Compiled Plans for In-Memory Path-Counting Queries

Brandon Myers, Jeremy Hyrkas, Daniel Halperin, and Bill Howe

Department of Computer Science and Engineering, University of Washington, Seattle, WA, U.S.A.
bdmyers,hyrkas,dhalperi,billhowe @cs.washington.edu

Abstract. Dissatisfaction with relational databases for large-scale graph processing has motivated a new class of graph databases that offer fast graph processing but sacrifice the ability to express basic relational idioms. However, we hypothesize that the performance benefits amount to implementation details, not a fundamental limitation of the relational model. To evaluate this hypothesis, we are exploring code-generation to produce fast in-memory algorithms and data structures for graph patterns that are inaccessible to conventional relational optimizers.

In this paper, we present preliminary results for this approach on path-counting queries, which includes triangle counting as a special case. We compile Datalog queries into main-memory pipelined hash-join plans in C++, and show that the resulting programs easily outperform PostgreSQL on real graphs with different degrees of skew. We then produce analogous parallel programs for Grappa, a runtime system for distributed memory architectures. Grappa is a good target for building a parallel query system as its shared memory programming model and communication mechanisms provide productivity and performance when building communication-intensive applications. Our experiments suggest that Grappa programs using hash joins have competitive performance with queries executed on Greenplum, a commercial parallel database. We find preliminary evidence that a code generation approach simplifies the design of a query engine for graph analysis and improves performance over conventional relational databases.

1 Introduction

Increased interest in the analysis of large-scale graphs found in social networking, web analytics, and bioinformatics has led to the development of a number of graph processing systems [1, 23]. These specialized systems have been developed, in part, because relational DBMSs are perceived as being too slow for graph algorithms. These systems thus sacrifice full relational algebra support in favor of improved performance for graph tasks. However, realistic applications typically involve relations as well as graphs—e.g., Facebook's content is richer than just its friend network—suggesting that relational models and languages should not be completely abandoned.

As a logical data model, relations arguably subsume graphs. Every graph can be trivially represented as an edge relation, but even simple relational schemas must be transformed in non-trivial ways to "shred" them into graphs. Consider a relation `Order(customer,part,supplier)`. Each tuple in this relation is a hyper-edge relating a particular customer, a particular part, and a particular supplier. To represent a tuple (c, p, s) as part of a graph, three edges (c, p),(p, s),(s, c) must be represented and exposed to the user for manipulation in queries. As another example, consider two relations `Friend(person1, person2)` and `Sibling(person1, person2)`. A natural graph representation would create one edge for each tuple in `Friend` and one edge for each tuple in `Sibling`. But to distinguish friends from siblings, each edge

needs to be labeled. Besides increasing the space complexity, this extra label must be manipulated by the user explicitly in queries.

So perhaps the relational model is preferable as a logical interface to the data, but the value proposition of graph databases is typically performance. By using specialized data structures and algorithms and operating primarily in main memory, these systems can outperform relational databases at graph-oriented tasks. However, we hypothesize that these specializations are essentially implementation details, and that there is no fundamental reason that a relational engine could not exploit them when appropriate.

To test this idea, we use code generation to produce fast in-memory query plans for simple graph pattern queries, in two contexts. First, we generate pipelined query plans over associative data structures in C++ and show that these programs significantly outperform tuned and indexed RDBMS implementations. Second, we show how analogous query plans targeting a parallel computation framework called Grappa can compete with a tuned and indexed MPP database.

In our experiments, we consider a class of *path-counting* queries (defined precisely in Section 3), which includes triangle counting [24] as a special case. These queries arise in both graph and relational contexts, including in credibility algorithms for detecting spam [6] and in probabilistic algorithms [30]. To handle "hybrid" graph-relational applications, we retain the relational data model and a relational query language—Datalog.

While only non-recursive queries are explored in this paper, Datalog with recursion will be targeted in future experiments.

Our approach is inspired by other work in compiling relational algebra expressions and SQL queries [20], but our goal is different. We wish to support a variety of backend runtime systems and explore various algorithms for specific graph patterns, rather than generate the fastest possible machine code for individual relational operator algorithms.

For parallel evaluation, we are concerned with the competing factors of scaling up: distributing data allows for higher bandwidth access but greater network usage amidst random access. For workloads with irregular memory access, like that in sequences of hash joins, high throughput can be achieved in modern processors given sufficient concurrency [18]. With this observation in mind, we employ a novel parallel runtime system, Grappa [19], designed for irregular workloads. Grappa targets commodity clusters but exposes a partitioned global address space to the programmer. This abstraction allows us to write code that is structurally similar to that of our serial C++ runtime, while allowing our algorithms and the Grappa engine to apply optimizations that exploit locality.

