O'REILLY®

2015 Edition

Big Data Now

Current Perspectives from O'Reilly Media

Strata+ Hadoop WORLD

Make Data Work

strataconf.com

Presented by O'Reilly and Cloudera, Strata + Hadoop World is where cutting-edge data science and new business fundamentals intersect—and merge.

- Learn business applications of data technologies
- Develop new skills through trainings and in-depth tutorials
- Connect with an international community of thousands who work with data

Job # 15420

Big Data Now

2015 Edition

O'Reilly Media, Inc.

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Big Data Now: 2015 Edition
by O'Reilly Media, Inc.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Nicole Tache
Production Editor: Leia Poritz
Copyeditor: Jasmine Kwityn
Proofreader: Kim Cofer

Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January 2016: First Edition

Revision History for the First Edition
2016-01-12: First Release

978-1-491-95057-9

[LSI]

Table of Contents

Introduction v

1. Data-Driven Cultures 1
How an Enterprise Begins Its Data Journey 1
Improving Corporate Planning Through Insight Generation 5
On Leadership 7
Embracing Failure and Learning from the Impostor Syndrome 10
The Key to Agile Data Science: Experimentation 12

2. Data Science 19
What It Means to "Go Pro" in Data Science 20
Graphs in the World: Modeling Systems as Networks 28
Let's Build Open Source Tensor Libraries for Data Science 37

3. Data Pipelines 43
Building and Deploying Large-Scale Machine Learning Pipelines 43
Three Best Practices for Building Successful Data Pipelines 48
The Log: The Lifeblood of Your Data Pipeline 55
Validating Data Models with Kafka-Based Pipelines 61

4. Big Data Architecture and Infrastructure 65
Lessons from Next-Generation Data-Wrangling Tools 66
Why the Data Center Needs an Operating System 68
A Tale of Two Clusters: Mesos and YARN 74
The Truth About MapReduce Performance on SSDs 81

Accelerating Big Data Analytics Workloads with Tachyon 87

5. The Internet of Things and Real Time. 95
A Real-Time Processing Revival 96
Improving on the Lambda Architecture for Streaming Analysis 98
How Intelligent Data Platforms Are Powering Smart Cities 105
The Internet of Things Has Four Big Data Problems 107

6. Applications of Big Data. 111
How Trains Are Becoming Data Driven 112
Multimodel Database Case Study: Aircraft Fleet Maintenance 115
Big Data Is Changing the Face of Fashion 127
The Original Big Data Industry 128

7. Security, Ethics, and Governance. 131
The Security Infusion 132
We Need Open and Vendor-Neutral Metadata Services 136
What the IoT Can Learn from the Healthcare Industry 138
There Is Room for Global Thinking in IoT Data Privacy Matters 141
Five Principles for Applying Data Science for Social Good 144

Introduction

Data-driven tools are all around us—they filter our email, they recommend professional connections, they track our music preferences, and they advise us when to tote umbrellas. The more ubiquitous these tools become, the more data we as a culture produce, and the more data there is to parse, store, and analyze for insight. During a keynote talk at Strata + Hadoop World 2015 in New York, Dr. Timothy Howes, chief technology officer at ClearStory Data, said that we can expect to see a 4,300% increase in annual data generated by 2020. But this striking observation isn't necessarily new.

What *is* new are the enhancements to data-processing frameworks and tools—enhancements to increase speed, efficiency, and intelligence (in the case of machine learning) to pace the growing volume and variety of data that is generated. And companies are increasingly eager to highlight data preparation and business insight capabilities in their products and services.

What is also new is the rapidly growing user base for big data. According to Forbes, 2014 saw a 123.60% increase in demand for information technology project managers with big data expertise, and an 89.8% increase for computer systems analysts. In addition, we anticipate we'll see more data analysis tools that non-programmers can use. And businesses will maintain their sharp focus on using data to generate insights, inform decisions, and kick-start innovation. Big data analytics is not the domain of a handful of trailblazing companies; it's a common business practice. Organizations of all sizes, in all corners of the world, are asking the same fundamental questions: How can we collect and use data successfully?

