G* Studio: An Adventure in Graph Databases, Distributed Systems, and Software Development

By Alan Labouseur, Shane Crumlish, Cassandra Graves, Melissa J. Iori, Gregory Miller, and Thomas J. Wojnar,
Marist College

The e-mail from the department chair was urgent. There were several graduate students with no classes to take. "Would somebody please run an independent study?" she asked. The semester was already a few days old. Alan had to strike fast. "I'm in," he wrote, "I'll put them to work on my graph database research." With that, Alan and his new team, which would become known as the G-stars, began a two-semester adventure in graph databases, distributed systems, and software development that resulted in more than 8,000 lines of code over 520 Git commits. This is their story.

Some key contributions of this story include the following:

- The authors discuss a large, independent study software development project in the new and original context of graph processing and analytics.
- They tell the story of successful full stack development using, at the time of this writing, cutting-edge hybrid application development tools.
- The authors show an example of large-scale application development that educators can use to send a message to their own students: "It can be done. Here's what they did, why they did it, and how they accomplished it."
- They note some current best practices for bleeding-edge software architecture in a modern, blended software development environment, using many of the tools commonly found in industry today, at the same time documenting a practical experience that's day-one applicable in our students' post-college careers. If we are to balance theory and practice, projects like this make for great practice.

BACKGROUND

THE PROBLEM

We are awash in a daily deluge of data, much of which looks like a network or which a network can model. These networks are constantly growing with ever more products, services, messages, and transactions. They are constantly changing with connections added, removed, and modified all the time. In domains as diverse as marketing, transportation, pharmacology, communication, finance and others, real-world networks tend to be large (*big* data) and dynamic, evolving over time (*long* data). Piling up data is easy. Gaining insight from the data pile is hard.

Q: How do we gain insight from large, evolving networks?
A: Treat them like dynamic graphs.

Modeling network evolution as a series of graphs—whose vertices represent entities and edges represent relationships between entities—allows us to capture network evolution as a series of graph snapshots, each representing the network at a different point in time.

Most of today's graph systems store and analyze only one graph at a time, unable to handle efficiently the complexity and subtlety inherent in dynamic graphs. Even the most modern of relational database systems (including in-memory systems) cannot effectively support analytics on large, evolving graphs because of the $O(n^2)$ nature of self-joins. Yet modern analytics on real-world data requires systems capable of storing and processing large series of graph snapshots. What can we do?

```
Welcome to G*.
gstar.Master is running.
gstar.Master is waiting to be contacted by 3 Workers.
gstar.Master registered Worker 0.
gstar.Master registered Worker 1.
gstar.Master registered Worker 2.
gstar.Master registered 3 Workers.
gstar.Worker(0) has started its graph manager and query engine.
gstar.Worker(1) has started its graph manager and query engine.
gstar.Worker(2) has started its graph manager and query engine.

Welcome to G* version 1.0.0 Terminal version 0.2
$ cd default
moved to /default

$ ls
0.0   # graph containing 2 vertices and 1 edges
1.0   # graph containing 4 vertices and 2 edges

$ operator vertex@* = VertexOperator([], [0,1]);
$ operator degree@* = ProjectionOperator([vertex@local], [id, (cardinality(outgoing_edges) +
cardinality(incoming_edges)), graph.id], [vid, total_degree, graph.id]);
$ operator union@0 = UnionOperator([degree@*]);
run union@0
{vid=a, total_degree=1, graph.id=[0.0]}
{vid=b, total_degree=1, graph.id=[0.0]}
{vid=a, total_degree=2, graph.id=[1.0]}
{vid=b, total_degree=1, graph.id=[1.0]}
{vid=c, total_degree=1, graph.id=[1.0]}
{vid=d, total_degree=0, graph.id=[1.0]}
```

Figure 1: G* Terminal

A PARTIAL SOLUTION

G* (pronounced "JEE-star") is a distributed system for storing and processing series of graph snapshots [8]. G* compresses dynamic graph data based on commonalities among snapshots, providing deduplicated storage across multiple workers to save space. G* supports multiple cores for scale up and multiple servers for scale out. In this manner, I/O bound analytic tasks benefit from parallel reads and writes among workers. G* executes analytic queries on large graphs using distributed operators for parallel processing. It speeds up these queries by processing graph commonalities only once and sharing results across relevant graphs and workers. This architecture not only provides scalability, but since G* is not limited to processing only what is available in RAM, its analytic capabilities exceed those of systems constrained by what they can hold in memory.

The G* engine works quite nicely [8,9] but, when we started this project, it was only accessible to end users through a text-based command-line interface uninspiringly called "Terminal." (See Figure 1.) We needed something better.

OUR CHALLENGE: THE REST OF THE SOLUTION

The text-based command line terminal worked, but it was "research code" (i.e., utilitarian and somewhat unpleasant to use). We needed something less technical and more enjoyable to use in order to introduce the world to our dynamic graph database and encourage adoption. We needed something visual—something everybody could run on any system. Realizing that G* was only one part of an overall solution, we set out to develop G* Studio—an interactive environment for G* available as an embeddable component or as a complete application. Its features would include a syntax-highlighting graph editor and console, visualization tools, query and analysis tools, as well as configuration analysis and management.

We also realized that G* Studio would need to be able to teach its users how it works. Further, although we did not see this at first, we soon learned, from external feedback, that in order for users to use G* Studio effectively, they would need to know about graph theory and graph databases, and why they are awesome.