The contributions of this paper are:

1. A code generator that translates path-counting queries expressed in Datalog into fast C++ programs that implement join-based query plans over associative data structures.
2. In-memory algorithms for parallel path-counting queries in Grappa, along with generic templates compatible with our code generation framework.
3. Experimental results comparing generated C++ programs against the serial relational database PostgreSQL, showing the generated plans to be 3.5×–7.5× faster than tuned and optimized relational query plans.
4. Experimental results comparing path-counting queries in Grappa to the Greenplum commercial parallel RDBMS.

In the next section, we briefly describe the Grappa parallel framework that we use as a compilation target for parallel plans. In Section 3, we describe our code generation approach and evaluate the performance of the resulting plans in Section 4.

2 Grappa: programming for irregular applications

Grappa is a C++11 runtime for commodity clusters that is designed to provide high performance for massively parallel *irregular* applications, which are characterized by unpredictable access patterns, poor locality, and data skew. In these situations, communication costs dominate runtime for two reasons: random access to large data does not utilize caches and commodity networks are not designed for small messages. Interconnects like Ethernet are Infiniband are designed to achieve maximum bisection bandwidth for packet sizes of 10KB–1MB, while irregular accesses may be on the order of 64 bytes—the size of a typical cache line. To simplify programming for irregular applications, Grappa provides the following features:

- a **partitioned global address space** (PGAS) to enable programmer productivity without hindering the ability to optimize performance for NUMA shared memory and distributed memory systems
- **task and parallel loop constructs** for expressing abundant concurrency
- **fine-grained synchronization and active messages** to allow for asynchronous execution and low cost atomic operations, respectively
- **lightweight multithreading** to provide fast context switching between tasks
- a **buffering communication layer** that combines messages with the same destination to utilize the network better than fine-grained messages
- **distributed dynamic load balancing** to scalably cope with dynamic task imbalance

Grappa provides an appropriate level of abstraction as a target platform for our query compilation approach. The global address space allows us to generate relatively simple code, and parallel loop constructs preclude the need to emit explicitly-multi-threaded routines. However, Grappa is sufficiently expressive to allow us to optimize for locality, and we can use lower-level communication abstractions to build distributed algorithms for special situations.

Concurrency can be expressed with a variety of arbitrarily nestable parallel loop constructs that exploit spatial locality when it exists; these idioms are a natural fit for pipelined query plans.

3 Code Generation for Path-Counting Queries

Following Seo, Guo, and Lam [26], we adopt a Datalog syntax for expressing graph queries. In this paper, we show only preliminary results of the efficacy of the code generation approach rather than a full Datalog implementation.

We focus on *path-counting queries*, of which triangle counting is a special case. Each query is of the form

$$\gamma_{count}(\sigma_c(\sigma_1 R_1 \bowtie \sigma_2 R_2 \bowtie \ldots \bowtie \sigma_N R_N)) \tag{1}$$

where γ is an aggregation operator for counting the final results. The extra selection operator σ_c can enforce relationships between non-adjacent vertices in the path. In particular, this condition can enforce that the path form a cycle, as in the triangle queries.

Each leaf and each internal node in the plan may be filtered by a select operator. The internal selection conditions allow us to express, for example, a triangle counting query (see below). The count operation may optionally remove duplicates before counting, which results in significantly different performance in all tested systems.

We consider a graph represented as a relation with the schema `(src:int, dst:int)`. Additional attributes are allowed, but are not considered in these preliminary experiments. Each tuple (a, b) represents an outgoing edge from vertex a to vertex b. While a table is not the most memory efficient way of representing a graph [26], it allows us to easily apply concepts of relational algebra to the graph problems presented.

Through the lens of relational algebra, paths in a graph are expressed as a series of joins on the edge relations. For example, the two-hops (or friends of friends) query is a single join on the $edges$ table [15]. In Datalog, this query is expressed as

```
Twohop(s,d) :- edges(s,m), edges(m,d).
```

A three-hop query would add one additional join to the query above. A popular case of the three-hop query in large graphs is triangle counting [12, 22], where a triangle must be a cycle. Triangles in a graph represent a three-hop where the source s and destination d are the same vertex. In Datalog, directed triangles are expressed as:

```
Triangle(x,y,z) :- edges(x,y), edges(y,z),
                   edges(z,x), x < y, y < z.
```

The final conditions in the Datalog rule ensure that directed triangles are only counted once, instead of three times (i.e. $1, 2, 3$ is counted, but $3, 1, 2$ is not). These conditions correspond to selects in relational algebra.

