DataStax Developer Blog

« PHP Driver 1.0 GA

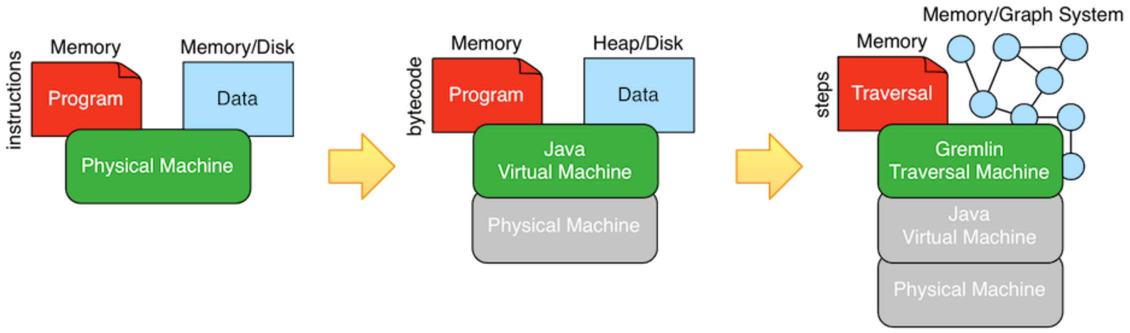
Exciting Changes to KillrVideo Sample Application and Website »

The Benefits of the Gremlin Graph Traversal Machine

BY MARKO A. RODRIGUEZ - SEPTEMBER 14, 2015 | COMMENTS OFF ON THE BENEFITS OF THE GREMLIN GRAPH TRAVERSAL MACHINE

A computer is a machine that evolves its state (data) according to a set of rules (program). Simple "one off" computers have their programs hardcoded. Programmable computers have an instruction set whereby parameterized instructions can be arbitrarily composed to yield different algorithms (different instruction sequences). If a programmable computer has a sufficiently complex instruction set, then not only can it simulate any "one off" computer, but it can also simulate any programmable computer. Such general-purpose computers are called universal. An example of a popular universal computer is the machine being used to read this blog post. Virtual machines emerged to enable the same program to run on different operating/hardware platforms (i.e. different machines). Popular virtual machines, such as the Java Virtual Machine, are universal in that they can be used to simulate other universal machines, including themselves. In the domain of graph computing, the computer is a *graph traversal machine*. The data of this machine is a graph composed of vertices (dots, nodes) and edges (lines, links). The instruction set is called the machine's *step library*. Steps are composed to form a program called a traversal. An example universal graph traversal machine is the Gremlin traversal machine.

Physical machine	Java virtual machine	Gremlin traversal machine
instructions	bytecode	steps
program	program	traversal
program counter	program counter	traverser
memory	heap	graph

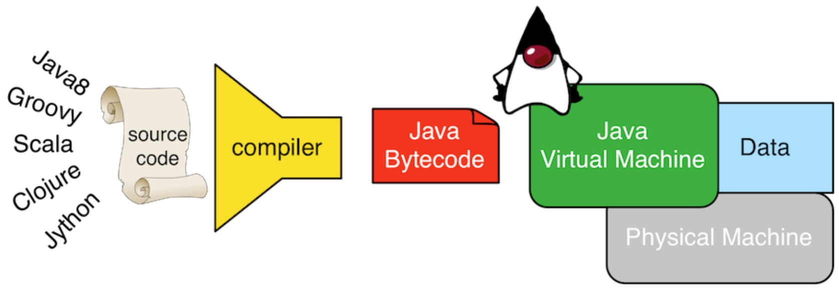


This blog post will review the benefits of Apache TinkerPop's Gremlin graph traversal machine for both *graph language designers* and *graph system vendors*. A graph language designer develops a language specification (e.g. SPARQL, GraphQL, Cypher, Gremlin) and respective compiler for its evaluation over some graph system. A graph system vendor develops an OTLP graph database (e.g. Titan, Neo4j, OrientDB) and/or an OLAP graph processor (e.g. Titan/Hadoop, Giraph, Hama) for storing and processing graphs. The benefits of the Gremlin traversal machine to these stakeholders are enumerated below and discussed in depth in their respective sections following this prolegomenon.

