



# The Future of Big Data Is ... JavaScript?

Jimmy Lin and Kareem El Gebaly • University of Waterloo

**T**he future of Big Data is ... JavaScript? No, seriously, hear us out.

JavaScript has already made significant inroads as an integrated technology stack for building user-facing applications, both on the front end (JavaScript running in the browser) and the back end (JavaScript running in Node.js). In this article, we explore the idea of using JavaScript in Big Data platforms, both on the front end (for example, how data scientists interact with analytical tools) and on the back end (such as distributed execution infrastructure). Could the future be ... JavaScript everywhere?

Given the anticipated wince-inducing reaction to this question from any sane developer, let's get this out of the way first:

*JavaScript is a terrible language.*

JavaScript is terrible in so many ways. Instead of recounting them all, we'll just refer to "WAT," Gary Bernhardt's incredibly funny talk.<sup>1</sup> Nevertheless, we should cut Brendan Eich (the creator of JavaScript) some slack, considering that he developed the language in 10 days. We're sure neither he nor anyone else expected JavaScript to become so successful and ubiquitous. Anywhere there's a browser, there's (almost always) JavaScript. Your connected refrigerator or toaster in the near future? It might have a browser for its user interface, which means it'll likely run JavaScript.

But we're getting ahead of ourselves. Let's start with a quick recap of computing history to see how we got here.

## A Brief History

In the mid-1990s, Sun Microsystems (gobbled up by Oracle in 2010) came up with a brilliant slogan, "write once, run anywhere," to describe the promises of Java in offering seamless cross-platform development. In the early days of the Web, one of the main selling points of Java was its ability to run inside the browser via applets – so you can push server-side code over to the client with minimal hassle. It sounded like a great idea, except that it didn't actually work. Java applets were clunky, slow, and insecure. Eventually, the world relegated them to the dustbin of computing history – at best a historical curiosity today.

In the end, of course, Java did just fine. Although today hipster languages such as Scala and Clojure jockey for attention, the broader ecosystem of the Java Virtual Machine (JVM) is well-entrenched on the server. Thanks to Hadoop, Spark, Flink, and other Big Data platforms, the JVM has become the de facto execution environment for Big Data applications. On the client side, Java runs on more than a billion devices around the world via Android. Even though it didn't deliver on the "write once, run anywhere" promise, by any standard Java is a smashing success.

Despite the similarity in names, which was largely a marketing ploy, JavaScript is entirely unrelated to Java (except that both share C-like syntax). It was originally designed as a lightweight in-browser interpreted language for non-professional programmers, but over the years JavaScript has become the dominant language for Web development, practically

## Big Data Bites

“Big Data Bites” is a regular department in *IEEE Internet Computing* that aims to deliver thought-provoking and potentially controversial ideas about all aspects of Big Data. Interested in contributing? Drop me a line!

—Jimmy Lin

unchallenged in its hegemony. In fact, front-end development today is synonymous with HTML, CSS, and JavaScript. In the beginning, JavaScript was limited to those dumb “see, I can compute factorial in the browser” and “oh look, I can change the background color of the page with a button click” scripts. Today, Web apps written in JavaScript offer user experiences rivaling that of native desktop apps. For example, Gmail is comprised of more than 400,000 lines of JavaScript (as of 2010).<sup>2</sup>

On the client side, JavaScript comes much closer to delivering the promise of “write once, run anywhere” than Java ever did – evidenced by the fact that JavaScript-rich Web apps “just work” nearly all the time. They work on Intel chips. They work on ARM chips. They work on Windows, Linux, Mac, and other operating systems. They work on Firefox, Chrome, Safari, and other browsers. They work on desktops, laptops, tablets, and mobile phones. Sure, cross-platform Web development remains a herculean effort with endless shims and special handling of weird incompatibilities; but for the average end user, the experience is fairly seamless. That’s a pretty remarkable feat.

### The JavaScript Server Incursion

And so for a while, we lived in a stable bipolar world. JavaScript reigned in the browser on the client side, and Java held a dominant position on the server (although with competition from PHP, Rails, Python, and other stacks). REST APIs served as platform- and language-agnostic connections between front ends and back ends. All was well.