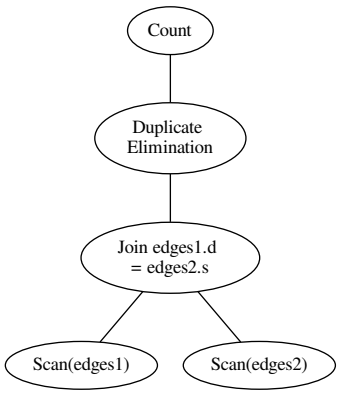
3.1 Generating Code from Datalog

To generate implementations of path-counting queries, we constructed a query compilation framework that can produce pipelined query plans over associative data structures. The overall structure is similar to a typical relational plan, but the use of nested data structures admits some algorithmic improvements that are relatively simple but difficult for typical optimizers to exploit, as we will see. The input is a graph query written in Datalog, and the output is imperative source code. The compiler is written in Python using the NetworkX package [10] for manipulating the query graph. Each rule in the source program[1] is converted to a logical relational algebra expression and then to a physical plan.

In general, a path of k edges through a graph involves $k - 1$ self-joins on the $edges$ relation. To avoid the cost of constructing the results of each join in memory for the next join, we emit pipelined plan consisting of a series of nested hash-joins: a lookup table (tree-based rather than a hash-based; see Section 3.2) is constructed over the join column for the left relation, and then probed with tuples from the right relation. Pseudocode plans for the two-hop and triangle queries appear in Figures 1 and 2 respectively, along with each query's relational plan.

The two-hop query explored in this paper requires duplicate elimination, as there may be multiple paths from vertex s to vertex d. We perform duplicate elimination by inserting results into a set data structure. Since the start vertex is unique for all paths

[1] All queries considered in this paper can be expressed with a single Datalog rule.




```
1  count = 0;
2  #iterating over map simulates
3  #looping over an adjacency list
4  for all x in map:
5    tuple_list1 = map[x];
6    #holds distinct two-hops
7    #from x
8    dup_elim_set;
9    for all edge1 in tuple_list1:
10     tuple_list2 = map[edge1.d];
11     for all edge2 in tuple_list2:
12       tup = edge1.s,edge2.d;
13       dup_elim_set.insert(tup);
14   count += dup_elim_set.size();
15   dup_elim_set.clear();
```

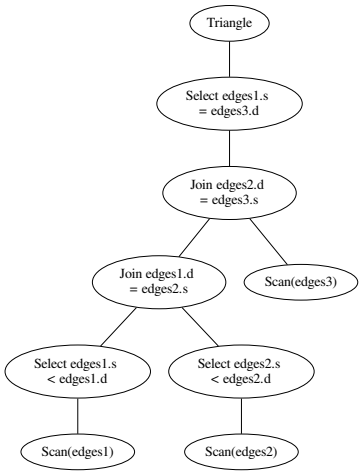
Fig. 1: Relational algebra and pseudocode for the distinct two-hop query. In the code on the right, the map variable maps each vertex x to all edges (x, y). The outer loop behaves as a relational group-by, and the set for duplicate elimination is moved inside the group-by (line 8) and cleared after each iteration (line 15). This approach allows for for better memory-usage.

from k, we can optimize the memory usage by logically grouping by start vertex and iterating through the groups one at a time. This optimization reduces the worst case size of the set by $O(|V|)$. The improvement in lookup time yielded a $5\times$ decrease in runtime for two-hop on the datasets in our experiments. Consequently, whenever a query requires a distinct variable from the outer relation, we perform a group-by on the outer relation. This technique is a scheduling optimization, the motivation for which is similar to that of depth-first search: by exploring all paths from s first, we can reduce memory utilization. We believe this type of custom optimization is a perfect fit for a code generation technique; as new patterns are recognized, the known optimal code for that pattern of query can be automatically generated. This gives our code generation approach potential to generate algorithms that are traditionally not considered by relational DBMS optimizers (such as algorithms used by graph databases), as well as algorithms that are more efficient than those of graph databases. . Pseudocode for the distinct source-destination optimization is shown in Figure 1.

3.2 C++ Code generation

The first language targeted by our code generation model is C++. The logic for generating code described in 3.1 remains unchanged, but there are some language specific features. For example, in our C++ code, the "hash" table is an STL `map`. Similarly, duplicate elimination is performed by inserting results into an STL `set`. Both associative data structures are implemented with a red-black tree, but in experiments this was never a performance factor.