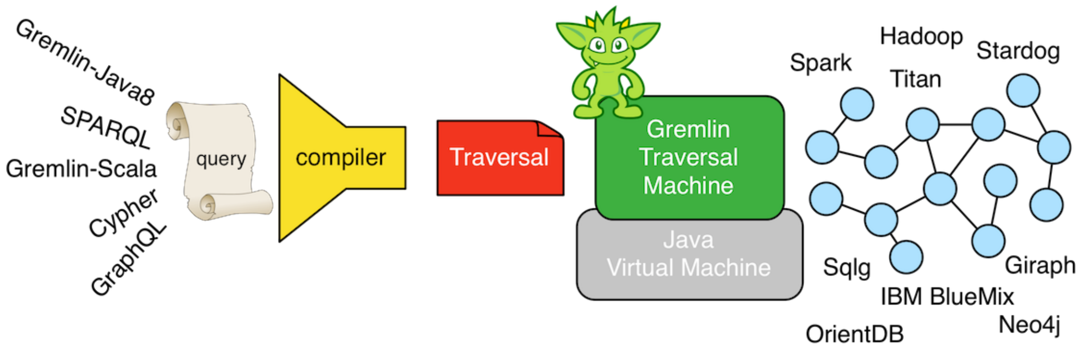
1. **Language/system agnosticism**: A graph language designer need only create a Gremlin traversal compiler for their language to execute over any graph system that supports the Gremlin traversal machine. Likewise, a graph system vendor that supports the Gremlin traversal machine supports any language that compiles to a Gremlin traversal. [Section 1]
2. **Provided traversal engine**: A graph language designer need only concern themselves with writing a Gremlin traversal compiler as they can rely on the Gremlin machine to handle traversal optimization and execution. Similarly, a graph system vendor only has to implement a few core abstractions to support the Gremlin traversal machine and can further advanced it with vendor-specific optimizations. [Section 2]
3. **Native distributed execution**: A graph language designer need not concern themselves with the sequential, parallel, or distributed execution of their language as the Gremlin traversal machine can operate on a single machine or a multi-machine compute cluster. Additionally, if a graph system vendor implements the `GraphComputer` API, then Gremlin traversals can execute over their multi-machine partitioned graphs. [Section 3]
4. **Open source and Apache2 licensed**: A graph language designer can contribute to the evolution of the Gremlin traversal machine. In turn, graph system vendors can also help advance Gremlin as Apache TinkerPop is part of the Apache Software Foundation and thus, is open source and free to use. [Section 4]

Language/System Agnosticism: "Many Graph Languages for Many Graph Systems"

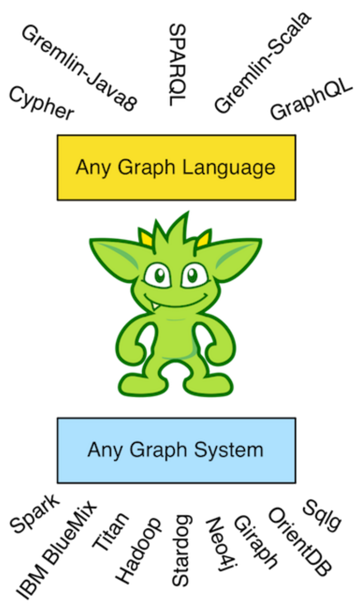
Java is both a virtual machine and a programming language. The Java virtual machine (JVM) is a software representation of a programmable machine. The benefit of the JVM is that its software (a Java program) can be executed on any JVM-enabled operating system without requiring it to be rewritten/recompiled to the instruction set of the underlying hardware machine. This feature is espoused by Java's popular slogan "write once, run anywhere." The Java programming language is a human writable language that when compiled by the Java compiler (`javac`), a sequence of instructions from the JVM's instruction set is generated called Java bytecode.



Gremlin, like Java, is both a virtual machine and a programming language (or query language). The Gremlin traversal machine maintains a step library (i.e. instruction set) whose steps can be arbitrarily composed to enact *any computable* graph traversal algorithm. In other words, the Gremlin traversal machine is a programmable, universal machine (see *The Gremlin Graph Traversal Machine and Language*). Moreover, the Gremlin traversal machine can be supported by any graph system vendor such that Gremlin expresses the same "write once, run anywhere"-mantra. The Gremlin traversal language (aka Gremlin-Java8) is a human writable language that when compiled, a traversal is generated that can be executed by the Gremlin traversal machine.