Finally, with what to do firmly in hand, we began to think about why we would be doing it. After some reflection, we agreed that full stack development experience is critical for success in our modern world and the Internet of Things. Just as balancing theory with practice is essential regardless of domain, we must embrace a range of software development talents suitable for modern application development, from databases through servers to APIs and clients. Regardless of domain, responsibility for the entire stack is important. To that end, we decided that the students would

- make all front-, back-, and middle-end design choices (with guidance towards best practices)
- Divide and conquer the work on their own
- set and enforce their own source code style guidelines
- have root access to the servers

Armed with unanimous buy-in on what to do and why we would be doing it, we began our software development journey by asking the next question: How would we do it?

GETTING STARTED

The first decisions we made about how to proceed focused on our development environment: hosting, source code control, and team collaboration. To avoid using our school's inconveniently secure cloud—it's really more of a bunker than a cloud—we chose to host our system on Amazon Web Services (AWS) [2] Elastic Compute Cloud (EC2). Amazon offers one year of free service, perfect for a two-semester project. We managed our source code

with Git and stored it in the cloud on GitHub [5]. We would all develop locally, commit our code and push it to Git, and then deploy to our EC2 instance in the cloud. We used Slack [12] for team collaboration in addition to our weekly in-person meetings.

A note from the professor—Many students are beset with anxiety on two fronts when starting out with source code control. One challenge comes from their not understanding the technical details of how it works, what it does, and where it's stored. (Thankfully, everyone seems to understand why we need it.) Teaching our students how it works and what it does and where it's stored easily remedied this problem. The second source of anxiety is far more pernicious. It's social anxiety stemming from fear of "breaking it." (See sidebar.) While there tends to be considerable pressure to avoid "breaking the build" with untested code (another thing that's easy to fix by teaching our students about testing and continuous integration), the real problem is the students' fear of corrupting other people's code by somehow misusing source code control. They worry about overwriting others' code, accidentally reverting the repository to an earlier state, or somehow losing code. These fears are not without merit. Git, for example, is complex to say the least. There is "pull" and "fetch," which sound the same but are actually different. There's "update" which does not. And what the heck is "rebase?" (I'm still not sure, but I know it's not "reset.") As my students note in the sidebar, having the team work together in person, experimenting as a group with adding, committing, pushing, and pulling code may relieve this anxiety thanks to the technical knowledge they share and the social experience of having safely done it together.

Our next decision involved the target platform. After briefly considering building native mobile and tablet apps for iOS and Android, we quickly decided that the best way for us to implement a visual environment capable of running on any system would be to take advantage of the HTML Document Object Model (DOM) and JavaScript execution environment present in all web browsers.

Having chosen a web-based interactive environment for G* Studio, we set out to tackle a few more fundamental issues such as look and feel, tools for building browser-based applications (once called "Rich Internet Apps," today called "apps"), and how to communicate with the G* database from inside the browser.

For look and feel, we took inspiration from R Studio [11] and a few browser-based operating systems projects [13] that other students had recently written. Both R Studio and those operating systems provide a lot of information about complex systems in user-friendly ways.

We decided to adopt three currently standard tools for building browser-based applications: (1) the Bootstrap GUI framework for responsive page layout, typography, and user interaction [3]; (2) the jQuery JavaScript library for DOM manipulation and AJAX/JSON functionality [6]; and (3) the D3 JavaScript visualization library for drawing graphs and charts [4]. Using these common tools allowed us to focus our development efforts on building valuable application-specific user-facing functionality without spending significant time on "plumbing." Further, the responsive design features of Bootstrap moved us towards having a mobile browser app with no additional effort. (Adding a library like jQuery Mobile would get us the rest of the way there.) Experience with these tools makes for excellent additions to our students' resumes, since industry uses them routinely.

STUDENT STORY: SCARED TO GIT

I was scared to use Git, even though the point of Git should have been to make me feel better about the safety of the code. I made it through a class requiring that I use Git myself, minimally committing changes to GitHub while trying to "commit early and commit often" as the semester progressed. In my last year at school I became a member of the G-stars. The professor wanted source control, which was smart and useful, as we had several developers working together. But I had never contributed to code like that in any of my previous projects. So I was back to feeling like I had before, practically back to "How do I GitHub?" Luckily I had smart, supportive teammates who helped me get up to speed with Git and GitHub, but I was still scared of committing my own code. I'd send my code to teammates and they'd add it into theirs in their next commit. I did that until my professor called me out for not committing. I explained that I didn't want to mess anything up, but that didn't fly, and by the end of that meeting, I felt the same as I did when I first ever tried using GitHub. So, I had the rest of the team practically hold my hand as they helped me walk through the process. I really didn't want to let the team down, and they were all really supportive and patient with me and my timidness towards the idea. We went from the beginning to the end of a commit, and I felt much more confident about it. I didn't break anything, and I even helped contribute towards the code-this time with my name on it!

Another note from the professor—Projects extending more than a single semester also make wonderful additions to students' resumes. They get to brag about exercising their vital software development powers in projects affording them scope beyond in-class assignments. If their projects involve shiny terms like API development, graph databases, and analytics, so much the better. Throw in some distributed systems—as they are everywhere today and experience with solving common distributed systems problems is great—with a dash of popular tools like Git and Slack, and our students will have a fine resume replete with valuable experience by the time they are done with us. Stories like this one, where students learn hard and soft skills that enhance their career prospects after graduation, make for great answers to parents asking about educational ROI.