Then, some developers had to upset this balance by exploring the question, “What if we started running JavaScript on the server also?” Server-side JavaScript is an idea that dates back to 1995, soon after JavaScript appeared in the browser, but no one took it seriously until the mid-2000s. Developers started exploring the idea for a vari-

ety of reasons: One of the most compelling is the reduction of impedance mismatch between front- and back-end development, particularly for building rich, interactive Web apps. Some of the challenges were technical – the friction that comes from switching between different languages and environments when the end goal is to deliver a seamless user experience,<sup>3</sup> but some were organizational – writing everything in JavaScript breaks down team structures that segmented developers into front- and back-end teams, with all the associated communications barriers of silos.<sup>4</sup>

All of this became practical with the advent of high-performance JavaScript engines, most notably Google’s open source V8 (<https://developers.google.com/v8/>). Everything came together with Node.js, the popular open source runtime environment for server-side JavaScript.

There have been a number of high-profile success stories of Node.js. In 2012, LinkedIn mobile moved from running Ruby on Rails on the back end to Node.js and was able to replace 30 servers with just three to serve the same traffic.<sup>5</sup> Netflix, PayPal, Uber, and many others are all big users of Node.js. Development with Node.js meshes well with the “microservices” architecture in vogue today, and the advent of Docker ([www.docker.com](http://www.docker.com)) simplifies many aspects of deployment. All of these technologies fit together nicely to offer an integrated stack for building user-facing applications.

Before moving on, there’s another historical footnote worth mentioning. Beyond applets, there has been another concerted attempt at pushing

Java onto the client. Google Web Toolkit (GWT; [www.gwtproject.org](http://www.gwtproject.org)), initially released in 2006, is a framework that lets users develop Web apps all in Java. It powered the ill-fated Google Wave, if anyone remembers that. GWT is, for all intents and purposes, dead. Somewhat ironically, GWT cross-compiles Java into JavaScript that’s deployed on the front end, so this attempted incursion by Java into the browser is perhaps best described as a Trojan Horse.

### JavaScript and Big Data

So far, we’ve (hopefully) established that for user-facing applications, adopting a pure JavaScript stack (both on the front end and back end) is a fairly compelling choice. Let’s now examine the radical proposition of this article – that there are similarly compelling reasons to adopt JavaScript for Big Data applications, both on the front end and back end. In contrast to user-facing applications, Big Data applications are internal-facing, used mostly by an organization’s data scientists (and maybe higher-level pointy-hair types via dashboards). To be clear: by “front end,” we mean the tools that data scientists use to access analytical capabilities over large datasets, and by “back end,” we mean the distributed processing infrastructure (for example, Hadoop and Spark) that provides those capabilities.

What’s the front end for data scientists? Traditionally, it’s just been a shell – in the old days, this might have been Pig’s “Grunt” shell or the Hive shell; but more recently, it’s likely to be the Spark shell. Good old command-line interfaces!

However, changes are brewing. Browser-based notebooks (such as Jupyter; <http://jupyter.org>) have gained tremendous popularity with data scientists in recent years, for a variety of reasons: The tight integration of code and execution output elevates the analytical process and its products to first-class citizens, because the notebook itself can be serialized, reloaded, and shared. The ability to manipulate, rearrange, and insert snippets of code (in “cells”) lines up well with the iterative nature of data science and a wide range of analytics tasks. With the seamless integration of browser-based notebooks and scalable data analytics platforms, code written in a browser-based notebook can be executed on a cluster and the results can be further manipulated in the notebook. The commercial Databricks platform represents a big bet that browser-based Big Data analytics will be the way of the future.

This means that the browser may become the shell. And remember, JavaScript reigns in the browser.

Furthermore, the end product of data science is often a visualization to tell a story. In many cases, these visualizations are interactive, giving the data scientist (and her colleagues, managers, clients, and so on) a chance to twiddle knobs and drag sliders in exploring the data. In other words, the “output” of data science is actually a custom-built Web app that encapsulates the analytical results! The preferred visualization tool these days is D3.js (<https://d3js.org>). Guess what – that’s JavaScript, too. In other words, ultimately Big Data is user-facing also – with just a different set of users.