As tuples are scanned from disk, they are converted to structs and inserted into an STL `vector`. Each relation is scanned only once, regardless of how many times it appears in the query. Query processing then proceeds as described in Section 3.1.




```
1  triangle = 0;
2  tuples_list1 = edges_list;
3  for all edge1 in tuple_list1:
4    #select over edge1
5    if not edge1.s < edge1.d:
6      continue;
7    tuple_list2 = map[edge1.d];
8    for all edge2 in tuple_list2:
9      #select over edge2
10     if not edge2.s < edge2.d:
11       continue;
12     tuple_list3 = map[edge2.d];
13     for all edge3 in tuple_list3:
14       #final select
15       if edge1.s == edge3.d :
16         triangle++;
```

Fig. 2: Relational algebra and pseudocode for the triangle query. The select conditions for *edges2*, *edges2*, and the final *edges1.s=edges3.d* are shown in lines 5, 10, and 15. We note that in this example, more efficient code could be generated by pushing the selection conditions in lines 5 and 10 into the construction of the hash maps used in the join operations.

End to end, this compiler allowed us to generate C++ code for path-counting queries from Datalog. Queries without grouping (such as the triangle query) generate code similar to the code shown above. Path queries requiring a distinct source and destination were generated using a "distinct mode", with a group-by structure as shown in Section 3.1. In Section 6, we discuss compiler extensions to support more general graph queries.

3.3 Queries in Grappa

Recall that Grappa provides a global address space and parallel for loop constructs. Each relation is scanned and loaded into an array, which Grappa distributes by default in a block-cyclic fashion.

To compute a hash join, the build relation is hashed on the join key and used to populate a distributed hash table. The hash table representation is essentially equivalent to an adjacency list representation used in graph systems, but general enough to support arbitrary relations. This point is important: it is not obvious that there is a fundamental difference between a join-based evaluation approach and a graph-traversal approach. To compute a chain of joins, we use Grappa's abstractions for shared memory and parallel for-loops to produce nested, pipelined plans that are analogous to the serial case.

The parallel execution strategy for the nested parallel for-loops of a single pipeline is handled by the runtime system. Grappa uses recursive decomposition to spawn tasks for loop iterations and schedules them in depth-first order to use memory only linear in the number of threads. This approach is inspired by Cilk [5], but in Grappa the number of "threads" is determined by the number of concurrent tasks required to tolerate the latency to access distributed memory. Grappa's parallel for-loop mechanism supports different levels of granularity—that is, the number of consecutive loop-body iterations that each task executes. Currently, the level of granularity is fixed at compile time, but we expect that dynamic adjustments will be an important defense against unpredictable result sizes.

To exploit inter-pipeline parallelism, generated code spawns each pipeline as a separate task. No synchronization is required between independent subplans, but for joins, probes of the hash table block until the build phase is complete.

The HashMultiMap and HashSet used for joins and duplicate elimination are designed similarly to typical shared memory implementations, except the data structures are distributed. The entry point is a single shared array of buckets distributed block-cyclically across all cores in the cluster.

The critical part of our shared memory lookup structures for Grappa is how concurrent `insert` and `lookup` are implemented efficiently. By ensuring that a given `insert` or `lookup` touches only data localized to a single bucket, we can implement these operations as active messages. Grappa can mitigate skew by putting all edge lists in one block-cyclic distributed array as in compressed sparse row format, but in the experiments for this paper, we use a data structure that puts each edge list entirely on one machine. All operations on a given bucket are performed atomically, by the simple fact that each core has exclusive access to its local memory. This example shows how Grappa's partitioning of the global address space admits locality-aware optimizations.

Since Grappa uses multithreading to tolerate random access latency to distributed memory, execution flow of hash join probes looks similar to the explicit prefetching schemes by Chen et al. [7]. Specifically, prefetching a collection of independent hash buckets before performing loads is analogous to how Grappa switches out a thread on a remote memory operation.

Since the insertion for duplicate elimination is the last operation in the probing pipeline, the task does not need to block waiting for the insertion to complete, saving an extra context switch. We found that this optimization reduces the cost of duplicate elimination in Grappa by roughly $2\times$ relative to using blocking inserts, which contributes to the particularly good performance of Grappa code for the two-hop query (see Section 4.4). This technique generalizes to continuation passing optimizations for certain types of queries, which is a subject for future work.

4 Evaluation

As a first step in studying our approach, we want to answer two questions experimentally. First, does our generated C++ code significantly outperform traditional DBMSs? Second, is Grappa an effective platform for parallel path query execution even without clever partitioning and other customizations for this task?