[1] Why REST? We like that REST is based on existing HTTP verbs and therefore requires no additional infrastructure. (We're always seeking to minimize dependencies.) Its URL encoding scheme makes it universal to today's internet. We considered XML for a few seconds, but immediately thought better of it.

We chose to design and implement a REST[1] API to communicate with our graph database. It would be callable from the browser or any client with an internet connection via standard HTTP verbs. It would respond with objects in JavaScript Object Notation (JSON). To that end, we quickly discovered the utility of JSONLint [7], a JSON validator that helped us debug our API responses. Since REST uses HTTP verbs, we would need an HTTP server to implement our API. Enamored of the KISS principle, and wanting a modular, testable solution with as few dependencies as possible, we chose to forego bulky middleware like JBoss, WebSphere, and (the accursed) Struts in favor NanoHTTPD [10], a tiny, embeddable HTTP server written in Java.

<aside>

We can hear you asking, "Wait… what? Why use NanoHTTPD instead of something standard and easy like PHP or Python?" We thought of that. But G* is written in Java. Our core API functionality needed to be tightly coupled with the G* engine so it could call G*'s graph management and analytic routines. Since we were trying to minimize dependencies (always a best practice) we took a dim view of building our system around heavyweight Java containers like Tomcat, JBoss, WebLogic, WebSphere, etc. because bridging the server environment of PHP or Python to G*'s Java environment would be a problem. While there exist PHP/Java integration bridges (the accurately named and open source *php-java-bridge* and *Zend*, for example), they typically require Java application containers (Tomcat and WebLogic, respectively) to work. Having decided to forego heavyweight dependencies like those introduced by Java containers and application servers, integration with Python or PHP became untenable. We needed something lightweight and with no dependencies other than a JAR. Enter NanoHTTPD.

</aside>

We separated the core of our API from its REST and JSON implementation. This internal separation of the core API and the REST (of the) API makes our solution both modular and extensible. The core API (called, appropriately enough, *APIcore*) is responsible for "tactics" in that it has to figure out **how** to get results by making calls into G*'s database internals and returning those results in raw strings. *APIrest* is responsible for "strategy" in that it receives requests for **what** to do from the browser via HTTP, calls the appropriate routine in *APIcore*, and then takes the resulting raw strings[2] from *APIcore* and builds the JSON response returned to the browser. If, in the future, we wanted to implement an XML API (or any other form of API) we could implement XML-returning "strategy" code while leaving the *APIcore* "tactics" untouched. Figure 2 shows this architecture.

DEVELOPMENT AND DEMOS

Avoiding embarrassment is a great motivator. And that's why demonstrations make great development milestones. The first thing we did was set up a calendar of important dates that included functionality/development milestones and demonstrations. Finding a willing audience was easy in our academic environment, so we arranged demos for our faculty and dean. Also, any time someone from industry came on campus to speak to a class or present to a group we tried to show them our software. Several G* Studio features came from suggestions, questions, and discussions incorporating people outside of our school and outside of our field (professionals in finance, supply chain, pharmaceuticals, and more.)

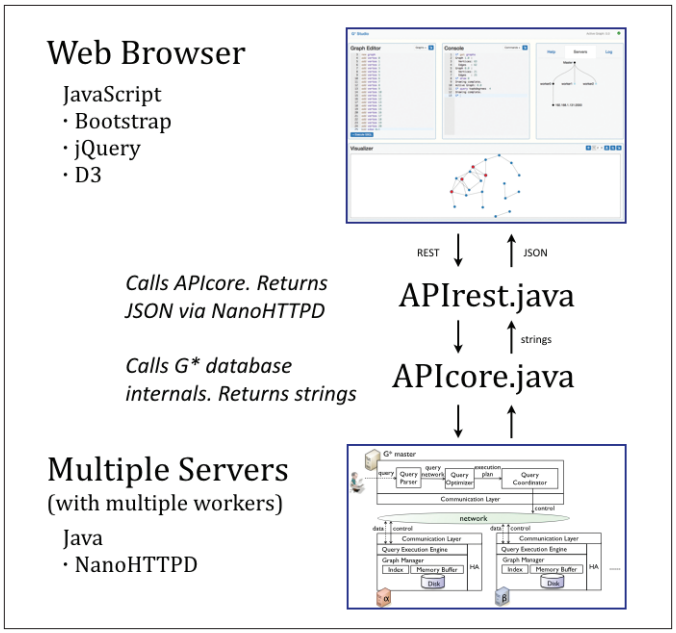

Figure 2: Architecture

DEVELOPMENT

We separated our functionality/development milestones into three phases: Phase one, API request and response; phase two, GUI / web interface to the API; and phase three, additional features. We considered phases one and two as "need to have" features without which the project would fail. Phase three consisted of "nice to have" features that made our system more powerful and easier to use.

PHASE ONE:
API REQUEST AND RESPONSE

The first task we tackled was developing simple request-response connectivity, sending browser-based REST requests to our server-based graph database and getting JSON responses from our server back to the browser. Once we integrated NanoHTTPD and verified that we could indeed interact with it from the browser, we started developing *APIrest* (see Figure 2) by implementing elementary GET[3] commands such as *version* and *time*. These simple commands did not require any interaction with the G* database and their JSON representation was quite simple.

[2] In this context, "raw" strings are minimally formatted or un-formatted strings. They are not JSON. They are not XML. They are just packed arrays of (raw) character data.

