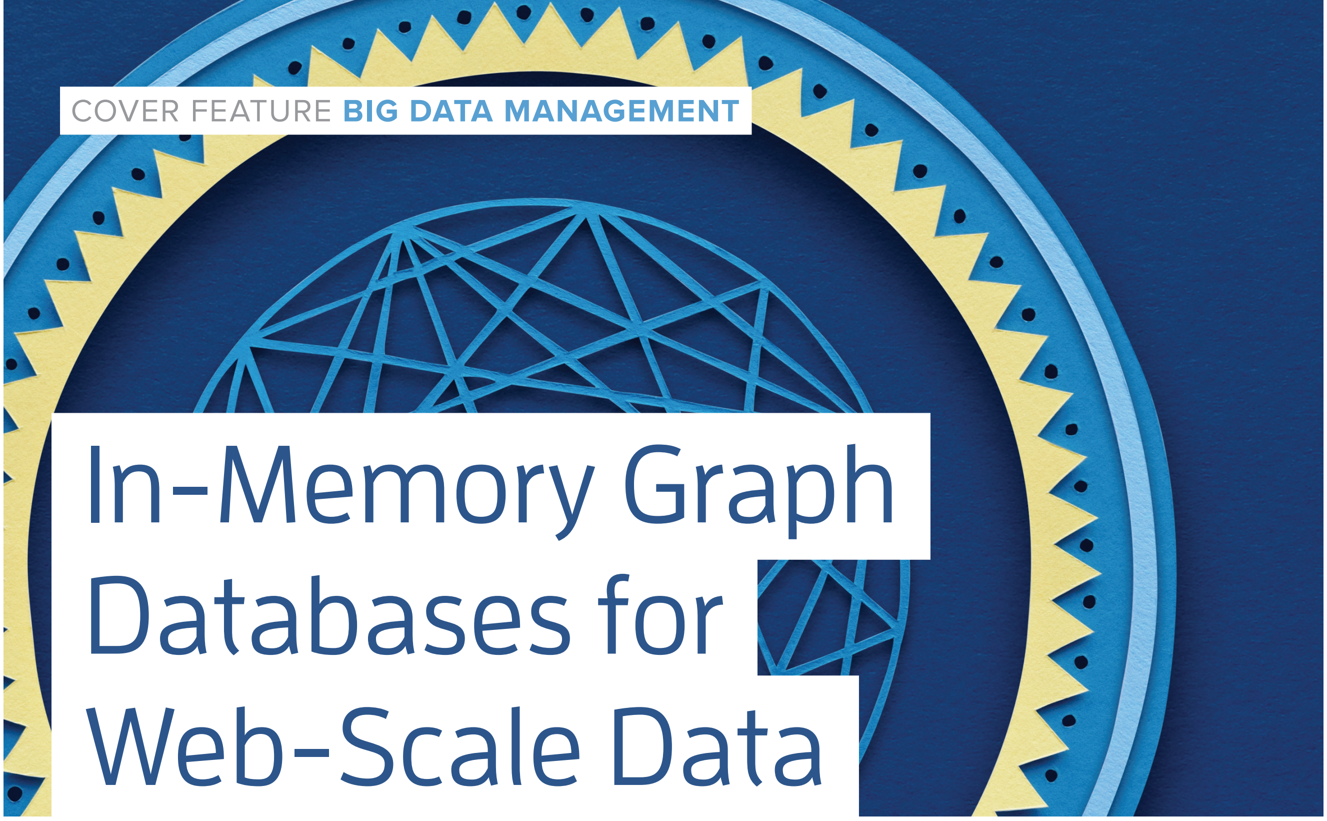
COVER FEATURE **BIG DATA MANAGEMENT**



In-Memory Graph Databases for Web-Scale Data

Vito Giovanni Castellana, Alessandro Morari, Jesse Weaver, Antonino Tumeo, and David Haglin, Pacific Northwest National Laboratory

Oreste Villa, NVIDIA Research

John Feo, Context Relevant

A software stack relies primarily on graph-based methods to implement scalable resource description framework databases on top of commodity clusters, providing an inexpensive way to extract meaning from volumes of heterogeneous data.

Arguably, the most significant obstacle to handling the digital age's data deluge is finding a database model that addresses the "too big, too fast, too hard challenges"[1] that correspond to big data's high volume, velocity, and variety. To meet these challenges, the model must not only solve technical issues related to volume and velocity, but it must also support decision making, insight discovery, and process optimization, which relate to data variety.

Work on the Semantic Web has at least partly addressed the high-variety challenge by promoting the Resource Description Framework (RDF), a flexible model that lets data publishers explicitly map Web resource names and descriptions with associated semantics. In response to the World Wide Web Consortium's recommendation to use the RDF, communities from finance to healthcare have embraced it as a way to publish linked data. Major search engines, including Bing, Google, and Yahoo, recommend using schema.org annotations that help aggregate RDF data into datasets that are ready for additional processing, querying, and analysis.

Although increased RDF adoption is certainly addressing the variety challenge, it falls short of handling the massive volume as the Web continues to churn out data on an unprecedented scale. Addressing volume requires considering how RDF databases relate to graph methods and how graph methods relate to high-performance computing solutions.