Who can help us establish an effective working relationship with data?

Big Data Now recaps the trends, tools, and applications we've been talking about over the past year. This collection of O'Reilly blog posts, authored by leading thinkers and professionals in the field, has been grouped according to unique themes that garnered significant attention in 2015:

- Data-driven cultures (Chapter 1)
- Data science (Chapter 2)
- Data pipelines (Chapter 3)
- Big data architecture and infrastructure (Chapter 4)
- The Internet of Things and real time (Chapter 5)
- Applications of big data (Chapter 6)
- Security, ethics, and governance (Chapter 7)

CHAPTER 1

Data-Driven Cultures

What does it mean to be a truly data-driven culture? What tools and skills are needed to adopt such a mindset? DJ Patil and Hilary Mason cover this topic in O'Reilly's report "Data Driven," and the collection of posts in this chapter address the benefits and challenges that data-driven cultures experience—from generating invaluable insights to grappling with overloaded enterprise data warehouses.

First, Rachel Wolfson offers a solution to address the challenges of data overload, rising costs, and the skills gap. Evangelos Simoudis then discusses how data storage and management providers are becoming key contributors for insight as a service. Q Ethan McCallum traces the trajectory of his career from software developer to team leader, and shares the knowledge he gained along the way. Alice Zheng explores the impostor syndrome, and the byproducts of frequent self-doubt and a perfectionist mentality. Finally, Jerry Overton examines the importance of agility in data science and provides a real-world example of how a short delivery cycle fosters creativity.

How an Enterprise Begins Its Data Journey

by Rachel Wolfson

You can read this post on oreilly.com here.

As the amount of data continues to double in size every two years, organizations are struggling more than ever before to manage,

ingest, store, process, transform, and analyze massive data sets. It has become clear that getting started on the road to using data successfully can be a difficult task, especially with a growing number of new data sources, demands for fresher data, and the need for increased processing capacity. In order to advance operational efficiencies and drive business growth, however, organizations must address and overcome these challenges.

In recent years, many organizations have heavily invested in the development of enterprise data warehouses (EDW) to serve as the central data system for reporting, extract/transform/load (ETL) processes, and ways to take in data (data ingestion) from diverse databases and other sources both inside and outside the enterprise. Yet, as the volume, velocity, and variety of data continues to increase, already expensive and cumbersome EDWs are becoming overloaded with data. Furthermore, traditional ETL tools are unable to handle all the data being generated, creating bottlenecks in the EDW that result in major processing burdens.

As a result of this overload, organizations are now turning to open source tools like Hadoop as cost-effective solutions to offloading data warehouse processing functions from the EDW. While Hadoop can help organizations lower costs and increase efficiency by being used as a complement to data warehouse activities, most businesses still lack the skill sets required to deploy Hadoop.

Where to Begin?

Organizations challenged with overburdened EDWs need solutions that can offload the heavy lifting of ETL processing from the data warehouse to an alternative environment that is capable of managing today's data sets. The first question is always *How can this be done in a simple, cost-effective manner that doesn't require specialized skill sets?*

Let's start with Hadoop. As previously mentioned, many organizations deploy Hadoop to offload their data warehouse processing functions. After all, Hadoop is a cost-effective, highly scalable platform that can store volumes of structured, semi-structured, and unstructured data sets. Hadoop can also help accelerate the ETL process, while significantly reducing costs in comparison to running ETL jobs in a traditional data warehouse. However, while the benefits of Hadoop are appealing, the complexity of this platform contin-

ues to hinder adoption at many organizations. It has been our goal to find a better solution.