[3] This is the HTTP verb GET, which requests data from a server. The POST verb, which we'll use later, submits data.

GET Request: `http://ec2-gstar.amazonaws.com:8080/version`
JSON Response:
```
{ "version" : "42.007.2112.8675309" }
```

After debugging our API responses—with help from JSONLint—we moved on to the logging module. At first, we recorded server-side activity in a text file that we would *tail* so we could monitor it. Later, in phase three, we incorporated logging to a relational database via ODBC.

Once we had server-side logging and our request–response cycle working, we began implementing a few "informational" graph commands (GET *graphs*, GET *vertices*, GET *edges*) by developing *APIcore*, which is tightly coupled with G*'s database internals. (See Figure 2). This required that we parse GET requests in *APIrest* and call the appropriate routine in *APIcore*, which in turn would call the appropriate routine(s) in G* and return the results to APIrest as a raw string for formatting into the JSON response to the browser. We verified these web-based results against those from our Terminal (see Figure 1).

GET Request: `http://ec2-gstar.amazonaws.com:8080/graphs`
JSON Response:
```
[{"graph" : "0.0", "vertices" : 2, "edges" : 1},
 {"graph" : "1.0", "vertices" : 4, "edges" : 2}]
```

With a few graph information commands implemented, we spent the final part of our initial phase implementing graph creation in the API via HTTP POST commands. Once again, we added to *APIrest* for the "strategy" (what to do) and *APIcore* for the "tactics" (how to do it). And, again, we verified our API-based results against those from our Terminal.

POST Request: `http://ec2-gstar.amazonaws.com:8080/graphs/5`
JSON Response:
```
{ "message" : "New graph 5.0 was created." }
status: success
```

We ran into some problems where the browser would block our request-response round trips due to cross-origin POST restrictions. This surprised us because both our browser-based client and our Java-based server were running on the same IP address both locally and on EC2. But the client app runs on port 80 while our server listens on port 8080. Much to our surprise, we learned that using different ports—even on the same IP address—is enough to trigger cross-origin security precautions implemented by most modern browsers. To avoid this, we added an OPTIONS request before every POST to permit the next cross-origin request by telling the browser to allow GET, POST, PUT, DELETE, and OPTIONS operations by including the following in the OPTIONS response:

OPTIONS Response:
```
200 OK
Allow: GET,POST,PUT,DELETE,OPTIONS
```

We also added three "Access-Control-Allow" headers to all responses:

```
Access-Control-Allow-Methods, DELETE, GET, POST, PUT, OPTIONS
Access-Control-Allow-Origin, *
Access-Control-Allow-Headers, X-Requested-With, Content-Type
```

With API request and response finally working, we moved on to phase two.

PHASE TWO: GUI/WEB INTERFACE TO THE API

Pablo Picasso might have once mentioned that, "Good artists copy; great artists steal." (Definitive attribution of this is difficult to find.) One of our early design decisions was to ~~copy~~ steal R Studio's layout. While we certainly did not (and do not) consider ourselves great artists, we recognized in R Studio what so many others have: it provides the concise power of a command line interface while simultaneously presenting an easy-to-use interactive graphical environment. Where it provides an R language editor, we provide our graph language editor. Where it provides an interactive console to the R engine, we provide our interactive console to the G* engine. Where it provides a plotter for charts and graphs, we provide a visualizer for graphs and charts. There are even similarities in our help system and tutorials. Using the responsive grid layout tools provided by Bootstrap, we structured our R Studio–inspired web app around a graph editor, a graph console, a visualizer, and a tabbed help/tutorial/log panel.

For the graph editor we used ACE [1], an open source high performance code editor for the web. By embedding an ACE instance in a Bootstrap *well* contained in a (grid) *row* (the *well* in the *row* effectively becoming a container), we were able to provide text editing functionality, including syntax highlighting, without reinventing a code-editor ourselves. Implementing syntax highlighting for our custom grammar was not as easy as the examples on the ACE web site might have led us to believe. Eventually, we found that modifying the LISP definition file *(mode-LISP.js)* for our own syntax was an efficient way to bypass the brain damage of manual configuration from scratch. In hindsight, we would also recommend using the "Ace Mode Creator" utility on their web site.

A note from one of the G-stars—For me, the most challenging part of the project was dealing with the libraries we used for different functions of the user interface. D3 and the Ace editor provided countless hours of frustration and stress as we tried to work through the limited, vague, or wrong documentation, misleading tutorials, or just broken functionality. It should have been enough to drive most people crazy and just give up, but we stuck with it and somehow we were always able to overcome the issues we faced.

Once we got good at it, ACE became the basis for our graph console as well. We wrote a special key-handling function to process the *home*, *end*, *arrow*, and *backspace* keys in order to implement the prompt, command history and recall, and other behaviors necessary to provide the interactive, line-by-line experience of the console as opposed to the page-based experience of the editor. To do this, we assigned our special key-handling function to the ACE instance's `keyBinding.onCommandKey` event and processed keystrokes according to their *keycode* parameters.