An RDF organizes data as subject-predicate-object triples, and RDF statements naturally map to a directed labeled graph. To explore an RDF dataset, analysts can use query languages such as SPARQL, which expresses a query as a set of graph pattern-matching operations. Employing RDF and SPARQL potentially allows data mining through graph methods and crawling on graph-based data structures, both of which can reduce volume and increase performance relative to table-based relational databases. Size reductions stem from the graph structure's more efficient use of space, while performance increases come from exploiting graph methods' inherent parallelism.

However, fully realizing these benefits requires a new software framework paradigm. Most existing RDF databases, even those that store and retrieve data as triples, rely on the conventional techniques typical of relational databases, such as Virtuoso and 4store. These techniques neither exploit high-performance computing solutions nor account for a graph method's irregular behavior. Indeed, with conventional techniques, the traversal of a very large graph can result in unpredictable, fine-grained data accesses to any memory location, which is incompatible with high-performance computing solutions. The sidebar "Aligning Graph Methods and High-Performance Computing" describes this problem in more detail.

To address the need for a new paradigm, we developed the Graph database Engine for Multithreaded Systems (GEMS), a complete software framework for implementing RDF databases on commodity, distributed-memory, high-performance clusters. Unlike most RDF databases, databases structured with the GEMS software stack are customized to address the application of graph methods to large-scale datasets on clusters. They also incorporate principles that enable the automatic, efficient translation of SPARQL queries to C++.

ALIGNING GRAPH METHODS AND HIGH-PERFORMANCE COMPUTING

Graphs with many arcs, such as those in representations of large, complex RDF databases, are difficult to partition without generating load imbalance. Graph algorithms usually perform fine-grained, unpredictable data accesses, and often have high synchronization intensity, resulting in irregular behavior. In contrast, modern high-performance computing systems rely on powerful multicore processors, optimized for regular computations and for locality through deep cache hierarchies. Typically, these systems are distributed-memory clusters with nodes interconnected through high-performance networks that reach their peak bandwidth with large, batched data transfers and perform poorly with fine-grained communication. Consequently, algorithms for large-scale graphs are difficult to implement on clusters, and they often scale poorly on these systems.

The high-performance computing (HPC) community has invested significant effort in providing more efficient solutions for parallel processing large graphs.[1] Although controversial because of the benchmark used in its evaluation experiments and results applicability, the Graph 500 list (www.graph500.org) demonstrates the progress of modern HPC systems in graph processing. However, approaches to implementing an RDF database by fully exploiting graph methods should look at providing not only the substrate to efficiently execute graph algorithms but also an effective way to convert the queries to graph algorithms, thus minimizing any overhead and limiting any additional layers.

Reference

1. A. Lumsdaine et al., "Challenges in Parallel Graph Processing," *Parallel Processing Letters*, vol. 17, no. 1, 2007, pp. 5–20.

SCALABLE GRAPH-BASED FRAMEWORK

Figure 1 shows GEMS' three main components. Through a Web client interface, analysts load RDF databases and write or compile SPARQL queries, which GEMS can launch in parallel on the same database. Database loading consists of an *ingest* phase, which generates a graph and related dictionary from sets of RDF triples (in the N3 format, for example). The dictionary associates string labels with unique integer identifiers. Because layers are clearly separated, analysts can hand-code queries in C++ as graph pattern-matching operations.

Managing skewed data

SGLib manages the graph data structure and dictionary, encoding each RDF triple as three unique identifiers and partitioning the index range so that each node in the cluster gets an almost equal number of triples. The index of RDF triples is in subject-predicate-object or object-predicate-subject order. By exploiting the Global Memory and Threading (GMT) library's partitioned global address space, SGLib can automatically distribute among nodes any subject-predicate and object-predicate pairs (indexes) with highly skewed proportions of associated triples, thus avoiding the load imbalance from skewed data.

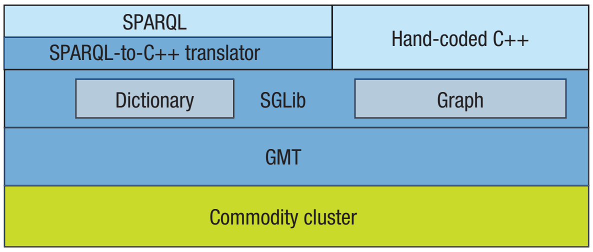


FIGURE 1. Graph database Engine for Multithreaded Systems (GEMS) layers on top of a commodity cluster. From the top, the layers comprise the SPARQL-to-C++ compiler; the Semantic Graph Library (SGLib), which provides a library of graph methods and supporting data structures; and the Global Memory and Threading (GMT) runtime library, which provides lightweight software multithreading and a partitioned global address space.

Before building the indexes, SGLib computes the average count for such pairs as well as the count's standard deviation. Given that information, it uses a simple heuristic to test the degree of skew in the pairs: a pair is highly skewed if it occurs more often than the average plus the standard deviation.

For each highly skewed subject-predicate (or object-predicate) pair, SGLib range-partitions the associated triples across all nodes and then separately range-partitions triples not associated with a highly skewed pair. SGLib supports graph explorations, including edge traversal, list retrieval, and label lookups. It also supports query processing by providing tables to store results and operations such as deduplication and grouping (for example, through the `Distinct` and `Group By` SPARQL constructs).