Using Tools to Offload ETL Workloads

One option to solve this problem comes from a combined effort between Dell, Intel, Cloudera, and Syncsort. Together they have developed a preconfigured offloading solution that enables businesses to capitalize on the technical and cost-effective features offered by Hadoop. It is an ETL offload solution that delivers a use case–driven Hadoop Reference Architecture that can augment the traditional EDW, ultimately enabling customers to offload ETL workloads to Hadoop, increasing performance, and optimizing EDW utilization by freeing up cycles for analysis in the EDW.

The new solution combines the Hadoop distribution from Cloudera with a framework and tool set for ETL offload from Syncsort. These technologies are powered by Dell networking components and Dell PowerEdge R series servers with Intel Xeon processors.

The technology behind the ETL offload solution simplifies data processing by providing an architecture to help users optimize an existing data warehouse. So, how does the technology behind all of this actually work?

The ETL offload solution provides the Hadoop environment through Cloudera Enterprise software. The Cloudera Distribution of Hadoop (CDH) delivers the core elements of Hadoop, such as scalable storage and distributed computing, and together with the software from Syncsort, allows users to reduce Hadoop deployment to weeks, develop Hadoop ETL jobs in a matter of hours, and become fully productive in days. Additionally, CDH ensures security, high availability, and integration with the large set of ecosystem tools.

Syncsort DMX-h software is a key component in this reference architecture solution. Designed from the ground up to run efficiently in Hadoop, Syncsort DMX-h removes barriers for mainstream Hadoop adoption by delivering an end-to-end approach for shifting heavy ETL workloads into Hadoop, and provides the connectivity required to build an enterprise data hub. For even tighter integration and accessibility, DMX-h has monitoring capabilities integrated directly into Cloudera Manager.

With Syncsort DMX-h, organizations no longer have to be equipped with MapReduce skills and write mountains of code to take advantage of Hadoop. This is made possible through intelligent execution that allows users to graphically design data transformations and focus on business rules rather than underlying platforms or execution frameworks. Furthermore, users no longer have to make application changes to deploy the same data flows on or off of Hadoop, on premise, or in the cloud. This future-proofing concept provides a consistent user experience during the process of collecting, blending, transforming, and distributing data.

Additionally, Syncsort has developed SILQ, a tool that facilitates understanding, documenting, and converting massive amounts of SQL code to Hadoop. SILQ takes an SQL script as an input and provides a detailed flow chart of the entire data stream, mitigating the need for specialized skills and greatly accelerating the process, thereby removing another roadblock to offloading the data warehouse into Hadoop.

Dell PowerEdge R730 servers are then used for infrastructure nodes, and Dell PowerEdge R730xd servers are used for data nodes.

The Path Forward

Offloading massive data sets from an EDW can seem like a major barrier to organizations looking for more effective ways to manage their ever-increasing data sets. Fortunately, businesses can now capitalize on ETL offload opportunities with the correct software and hardware required to shift expensive workloads and associated data from overloaded enterprise data warehouses to Hadoop.

By selecting the right tools, organizations can make better use of existing EDW investments by reducing the costs and resource requirements for ETL.

This post is part of a collaboration between O'Reilly, Dell, and Intel. See our statement of editorial independence.

Improving Corporate Planning Through Insight Generation

by Evangelos Simoudis

You can read this post on oreilly.com here.

Contrary to what many believe, insights are difficult to identify and effectively apply. As the difficulty of insight generation becomes apparent, we are starting to see companies that offer insight generation as a service.

Data storage, management, and analytics are maturing into commoditized services, and the companies that provide these services are well positioned to provide insight on the basis not just of data, but data access and other metadata patterns.

Companies like DataHero and Host Analytics are paving the way in the insight-as-a-service (IaaS) space.[1] Host Analytics' initial product offering was a cloud-based Enterprise Performance Management (EPM) suite, but far more important is what it is now enabling for the enterprise: It has moved from being an EPM company to being an insight generation company. This post reviews a few of the trends that have enabled IaaS and discusses the general case of using a software-as-a-service (SaaS) EPM solution to corral data and deliver IaaS as the next level of product.

