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 applications have motivated a new class of graph databases emphasizing in-memory graph traversal primitives instead of join-oriented query processing. However, we hypothesize that code-generation techniques and fast in-memory algorithms on modern hardware can reduce the overhead associated with a relational implementation of graph analysis tasks.
In this paper, we present preliminary results for this approach on path-counting queries, which includes triangle counting as a special case. To evaluate performance, we automatically generate in-memory pipelined hash-join query plans in C++ directly from Datalog queries, 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 parallel run time system for distributed memory architectures that provides a shared-memory abstraction and mechanisms for reducing communication costs. Our experiments show that the generated Grappa programs perform well against Greenplum, a commercial parallel database platform. We find 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 the large-scale graphs found in social networking, web analytics, and bioinformatics has led to a number of specialized graph processing systems [6, 1]. However, large graphs and large relations tend to coexist in realistic applications, suggesting that relational models and languages should not be completely abandoned. We consider whether in-memory approaches to relational query processing can significantly outperform conventional relational databases for graph-oriented queries, reducing the need for specialized graph-only systems.

In this paper, we present preliminary results using code generation, straightforward in-memory query plans, and a novel parallel runtime system to evaluate a class of *path-counting* queries, which includes triangle counting [19] as a special case. These queries arise in both graph and relational contexts, including in credibility algorithms for detecting spam [4] and in probabilistic algorithms [23]. To handle "hybrid" graph-relational applications, we retain the relational data model and a relational query language (a subset of Datalog). We show that improving performance is more a function of reducing overhead and exploiting modern hardware than fundamentally changing the evaluation strategy.

For sequential evaluation, we adopt conventional relational query processing strategies (pipelined plans, hash joins), but generate simple, minimalist C++ programs that implement a single query plan with very little overhead in memory allocation or data structure manipulation. Our approach is in part inspired by other work in compiling relational algebra expressions and SQL queries [17], but our overarching goal is to support a variety of backend runtime systems rather than generating the fastest possible machine code. Our interest in Datalog is to extend our approach to handle recursive queries and therefore more complex graph queries, but we do not consider recursive queries in this paper.

For parallel evaluation, we are concerned with the competing factors of scaling up: distributing the data allows for higher bandwidth access to it but greater network usage for random accesses. For workloads with irregular memory access, like that in sequences of hash joins, high throughput can still be achieved in modern processors with sufficient concurrency [15]. With this in mind, we employ a novel parallel runtime system, Grappa [16], 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, simple C++ programs that implement join-based query plans.
2. A suite of 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 3.5X-7.5X faster than tuned and optimized relational query plans.
4. Experimental results comparing path-counting queries in Grappa to the Greenplum commercial parallel RDBMS.

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, 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. Grappa exploits large amounts of concurrency for high throughput. 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

We argue that Grappa provides an appropriate level of abstraction for a parallel graph query runtime. The global address space allows us to generate relatively simple code, and parallel loop constructs avoid the need to emit explicitly multi-threaded routines. However, Grappa does not preclude us from optimizing for locality and emitting more sophisticated distributed algorithms for special situations using lower-level communication abstractions. Concurrency can be expressed with a variety of arbitrarily nestable parallel loop constructs that respect spatial locality when it exists; this idiom is a good fit for pipelined query plans.

3 Code Generation for Path-Counting Queries

We explore a code generation approach for high-performance in-memory query processing. We favor code generation over query plan interpretation for two main reasons: First, relative to disk-based processing, the higher speed of memory exacerbates the overhead of iterator-based interpretation unless you give up pipelining, as pointed out by Neumann [17]. Second, Kim et al. [9] and others have demonstrated dramatic performance improvements for relational operators by using features of modern processors. We think a code generation approach has more potential to incorporate these kinds of techniques, as well as compiler optimizations over whole queries, than interpreted-plan approaches.