There is a noteworthy consequence of the virtual machine/programing language distinction. The Java virtual machine does not require the Java programming language. That is to say, any other programming language can have a compiler that generates Java bytecode (JVM instructions). It is this separation that enables other JVM languages to exist. Examples include Groovy, Scala, Clojure, JavaScript (Rhino), etc. Analogously, the Gremlin traversal machine does not require the Gremlin traversal language (Gremlin-Java8). Any other graph language can be compiled to a Gremlin traversal.[1] It is because of this separation that other graph traversal languages exist such as Gremlin-Groovy, Gremlin-Scala, and Gremlin-JavaScript. It is arguable that these Gremlin variants simply leverage the fluent interface of the Gremlin(-Java8) language within the programming constructs of their respective host language. Regardless, nothing prevents any other graph language from being executed by a Gremlin traversal machine such as SPARQL, GraphQL, Cypher, and the like.[2] How is this possible? — the Gremlin traversal machine is universal and maintains an extensive step library of the common query motifs found in most every graph language.[3]

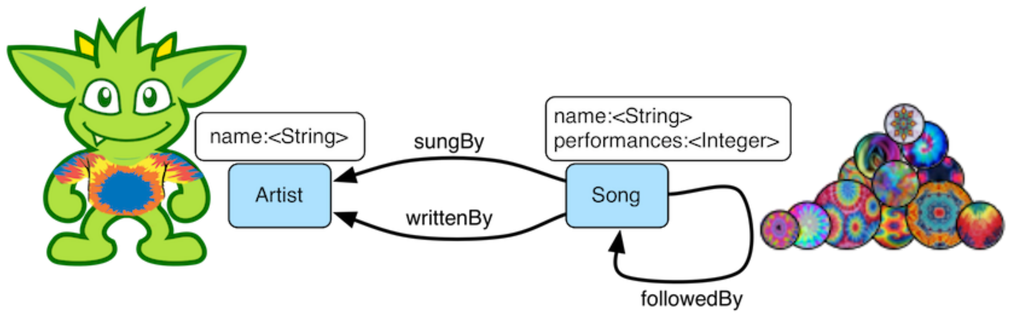


If there exists a compiler that translates language X to Gremlin traversals and graph system Y supports the Gremlin traversal machine, then graph language X can execute against graph system Y.

Provided Traversal Engine: “SPARQL on the Gremlin Traversal Machine”

SPARQL is a graph query language used extensively in the RDF/SemanticWeb community and supported by every RDF graph database vendor. SPARQL is not as popular in the property graph database space because the underlying data model is different. The primary mismatch is that property graph edges can have an arbitrary number of key/value pairs (called properties). However, this difference is minor when compared to their numerous similarities. Capitalizing on their alignment, it is possible to compile SPARQL to a Gremlin traversal and thus, have a SPARQL query execute on any graph database/processor that supports the Gremlin traversal machine.

The examples to follow throughout the remainder of this post make use of the Grateful Dead graph distributed by Apache TinkerPop. The Grateful Dead graph is composed of song and artist vertices and respective interrelating edges. The graph schema is diagrammed below.

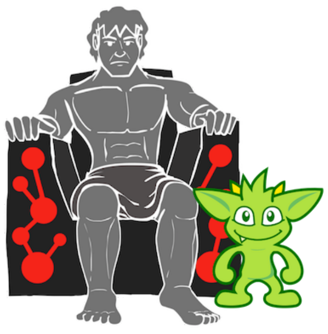


A question that can be asked of this graph is:

Which song writers wrote songs that were sung by Jerry Garcia and performed by the Grateful Dead more than 300 times?

This query is expressed in Gremlin-Java8 as:

```
g.V().match(
  as("song").out("sungBy").as("artist"),
  as("song").out("writtenBy").as("writer"),
  as("artist").has("name","Garcia"),
    where(as("song").values("performances").is(gt(300)))
).select("song","writer").by("name")
```



It is possible to express this same query in SPARQL. However, in order for it to execute against a Gremlin-enabled graph system, it must be compiled to a Gremlin traversal. SPARQL-Gremlin was developed for this purpose.[4] Apache Jena provides programmer access to their SPARQL compiler called ARQ. The Jena SPARQL compiler builds a syntax tree that can be walked and as the walk proceeds, Gremlin steps are concatenated and nested. The Gremlin-Java8 query above is represented in SPARQL below, where `e:` means “edge label” and `v:` means “property value.” The SPARQL query can be executed via a Java application or via the Gremlin-Console (REPL). A Gremlin-Console session using SPARQL-Gremlin over Titan (version 1.0) is provided below.[5]