To process queries on the RDF graph, SGLib uses a constrained subgraph homomorphism algorithm, expressed as a collection of nested first-order function calls, one for each pattern edge. SGLib exposes primitives that explore graph edges or vertices and that can potentially spawn a task for each matching element. Because the graph might include several billion edges and vertices, it contains a large amount of parallelism, which GEMS' GMT library layer is well suited to exploit.

Managing queries

The GMT library[2] has three main features that hide the complexity of managing and querying a graph database on top of a commodity cluster:

- a virtual global address space that spans the memories of all nodes in the cluster,
- lightweight software multithreading, and
- network message aggregation (coalescing).

These features address some of the shortcomings that existing algorithms exhibit when applied to large graphs on distributed-memory clusters. For example, exploring large graphs usually generates fine-grained accesses to unpredictable memory locations, particularly for graphs with many edges. In contrast, HPC clusters with commodity nodes focus on exploiting locality and regular computation through deep cache hierarchies. Network interconnections reach their effective peak bandwidth only with large data transfers; small data transfers usually waste bandwidth for headers and control information.

Many solutions attempt to reduce data access latency. GMT employs lightweight software multithreading to *tolerate* it. When an algorithm needs to access data stored in a remote node, GMT quickly switches to another of the thousands of tasks available for each node, overlapping communication with computation. To maximize bandwidth utilization, GMT aggregates messages directed toward the same remote node.

Another problem is how to partition graph data structures on distributed memory. Partitioning algorithms might have greater computational complexity than the exploration algorithms, and uniform partitioning is prone to load unbalancing in strongly interconnected graphs. GMT's partitioned global address space removes the need to partition data structures by treating distributed memories across cluster nodes as a virtual shared address space. However, it still allows adding locality hints to optimize performance.

GMT provides the layers above it with a thin, customized API that allows task spawning in parallel loops, either through active messages on specific nodes or through cluster-wide load balancing. The API enables data freeing and movement (gets and puts) into and out of data structures allocated in the global address space. Data allocation can be either blocking or nonblocking.

Cooperation among layers

Although interacting through clearly separated abstractions, GEMS' three layers (SPARQL-to-C++ compiler, SGLib, and GMT) work together to provide the required functionalities and optimal performance. All layers see query processing as graph pattern matching, executed primarily through graph walking and finalized with table operations. The compiler sees only the methods to interact with the graph and the dictionary data structures that SGLib provides, so it is concerned with performing graph walks, retrieving edge lists, performing lookups, and executing operations on the results table.

If required, the compiler can interact with the GMT API to allocate additional data structures, but because of the partitioned global address space, it does not need to reason about data location. SGLib employs methods to manage the main database data structures, the additional data structures, and all the helper algorithms.

The current GEMS version requires a commodity x86 cluster, Linux with pthreads support, and a Message Passing Interface (MPI), mainly because of the basic requirements of its runtime layer.

PATTERN MATCHING

SPARQL queries retrieve information from graph-structured data by matching basic graph patterns (BGPs), which are sets of triple patterns on the data. BGP matching is relatively easy to model as a subgraph homomorphism algorithm. Our method draws from Ullmann's subgraph isomorphism algorithm,[3] which enumerates all possible vertices mappings in the graph pattern, *P*, onto the graph-data vertices, *D*, through a depth-first tree search.

A path from root to leaves in the tree denotes a complete mapping. The path represents a match if all vertices that are neighbors in the path are also neighbors in both *P* and *D* (preserve adjacency).

Pruning the search tree

As a purely structural subgraph homomorphism, this algorithm has exponential worst-case time complexity. However, the structure of RDF queries and RDF data offers substantial opportunities for pruning the search tree, making the adoption of similar pattern-matching algorithms feasible and profitable, even for large graph patterns and datasets.

Pruning is based primarily on value checking. All RDF graphs have a value, and any subject, predicate, and object has value as well. Thus, performing value checks while exploring the solution space can dramatically reduce the number of legal mappings. Further pruning opportunities stem from RDF query features; for example, queries can specify filtering constraints that highly increase their selectivity.

MANY SOLUTIONS ATTEMPT TO REDUCE DATA ACCESS LATENCY. THE GMT LIBRARY EMPLOYS LIGHTWEIGHT SOFTWARE MULTITHREADING TO *TOLERATE* IT.

GEMS models BGP matching as a sequence of graph walk operations (GWOs), each of which is associated with a triple pattern. However, GWOs alone cannot capture all the expressiveness and features that languages like SPARQL offer, so we have added table operations (TOs). GWOs retrieve information from the graph, and TOs elaborate it by sorting, projecting, aggregating, and combining data—operations that SGLib efficiently processes in parallel.

Figure 2 shows an example of how GWOs and TOs implement a SPARQL query.

The query in Figure 2a features a BGP constrained by two filters. One filter exploits negation: if the pattern specified under the `NOT EXISTS` clause matches the data, the overall match is not legal. The query has three stages: a GWO identifies matches for the first BGP, a second GWO identifies the nonlegal matches, and a sequence of TOs constructs the final results. Figure 2b shows the associated pattern graph.