Currently, we have a “canonical” Big Data analytics pipeline that looks something like this: Spark clusters running in the datacenter (the back end) talking to a Spark client (either Scala, Python, or R) that might be running in a browser (notebooks), connected somehow to a Web app (an interactive D3.js visualization) that’s

the final analytical product. The connection between Spark clusters and Scala/Python/R is fairly well developed, but interfacing with D3.js today is likely accomplished via JavaScript Object Notation (JSON) or comma-separated value (CSV) dumps, which is clumsy. Why not cut out the middleman and just use JavaScript?

Let’s try to refine this argument by examining the modern data science workflow in more detail, leaving aside the final visualization step for now. It’s well known that the work of data scientists, particularly in cases of exploratory tasks, is highly iterative, where they “poke” at the data from many different angles. Frequently, these tasks focus on a small subset of the data – for example, the data warehouse stores six months of log data, but the data scientist is only interested in data from the last week. Here are some ways she can go about accomplishing her task:

1. She repeatedly queries the entire dataset, but filters for the subset of interest. Achieving low query latency is entirely dependent on the analytical engine properly optimizing away unneeded work, which is dependent on the data’s physical storage (how it’s partitioned, compressed, and so forth).
2. She materializes the data of interest. In an analytical database, this typically involves selecting into a temporary table or creating a materialized view; in Spark, this typically involves writing intermediate results onto the Hadoop Distributed File System (HDFS). Subsequent queries would then be posed against this smaller dataset.
3. She materializes the data of interest and then copies the data locally (to her laptop, for example) for further manipulation. This might involve running Spark locally or dumping data into a local relational database and then issuing additional queries.

There are issues with all three approaches. In the first, the analytics engine might not be smart enough to efficiently optimize the query, or there might be inherent limits to query execution speed. As a simple example, if the dataset is hash-partitioned, pulling recent records might still involve distributed scans with large fan-outs. After standard optimizations such as columnar layouts and predicate pushdowns, network latency and coordination overhead start becoming the bottlenecks.

In the second and third scenarios, queries over the materialized data might no longer run efficiently because the startup cost of the analytics engine (such as Spark) dominates the actual processing time. For some datasets, it might be easier to wrangle on the command line using `sed`, `awk`, and so on. By definition, efficient queries over small datasets isn’t a scenario for which Big Data processing platforms optimize.

In the second scenario, materializing temporary data creates a different set of challenges. In analytical databases, creating materialized views might require elevated privileges (not granted to everyone), and the ability to create temporary tables presents a different set of data management headaches. For example, who (data scientists themselves?) or what (automated processes?) “cleans up” 51 tables named variants of `tmp` or `test`? The same goes for Spark: go look in your HDFS home directory right now and count up all the directories named `test1`, `tmp42`, and so forth. Now try to remember how those intermediate results were generated.

The third scenario has the downside of an awkward workflow. For a relational database, this involves the data scientist copying data over to her laptop and ingesting the data locally before additional queries can be run. Furthermore, this approach essentially requires maintaining two separate analytics stacks (one on the client and one on the server). Say,

she was playing with a new unstable branch of Spark on her laptop to test out a new feature (which hasn't been rolled out to production yet), but in the new branch, the API has changed slightly. What about the custom machine-learning library that her organization deploys on top of Spark – is the version running on the cluster the same as the one running on her laptop?

Now imagine an alternative: What if we had something like Spark.js, where we could transparently write Spark code in JavaScript? Fundamentally, this isn't any different from PySpark, which is based on Py4J ([www.py4j.org](http://www.py4j.org)), a bridge that enables Python programs running in a Python interpreter to dynamically access Java objects in a JVM. Similarly, Spark.js could be written with the help of Rhino ([www.mozilla.org/rhino](http://www.mozilla.org/rhino)), an open source implementation of JavaScript written entirely in Java.

In the current Spark API, “collect” and related methods materialize a resilient distributed dataset (RDD), the results of which are sent over to the client. In Spark.js, JVM objects could be transparently converted into native JavaScript objects, ready for subsequent manipulation and seamless integration with D3.js for visualization. All of this would run in the browser (any standards-compliant one, on any platform) or alternatively Node.js.