Following Seo, Guo, and Lam [20], 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

$$\gamma_{count}(\sigma_c(\sigma R_1 \bowtie \sigma R_2 \bowtie \ldots \bowtie \sigma R_N))$$

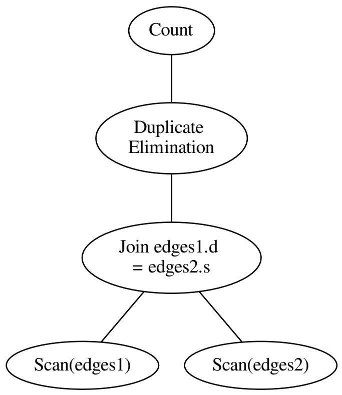
where γ is an aggregation operator for counting the final results. The extra selection operator σ_c can enforce relationships between non-adjacent vertices in the path. In particular, this condition can enforce that the path form a cycle, as in the triangle queries. Each leaf and each internal node in 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 to have 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 node a to node b. While a table is not the most memory efficient way of representing a graph [20], 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 [13]. In Datalog, this is expressed as

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

A three-hop query would simply add one additional join to the query above. A popular case of the three-hop query in large graphs is triangle counting[10][18], 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 node. In Datalog, directed triangles are expressed as:



```
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 two-hop query. In the code on the right, the map variable maps each node x to all edges (x, y). This allows the outer loop (line 4) to act as a relational group-by, and the set necessary for duplicate elimination can be moved inside the group-by (line 8) and cleared after each iteration (line 15), allowing for better memory-usage. The hash-join probe for the condition *edges1=edges2* shown in the relational algebra is displayed on line 10.

```
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. 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 [7] 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.

Two-hop requires duplicate elimination, as there may be multiple paths from node s to node d. We perform duplicate elimination by inserting results into a set data structure. In general, path queries may be interested in existence of a path of length k from node s to node d, but not in all such paths. In these cases, we found roughly a $5\times$

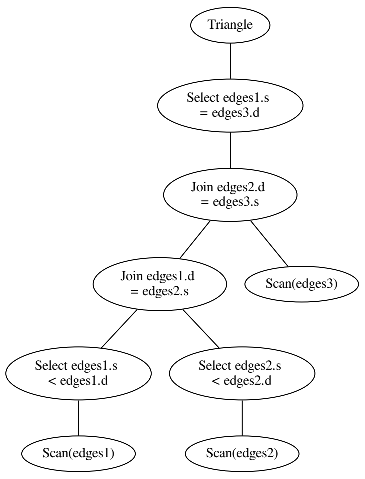
[1] All queries considered in this paper can be expressed with a single Datalog rule.

decrease in runtime by performing a relational group-by operation over the s variable and iterating over each group one at a time. The set could then be safely cleared after each iteration, preventing the set from becoming unnecessarily large. Consequently, whenever a query requires a distinct variable from the outer relation, we perform a group-by on the outer relation. This 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. 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`, which uses a red-black tree. Although inserts and lookups are asymptotically slower than a hash map in theory, in practice we found the red-black tree implementation was never a performance factor. Similarly, duplicate elimination is performed by inserting results into an STL `set`, which also uses a red-black tree internally.

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.

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 possible compiler extensions to support more general graph queries (including recursive queries).

3.3 Queries in Grappa

Each relation is scanned and loaded into an array, which Grappa distributes in a block-cyclic fashion to preserve segments of spatial locality while balancing load.

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. These abstractions allow us to produce nested, pipelined plans that are analogous to the serial case. For these plans, 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 tasks required to tolerate latency. To exploit inter-pipeline parallelism, Grappa can spawn each pipeline as a separate task. No synchronization is required between independent subplans; the only synchronization necessary is to ensure that probes of the hash join (the `lookup` method) block until the build phase is complete.

Grappa's parallel for-loop mechanism is flexible enough to support 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 at runtime against unpredictable result sizes.

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. All operations on a given bucket are performed as transactions, by the simple fact that each core has exclusive access to its local memory. This is an example of Grappa's PGAS model allowing for 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 in [5].

For insert operations during duplicate elimination, the task does not need to block waiting for the insertion to complete, saving an extra context switch. This effect appears to reduce the relative cost of duplicate elimination in Grappa, but we did not study this hypothesis quantitatively.

4 Evaluation

To understand whether this approach is promising, we want to answer two questions experimentally. First, does our generated C++ code outperform sequential DBMS query execution on graph queries? Second, is Grappa an effective platform for parallel path query execution even without clever partitioning and other customizations for this task?

Dataset	# Vertices	# Edges	# Distinct 2-hop paths	# Triangles
BSN	685 230	7 600 595	78 350 597	6 935 709
Twitter subset	166 317	4 532 185	1 056 317 985	14 912 950
com-livejournal	3 997 962	34 681 189	735 398 579	—
soc-livejournal	4 847 571	68 993 773	—	112 319 229

Table 1: Salient properties of the graphs studied.

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 cluster installations of Grappa and 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.