Figure 2c shows the pseudocode of the query as modeled in GEMS. The system implements graph walks by exploiting the `forEach(S, P, O, [args], CB)` primitive, where `CB` is the callback function. The primitive spawns several tasks, which match the subject-predicate-object triple in the data. Tasks also check values, if the elements are valued, and execute the CB. The CB checks for constraint satisfaction, buffers or updates data as part of composing temporary or results tables, and takes the graph walk through further `forEach` calls.

The `scan_table(CB)` function (line 26) executes multiple CB instances associated with table rows. In Figure 2c, `scan_table(CB)` starts the second GWO.

Translating queries to C++

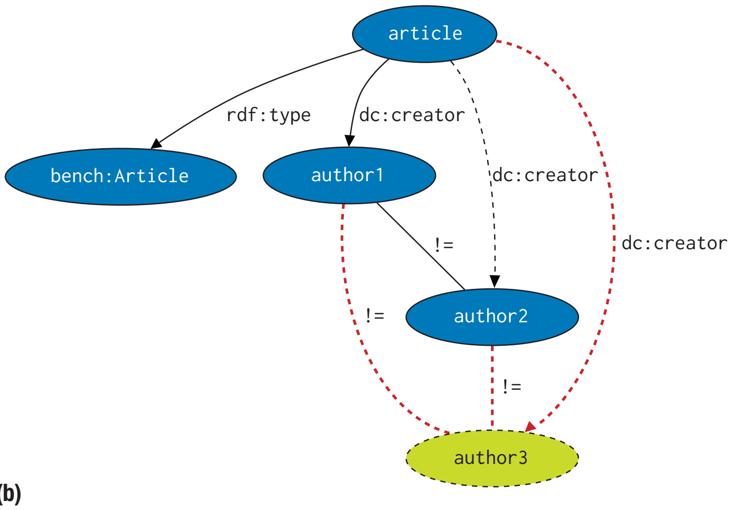
It is possible but not practical to implement the query through a single GWO. All the tasks finding a whole pattern match would have to notify other tasks that every match featuring the same `?article` variable is not legal. The required notifications would easily congest the network.

To avoid this issue, we designed several algorithms to implement SPARQL queries as compositions of GWOs and TOs and tailored them for the SPARQL-to-C++ translator.

As Figure 3 shows, the translation engine consists of the front-end

```
SELECT (count(?article) as ?numRes)
WHERE {
    ?article rdf:type bench:Article .
    ?article dc:creator ?author1 .
    ?article dc:creator ?author2 .
    FILTER(?author1 != ?author2)
    FILTER NOT EXISTS {
    ?article dc:creator ?author3
     FILTER((?author3!=?author2)
                 && (?author3!=?author1))
    }
}
```

(a)



(b)

```
fun2_1(tuple){
  args = {tuple.author1, tuple.author2};
  forEach(tuple.article, creator, ?author3, args, fun2_2);
}

fun1_2(article, creator, author2, args){
   if(args.author1!=author2)
          results.insert({article, args.author1, author2})
}
fun1_2(article, creator, author1){
  args = {author1};
  forEach(article, creator, ?author2, args, fun1_3);
}
fun1_1(article, type, article){
  forEach(article, creator, ?author1, fun1_2);
}

query_Q0(){
  //Graph Walk 1
  forEach(?article, type, Article, fun1_1);
  //Graph Walk 2
  results.scan_table(fun 2_1);
  //Computing final results
  results.update();
  results.count(article);
}
```

(c)

FIGURE 2. Applying basic graph pattern matching. (a) SPARQL query "Return the number of articles coauthored by exactly two people"; (b) pattern graph; and (c) query pseudocode in GEMS.

and back-end phases. Input language details are hidden from the back end, which exploits SGLib primitives as building blocks. Because the two phases are clearly separated, GEMS can support other input languages by changing only the front end, and different output APIs by modifying only the back end.

Front end. The front end is responsible for defining an optimized execution plan for the query, which it exposes to the back end through a low-level internal representation (LLIR). The front end is unaware of the underlying layers, producing an internal representation that is independent of GEMS' API.

The first step is to parse the input query, which generates a tree representation aligned with the SPARQL algebra as defined in the SPARQL standard. The front end purposely processes the algebraic representation to construct the query plan—a directed acyclic graph that defines a partial ordering among the basic operations for composing the query.

The front-end planner uses a cost model and a cardinality estimator to identify the most promising ordering among several candidates. The optimization process exploits an iterative dynamic planning algorithm to order the triple patterns to traverse within a graph walk; it also determines if multiple graph walks share components, which enables pattern matching only once within the same walk. As a final stage, the front end represents the optimal plan as a sequence of operations in JavaScript Object Notation (JSON), characterizing them by used variables and references to previous operations.