To answer these questions, we executed path queries to count distinct two-hop paths and triangles in standard public graph datasets. For the first question, we compared our generated C++ queries to Postgres, a well-studied DBMS. Though the Postgres installation was heavily indexed and our C++ code read un-indexed data from disk, our C++ code generated from Datalog was $3.5\times$–$5\times$ faster on a more skewed data set and $5\times$–$7.5\times$ faster on a less skewed data set.

For the second question, we compared Grappa with Greenplum, a commercial parallel DBMS. We evaluated Grappa on clusters comprising 2 to 64 nodes and an 8-node Greenplum installation. Without making any modifications to Grappa to support our application, the 8-node Grappa cluster completed queries as fast or faster than the 8-node Greenplum cluster, and scaled well to 32 nodes. We also found that Grappa's good performance extended across datasets and queries.

Table 1: Salient properties of the graphs studied

Dataset	# Vertices (M)	# Edges (M)	# Distinct 2-hop paths	# Triangles
BSN	0.685	7.60	78 350 597	6 935 709
Twitter subset	0.166	4.53	1 056 317 985	14 912 950
com-livejournal	4.00	34.7	735 398 579	—
soc-livejournal	4.85	69.0	—	112 319 229

4.1 Datasets

We used standard, public graph datasets for our evaluations: the Berkeley-Stanford Network (BSN) graph [14]; a subset of the Twitter follower graph [13]; and two SNAP LiveJournal graphs [29, 4]. We summarize salient properties of these graphs in Table 1. The Twitter subset is notable for its significant skew, leading to large intermediate results (discussed in 4.3). We evaluate the following queries.

4.2 Test Queries

In this paper, we are concerned primarily with relational, in-core execution techniques for graph-based tasks. We thus choose our queries and datasets to exercise these design points, namely choosing queries that will fit in the system memory but that are large enough to expose parallelism.

Two-path: count the number of distinct pairs (x, z) such that vertex x has a path of length two to z through any vertex y.

```
select count(*) from (select distinct a.src,b.dst from
    edges a, edges b where a.dst = b.src) z;
```

Triangle: count the number of ordered triangles in the graph.

```
select count(*) from edges a, edges b, edges c where a.src <
    a.dst and a.dst = b.src and b.src < b.dst and b.dst = c.
    src and c.dst = a.src;
```

A **variant three-path** query was also used to test the performance of queries involving multiple graph relations. For these experiments, the BSN and Twitter data sets were split into two disjoint relations, the larger of which contains roughly 90% of the original edges. The first hop is taken in the smaller relation, then the intermediate results are joined to the larger relation.

4.3 Single-node experiments: C++ vs. Postgres

We compared the generated C++ code against SQL queries in an indexed relational implementation using PostgreSQL.

All tests were performed on a shared-memory system running Linux kernel 2.6.32. The machine has 0.5TB of main memory and 4 sockets, each with a 2.0GHz, 12-core AMD Magny-cours processor. After code generation, the resulting C++ code was compiled using GCC 4.7 with -O3 optimization.

Postgres 8.4 was configured to use 64GB of shared buffer space and 50GB of working memory. Indexes were created on both the src and dst variables of the $edges$ relation, and then $edges$ was clustered on the src variable. These optimizations were applied in an

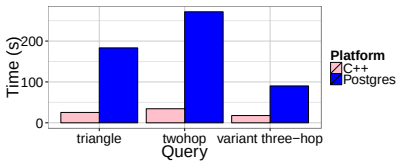


Fig. 3: Runtimes for the distinct two-hop, triangle, and variant three-hop queries on the BSN graph

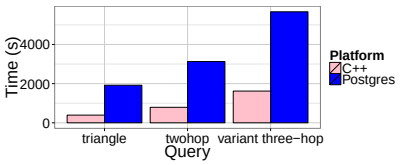


Fig. 4: Runtimes for the distinct two-hop, triangle and variant three-hop queries on the Twitter subset

attempt to minimize the runtime for the queries. For all three plans, the query optimizer chose to execute a sort-merge join. For the two path query, the table is already sorted by src, so one instance of the $edges$ relation is sorted by dst, and a standard sort-merge join was performed. In the case of triangle and three path, the intermediate result from the first join was sorted on $b.dst$, and then a sort-merge join was performed between the intermediate result and c. For comparison, we reran the Postgres experiments with sort-merge joins disabled; the resulting plans used a hash-join, like our code, but were slower than the original plans.

Figure 3 and Figure 4 show the runtime comparison of the two -path, three-path, and triangle queries for C++ vs. Postgres on the BSN and Twitter graphs. Our automatically generated C++ code runs $3.5\times$–$5\times$ faster than Postgres on the twitter data set and $5\times$–$7.5\times$ faster on the BSN data set.