4.1 Datasets

We used standard, public graph datasets for our evaluations: the Berkeley-Stanford Network (BSN) graph [12]; a subset of the Twitter follower graph [11]; and two SNAP LiveJournal graphs [22, 3]. 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;
```

Three-path: count the number of distinct pairs (x, w) such that vertex x has a path of length three to w through any vertices y and z.

```
select count(*) from (select distinct a.src,c.dst from
   edges_small a, edges_small b, edges_large c where a.dst =
    b.src and b.dst = c.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. This query demonstrates how a relational model can easily allow us to write complex queries that ask about more than the structure of a graph.

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

The generated C++ code was compared against SQL queries in Postgres. Postgres is a fully functional DBMS and provides a reasonable baseline for comparison.

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 so that it would not use disk after an initial scan. 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 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.

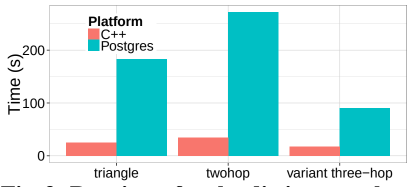


Fig. 3: Runtimes for the distinct two-hop, triangle, and variant three-hop queries on the BSN graph.

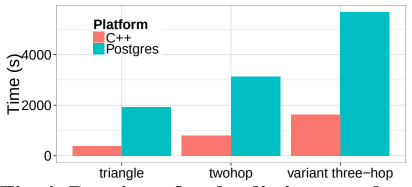


Fig. 4: Runtimes for the distinct two-hop, triangle and variant three-hop queries on the Twitter subset.

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× 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. As described in Section 3.2, the triangle query does not require this expensive step. 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.

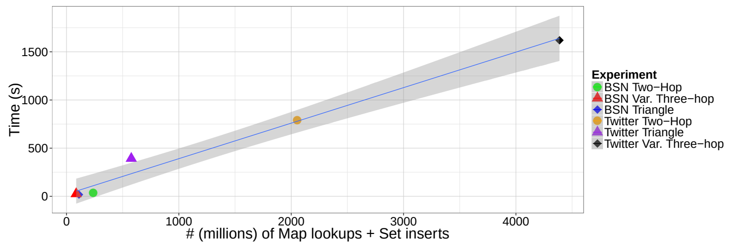


Fig. 5: There is a correlation between the sum of `map` lookups and `set` inserts and the runtime of the query generated as C++.

4.4 Parallel experiments: Grappa and Greenplum

We evaluate the scalability of Grappa and compare absolute performance that of Greenplum [21], 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. Though 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

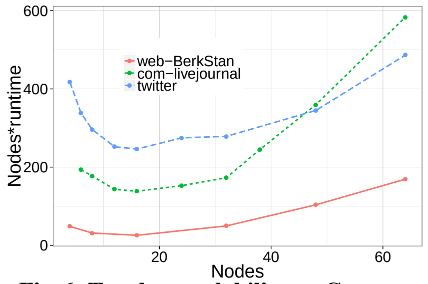


Fig. 6: Two-hop scalability on Grappa.

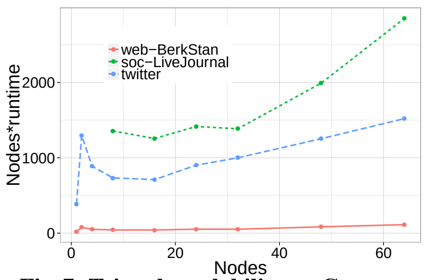


Fig. 7: Triangle scalability on Grappa.

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 is not surprising, because Grappa is designed for high throughput random access, which occurs in hash insertions. Although context switches 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.

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

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

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 augmented the generated C++ code with OpenMP pragmas (we did not parallelize the hash build since its runtime is small relative to the probe phase). Shown is the result with 16 cores, using dynamic scheduling on the outer two loops and guided scheduling with a chunksize of 8 on the inner loop. Due to single-node shared memory overheads in Grappa's implementation, for a single node, Grappa's performance with 16 cores equals that of 4 cores in OpenMP.

5 Related work

Some recent effort has been focused on generating and compiling code for SQL queries. DBToaster[2] compiles C++ code from SQL queries for fast delta processing for the view maintenance problem. More generally, [17] compiles TPC-H style analytics queries to LLVM bytecode and C. A `produce` and `consume` method is written for each operator to emit code. Instead of using a produce method, we just start processing the query plan at the leaves. The work focuses on speeding up queries found in common DBMS benchmarks, while our work is motivated by making graph-style queries efficient.