Insight generation is the identification of novel, interesting, plausible, and understandable relations among elements of a data set that (a) lead to the formation of an action plan, and (b) result in an improvement as measured by a set of key performance indicators (KPIs). The evaluation of the set of identified relations to establish an insight, and the creation of an action plan associated with a particular insight or insights, needs to be done within a particular context and necessitates the use of domain knowledge.

IaaS refers to action-oriented, analytics-driven, cloud-based solutions that generate insights and associated action plans. IaaS is a *distinct* layer of the cloud stack (I've previously discussed IaaS in "Defining Insight" and "Insight Generation"). In the case of Host Analytics, its EPM solution integrates a customer's financial plan-

1 Full disclosure: Host Analytics is one of my portfolio companies.

ning data with actuals from its Enterprise Resource Planning (ERP) applications (e.g., SAP or NetSuite, and relevant syndicated and open source data), creating an IaaS offering that complements their existing solution. EPM, in other words, is not just a matter of streamlining data provisions within the enterprise; it's an opportunity to provide a true insight-generation solution.

EPM has evolved as a category much like the rest of the data industry: from in-house solutions for enterprises to off-the-shelf but hard-to-maintain software to SaaS and cloud-based storage and access. Throughout this evolution, improving the financial planning, forecasting, closing, and reporting processes continues to be a priority for corporations. EPM started, as many applications do, in Excel but gave way to automated solutions starting about 20 years ago with the rise of vendors like Hyperion Solutions. Hyperion's Essbase was the first to use OLAP technology to perform both traditional financial analysis as well as line-of-business analysis. Like many other strategic enterprise applications, EPM started moving to the cloud a few years ago. As such, a corporation's financial data is now available to easily combine with other data sources, open source and proprietary, and deliver insight-generating solutions.

The rise of big data—and the access and management of such data by SaaS applications, in particular—is enabling the business user to access internal and external data, including public data. As a result, it has become possible to access the data that companies really care about, everything from the internal financial numbers and sales pipelines to external benchmarking data as well as data about best practices. Analyzing this data to derive insights is critical for corporations for two reasons. First, great companies require agility, and want to use all the data that's available to them. Second, company leadership and corporate boards are now requiring more detailed analysis.

Legacy EPM applications historically have been centralized in the finance department. This led to several different operational "data hubs" existing within each corporation. Because such EPM solutions didn't effectively reach all departments, critical corporate information was "siloed," with critical information like CRM data housed separately from the corporate financial plan. This has left the departments to analyze, report, and deliver their data to corporate using manually integrated Excel spreadsheets that are incredibly inefficient to manage and usually require significant time to under-

stand the data's source and how they were calculated rather than what to do to drive better performance.

In most corporations, this data remains disconnected. Understanding the ramifications of this barrier to achieving true enterprise performance management, IaaS applications are now stretching EPM to incorporate operational functions like marketing, sales, and services into the planning process. IaaS applications are beginning to integrate data sets from those departments to produce a more comprehensive corporate financial plan, improving the planning process and helping companies better realize the benefits of IaaS. In this way, the CFO, VP of sales, CMO, and VP of services can clearly see the actions that will improve performance in their departments, and by extension, elevate the performance of the entire corporation.

On Leadership

by Q Ethan McCallum

You can read this post on oreilly.com here.

Over a recent dinner with Toss Bhudvanbhen, our conversation meandered into discussion of how much our jobs had changed since we entered the workforce. We started during the dot-com era. Technology was a relatively young field then (frankly, it still is), so there wasn't a well-trodden career path. We just went with the flow.

Over time, our titles changed from "software developer," to "senior developer," to "application architect," and so on, until one day we realized that we were writing less code but sending more emails; attending fewer code reviews but more meetings; and were less worried about how to implement a solution, but more concerned with defining the problem and why it needed to be solved. We had somehow taken on leadership roles.