Back end. The back end processes the LLIR to construct an efficient query

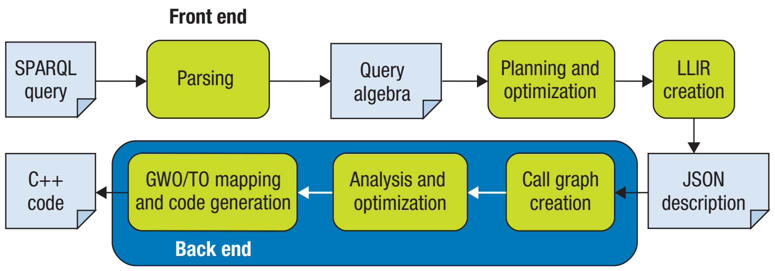


FIGURE 3. Automatic code generation flow in GEMS' SPARQL-to-C++ translator. The front end starts with the SPARQL query and ends with the creation of a low-level internal representation (LLIR), which it exposes to the back end. The back end is responsible for translating the LLIR to C++ code. JSON: JavaScript Object Notation; GWO: graph walk operation; TO: table operation.

implementation in C++. It begins by parsing the LLIR, which involves refining the variables and operations characterization and identifying definition of the SGLib data structures, such as tables, and their use points. Data structure definition and use information is not embedded in the input LLIR to keep it independent of the underlying APIs.

The second step is to create and analyze a call graph of the query on the basis of the precedence relations among operations. The graph's nodes are GWOs or TOs. A dataflow analysis identifies operations that might execute in parallel and partitions the call graph in sequences of data-dependent GWOs or TOs.

A liveness analysis follows, which evaluates the lifetime of variables and data structures, allowing the identification of the minimum set of variables that must be carried from call to call during a graph walk. Having a minimum set reduces the data passed as an argument to `forEach` functions and the variables that must be stored in tables, which in turn reduces the memory footprint of temporary data structures. In addition, liveness information is exploited to destroy (in the emitted code) data structures as soon as possible, freeing the allocated memory.

At this stage, all data structure layouts are defined, so a logical next step is to identify data structures with the same layout (store the same variables) that are reusable and remove all copies. GEMS then renames the removed copies with the identifier of the first occurring data structure instance. It then reanalyzes liveness, since copy removal might have changed data structure lifetimes. Variable lifetimes are not affected, so the algorithm does not require iteration.

The last step is to generate the C++ code. GEMS traverses the partitioned call graph in topological order and maps each operation to an implementation that follows a C++ template. This code generation strategy ensures that variable scoping is correct and identifies particular sequences of operations within the same partition, such as table filtering and projection, which SGLib might execute as a single operation.

FRAMEWORK EVALUATION AND COMPARISON

To evaluate GEMS, we ran it on the Olympus supercomputer at Pacific Northwest National Laboratory's Institutional Computing Program. Olympus is a cluster of 604 nodes interconnected through a QDR Infiniband switch with 648 ports. It has a theoretical peak of 4 Gbytes per second. Each Olympus node features two 2.1 GHz AMD Opteron 6272 processors and 64 Gbytes of DDR3 memory clocked at 1,600 MHz. Each socket hosts eight processor modules (two integer cores and one floating-point core per module) on two dies, for a total of 32 integer cores per node.

A module includes an L1 instruction cache of 64 Kbytes, two L1 data caches of 64 Kbytes, and a 2-Mbyte L2 cache. Each four-module die hosts a shared L3 cache of 8 Mbytes. Dies and processors communicate through HyperTransport.

Comparison parameters

The sidebar "RDF Database Design" describes alternative approaches to GEMS. Of these, YarcData's Urika version 6.04 is most similar, so we included a comparison in our evaluation. Like GEMS, Urika employs multithreading and global address space, but it implements these features in hardware. Both Urika and GEMS process the graph database in-memory.

Because Urika exploits a shared-memory abstraction, even when it executes on a single node, the database and queries can use the entire machine's memory. In contrast, GEMS scales in memory size as more cluster nodes are added.

Because our motivation centers on dataset size, we compared Urika and GEMS with equivalent available memory. Our Urika appliance has 4 Tbytes of shared memory, so we compared it with GEMS running on a cluster with up to 64 nodes, which collectively provide a global memory of 4 Tbytes.

We compared the systems executing queries from the SP2Bench SPARQL Performance Benchmark.[3] SP2Bench queries provide a complex environment for evaluating RDF engine performance. We were able to explore language expressiveness, covering various operator constellations combined with solution modifiers. The queries also describe different graph patterns, including long path chains, bushy patterns (single nodes with many successors), and combinations of these. Finally, they present different selectivity, dramatically varying memory requirements and time complexity, particularly when dataset size increases.

RDF DATABASE DESIGN

Solutions that try to map RDF databases on top of conventional relational SQL-based systems usually incur significant overhead from the performance penalties of feature-complete SPARQL-to-SQL translation. Translating SPARQL to SQL implies the use of relational algebra to perform optimizations and classical relational operators, such as joins and selects, to execute the query.

Many translation solutions incur overhead from intermediate data structures, which rely on Java-based front ends that generate more or less standardized representations and API calls. In contrast, GEMS translates SPARQL queries to graph pattern-matching operations and its full customization ensures efficient C++ code with calls and accesses to optimized APIs and data structures.

IN-MEMORY SOLUTIONS

Some solutions process queries in memory; others store data on disks and perform swapping. Apache Jena (https://jena.apache.org) and Sesame (http://rdf4j.org) provide libraries that natively implement in-memory RDF storage and support integration with some disk-based, SQL back ends. OpenLink's Virtuoso (http://virtuoso.openlinksw.com) implements an RDF/SPARQL layer on top of their SQL-based column store, providing support for multinode clusters.

GEMS stores all data structures in RAM, which enables a larger dataset, yet constant query throughput as nodes are added. Given the highly irregular data structures and algorithmic behavior of large graphs, in-memory processing is generally the approach of choice for processing them at the highest possible speed.