Queries on the Twitter graph were slower because they resulted in many more results (Table 1). The key insight is that Twitter has both larger average degree (27 vs 11) and more vertices with both large in-degree and out-degree. The maximum out-degree in Twitter is 20 383, two orders of magnitude greater than the BSN graph. These properties of the Twitter graph cause the intermediate results of self-joins to be orders of magnitude larger than in the BSN graph, and indeed we see that Twitter has $13\times$ as many two-paths.

The triangle query on a given data set is always faster than distinct two-hop, despite having larger intermediate join results—the extra join results in an order of magnitude more results than both two-hop and variant three-hop. The reason is that the most costly step in the computation is duplicate elimination. Indeed, when we counted the number of operations used to implement duplicate elimination (map lookups and set inserts), we found a strong correlation with the runtime of the program (Figure 5), reinforcing that duplicate elimination dominates costs.

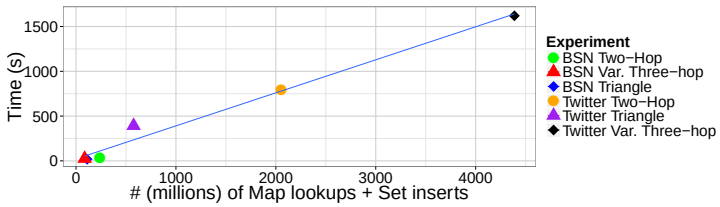


Fig. 5: There is a correlation between the sum of `map` lookups and `set` inserts and the runtime of the query generated as C++.

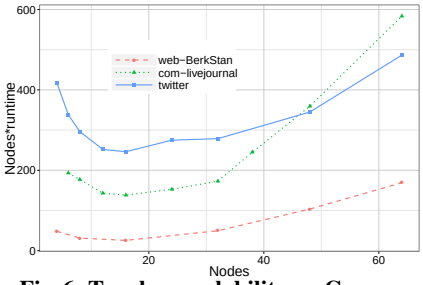


Fig. 6: Two-hop scalability on Grappa

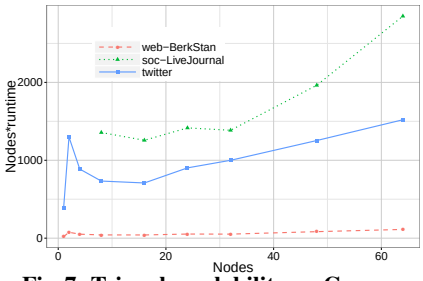


Fig. 7: Triangle scalability on Grappa

4.4 Parallel experiments: Grappa and Greenplum

We evaluate the scalability of Grappa and compare absolute performance that of Greenplum [27], a commercial parallel DBMS. To scale to larger systems, we used the same queries as above and extend to bigger datasets. For each dataset and query, we compare the runtime as we scale up the number of machines.

As Grappa is a research system for which we are still determining the best optimization strategies, we did not generate the Grappa code automatically. Instead, the query was manually coded in a style that is essentially isomorphic to the serial case.

We run the Grappa queries on allocations of a 144-node cluster of AMD Interlagos processors. Nodes have 32 cores (16 full datapaths) running at 2.1-GHz in two sockets, 64GB of RAM, and 40Gb Mellanox ConnectX-2 InfiniBand network cards. Nodes are connected via a QLogic InfiniBand switch.

Due to time and administration constraints, the Greenplum installation runs on a different system: an 8-node cluster of 2.0-GHz, 8-core Intel Xeon processors with 16GB of RAM. Although setup this does not provide an apples-to-apples comparison between the two systems, we can still provide some context for the performance of Grappa queries. Greenplum database was tuned to utilize as much memory as possible and configured to use 2 segments per node. We show the best results between Greenplum partitions on src, dst, or $random$ (although all runtimes were within about 10%).

First, we examine the parallel efficiency of the queries implemented on Grappa. Figure 7 and Figure 6 illustrate scalability using the metric of number of nodes multiplied by the runtime. With perfect scaling, this line would be flat (as doubling the number of cores would halve the runtime). Increasing values indicate suboptimal scaling.

On both queries, going from one node to two nodes incurs a performance penalty, as memory references must now go over the network. Four nodes is sufficient to gain back performance with parallelism. Two-hop on the Twitter subset scales well and runs in as little as 6.5s with 64 nodes. Most of the time is spent on insertions into the distinct set of results. For the other datasets, two-hop performance does not scale well beyond 32 nodes; in fact, for this query it degrades. Because 32 nodes utilizes all the data parallelism in these two datasets, the rising cost of set inserts over the network outweighs having more aggregate memory bandwidth. On triangles, Grappa scales well up to 32 nodes on com-livejournal and BSN but less efficiently on Twitter.