We've stuck with it. Toss now works as a principal consultant at Pariveda Solutions and my consulting work focuses on strategic matters around data and technology.

The thing is, we were never formally trained as management. We just learned along the way. What helped was that we'd worked with some amazing leaders, people who set great examples for us and recognized our ability to understand the bigger picture.

Perhaps you're in a similar position: Yesterday you were called "senior developer" or "data scientist" and now you've assumed a technical leadership role. You're still sussing out what this battlefield promotion really means—or, at least, you would do that if you had the time. We hope the high points of our conversation will help you on your way.

Bridging Two Worlds

You likely gravitated to a leadership role because you can live in two worlds: You have the technical skills to write working code and the domain knowledge to understand how the technology fits the big picture. Your job now involves keeping a foot in each camp so you can translate the needs of the business to your technical team, and vice versa. Your value-add is knowing when a given technology solution will really solve a business problem, so you can accelerate decisions and smooth the relationship between the business and technical teams.

Someone Else Will Handle the Details

You're spending more time in meetings and defining strategy, so you'll have to delegate technical work to your team. Delegation is not about giving orders; it's about clearly communicating your goals so that someone else can do the work when you're not around. Which is great, because you won't often be around. (If you read between the lines here, delegation is also about you caring more about the high-level result than minutiae of implementation details.) How you communicate your goals depends on the experience of the person in question: You can offer high-level guidance to senior team members, but you'll likely provide more guidance to the junior staff.

Here to Serve

If your team is busy running analyses or writing code, what fills your day? Your job is to do whatever it takes to make your team successful. That division of labor means you're responsible for the pieces that your direct reports can't or don't want to do, or perhaps don't even know about: sales calls, meetings with clients, defining scope with the product team, and so on. In a larger company, that may also mean leveraging your internal network or using your

seniority to overcome or circumvent roadblocks. Your team reports to you, but you work for them.

Thinking on Your Feet

Most of your job will involve making decisions: what to do, whether to do it, when to do it. You will often have to make those decisions based on imperfect information. As an added treat, you'll have to decide in a timely fashion: People can't move until you've figured out where to go. While you should definitely seek input from your team—they're doing the hands-on work, so they are closer to the action than you are—the ultimate decision is yours. As is the responsibility for a mistake. Don't let that scare you, though. Bad decisions are learning experiences. A bad decision beats indecision any day of the week.

Showing the Way

The best part of leading a team is helping people understand and meet their career goals. You can see when someone is hungry for something new and provide them opportunities to learn and grow. On a technical team, that may mean giving people greater exposure to the business side of the house. Ask them to join you in meetings with other company leaders, or take them on sales calls. When your team succeeds, make sure that you credit them—by name!—so that others may recognize their contribution. You can then start to delegate more of your work to team members who are hungry for more responsibility.

The bonus? This helps you to develop your succession plan. You see, leadership is also temporary. Sooner or later, you'll have to move on, and you will serve your team and your employer well by planning for your exit early on.

Be the Leader You Would Follow

We'll close this out with the most important lesson of all: Leadership isn't a title that you're given, but a role that you assume and that others recognize. You have to earn your team's respect by making your best possible decisions and taking responsibility when things go awry. Don't worry about being lost in the chaos of this new role. Look to great leaders with whom you've worked in the past, and their lessons will guide you.

Embracing Failure and Learning from the Impostor Syndrome

by Alice Zheng

You can read this post on oreilly.com here.

Lately, there has been a slew of media coverage about the *impostor syndrome*. Many columnists, bloggers, and public speakers have spoken or written about their own struggles with the impostor syndrome. And original psychological research on the impostor syndrome has found that out of every five successful people, two consider themselves a fraud.