At this point, the user interaction flows as follows: A user enters one or more graph commands in the editor and clicks/touches the execute button; or they enter one command in the

console followed by the enter key. JavaScript functions parse the command(s) "behind" the GUI using regular expression pattern matching. Each command is processed with two functions, the first doing error checking and reporting, the second implementing the command actions. (We split this code to improve readability and reusability.) Then, each command is checked for whether or not the user supplied a graph ID. If so, it is used. If not, the current graph ID (tracked by the GUI) is used unless there is none (because of a cold start or browser refresh), in which case an error is reported. If all goes well, the REST API request URL is created, typically GET or POST. (PUT and DELETE are other possibilities in REST but we have not yet made use of those HTTP verbs.) One of two jQuery functions (`.getJSON` for GET or `.ajax` for POST) send the REST requests to the server. At this point, the *server* receives the browser requests in *APIrest* where it decodes the request and calls the appropriate routine in *APIcore*. *APIcore* uses G* internals to execute the command actions and collect the results, which are returned to *APIrest* as a raw string (or possibly a list or a set of raw strings). *APIrest* formats the data from *APIcore* as JSON and sends it to the browser. These responses from the server trigger a JavaScript callback function that examines the *status* (200 is success, others less so) and the *data* (the JSON object, one hopes). If all is good, the JSON responses are instantiated as JavaScript objects, formatted, and displayed in the GUI's console, visualizer, or data panel. Meanwhile, the editor and console have been waiting for the next command.

Once we had our graph editor and graph console working, we began testing G* functionality via the GUI. We verified the responses in the GUI against the results of the same commands on the same graphs executed in Terminal. When all was looking good, we turned our attention to the visualizer.

After considering several JavaScript visualization libraries, we chose to implement our visualizer with D3, specifically its force-directed graph. The D3 web site had some helpful examples, one of which was a graph of character co-occurrence in *Les Misérables*, which was very similar to the kind of graph we wanted to show. Some of the challenges in getting D3 integrated into our visualizer involved getting the z-index right for the multiple overlays that comprise the visualizer display, figuring out how to use DOM selectors to access and modify the SVG graphic objects D3 painted, and keeping D3's many global variables properly updated with our (more modular) state.

We once again turned to Bootstrap for the tabbed help/tutorial/log panel, using its *nav-tabs* class with divs as *wells* (some of which contained an *iframe*) for each tab.

With all the pieces of our GUI / web interface in place (see Figure 3), we noticed that we rarely used the API any more. In fact, we sometimes forgot to update the server-side API help and documentation after adding new client functionality because the GUI was so pleasant to use and working so well.