Unique among the systems studied, Grappa performs better on two-hop than on triangles. This improvement is not surprising, because Grappa is designed for high throughput random access, which occurs in hash insertions. Although context switches

Table 2: Performance of code manually written for Grappa and OpenMP according to our generation technique and SQL queries on Greenplum

System	# Nodes	Query	Dataset	Runtime
Greenplum	8	two-hop distinct	com-livejournal	265.5s
Grappa	8	two-hop distinct	com-livejournal	24.0s
Grappa	32	two-hop distinct	com-livejournal	5.3s
Greenplum	8	triangle	Twitter	84.3s
Grappa	8	triangle	Twitter	91.6s
Grappa	64	triangle	Twitter	24.0s
OpenMP	1	triangle	Twitter	110.8s

are fast, eliminating them in the inner loop can increase performance. Triangles requires more of them: one path through the triangles pipeline requires a task to do two blocking lookups (joins), while one path through the two-hop pipeline requires a task to do just one blocking lookup (join) and one non-blocking insert (distinct). Since set insertions are *fire-and-forget* and the inner loop of triangles contains no remote memory operations, setting the parallel granularity of the inner loop to be large (around 128) gave the best performance.

In Table 2, we list results for Greenplum and Grappa. These results were collected on different machines and networks; however, they indicate that graph path queries compiled to Grappa have the potential to be competitive with a commercial parallel RDBMS, especially in the case of duplicate elimination. To get an indication of the parallel performance of a single shared memory node using our code generation, we also wrote the triangle query by augmenting the generated C++ code with OpenMP pragmas. Shown is the best result with 16 cores, tuned to use dynamic scheduling on the outer two loops and guided scheduling with a chunksize of 8 on the inner loop. Due to additional overhead to share memory between cores in Grappa, for a single node, Grappa's performance with 16 cores equals that of 4 cores in OpenMP.

5 Related work

DBToaster [2] compiles C++ code from SQL queries for fast delta processing for the view maintenance problem. More generally, Neumann compiles TPC-H style analytics queries to LLVM bytecode and C [20] using a typical iterator framework. In contrast, we seek specialized algorithms for specific graph query patterns.

Vertex-centric programming models including Pregel [17] and GraphLab [9] have become popular for graph computations. GraphLab supports asynchronous computation and prioritized scheduling, but offers no declarative query interface and cannot express simple relational idioms. By making Grappa one of our compiler targets, we can also support asynchronous execution.

Neo4j [1] is a graph database, with its own data model and graph query language. Unlike Grappa, the entire graph is replicated on all compute nodes, so it cannot scale to large graphs. Recent work has shown that SQL DBMSs can compete with graph databases for real world network datasets and shortest path queries [28].

SPARQL is a pattern language for graphs represented as a set of (subject, predicate, object) triples. A number of systems have focused on executing and optimizing

SPARQL queries [8, 21], but we find SPARQL to be neither necessary nor sufficient for graph manipulation: Datalog is strictly more expressive than SPARQL without v1.1 path expressions [3], and the semantics of path expressions make query processing intractable [16].

Parallel databases like Greenplum are like conventional relational DBMSs but parallelize individual queries across shared-nothing architectures. Vertica [11] is a parallel DBMS designed for analytics and includes just-in-time compilation techniques. Grappa provides shared memory, as well as active messages, to the database system programmer. Since we are concerned with in-memory execution, we are exploring compiling rather than interpreting queries.

6 Future Work

We have focused on only a narrow class of path-counting queries; we plan to extend the framework to handle full Datalog. This allows for a much larger scope of graph queries, such as conjunctive regular path queries. Datalog is sufficient to express most or all of the queries handled in graph databases.

Our code generation framework will be extended to generate specialized code for some recognizable patterns (such as the two-hop optimization explored in this paper) and "canned" best-known algorithms for some specific queries. While a conventional DBMS can be extended similarly, a code generation approach suggests richer opportunities for incorporating user code into optimizations and for library-writing level users to write generators for special classes of queries, as Rompf et al [25] demonstrated for domain specific languages.

7 Conclusions

We wish to show that relational query languages are an attractive option for modern graph queries on complex data. Our experiments demonstrate that generated C++ code and analogous Grappa code can outperform traditional DBMSs and parallel DBMSs for non-recursive graph queries. Query execution code for Grappa is simple, being symmetric in structure to sequential C++. This simplicity, combined with Grappa's good scalability, makes our code generation an easy and efficient method for relational queries on real-world graphs in a distributed setting.