I'm certainly no stranger to the sinking feeling of being out of place. During college and graduate school, it often seemed like everyone else around me was sailing through to the finish line, while I alone lumbered with the weight of programming projects and mathematical proofs. This led to an ongoing self-debate about my choice of a major and profession. One day, I noticed myself reading the same sentence over and over again in a textbook; my eyes were looking at the text, but my mind was saying *Why aren't you getting this yet? It's so simple. Everybody else gets it. What's wrong with you?*

When I look back on those years, I have two thoughts: first, *That was hard*, and second, *What a waste of perfectly good brain cells! I could have done so many cool things if I had not spent all that time doubting myself.*

But one can't simply snap out of the impostor syndrome. It has a variety of causes, and it's sticky. I was brought up with the idea of holding myself to a high standard, to measure my own progress against others' achievements. Falling short of expectations is supposed to be a great motivator for action…or is it?

In practice, measuring one's own worth against someone else's achievements can hinder progress more than it helps. It is a flawed method. I have a mathematical analogy for this: When we compare our position against others, we are comparing the static value of functions. But what determines the global optimum of a function are its derivatives. The first derivative measures the *speed of change*, the second derivative measures *how much the speed picks up over time*, and so on. How much we can achieve tomorrow is not just determined by where we are today, but how fast we are learning,

changing, and adapting. The rate of change is much more important than a static snapshot of the current position. And yet, we fall into the trap of letting the static snapshots define us.

Computer science is a discipline where the rate of change is particularly important. For one thing, it's a fast-moving and relatively young field. New things are always being invented. Everyone in the field is continually learning new skills in order to keep up. What's important today may become obsolete tomorrow. Those who stop learning, stop being relevant.

Even more fundamentally, software programming is about tinkering, and tinkering involves failures. This is why the hacker mentality is so prevalent. We learn by doing, and failing, and re-doing. We learn about good designs by iterating over initial bad designs. We work on pet projects where we have no idea what we are doing, but that teach us new skills. Eventually, we take on bigger, real projects.

Perhaps this is the crux of my position: I've noticed a cautiousness and an aversion to failure in myself and many others. I find myself wanting to wrap my mind around a project and perfectly understand its ins and outs before I feel comfortable diving in. I want to get it right the first time. Few things make me feel more powerless and incompetent than a screen full of cryptic build errors and stack traces, and part of me wants to avoid it as much as I can.

The thing is, everything about computers is imperfect, from software to hardware, from design to implementation. Everything up and down the stack breaks. The ecosystem is complicated. Components interact with each other in weird ways. When something breaks, fixing it sometimes requires knowing how different components interact with each other; other times it requires superior Googling skills. The only way to learn the system is to break it and fix it. It is impossible to wrap your mind around the stack in one day: application, compiler, network, operating system, client, server, hardware, and so on. And one certainly can't grok it by standing on the outside as an observer.

Further, many computer science programs try to teach their students computing concepts on the first go: recursion, references, data structures, semaphores, locks, and so on. These are beautiful, important concepts. But they are also very abstract and inaccessible by themselves. They also don't instruct students on how to succeed in real software engineering projects. In the courses I took, program-

ming projects constituted a large part, but they were included as a way of illustrating abstract concepts. You still needed to parse through the concepts to pass the course. In my view, the ordering should be reversed, especially for beginners. Hands-on practice with programming projects should be the primary mode of teaching; concepts and theory should play a secondary, supporting role. It should be made clear to students that mastering all the concepts is not a prerequisite for writing a kick-ass program.

In some ways, all of us in this field are impostors. No one knows everything. The only way to progress is to dive in and start doing. Let us not measure ourselves against others, or focus on how much we don't yet know. Let us measure ourselves by how much we've learned since last week, and how far we've come. Let us learn through playing and failing. The impostor syndrome can be a great teacher. It teaches us to love our failures and keep going.

O'Reilly's 2015 Edition of Women in Data *reveals inspiring success stories from four women working in data across the European Union, and features interviews with 19 women who are central to data businesses.*

The Key to Agile Data Science: Experimentation

by Jerry Overton

You can read this post on oreilly.com here.