Vertex-centric programming models have become popular. Because they involve expressing a computation on each datum, like MapReduce, they can naturally parallelize a program and abstract away distributed computing concerns. Pregel [14] and GraphLab [6] adopt this model. GraphLab supports asynchronous computation, allowing for prioritization, which is important for faster convergence of certain iterative machine learning algorithms. We target the space of graph queries and want to provide a declarative programming abstraction, as well as stay within a relational model. By making Grappa one of our compiler targets, we can also support asynchronous execution.

Neo4j [1] is a graph database. It has its own data model and graph query language. The motivation for not using the relational model is performance. This system represents the data as a graph. Currently, the entire graph is replicated on all compute nodes, so unlike Grappa it is not truly distributed.

Parallel databases like Greenplum are like conventional relational DBMSs but parallelize individual queries across shared-nothing architectures. Vertica [8] is a parallel DBMS designed for analytics and includes just-in-time compilation techniques. Grappa provides shared-memory to the 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; in future work, we will extend the framework to support any valid Datalog query, including recursive queries.

Some work has shown that more natural graph data structures, such as adjacency list-like structures, can be used in databases while harnessing all of the relational power usually applied to tables in SQL[20]. We hope to expand our data representation model in both C++ and Grappa to reflect a more natural graph structure, and explore how nested data structures can be applied to relations that are more complex than edges in a graph.

The ideas presented for compiling graph-path queries to C++ can be applied to compiling Grappa code, which is more complex than standard C++. While the high-level structure of the code would be very similar to the C++ version, generated code for Grappa would also include code to get efficient parallelism out of the Grappa system. We believe that a Grappa code generator could make utilizing the Grappa system much easier for both experienced users and new users.

7 Conclusions

The experiments shown demonstrate that generated C++ code and analogous Grappa code can easily outperform traditional DBMSs and distributed DBMSs for non-recursive graph queries. The symmetry of the C++ and Grappa code points to the ability to easily generate Grappa code from Datalog, which, combined with Grappa's scalability, could combine an easy, efficient method for relational evaluation of real-world graphs in a distributed setting. Future work should lead to the ability to generate code for recursive graph queries and an efficient method for graph analytics without giving up on the power of the relational model.

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. L. Backstrom et al. Group formation in large social networks: membership, growth, and evolution. In *ACM KDD*, pages 44–54, 2006.
4. J. Caverlee and L. Liu. Countering web spam with credibility-based link analysis. In *ACM Principles of Distributed Computing (PODC)*, pages 157–166, 2007.
5. 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.
6. 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.
7. A. A. Hagberg et al. Exploring network structure, dynamics, and function using NetworkX. In *Python in Science Conference (SciPy)*, pages 11–15, Aug. 2008.
8. HP-Vertica. Vertica analytics platform. `http://www.vertica.com`, June 2013.
9. Kim and et al. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. *Proc. VLDB Endow.*, 2(2):1378–1389, Aug. 2009.
10. 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.
11. 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.
12. 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.
13. B. T. Loo et al. Declarative routing: extensible routing with declarative queries. *SIGCOMM Comput. Commun. Rev.*, 35(4):289–300, Aug. 2005.
14. G. Malewicz et al. Pregel: A system for large-scale graph processing. In *ACM SIGMOD*, pages 135–146, 2010.
15. A. Mandal, R. Fowler, and A. Porterfield. Modeling memory concurrency for multi-socket multi-core systems. *Performance Analysis of Systems Software (ISPASS)*, March 2010.
16. J. Nelson et al. Crunching large graphs with commodity processors. In *USENIX Conference on Hot Topics in Parallelism (HotPar)*, pages 10–10, 2011.
17. T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.
18. A. Pavan, K. Tangwongan, and S. Tirthapura. Parallel and distributed triangle counting on graph streams. Technical report, IBM, 2013.
19. M. Przyjaciel-Zablocki et al. RDFPath: path query processing on large RDF graphs with MapReduce. In *Extended Semantic Web Conference (ESWC)*, 2011.
20. 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.

21. F. M. Waas. Beyond conventional data warehousing–Massively parallel data processing with Greenplum database. *Springer LN Business Information Processing*, 27:89–96, 2009.
22. 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.
23. W. Zhang, D. Zhao, and X. Wang. Agglomerative clustering via maximum incremental path integral. *Pattern Recognition*, (0):–, 2013.