References

1. neo4j open source graph database. `http://neo4j.org/`, May 2013.
2. Y. Ahmad and C. Koch. DBToaster: a SQL compiler for high-performance delta processing in main-memory databases. *Proc. VLDB Endow.*, 2(2):1566–1569, Aug. 2009.
3. R. Angles and C. Gutierrez. The expressive power of SPARQL. In *The Semantic Web - ISWC 2008*, volume 5318 of *Lecture Notes in Computer Science*, pages 114–129. Springer Berlin Heidelberg, 2008.
4. L. Backstrom et al. Group formation in large social networks: membership, growth, and evolution. In *ACM KDD*, pages 44–54, 2006.
5. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.
6. J. Caverlee and L. Liu. Countering web spam with credibility-based link analysis. In *ACM Principles of Distributed Computing (PODC)*, pages 157–166, 2007.

7. S. Chen, A. Ailamaki, P. Gibbons, and T. Mowry. Improving hash join performance through prefetching. In *Intl. Conference on Data Engineering (ICDE)*, pages 116–127, 2004.
8. O. Erling and I. Mikhailov. Virtuoso: RDF support in a native RDBMS. In *Semantic Web Information Management*, pages 501–519. Springer Berlin Heidelberg, 2010.
9. J. E. Gonzalez et al. PowerGraph: distributed graph-parallel computation on natural graphs. In *USENIX Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2012.
10. A. A. Hagberg et al. Exploring network structure, dynamics, and function using NetworkX. In *Python in Science Conference (SciPy)*, pages 11–15, Aug. 2008.
11. HP-Vertica. Vertica analytics platform. `http://www.vertica.com`, June 2013.
12. T. G. Kolda, A. Pinar, T. Plantenga, C. Seshadhri, and C. Task. Counting triangles in massive graphs with MapReduce. *arXiv preprint arXiv:1301.5887*, 2013.
13. H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Intl. Conference on World Wide Web (WWW)*, pages 591–600, 2010.
14. J. Leskovec et al. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *CoRR*, abs/0810.1355, 2008.
15. B. T. Loo et al. Declarative routing: extensible routing with declarative queries. *SIGCOMM Comput. Commun. Rev.*, 35(4):289–300, Aug. 2005.
16. K. Losemann and W. Martens. The complexity of evaluating path expressions in SPARQL. In *Proceedings of Principles of Database Systems (PODS)*, 2012.
17. G. Malewicz et al. Pregel: A system for large-scale graph processing. In *ACM SIGMOD*, pages 135–146, 2010.
18. A. Mandal, R. Fowler, and A. Porterfield. Modeling memory concurrency for multi-socket multi-core systems. *Performance Analysis of Systems Software (ISPASS)*, March 2010.
19. J. Nelson et al. Crunching large graphs with commodity processors. In *USENIX Conference on Hot Topics in Parallelism (HotPar)*, pages 10–10, 2011.
20. T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.
21. T. Neumann and G. Weikum. x-RDF-3X: Fast querying, high update rates, and consistency for RDF databases. In *Proceedings of the 36th international conference on Very Large Data Bases*, PVLDB'13, 2010.
22. A. Pavan, K. Tangwongan, and S. Tirthapura. Parallel and distributed triangle counting on graph streams. Technical report, IBM, 2013.
23. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. In I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, and L. Aroyo, editors, *The Semantic Web - ISWC 2006*, volume 4273 of *Lecture Notes in Computer Science*, pages 30–43. Springer Berlin Heidelberg, 2006.
24. M. Przyjaciel-Zablocki et al. RDFPath: path query processing on large RDF graphs with MapReduce. In *Extended Semantic Web Conference (ESWC)*, 2011.
25. T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *SIGPLAN Not.*, 46(2):127–136, Oct. 2010.
26. J. Seo, S. Guo, and M. S. Lam. SociaLite: Datalog extensions for efficient social network analysis. In *29th IEEE International Conference on Data Engineering*. IEEE, 2013.
27. F. M. Waas. Beyond conventional data warehousing–Massively parallel data processing with Greenplum database. *Springer LN Business Information Processing*, 27:89–96, 2009.
28. A. Welc, R. Raman, Z. Wu, S. Hong, H. Chafi, and J. Banerjee. Graph analysis: do we have to reinvent the wheel? In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 7:1–7:6, New York, NY, USA, 2013. ACM.
29. J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *ACM SIGKDD Workshop on Mining Data Semantics*, pages 3:1–3:8, 2012.
30. W. Zhang, D. Zhao, and X. Wang. Agglomerative clustering via maximum incremental path integral. *Pattern Recognition*, (0):–, 2013.