Wait, it gets even better. At this point, Node.js already runs your user-facing applications, right? Remember that complex extract, transform, and load (ETL) pipeline for pumping log data into Kafka (<http://kafka.apache.org>) and eventually HDFS? All that custom code for data cleaning, schema transformation, and so on can now be written in JavaScript. Developers often write log messages in JSON these days, right? Note what the “J” in JSON stands for! So, with everything in JavaScript, you can retain the convenience of what you were already doing for user-facing appli-

cations but gain seamless integration with the data collection and analytics stack. This is a win from the friction-reduction perspective and potentially the performance perspective as well – ETL pipelines in JavaScript might reduce the amount of unnecessary data transformation, thus decreasing latency and increasing throughput. Although textual JSON isn't particularly efficient as a storage format (even when compressed), the physical layout could be easily optimized using Parquet (<http://parquet.io>), which is essentially built around a JSON data model – thus reaping the benefits of columnar layouts. You can have your cake and eat it, too!

One final leap: Note that the actual execution of Big Data analytics still occurs on Spark, inside the JVM. Spark.js merely provides convenient JavaScript integration. Why don't we “go all the way” and replace Spark completely with a native JavaScript analytics engine? We wonder, what would a Big Data analytics platform written in the style of microservices look like? This is perhaps not an outrageous idea, because such a platform would be the logical endpoint of a “JavaScript everywhere” vision.

### A Small Step

To examine the feasibility of this “JavaScript everywhere” vision, we recently implemented a small prototype analytical RDBMS called *Afterburner* in JavaScript.<sup>6</sup> The entire system runs completely inside a browser with no external dependencies (or alternatively, in Node.js). Afterburner generates compiled query plans<sup>7,8</sup> that exploit two JavaScript features: typed arrays and asm.js, which we explain next.

Array objects in JavaScript can store elements of any type and aren't arrays in a traditional sense (compared to say, C), because consecutive elements might not be contiguous; furthermore, the array itself can dynamically grow and shrink. This flexibility makes the data structure

accessible for novice programmers but limits optimizations that JavaScript engines can perform both during compilation and at runtime. In the evolution of JavaScript, it became clear that the language needed more efficient methods to quickly manipulate binary data: typed arrays are the answer.

In conjunction with typed arrays, Afterburner takes advantage of asm.js, a strictly typed subset of JavaScript that's designed to be easily optimizable by an execution engine. A key feature of asm.js is the use of type hints, which essentially introduces a static type system while retaining backwards compatibility with “vanilla” JavaScript.

Any JavaScript block of code can request validation as valid asm.js via a special prologue directive, which happens when the source code is loaded. Validated asm.js code is amenable to ahead-of-time (AOT) compilation, in contrast to just-in-time (JIT) compilation with vanilla JavaScript. Executable code generated by AOT compilers can be quite efficient, through the removal of runtime type checks, operations on unboxed (that is, primitive) types, and the removal of garbage collection.

At a high level, Afterburner translates SQL into the string representation of an asm.js module (that is, the physical query plan), calls `eval` on the code, which triggers AOT compilation and links the module to the calling JavaScript client, and finally executes the module. The typed array storing all the tables (essentially, the entire database) is passed into the module as a parameter and the results are returned by the module.

Before summarizing some preliminary results, it's worthwhile to highlight how far JavaScript has come as an efficient execution platform, from its much-maligned performance in the early days. Today, modern browsers embed powerful JavaScript engines capable of running real-time collaboration tools and online multiplayer

games. One impressive feat is QuakeJS ([www.quakejs.com](http://www.quakejs.com)), a port of the classic first-person shooter Quake 3, that *runs completely in the browser*. The original C code was compiled into asm.js using Emscripten (<http://emscripten.org>), which takes LLVM bitcode and emits JavaScript as its target output. The game runs at a good frame rate and is definitely playable. As a rough heuristic, C code compiled to asm.js (which, recall, is just JavaScript) using Emscripten runs at about half the speed of the C program running natively (directly on Linux, for example). For many applications, the tradeoff of performance for ubiquitous execution inside a browser is worthwhile.

We evaluated Afterburner against the popular open source analytical database MonetDB,<sup>9</sup> using data from the TPC-H benchmark. At a scaling factor of 1 gigabyte, which yields a lineitem table with 6 million rows and an orders table with 1.5 million rows, experiments with a few simple queries show that our prototype achieves comparable performance to MonetDB running natively on the same machine. See our report for additional details.<sup>6</sup> Of course, we're quick to qualify that Afterburner is a prototype with limited functionality, but our evaluations affirm at least the *feasibility* of in-browser analytics using JavaScript.