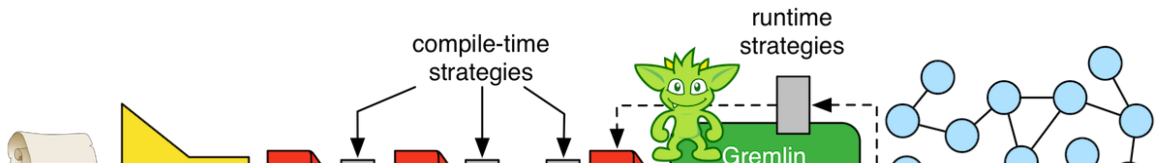
```
SELECT ?songName ?writerName WHERE {
  ?song e:sungBy ?artist .
  ?song e:writtenBy ?writer .
  ?song v:name ?songName .
  ?writer v:name ?writerName .
  ?artist v:name "Garcia" .
  ?song v:performances ?performances .
    FILTER (?performances > 300)
}
```

```
$ bin/gremlin.sh

         \,,,/
         (o o)
-----oOOo-(3)-oOOo-----
plugin activated: aurelius.titan
plugin activated: tinkerpop.server
plugin activated: tinkerpop.utilities
plugin activated: tinkerpop.hadoop
plugin activated: tinkerpop.tinkergraph
gremlin> :install com.datastax sparql-gremlin 0.1
==>Loaded: [com.datastax, sparql-gremlin, 0.1]
gremlin> :plugin use datastax.sparql
==>datastax.sparql activated
gremlin> graph = TitanFactory.open('conf/titan-cassandra.properties')
==>standardtitangraph[cassandrathrift:[127.0.0.1]]
gremlin> :remote connect datastax.sparql graph
gremlin> query = """
  SELECT ?songName ?writerName WHERE {
    ?song e:sungBy ?artist .
    ?song e:writtenBy ?writer .
    ?song v:name ?songName .
    ?writer v:name ?writerName .
    ?artist v:name "Garcia" .
    ?song v:performances ?performances .
      FILTER (?performances > 300)
  }
"""
gremlin> :> @query
==>[songName:BERTHA, writerName:Hunter]
==>[songName:TENNESSEE JED, writerName:Hunter]
==>[songName:BROWN EYED WOMEN, writerName:Hunter]
==>[songName:CHINA CAT SUNFLOWER, writerName:Hunter]
==>[songName:CASEY JONES, writerName:Hunter]
==>[songName:BLACK PETER, writerName:Hunter]
==>[songName:RAMBLE ON ROSE, writerName:Hunter]
==>[songName:WHARF RAT, writerName:Hunter]
==>[songName:LADY WITH A FAN, writerName:Hunter]
==>[songName:HES GONE, writerName:Hunter]
==>[songName:LOSER, writerName:Hunter]
==>[songName:DEAL, writerName:Hunter]
==>[songName:SUGAREE, writerName:Hunter]
==>[songName:DONT EASE ME IN, writerName:Traditional]
==>[songName:UNCLE JOHNS BAND, writerName:Hunter]
==>[songName:SCARLET BEGONIAS, writerName:Hunter]
==>[songName:EYES OF THE WORLD, writerName:Hunter]
==>[songName:US BLUES, writerName:Hunter]
==>[songName:TERRAPIN STATION, writerName:Hunter]
==>[songName:STELLA BLUE, writerName:Hunter]
gremlin>
```

The generated Gremlin traversal “bytecode” (steps) is provided below. Unlike hardware machine code (or Java bytecode), a traversal is not just a linear concatenation of steps. It is possible for the parameters of a step to be a traversal (called an anonymous traversal). Step concatenation and traversal nesting yield the global structure of a Gremlin traversal.

```
[GraphStep([],vertex),
  MatchStep(AND,[
    [MatchStartStep(song), VertexStep(OUT,[sungBy],vertex), MatchEndStep(artist)],
    [MatchStartStep(song), VertexStep(OUT,[writtenBy],vertex), MatchEndStep(writer)],
    [MatchStartStep(song), PropertiesStep([name],value), MatchEndStep(songName)],
    [MatchStartStep(writer), PropertiesStep([name],value), MatchEndStep(writerName)],
    [MatchStartStep(artist), PropertiesStep([name],value), IsStep(eq(Garcia)), MatchEnd
    [MatchStartStep(song), PropertiesStep([performances],value), MatchEndStep(performan
  WhereTraversalStep([WhereStartStep(performances), IsStep(gt(300))]), SelectStep([song
```