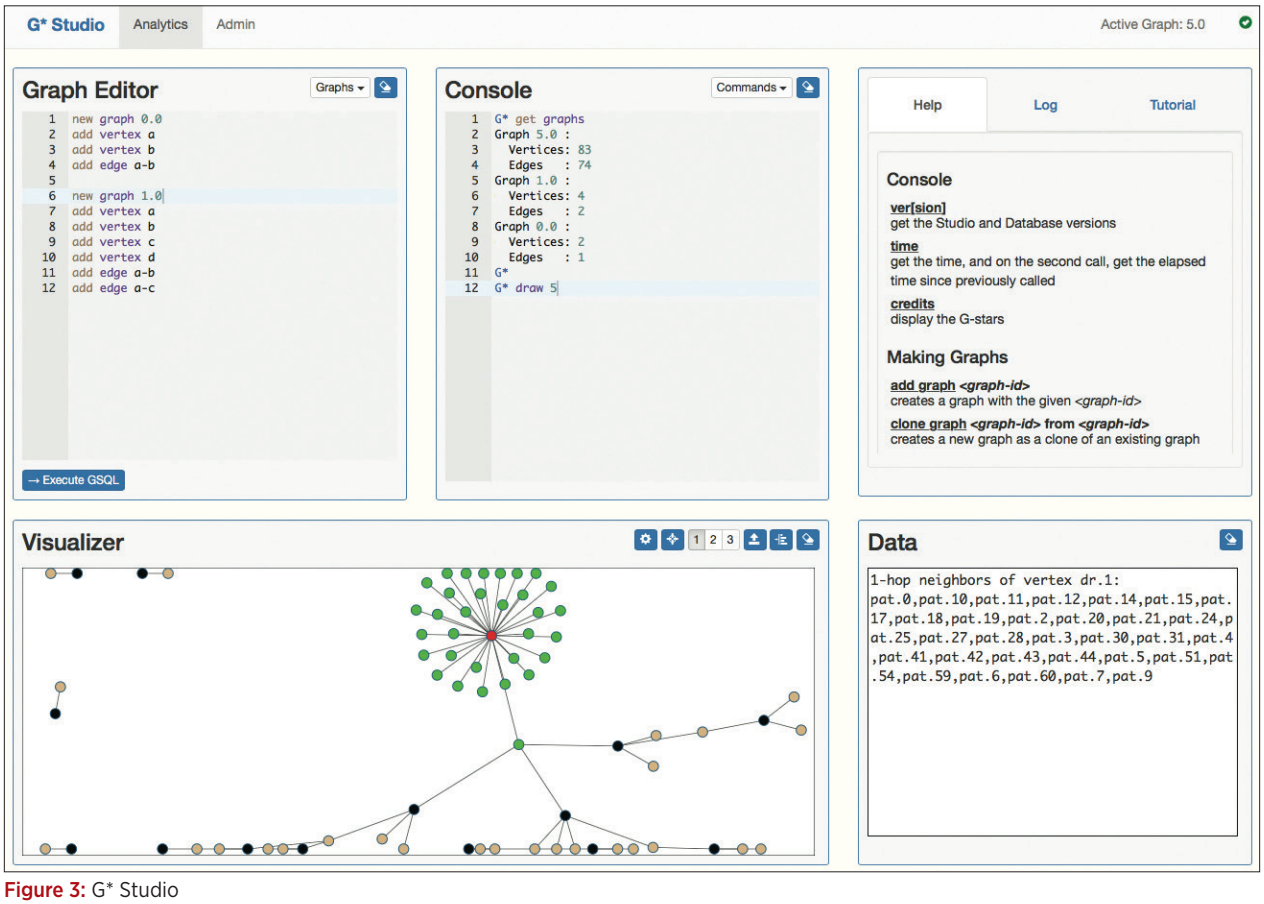

Figure 3: G* Studio

PHASE THREE: ADDITIONAL FEATURES

With the system architecture done and the basic GUI complete, our "need to have" features were in place, and we turned our attention to teaching our users about graphs, graph databases, their awesomeness, and how to use G* Studio. In the process of doing this we added some other "nice to have" features to make our system even more powerful and easy to use.

We developed an interactive tutorial (see Figure 4 for a tiny part of it) that teaches the user about graphs and graph databases. It also shows them how to use G* Studio. After starting out with a little bit of graph theory, the tutorial walks the user through various GUI elements and features of G* Studio (see Figure 3). We made the tutorial interactive within the GUI. This was easy with the tutorial embedded inside the GUI and both written in JavaScript, so simple messaging and event processing worked quite well to integrate the two. In that sense, rather than being a passive learning "read about it" experience, our tutorial is very much an active learning "try it and see" experience. As the user interacts with the tutorial the results are presented (as new graphs, drawings, queries, etc.) in G* Studio itself.

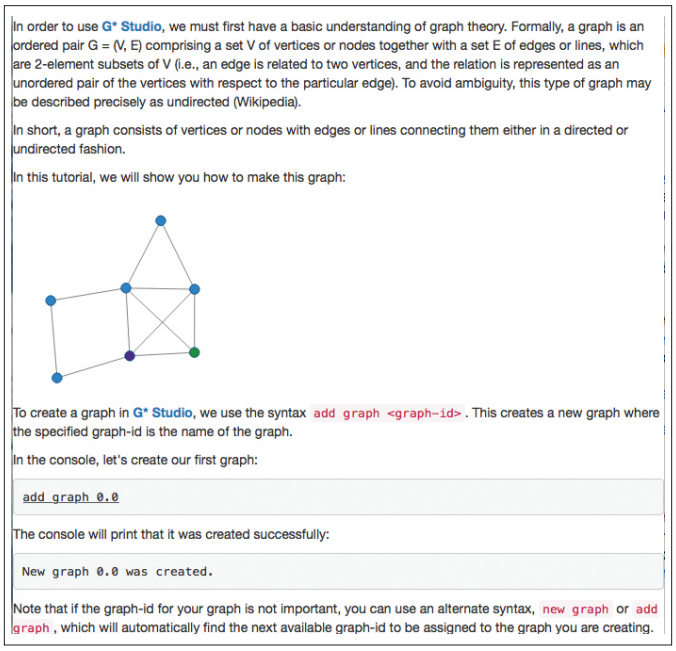

Figure 4: Interactive Tutorial

Once users have gone through the interactive tutorial once or twice, all they really need from that point on is a standard help facility of reminders and command shortcuts. So we built one. Like our tutorial, it is also interactive with the GUI.

Developing the tutorial with an eye towards teaching users about graphs and graph databases caused us to think about illustrative use cases. To facilitate graph theory education, we pre-loaded several common graph definitions and use cases inside the graph editor, including 8-vertex full, 32-vertex ring, 32-vertex bipartite, 63-vertex binary tree, 64-vertex star, 64-vertex Erdős-Rényi random, and an incrementally-evolving graph consisting of four consecutive snapshots. Now, all the user needed to do to create any of these graphs was to select one from the menu in the graph editor and execute it. The commands are all synchronously piped to the console and the graph is created. With the success of pre-loaded common graphs, we quickly realized that we could pre-load some common graph queries as well: degree distribution, top-k vertices by total degree, and trends of rising or falling stars across contiguous graph snapshot pairs by total degree centrality.

> *One more note from the professor*—Anytime we can work into a practical project material that's primarily theoretical—graph theory in this case—we win. Talking about teaching a balance of theory and practice is fine, in theory. Demonstrating it in practice, like this, is great.

Finally, as our use cases and examples became more complex and more apropos of the "real world," we found ourselves wanting more features. We enhanced our visualizer to show 1-hop, 2-hop, and 3-hop influence from any vertex in the graph by coloring the vertices. (Note the green vertices of 1-hop influence from the red vertex shown in the graph visualized in Figure 3.) We also implemented the ability to update vertices and edges (rather than just adding new ones), setting and getting arbitrary attributes (as strings) on vertices and edges, and easy graph cloning (for those evolutionary graph snapshot examples). On the systems management side, we implemented a new visualizer pane to show master and worker configuration hardware details (and IP addresses) for configuration analysis and (eventually) management. While designing, developing, testing, and integrating those features into our system, some of us got mischievous[4] and implemented hidden features. While the non-mischievous team member was fighting Heisenbugs by implementing a facility to force checkpoints in the storage manager components of the underlying distributed database, others implemented "festive mode," the details of which are highly classified.[5]

We tracked our progress by looking at our Git commits over two semesters in time and three phases of development. In the commit chart for the backend G* database code (see Figure 5) we can see the initial commit in September followed by many commits in October and November as we developed *APIcore* and *APIrest* in phase one. The addition of new features once the initial version of G* Studio was working (phase three) caused the bump in February. In contrast, the commit chart for G* Studio (see Figure 6) shows most of the work in February through April, when we developed most of G* Studio (phases two and three), having put together only a minimally viable version by the previous December.