### From Split Execution to the Connect Refrigerator

Recall that one of the advantages of JavaScript everywhere is eliminating the distinction between front- and back-end code; see, for example, discussion by Netflix.<sup>3</sup> Returning to our data scientist, she can perform some analytics tasks on the cluster, bring intermediate data into her browser, and continue working without interruption. Front-end code can be pushed to the back end, and back-end code can be brought to the front end, all seamlessly.

In such a scenario, the data scientist explicitly decides which parts of her

queries run where – the next logical step would be for a query optimizer to figure that out for her automatically. In fact, researchers have already explored this idea in the 1990s;<sup>10</sup> in some cases, the placement of query operators on the client can lead to better performance, and obviously, reduces load on the back end. JavaScript brings a fresh take on this decades-old idea of splitting query execution across the client and server: one of the challenges with this thread of previous work is the proper encapsulation of the execution context to facilitate operator placement. With JavaScript, this is substantially simplified.

A major selling point of JavaScript is its ubiquity: everywhere there's a browser, there's JavaScript. The browser might even be on a phone: we've gotten Afterburner to run on a mobile phone, just to show it's possible. One not-too-unrealistic scenario might be to run something like Afterburner on a tablet to manipulate data: spreadsheets already run on tablets today – how would this be substantially different? Imagine running a Spark job from the browser inside your tablet to pull some data from the cluster, getting on an airplane, and continuing to explore the data while completely offline. Today, you could do this by exporting data into a spreadsheet app on the tablet, which is just a variant of the aforementioned third scenario, with all of its awkwardness.

If you've accepted our story so far, then it's not a particularly big leap to imagine extending the "JavaScript everywhere" vision to the Internet of Things (IoT) future: for example, check out Node-RED (<http://nodered.org>). In the not-so-distant future, imagine returning to your fully connected house, asking yourself, "Why is it so hot on the second floor?" and issuing the following query from the browser console on your fridge:

```
SELECT room, temp FROM sensors
WHERE floor = 2 and time = NOW;
```

This translates into a distributed query plan that gets pushed to all the sensors around the house, the results of which are then gathered back at the fridge and rendered as a heatmap – all in JavaScript.

Don't laugh – something like this has already been developed, built, and deployed in the context of distributed sensor networks: it's called TinyDB.<sup>11</sup> The only difference here is that we propose to implement everything in JavaScript.

Consider another example: you're at the grocery store and wonder if you need to buy milk. Pull out your phone and ask your fridge:

```
SELECT COUNT(*) FROM fridge
WHERE item = "milk";
```

Of course, actual SQL queries are likely to be hidden behind a snazzy UI within an app (or connected via intelligent agents such as Siri or Cortana), but behind the scenes, it could all be running JavaScript.

In fact, this vision of the IoT has one significant advantage over current trends – it can potentially serve as a counterweight to the desire of companies to centralize everything. Nearly all IoT gadgets today, including personal fitness trackers, pipe all data streams into the cloud, and then offer users an interface (or API) to access their own data (which of course the companies can change at any time). In other words, they control all your personal data.

Just how creepy is this? Samsung's SmartTV comes with voice recognition that allows you to issue verbal commands to it. It's a cute trick, but comes with a warning to watch what you say in front of your TV:

*Please be aware that if your spoken words include personal or other sensitive information, that information will be among the data captured and transmitted to a third party.<sup>12</sup>*

Seriously? This is, literally, straight out of 1984.<sup>13</sup>

One major advantage of the “JavaScript everywhere” vision is that, by erasing the distinction between front- and back-end code, we potentially lower the bar (by decreasing the complexity) of building distributed solutions, which will help counteract the dangers of centralization. In other words, JavaScript can help re-decentralize the Web.<sup>14</sup> Of course, we aren’t so naive to think that companies will easily give up their data collection efforts (which lie at the heart of their business models), but the ability to seamlessly push queries down to physical devices provides a workable mechanism that might give users greater control over what is and isn’t transmitted to the cloud.

