight Data Management with ViDa. In *CIDR*.

[37] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. 2014. Adaptive Query Processing on RAW Data. *PVLDB* 7, 12 (2014), 1119–1130.

[38] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building efficient query engines in a high-level language. *PVLDB* 7, 10 (2014), 853–864.

[39] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael and Yoder. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*.

[40] Konstantinos Krikellas, Stratis D Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *ICDE*. IEEE, 613–624.

[41] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale graph computation on just a pc. USENIX.

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

[43] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *VLDB* 5, 8 (2012), 716–727.

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

[45] Walaa Eldin Moustafa, Vicky Papavasileiou, Ken Yocum, and Alin Deutsch. 2016. Datalography: Scaling datalog graph analytics on graph processing systems. In *BigData*. IEEE, 56–65.

[46] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *SOSP*. ACM, 439–455.

[47] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550.

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

[49] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[50] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. 2017. Weld: A Common Runtime for High Performance Data Analytics. In *CIDR*.

[51] Marcus Paradies, Wolfgang Lehner, and Christof Bornhövd. 2015. GRAPHITE: an extensible graph traversal framework for relational database management systems. In *SSDBM*. ACM, 29.

[52] Yonathan Perez, Rok Sosič, Arijit Banerjee, Rohan Puttagunta, Martin Raison, Pararth Shah, and Jure Leskovec. 2015. Ringo: Interactive graph analytics on big-memory machines. In *SIGMOD*. 1105–1110.

[53] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - a vector algebra for portable database performance on modern hardware. *VLDB* 9, 14 (2016), 1707–1718.

[54] Jun Rao, Hamid Pirahesh, C Mohan, and Guy Lohman. 2006. Compiled query execution engine using JVM. In *ICDE*. IEEE, 23–23.

[55] Tiark Rompf and Nada Amin. 2015. Functional pearl: a SQL to C compiler in 500 lines of code. In *ICFP*. ACM, 2–9.

[56] Tiark Rompf and Martin Odersky. 2012. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM* 55, 6 (2012), 121–130.

[57] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP*. ACM, 472–488.

[58] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. 2013. The Graph Story of the SAP HANA Database.. In *BTW*.

[59] Sherif Sakr, Sameh Elnikety, and Yuxiong He. 2014. Hybrid query execution engine for large attributed graphs. *Information Systems* 41 (2014), 45–73.

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

[61] Amir Shaikhha, Ioannis Klonatos, Lionel Emile Vincent Parreaux, Lewis Brown, Mohammad Dashti Rahmat Abadi, and Christoph Koch. 2016. How to Architect a Query Compiler. In *SIGMOD*.

[62] Eugene Sharygin, Ruben Buchatskiy, Roman Zhuykov, and Arseny Sher. 2017. Runtime Specialization of PostgreSQL Query Executor. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer, 375–386.

[63] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big data analytics with datalog queries on spark. In *SIGMOD*. ACM, 1135–1149.

[64] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, Vol. 48. ACM, 135–146.

[65] David Simmen, Karl Schnaitter, Jeff Davis, Yingjie He, Sangeet Lohariwala, Ajay Mysore, Vinayak Shenoi, Mingfeng Tan, and Yu Xiao. 2014. Large-scale graph analytics in Aster 6: bringing context to big data discovery. *VLDB* 7, 13 (2014), 1405–1416.

[66] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM TECS* 13, 4s (2014), 134.

[67] Arvind K. Sujeeth, HyoukJoong. Lee, Kevin J. Brown, Tiark Rompf, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: an Implicitly Parallel Domain-Specific Language for Machine Learning. In *ICML*.

[68] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. 2015. SQLGraph: an efficient relational-based property graph store. In *SIGMOD*. ACM, 1887–1901.

[69] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *SIGMOD*. 307–322.

[70] Ruby Y Tahboub and Tiark Rompf. 2016. On supporting compilation in spatial query engines:(Vision paper). In *SIGSPATIAL*.

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

[72] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *GRADES*. 7.

[73] Todd L Veldhuizen. 2014. Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm. (2014).

[74] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. 2015. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB* 8, 12 (2015), 1542–1553.

[75] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. [n.d.]. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine. In *OSDI*.

[76] Theodore L. Willke and Nilesh Jain. 2012. GraphBuilder – A Scalable Graph Construction Library for Apache TM Hadoop TM. In *NIPS*.

[77] Konstantinos Xirogiannopoulos, Udayan Khurana, and Amol Deshpande. 2015. Graphgen: Exploring interesting graphs in relational data. *VLDB* 8, 12 (2015), 2032–2035.

[78] Da Yan, Hongzhi Chen, James Cheng, M Tamer Özsu, Qizhen Zhang, and John Lui. 2017. G-thinker: big graph mining made easier and faster. *arXiv* (2017).

[79] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning.. In *USENIX ATC*.