DEMONSTRATIONS

Armed with a full-featured graph database and our brand new interactive GUI with which to show it off, we set out to spread the word about graphs and graph databases. As previously

[4] The mischievous "us" refers to graduate students, of course.
[5] Like Portal's cake, this too is a lie.

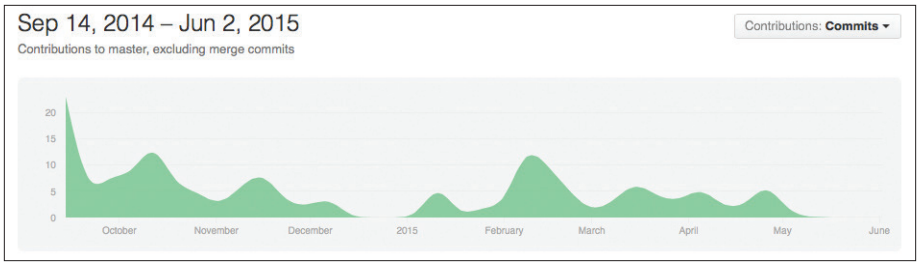

Figure 5: G* database commits

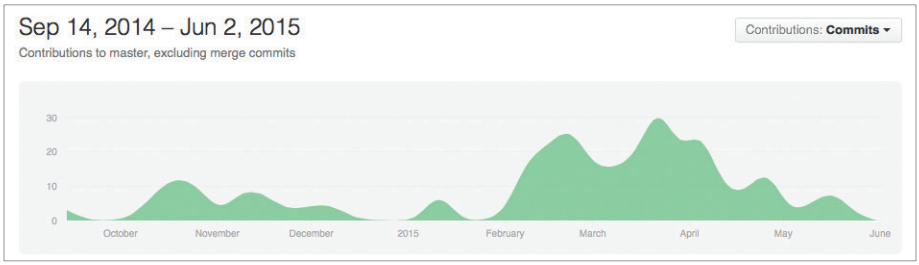

Figure 6: G* Studio commits

mentioned, demonstrations are wonderful motivators. We always pushed ourselves to sprint to the next demo date, lest we be embarrassed. In addition to driving development, perhaps the biggest benefit of our many demonstrations was that they conveyed to people who do not live and breathe graph theory and distributed systems what a graph is, what a graph database does, how various enterprises can use them, and why using them can be both interesting and beneficial. Repeatedly putting these technical concepts in lay terms greatly improved our thinking and helped to shape and enhance G* Studio.

In addition to showing G* Studio to our student and faculty peers, we invited many people from many industries to see it. In every case we heard new ideas, were asked new questions, and talked about new scenarios in areas as diverse as marketing, transportation, pharmacology, communication, finance and more. This too helped to clarify our thinking and sharpen many aspects of G* Studio.

THE G-STARS REFLECT

Gregory: What made the project so great? We created it from scratch and got to make all the decisions as a group. Sure, our professor led some of the decision-making, partially from his experience and knowledge, and partially because it was using his research. Even with that, most of the design and decision-making was still a discussion, something I was not used to in school. To me, it was amazing to have these discussions then actually build something from the ground up, something that is cutting-edge. I knew that we were doing something that many people will never get to experience in school. It also exposed me to developing things that I never even heard of before. I loved learning REST and programming things that I know I would have never done until maybe having to do them for a job someday.

Cassie: I owe everything to this team. I really enjoyed meeting up with and doing work with the team. Collaborating on this project gave me experience that I'm using each day at work. The camaraderie we shared between work and play was an experience I will miss and treasure forever. We may have all moved away from each other for now, but I hope for the day when we are all back in the Norton room at Marist, working through the twilight hours together again.

Thomas: Just the idea of working on something from scratch, but at the same time, being on a team, was an amazing experience. Instead of just contributing to a pre-existing code base, we were designing things how we wanted (with some set goals and basic parameters). Though we regretted some of our choices later on down the road, that is part of the learning experience. As a team, we were successful. There were definitely frustrating moments, whether from APIs that lacked correct documentation or libraries that seemed to work primarily on magic. That said, this project still contains some harnessed magic, and that would not have been possible without the team and our working relationship. We spent many long nights trying to make everything perfect, and though there is always more work that can be done, I do not regret a single second of it. This is my favorite project I have ever been a part of, and easily the best team.

Shane: Initially, my biggest fear about this project was the scale. After our professor described the project to the team for the first time, it seemed like a massive undertaking and I know I wasn't alone in having no idea where to start. I had never worked on such a large project with so many integrated parts and features. Many of the GUI features we talked about implementing were very complex and required the use of pre-existing libraries, most of which I had never heard of. However, as we started to progress, we quickly realized how prepared we were for such a challenge.

CONCLUSION

Projects like this allow students to take a longish view of decision-making, implementation, and maintenance/enhancement. This was a two-semester project and the students experienced first-hand the impact of their own earlier decisions such as

- choosing NanoHTTPD over a heavyweight application server or container,
- deciding on JSON as a response protocol instead of XML,
- selecting D3 as our visualization library instead of writing the graphics code from scratch or adopting a different library (like Sigma.js or Arbor.js).