SCALABILITY ON CLUSTERS

Some RDF databases leverage MapReduce frameworks to achieve scalability on clusters. For example, Shard[1] is a native triple-store database built on top of Hadoop. Yet Another RDF Store 2 (YARS2)[2] is a bulk-synchronous, distributed, query answering system. Both exploit hash partitioning to distribute triples across nodes. Other frameworks map SPARQL queries to Pig,[3] abstracting operations over that language, which is then compiled into MapReduce jobs.

These approaches work well for simple index lookups, but they also present high communication overhead from moving data through the network with more complex queries, and they introduce load-balancing issues with skewed data. GEMS avoids this overhead by abstracting operations at a lower level using data structures and primitives from SGLib.

4store and its successor 5store[4] directly interface to low-level operations. 4store uses distinct processing and storage nodes. Although processing back ends can execute on the same node with storage back ends, they still incur TCP/IP communication overhead. Dataset segments are nonoverlapping and uniformly distributed, but segments with significant data skew can be replicated. Storage nodes do not communicate directly, and processing nodes always send a single request at a time to storage nodes. Because a storage node can host multiple segments, and each segment has a different connection, processing nodes can send a request to all the segments and 4store can aggregate all the replies.

Theoretically, 4store can scale up to 32 nodes, and its developers have used up to nine nodes.

Results

Tables 1 through 4 show execution latencies for the sample query in Figure 2a (Q0) and SP2Bench queries 1 through 10 with dataset sizes of 10 million, 100 million, and one billion triples. Table 1 shows execution time with Urika only; Tables 2 through 4 show the number of result tuples and execution time for GEMS with different numbers of nodes. Our execution time threshold was three hours.

On average, the C++ code generation process required 0.12 seconds, while the compilation of the generated implementations required 3.86 seconds. For low-latency queries, this cost is often higher than the actual execution time. However, we assume that ultimately compilation time will become almost negligible for high-latency and frequently used queries, since such queries require only one compilation.

To evaluate GEMS' scalability, we analyzed the execution latencies of all the queries while varying the number of nodes allocated

Developers of 5store, its commercial successor, project that it will reach thousands of nodes, although scaling appears to be focused on storage, rather than query throughput. In contrast, GEMS adopts a distinct, custom runtime to implement hardware-specific optimizations that enable both dataset and performance scaling.

EXPLOITING PARALLELISM

The recent upsurge in research to enhance the parallel processing of large graphs on high-performance computing systems has led to general graph libraries, such as Pregel, Giraph (http://incubator.apache.org/giraph), and GraphLab (http://graphlab.org). Once their source data is converted, these libraries have potential as building blocks for RDF databases, but they require many additions. Moreover, they rely on bulk-synchronous, parallel models that do not perform well for large and complex SPARQL queries composed of many graph walks. In contrast, GEMS builds on research to enhance parallel graph processing on HPC systems by providing specific features to support an RDF database.

YarcData's Urika[5] also builds on parallel graph processing research. Urika, a commercial shared-memory system targeted for big data analytics, employs a Cray XMT 2 machine, which has custom nodes with barrel processors with up to 128 threads and a very simple cache. Beside hardware multithreading, which allows tolerating latencies for accessing data on remote nodes, the system has hardware support for a scrambled global address space and fine-grained synchronization. These features allow more efficient execution of irregular algorithms, such as graph methods. On top of this hardware, YarcData interfaces with the Jena framework to provide a front-end API. In contrast, GEMS uses a runtime layer that implements features similar to the XMT 2 in software, and runs on clusters built with commodity components. Commodity clusters are cheaper to acquire and maintain, and evolve more rapidly than custom hardware. On the other end, we purposely built GEMS' SGLib and SPARL-to-C++ translator on top of the runtime layer rather than adapting them from existing solutions.

References

1. K. Rohloff and R.E. Schantz, "High-Performance, Massively Scalable Distributed Systems Using the MapReduce Software Framework: The Shard Triple-Store," *Programming Support Innovations for Emerging Distributed Applications*, 2010, pp. 4:1–4:5.
2. A. Harth et al., "YARS2: A Federated Repository for Querying Graph Structured Data from the Web," *Proc. 6th Int'l Semantic Web and 2nd Asian Semantic Web Conf.* (ISWC 07/ASWC 07), 2007, pp. 211–224.
3. S. Kotoulas et al., "Robust Runtime Optimization and Skew-Resistant Execution of Analytical SPARQL Queries on Pig," *Proc. 11th Int'l Semantic Web Conf.* (ISWC 12), 2012, pp. 247–262.
4. S. Harris, N. Lamb, and N. Shadbolt, "4store: The Design and Implementation of a Clustered RDF Store," *Proc. 5th Int'l Workshop Scalable Semantic Web Knowledge Base Systems* (SSWS 09), 2009, pp. 94–109.
5. YarcData, Inc., "Urika Big Data Graph Appliance," 2015; www.cray.com/products/analytics/urika-gd.

on the cluster, according to dataset size. For complex queries, characterized by higher latencies, the speedup was linear with the number of cluster nodes, which demonstrates the customized runtime layer's effectiveness.