Generalizing a bit, we could envision an entirely different class of “verb-centric” APIs. Almost all APIs today are “noun-centric,” in that they hardcode the “verbs” in specific endpoints. For example, consider the common CRUD (create, read, update, delete) operations provided by many REST APIs: to invoke the API, the client supplies the “subject” (and sometimes the “object”) while the “verbs” are fixed (that is, hardcoded in the API endpoint itself). What if, instead, the API fixed the subject and object while letting the client specify the verb? That’s exactly what the aforementioned fridge query is doing – the client says, I know the data model that’s stored in a distributed fashion on the device (the crucial distinction here, as opposed to a centralized data warehouse) – here’s the verb, expressed as an SQL query that I want to run on the data. Give me back the results.

In fact, we’ve already been doing this for at least a decade – we’re just describing MapReduce! A verb-centric API is just another way of saying “move the processing to the data.” We ship the verb (that is, the mappers and reducers) over to the cluster, where the

code executes. Spark takes it to the next level by providing an interactive shell that allows closures (the verbs) to be dynamically serialized, shipped over to the cluster, and executed in a distributed manner. JavaScript makes this possible everywhere. We can just reuse existing infrastructure – it might offend the sensibilities of many developers, but what’s wrong with code shipping in textual form? It’s already being done billions of times per day – JavaScript code on the Web is distributed as plain text! What’s the alternative? Shipping VM images? Shipping Docker images? Both seem too heavyweight for widespread, democratized use.

Yes, there are many issues with verb-centric APIs that we’re mostly sweeping under the rug, but we can respond briefly to a few obvious ones. What about security if a client is shipping verbs? First, we can restrict clients to trusted sources – you’re probably not going to run an injection attack on your own fridge, in which case, perimeter defenses are a good start (a reasonable first-order approximation of how Hadoop security is handled these days). Second, this issue isn’t very different from JavaScript injection attacks, something that’s already well studied and continues to be actively researched. What about the exfiltration of potentially sensitive results from running the code? Leaving aside the “friendly clients” defense, this is a hot topic in security and privacy these days, which means that lots of smart people are thinking about it. Tying the challenges of JavaScript everywhere to existing research problems greatly increases the chances that we (collectively) will come up with workable solutions.

In many ways, JavaScript is like the hobbits of Middle Earth: both have been criticized to be slow, clumsy, and weak. They’re also known to be

very friendly (JavaScript is easy to learn). They live away from danger at the edge of the realms in lovely dwellings (D3.js visualizations), far from the troubles at the center of the action (the back end). However, when tested (Node.js), they have proven to be worthy. JavaScript, instead of successfully destroying the One Ring, might be successful in wielding it. Front ends, back ends, applications, and analytics: JavaScript will, in the light, bind them. ☐

## Acknowledgments

We thank Ashraf Aboulmaga and Todd Lipcon for comments on earlier drafts of this article.


## References

1. G. Bernhardt, “WAT,” *Destroy All Software*, software screencast, 2012; [www.destroyallsoftware.com/talks/wat](http://www.destroyallsoftware.com/talks/wat).
2. A. deBoor, “Gmail: Past, Present, and Future,” *Proc. Conf. Web Application Development*, 2010; [www.usenix.org/legacy/event/webapps10/tech/slides/deboor.pdf](http://www.usenix.org/legacy/event/webapps10/tech/slides/deboor.pdf).
3. K. Baxter, *Making Netflix.com Faster*, blog, 2015; <http://techblog.netflix.com/2015/08/making-netflix-com-faster.html>.
4. J. Harrell, *Node.js at PayPal*, blog, 2013; [www.paypal-engineering.com/2013/11/22/node-js-at-paypal](http://www.paypal-engineering.com/2013/11/22/node-js-at-paypal).
5. R. Paul, “A Behind-the-Scenes Look at LinkedIn’s Mobile Engineering,” *Ars Technica*, 2 Oct. 2012; <http://arstechnica.com/information-technology/2012/10/a-behind-the-scenes-look-at-linkedin-mobile-engineering>.
6. K. El Gebaly and J. Lin, “Afterburner: The Case for In-Browser Analytics,” 2016, <https://arxiv.org/pdf/1605.04035>.
7. K. Krikellas, S. Viglas, and M. Cintra, “Generating Code for Holistic Query Evaluation,” *Proc. 26th Int’l Conf. Data Eng.*, 2010, pp. 613–624.
8. T. Neumann, “Efficiently Compiling Efficient Query Plans for Modern Hardware,” *Proc. Very Large Data Bases*, vol. 4, no. 9, 2011, pp. 539–550.
9. P.A. Boncz, M. Zukowski, and N. Nes, “MonetDB/X100: Hyper-Pipelining Query Execution,” *Proc. 2nd Biennial Conf. Innovative Data Systems Research*, 2005, pp. 225–237.