Our early decision to have the students to make front-, back, and middle-end design choices turned out very well. There were

some mistakes made along the way, but mistakes that don't kill you are excellent teachers, even (especially?) if remedying them is painful.

Putting the students in charge of dividing the work on their own also worked well. They arranged themselves into natural and efficient teams. Certainly, the fact that they were graduate students at the end of their college careers helped this. Less experienced students might not be as successful in this area.

Letting the students set and enforce their own source code style guidelines was a mistake. They never managed to agree on a consistent style, in spite of the professor telling (or yelling at) them that being consistent is more important than the details of the style. This resulted in time wasted on code reformatting. Considering the fact that industry tends to enforce code style strictly, we should have imposed a strict style for this project.

Giving students root access to the servers was great. It set up a situation engendering personal responsibility, peer pressure, and trust, all at the same time. Trusted with root access, the students were empowered to fix anything and everything they broke when (not if) they deployed bad code, and to do so before anyone noticed, lest they risk the wrath of their peers.

Some of the strongest learning outcomes came from using new technologies, experiencing the value (and, at times, valor) of teamwork, and end-user focused development in a hard-core CS class. Many times, the analysis and design parts of a CS or software development curriculum are isolated from implementation courses. (There are many reasons for that, and not all of them bad.) It's one thing for the analysis for students to produce pretty diagrams or useless UML. It's quite another to actually write the code to implement those designs and make it work. That lends a new and valuable perspective to the entire enterprise.

PARTING THOUGHTS:

- Separate strategy from tactics.
- Minimize dependencies.
- Divide features into "need to have" and "nice to have."
- Demonstrations make great development milestones.
- Let (or make) the students make decisions and then make (or let) them live with the consequences.
- Watch out for cross-site scripting security exceptions. They are really annoying, and relentless.
- Never underestimate the necessity of consistent JSON formatting.
- Be sure to put delimiters on your regular expressions, lest they match too much.
- Be careful with the HTML z-index attribute.
- The documentation for third party libraries is rarely as good as you wish it were. Sometimes they seem to rely on magic, which can be troublesome to harness.
- You and your student developers are too close to the project to see the proverbial forest for the equally proverbial trees. Seek outside critique from external stakeholders early and often.
- While tools like GitHub and Slack promote collaboration, teamwork, and working at all hours of the day and night (for better or worse), they are no substitute for in-person meetings. In-person teamwork promotes a sense of community, shared experience, and harnesses the comfort offered by safety in numbers to help mitigate Git shyness and other commitment issues.
- Enforce disciplined source code style.
- Promote disciplined procedures for source code control: Pull the new code before you begin working each time. As you work, commit early and often, with descriptive commit messages. Commit and push from time to time as you work, and always when you're finished for that session. Discuss any merge conflicts with the team either in person or on Slack so they can be resolved quickly.
- The value of buy-in and fun cannot be overstated, especially fun. Remember to mix work and play. The more everyone enjoys it, the better the results. So please, embrace mischievousness[6] and **join the fun**. ❖

Acknowledgements

The authors thank Jeong-Hyon Hwang, the original G-star and a giant upon whose shoulders we stand. We thank as well all the people who attended our demonstrations of the system and provided feedback. Finally, we wish to thank the *Inroads* reviewers for their thoughtful comments. Writing papers like this, just as with writing software, relies on critical feedback from external stakeholders.

References

1. *Ace*; http://ace.c9.io. Accessed 2016 February 27.
2. *Amazon Web Services*; https://aws.amazon.com. Accessed 2016 February 27.
3. *Bootstrap*; http://getbootstrap.com. Accessed 2016 February 27.
4. *D3*; http://d3js.org. Accessed 2016 February 27.
5. *GitHub*; https://github.com. Accessed 2016 February 27.
6. *jQuery*; https://jquery.com. Accessed 2016 February 27.
7. *JSONLint* - The JSON Validator; http://jsonlint.com. Accessed 2016 February 27.
8. Labouseur, A. et al. "The G* Graph Database: Efficiently Managing Large Distributed Dynamic Graphs." *Distributed and Parallel Databases* 33, 4 (2015), 479-514.
9. Labouseur, A., et al. "A Demonstration of the G* Graph Database System." *ICDE 2013*, 1356–1359.
10. *NanoHTTPD* - The tiny, easily embeddable HTTP server in Java; http://nanohttpd.org. Accessed 2016 February 27.
11. *R Studio*; http://www.rstudio.com. Accessed 2016 February 27.
12. *Slack*; https://slack.com. Accessed 2016 February 27.
13. *Students' Operating Systems*; http://www.labouseur.com/courses/os. Accessed 2016 February 27.

Alan Labouseur,* *Alan.Labouseur@Marist.edu*

Shane Crumlish, *Shane.Crumlish1@Marist.edu*

Cassandra Graves, *Cassandra.Graves1@Marist.edu*

Melissa J. Iori, *Melissa.Iori1@Marist.edu*

Gregory Miller, *Gregory.Miller2@Marist.edu*

Thomas J. Wojnar, *Thomas.Wojnar1@Marist.edu*

Marist College
School of Computer Science and Mathematics
3399 North Road, Poughkeepsie, New York 12601 USA

* corresponding author

DOI: 10.1145/2896823

[6] Okay, so Festive Mode is not highly classified. It's seasonal icons (hearts, snowflakes, graduation caps) and random silliness (companion cubes, fire).