For simple queries, performance was flat as the system scaled. In queries Q1, Q8, and Q10, higher communication costs slightly degraded performance. However, in all these cases, overall execution time was less than a second.

For most of the queries in Tables 2 through 4, GEMS provided valuable speedups. For some queries, both Urika and GEMS performance varied in interesting ways.

Queries Q0 and Q5a. When executing these queries, Urika quickly ran out of memory, even with the smallest dataset. GEMS successfully executed Q0 in a reasonable time, but could not complete an automatically translated Q5a within our time threshold.

Q5a features a simple BGP constrained by a FILTER condition. The BGP

TABLE 1. Time in seconds for Urika to execute the query in Figure 2a (Q0) and SP2Bench queries (Q1 through Q10) with 10 million, 100 million, and 1 billion triples.

Query	10M	100M	1B
Q0	†	†	†
Q1	1.297	1.244	1.105
Q2	4.224	9.909	64.221
Q3a	1.638	2.348	7.165
Q3b	1.095	1.193	1.594
Q3c	0.831	0.960	0.938
Q4	8.533	62.575	673.937
Q5a	†	†	†
Q5b	1.990	2.437	5.492
Q6	4.104	13.142	115.660
Q7	3.306	4.216	12.841
Q8	6.450	67.621	814.554
Q9	1.552	2.214	9.614
Q10	0.746	0.718	0.942

† memory exceeded

TABLE 2. Results tuples and time in seconds when GEMS executes the query in Figure 2a (Q0) and SP2Bench queries (Q1 through Q10) with 10 million triples and two, four, and eight nodes.

Query	No. of answers	Two nodes	Four nodes	Eight nodes
Q0	259,998	3.674926	2.289429	1.467329
Q1	1	0.016376	0.022226	0.026737
Q2	613,729	3.733764	3.106377	1.627502
Q3a	323,456	0.381339	0.324787	0.376205
Q3b	2,209	0.335273	0.325474	0.245333
Q3c	0	0.317341	0.312902	0.247462
Q4	40,087,273	113.29851	55.765167	377.994934
Q5a	+	‡	‡	‡
Q5a+	404,903	21.472024	16.537401	12.982555
Q5b	404,903	2.20076	1.470326	1.015784
Q6	852,649	7.174859	5.013053	4.233066
Q7	2,336	3.797751	3.646978	3.323373
Q8	493	226.715055	183.196768	145.081242
Q8+	493	0.119182	0.140208	0.169048
Q9	4	3.807123	2.849280	1.670042
Q10	656	0.010789	0.016690	0.018765

‡ timeout reached; + manual optimization of translated C++ code

TABLE 3. Results tuples and time in seconds when GEMS executes the query in Figure 2a (Q0) and SP2Bench queries (Q1 through Q10) with 100 million triples and 8, 16, and 32 nodes.

Query	No. of answers	Eight nodes	16 nodes	32 nodes
Q0	867,868	7.025832	3.493988	1.775647
Q1	1	0.041053	0.051353	0.127594
Q2	9,050,604	11.103515	7.191075	5.063025
Q3a	1,466402	0.159287	0.108663	0.102876
Q3b	10,143	0.141681	0.094862	0.099148
Q3c	0	0.139884	0.089665	0.100040
Q4	460,135,248	244.082552	149.840050	64.886178
Q5a	+	‡	‡	‡
Q5a+	2,016,996	102.614320	60.674561	34.977476
Q5b	2,016,996	3.492491	1.993362	1.311534
Q6	9,812,030	24.896199	14.595010	10.430950
Q7	14,645	13.882659	10.363371	8.919411
Q8	493	1,177.359682	747.210641	473.620011
Q8+	493	0.149563	0.216421	0.246394
Q9	4	12.250798	6.472046	3.496389
Q10	656	0.019665	0.033179	0.052079

‡ timeout reached; + manual optimization of translated C++ code

TABLE 4. Results tuples and time in seconds when GEMS executes the query in Figure 2a (Q0) and SP2Bench queries (Q1 through Q10) with one billion triples and 32 and 64 nodes.

Query	No. of answers	32 nodes	64 nodes
Q0	4,845,712	20.403551	8.627155
Q1	1	0.052522	0.088859
Q2	95,481,441	31.522307	18.305085
Q3a	10,157,400	0.308892	0.219484
Q3b	70,613	0.291832	0.210325
Q3c	0	0.283186	0.200547
Q4	4,509,421,916	671.622605	348.719609
Q5a	+	‡	‡
Q5a+	9,197,350	341.864665	182.436236
Q5b	9,197,350	5.692948	3.746873
Q6	120,405,337	75.462789	57.318893
Q7	20,441	286.427132	250.132178
Q8	493	5,604.322000	4,510.380000
Q8+	493	0.229329	0.302651
Q9	4	35.869012	18.887735
Q10	656	0.048982	0.086379

‡ timeout reached; + manual optimization of translated C++ code

is composed of two disjoint patterns, and FILTER condition evaluation is possible only at the last levels of the graph walk. The result is a high number of partial pattern matches. We manually modified Q5a, splitting the original graph walk in two distinct ones, corresponding to the two disjoint patterns, and evaluated the FILTER condition afterward. The modified Q5a completed execution for all the datasets in reasonable time. We are currently improving the compiler to automatically apply these optimizations. Nevertheless, this evaluation demonstrates the value of allowing the user to tune or modify the generated code.