Once the query is compiled to a Gremlin traversal, then the Gremlin machine can optimize it both prior to and during its execution. There are two types of *traversal strategies*: compile-time and runtime. *Compile-time strategies* analyze the traversal and rewrite particular suboptimal step sequences into a more efficient form. For instance, the step sequence

```
...out().count().is(gt(10))...
```

Is re-written to the more efficient form

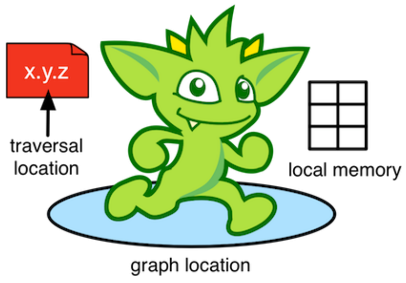
```
...outE().limit(11).count().is(gt(10))...
```

by means of the two compile-time strategies `AdjacentToIncidentStrategy` and `RangeByIsCountStrategy`. In the diagram above, traversal T_1 is transformed via a sequence of strategies that ultimately yield traversal T_n. Note that T_1, T_2, T_n, etc., if executed as such, would all return the same result set. In essence, traversal strategies do not effect the semantics of the query, only its execution plan. Next, if a graph system has a particular feature such as indexing, the vendor can provide custom strategies to the compilation process. Finally, `MatchStep` makes use of a *runtime strategy* called `CountMatchAlgorithm` to dynamically re-order pattern execution based on runtime set reduction statistics (amongst other analyses). In short, the larger the set reduction a particular pattern performs over time, the higher priority it has in the execution plan.

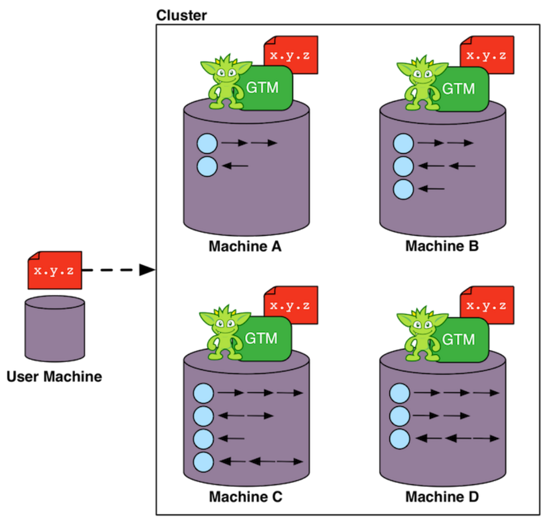
The Gremlin traversal machine will apply both compile-time and runtime optimizations (called traversal strategies) to any submitted traversal.

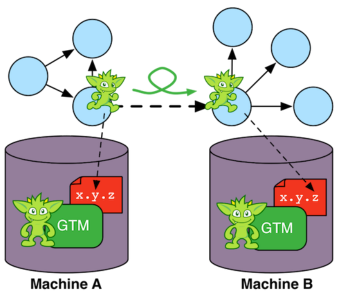
Native Distributed Execution: “A Gremlin Traversal over an OLAP Graph Processor”

The Gremlin traversal machine can not only execute a traversal compiled from any graph traversal language, but it also can execute *the same traversal* on a single machine or across a multi-machine compute cluster. There is no direct correlate to this on the Java virtual machine. If there were, it would be equivalent to saying that a distributed Java virtual machine provides a single address space across the the entire cluster and can execute the same JVM code regardless of how many physical machines are involved. Perhaps the closest analogy is BigMemory by Terracotta, though the graph data in distributed Gremlin can exist in both memory and on disk.



Gremlin executes a traversal using *traversers*. Traversers maintain a reference to their location in the graph (data reference), to their location in the traversal (program counter), and to a local memory data structure (registers). When a traverser is confronted with a decision in the graph structure (e.g. multiple incident `followedBy`-edges), the traverser clones itself and takes each path. In even the simplest traversals, it is possible for billions of such traversers to be spawned due to recursion (`repeat()`-step) and the complex branching structure found in natural graphs (see *Loopy Lattices Redux*). Fortunately, a "bulking" optimization exists to limit traverser enumeration. This optimization is extremely important in distributed Gremlin where parallel execution can easily lead to a large memory footprint. The distributed traversal is complete when all existing traversers have *halted* (i.e. no more steps to execute). The result of the query is the aggregate of the locations of all halted traversers.