10. M.J. Franklin, B.T. Jonsson, and D. Kossmann, "Performance Tradeoffs for Client-Server Query Processing," *Proc. 1996 ACM Sigmod Int'l Conf. Management of Data*, 1996, pp. 149–160.
11. S.R. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong, "TinyDB: An Acquisitional Query Processing System for Sensor Networks," *ACM Trans. Database Systems*, vol. 30, no. 1, 2005, pp. 122–173.
12. N. Lomas, "Today in Creepy Privacy Policies, Samsung's Eavesdropping TV," *Tech Crunch*, 2015; <http://techcrunch.com/2015/02/08/telescreen>.
13. P. Higgins, Twitter post, 8 Feb. 2015, 1:35 a.m.; <https://twitter.com/xor/status/564356757007261696>.
14. E. Mansour et al., "A Demonstration of the Solid Platform for Social Web Applications," *Proc. 25th Int'l Conf. Companion on World Wide Web*, 2016, pp. 223–226.

**Jimmy Lin** holds the David R. Cheriton Chair in the David R. Cheriton School of Computer Science at the University of Waterloo. His research lies at the intersection of information retrieval and natural language processing, with a particular focus on Big Data and large-scale distributed infrastructure for text processing. Lin has a PhD in electrical engineering and computer science from MIT. Contact him at [jimmylin@uwaterloo.ca](mailto:jimmylin@uwaterloo.ca).

**Kareem El Gebaly** is a PhD student in the David R. Cheriton School of Computer Science at the University of Waterloo. His research interests include data management and self-managing data analytics. El Gebaly has an MMath in computer science from the University of Waterloo. Contact him at [kareem.elgebaly@uwaterloo.ca](mailto:kareem.elgebaly@uwaterloo.ca).

 Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.

## ADVERTISER INFORMATION

### Advertising Personnel

Marian Anderson: Sr. Advertising Coordinator  
Email: [manderson@computer.org](mailto:manderson@computer.org)  
Phone: +1 714 816 2139 | Fax: +1 714 821 4010

Sandy Brown: Sr. Business Development Mgr.  
Email: [sbrown@computer.org](mailto:sbrown@computer.org)  
Phone: +1 714 816 2144 | Fax: +1 714 821 4010

### Advertising Sales Representatives (display)

Central, Northwest, Far East:  
Eric Kincaid  
Email: [e.kincaid@computer.org](mailto:e.kincaid@computer.org)  
Phone: +1 214 673 3742  
Fax: +1 888 886 8599

Northeast, Midwest, Europe, Middle East:  
Ann & David Schissler  
Email: [a.schissler@computer.org](mailto:a.schissler@computer.org), [d.schissler@computer.org](mailto:d.schissler@computer.org)  
Phone: +1 508 394 4026  
Fax: +1 508 394 1707

Southwest, California:  
Mike Hughes  
Email: [mikehughes@computer.org](mailto:mikehughes@computer.org)  
Phone: +1 805 529 6790

Southeast:  
Heather Buonadies  
Email: [h.buonadies@computer.org](mailto:h.buonadies@computer.org)  
Phone: +1 973 304 4123  
Fax: +1 973 585 7071

### Advertising Sales Representatives (Classified Line)

Heather Buonadies  
Email: [h.buonadies@computer.org](mailto:h.buonadies@computer.org)  
Phone: +1 973 304 4123  
Fax: +1 973 585 7071

### Advertising Sales Representatives (Jobs Board)

Heather Buonadies  
Email: [h.buonadies@computer.org](mailto:h.buonadies@computer.org)  
Phone: +1 973 304 4123  
Fax: +1 973 585 7071