Queries Q2, Q4, and Q6. Both Urika and GEMS could execute these queries, but GEMS outperformed Urika in most cases, particularly with larger datasets. All these queries produce a high number of result tuples and partial matches.

GEMS efficiently executed these queries because, unlike relational approaches, it does not generate temporary data structures during a graph walk. This feature not only significantly reduces the time complexity of GWOs, but also allows dataset scaling.

Query Q7. When targeting datasets with 10 million and 100 million triples, Urika and GEMS performed similarly. However, when the dataset increased to a billion triples, Urika performed significantly better.

Query 7 tests mainly nested, closed-world negation, and offers several opportunities for optimization that are based on reusing graph pattern results.[3] GEMS does not exploit optimizations of this type.

Query Q8. When both systems generated Q8 automatically, Urika's

ABOUT THE AUTHORS

VITO GIOVANNI CASTELLANA is a research scientist in the High Performance Computing Group at Pacific Northwest National Laboratory (PNNL). His research interests include embedded system design and electronic design automation, code transformation, compilation, and optimization. Castellana received a PhD in computer science and engineering from Politecnico di Milano, Italy. He is a member of IEEE. Contact him at vitogiovanni.castellana@pnnl.gov.

ALESSANDRO MORARI is a research scientist in the High Performance Computing Group at PNNL. His research interests are big data analytics, large-scale runtime systems, system software for high-performance computing, and performance modeling. Morari received a PhD in computer architecture from Universidad Politecnica de Catalunya in Spain. He is a member of ACM and IEEE. Contact him at alessandro.morari@pnnl.gov.

JESSE WEAVER was a research computer scientist in the Data Sciences Group at PNNL while performing the work reported in this article. His research interests include distributed graph and RDF databases, parallel reasoning systems, the Semantic Web, and linked data. Weaver received a PhD in computer science from Rensselaer Polytechnic Institute. Contact him at jrweaver@gmail.com.

ANTONINO TUMEO is a senior research scientist in the High Performance Computing Group at PNNL. His research interests include simulation and modeling of high-performance and embedded computer architectures, hardware and software coding, FPGA prototyping, and general-purpose computing on GPUs. Tumeo received a PhD in computer science and engineering from Politecnico di Milano. He is a member of IEEE and ACM. Contact him at antonino.tumeo@pnnl.gov.

DAVID HAGLIN is a senior research scientist in the Data Intensive Scientific Computing Group at PNNL. His research interests include data mining, big data, and graph algorithms. Haglin received a PhD in computer and information sciences from the University of Minnesota. He is a senior member of IEEE. Contact him at david.haglin@pnnl.gov.

ORESTE VILLA is a senior research scientist in the Architecture Group at NVIDIA Research. His research interests include computer architecture and simulation, irregular applications, and accelerators for scientific computing. Villa received a PhD in computer science and engineering from Politecnico di Milano. Contact him at ovilla@nvidia.com.

JOHN FEO is vice president of engineering at Context Relevant. His research interests include parallel computing, parallel application development, graph databases, functional languages, and performance studies. Feo received a PhD in computer science from the University of Texas at Austin. He is a member of ACM. Contact him at jfeo@contextrelevant.com.

performance was significantly better than GEMS'. Q8 presents the UNION of two partially overlapping patterns, characterized by triples with different selectivity. We modified the generated code, merging the matching of the two patterns in a single graph walk, and started the graph walk by matching the most selective triple pattern.

Exploiting selectivity had a tremendous impact on performance: our tuned query ran thousands of times faster than with Urika, and speed gains were even greater relative to GEMS' initial implementation.

Currently, GEMS' code generator cannot estimate a single triple pattern's selectivity. We are improving optimization by considering statistics collected through data probing, such as the number of graph vertices with a specific label. Exploiting this information should significantly improve performance for queries characterized by triple patterns with vastly different selectivity, such as Q8.

Our evaluation shows that, by targeting in-memory graph processing in all levels of the stack, GEMS can scale in performance and size as nodes are added to a commodity cluster. To further enhance performance, we are improving the query planner to account for data characteristics sets, which should result in more efficient C++ implementations of graph pattern-matching operations.

We believe that GEMS and its components, with their clean-slate and top-to-bottom design, represent a significant step toward providing more efficient and scalable RDF databases that can address the volumes of Web data and tame the "too big" aspects of big data. C

REFERENCES

1. S. Madden, "From Databases to Big Data." *IEEE Internet Computing*, vol. 16, no. 3, 2012, pp. 4–6.
2. A. Morari et al., "Scaling Irregular Applications through Data Aggregation and Software Multithreading," *Proc. IEEE 28th Int'l Parallel and Distributed Processing Symp.* (IPDPS 14), 2014, pp. 1126–1135.
3. J.R. Ullmann, "An Algorithm for Subgraph Isomorphism," *J. ACM*, vol. 23, no. 1, 1976, pp. 31–42.
4. M. Schmidt et al., "Sp2Bench: A SPARQL Performance Benchmark," *Semantic Web Information Management*, R. de Virgilio, F. Giunchiglia, and L. Tanca, eds., Springer, 2010, pp. 371–393.