For distributed OLAP graph processors such as Apache Giraph (via Giraph-Gremlin), the graph data set is partitioned across the machines in the cluster (top left figure). Each machine is responsible for a subset of the vertices in the graph. Each vertex has direct reference to its properties, its incoming and outgoing edges, as well as the properties of those incident edges. This atomic data structure is called a "star graph." The local edges of a vertex reference other vertices by their globally unique id. When a Gremlin traversal is submitted to the cluster it is copied to all machines. Each machine spawns a traverser at each vertex it maintains with the traverser's initial step being the first step in the traversal. When a traverser traverses an edge that leads to a vertex that is stored on another machine in the cluster, the traverser serializes itself and sends itself to the machine containing its referenced vertex (top right figure). Serialization is expensive so a good graph partition is desirable to reduce the number of "machine hops" the traversers have to take to solve the traversal.

In the Gremlin-Console session below, the previous SPARQL query is executed in a distributed manner using Apache Spark (via Spark-Gremlin). The data accessed by Spark is pulled from Titan (`CassandraInputFormat`). Titan is a distributed graph system that supports both OTLP graph database operations as well as OLAP graph processor operations.

```
gremlin> graph = GraphFactory.open('conf/titan-cassandra-hadoop.properties')
==>hadoopgraph[cassandrainputformat->gryooutputformat]
gremlin> g = graph.traversal(computer(SparkGraphComputer))
==>graphtraversalsource[hadoopgraph[cassandrainputformat->gryooutputformat], sparkgraph
gremlin> :remote connect datastax.sparql g
==>SPARQL[graphtraversalsource[hadoopgraph[cassandrainputformat->gryooutputformat], spa
gremlin> query = """
  SELECT ?songName ?writerName WHERE {
    ?song e:sungBy ?artist .
    ?song e:writtenBy ?writer .
    ?song v:name ?songName .
    ?writer v:name ?writerName .
    ?artist v:name "Garcia" .
    ?song v:performances ?performances .
```

```
      FILTER (?performances > 300)
  }
"""
gremlin> :> @query
[Stage 0:===>                                                  ]
[Stage 0:========>                                             ]
[Stage 0:==================>                                   ]
...
==>[songName:DONT EASE ME IN, writerName:Traditional]
==>[songName:EYES OF THE WORLD, writerName:Hunter]
==>[songName:BLACK PETER, writerName:Hunter]
==>[songName:HES GONE, writerName:Hunter]
==>[songName:STELLA BLUE, writerName:Hunter]
==>[songName:BERTHA, writerName:Hunter]
==>[songName:WHARF RAT, writerName:Hunter]
==>[songName:DEAL, writerName:Hunter]
==>[songName:TERRAPIN STATION, writerName:Hunter]
==>[songName:BROWN EYED WOMEN, writerName:Hunter]
==>[songName:SUGAREE, writerName:Hunter]
==>[songName:TENNESSEE JED, writerName:Hunter]
==>[songName:LADY WITH A FAN, writerName:Hunter]
==>[songName:LOSER, writerName:Hunter]
==>[songName:SCARLET BEGONIAS, writerName:Hunter]
==>[songName:RAMBLE ON ROSE, writerName:Hunter]
==>[songName:CASEY JONES, writerName:Hunter]
==>[songName:CHINA CAT SUNFLOWER, writerName:Hunter]
==>[songName:UNCLE JOHNS BAND, writerName:Hunter]
==>[songName:US BLUES, writerName:Hunter]
gremlin>
```

Any Gremlin traversal compiled from any query language can be executed by the Gremlin traversal machine on a single-machine or a multi-machine compute cluster.

Open Source and Apache2 Licensed: “Who/Where/When/What is The TinkerPop?”

Gremlin is designed and developed by Apache TinkerPop. TinkerPop is licensed Apache2 and thus, the software is free to use for any purpose commercial or otherwise. Apache TinkerPop is also open source with an active development community. There are four general types of stakeholders in the community.

- **TinkerPop committers**: The core developers of TinkerPop. This core designs, develops, and documents Gremlin. Furthermore, they communicate with the other stakeholders to ensure that the Gremlin machine architecture and language evolve accordingly.
- **Graph system vendors**: The commercial or open-source providers of graph system technology. These vendors ensure that their products are “TinkerPop-enabled” (i.e. have support for the Gremlin traversal machine). Furthermore, these vendors develop custom strategies to further optimize Gremlin traversals for their particular system.
- **Graph language designers**: The creators of graph traversal languages with compilers for the Gremlin traversal machine. The Gremlin-XYZ language line is embedded in popular programming languages such that the developer’s database query and application language are one in the same. With direct access to the Gremlin traversal machine, other languages that are not embedded in a host language (e.g. SPARQL, Cypher, GraphQL) can now be provided.
- **TinkerPop users**: The everyday users of the graph technology developed and distributed by the previous stakeholders. With TinkerPop being vendor agnostic, users are not locked into a particular system/language and thus, can explore the unique advantages that each system and language provides.

Apache TinkerPop is an open source project that is open to contributions of all kind and is free to use for any purpose commercial or otherwise.

Conclusion

Gremlin is both a graph traversal machine and graph traversal language. Any graph language can be compiled to Gremlin and evaluated against any of the numerous TinkerPop-enabled graph systems in existence today. Previous to this exposition, Gremlin had been presented as solely being agnostic to the underlying graph system. However, Gremlin is also agnostic to the user facing query language that reads and writes data to and from the underlying graph system. Simply put, **TinkerPop enables graph developers to use any query language with any graph system**.

Acknowledgements

This blog post was written by Marko A. Rodriguez under the inspiration of *The Gremlin Graph Traversal Machine and Language* article. Daniel Kuppitz developed SPARQL-Gremlin as a proof-of-concept demonstrating that any arbitrary graph language can be compiled to Gremlin. Matthias Bröcheler, upon reviewing this article, stated: “It would be sweet if somebody wrote SQL-Gremlin.” A quick Google search revealed JSQLParser which has an analogous architecture to the ARQ parser used by SPARQL-Gremlin. The Gremlins in the title logo are (from left to right) Grem Stefani, The Grem Reaper, Gremicide, Gremlin the Grouch, Ain’t No Thing But a Chicken Wing, Gremlivich, Gremopoly, Gremalicious, and Clownin’ Around. The contributors would like to thank the Apache Software Foundation for their continued support of TinkerPop and DataStax for sponsoring this research effort.

Thank you and good night.

Footnotes

1. Apache TinkerPop’s Gremlin traversal machine is written in Java and intended to be used by JVM-based graph systems. While most graph systems are JVM-based, there are some that aren’t. For those that aren’t, TinkerPop’s Gremlin can be leveraged via the Java Native Interface. However, the Gremlin traversal machine and language specifications are quite simple and thus, can be implemented in the native language of the underlying graph system wishing to capitalize on the benefits of the Gremlin traversal machine and language. ↩

2. Note that compilers for GraphQL and Cypher currently do not exist. The purpose of this discussion is to state that it is feasible for someone to write a Gremlin compiler for these languages. In this way, if a particular graph query language is more appreciated by the user, they can leverage it for whichever Gremlin-enabled graph system they wish. ↩

3. If the underlying graph system already has support for a particular query language, then it *may* be pointless to do a compilation to the Gremlin traversal machine. For instance, Cypher is the primary language of Neo4j and SPARQL is the primary language of Stardog. The designers of these graph systems focus on ensuring that the queries of their respective languages execute as fast as possible on their respective systems. However, the reason it "*may be pointless*" is because, like C++ vs. Java, if the optimizers of the Gremlin traversal machine are advanced enough, then in theory, a compilation of these languages to the Gremlin traversal machine may be faster than the respective supported language engine. ↩

4. SPARQL-Gremlin version 0.1 (September 17, 2015) is a prototype demonstrating the primary aspects of SPARQL being compiled to Gremlin. SPARQL-Gremlin version 0.1 only supports `SELECT`, `WHERE`, `FILTER`, and `DISTINCT`. Expanding its capabilities is a function of adding more translations to the compiler (e.g. `ORDER`, `GROUP BY`, etc.), where pull-requests are more than welcome. ↩

5. The Gremlin-Console is a thin wrapper around the Groovy Shell and thus, can only interpret Gremlin-Groovy (a superset of Gremlin-Java). As such, the SPARQL query is represented as a `String` and passed to the SPARQL-Gremlin compiler via the `datastax.sparql` Gremlin Plugin. ↩

« **PHP Driver 1.0 GA**

Exciting Changes to KillrVideo Sample Application and Website »