I lead a research team of data scientists responsible for discovering insights that generate market and competitive intelligence for our company, Computer Sciences Corporation (CSC). We are a busy group. We get questions from all different areas of the company and it's important to be agile.

The nature of data science is experimental. You don't know the answer to the question asked of you—or even if an answer exists. You don't know how long it will take to produce a result or how much data you need. The easiest approach is to just come up with an idea and work on it until you have something. But for those of us with deadlines and expectations, that approach doesn't fly. Companies that issue you regular paychecks usually want insight into your progress.

This is where being agile matters. An agile data scientist works in small iterations, pivots based on results, and learns along the way. Being agile doesn't guarantee that an idea will succeed, but it does decrease the amount of time it takes to spot a dead end. Agile data science lets you deliver results on a regular basis and it keeps stakeholders engaged.

The key to agile data science is delivering data products in defined time boxes—say, two- to three-week sprints. Short delivery cycles force us to be creative and break our research into small chunks that can be tested using minimum viable experiments. We deliver something tangible after almost every sprint for our stakeholders to review and give us feedback. Our stakeholders get better visibility into our work, and we learn early on if we are on track.

This approach might sound obvious, but it isn't always natural for the team. We have to get used to working on just enough to meet stakeholders' needs and resist the urge to make solutions perfect before moving on. After we make something work in one sprint, we make it better in the next only if we can find a really good reason to do so.

An Example Using the Stack Overflow Data Explorer

Being an agile data scientist sounds good, but it's not always obvious how to put the theory into everyday practice. In business, we are used to thinking about things in terms of tasks, but the agile data scientist has to be able to convert a task-oriented approach into an experiment-oriented approach. Here's a recent example from my personal experience.

Our CTO is responsible for making sure the company has the next-generation skills we need to stay competitive—that takes data. We have to know what skills are hot and how difficult they are to attract and retain. Our team was given the task of categorizing key skills by how important they are, and by how rare they are (see Figure 1-1).

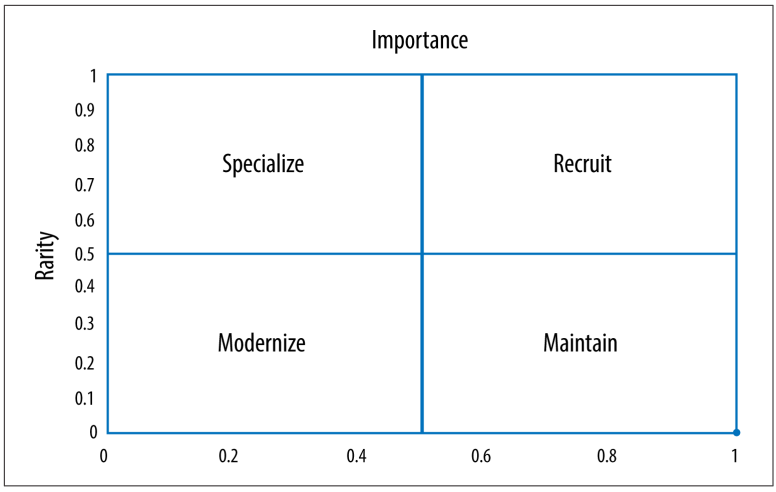


Figure 1-1. Skill categorization (image courtesy of Jerry Overton)

We already developed the ability to categorize key skills as important or not. By mining years of CIO survey results, social media sites, job boards, and internal HR records, we could produce a list of the skills most needed to support any of CSC's IT priorities. For example, the following is a list of programming language skills with the highest utility across all areas of the company:

Programming language	Importance (0–1 scale)
Java	1
SQL	0.4
Python	0.3
C#	0.2
C++	0.1
Perl	0.1

Note that this is a composite score for all the different technology domains we considered. The importance of Python, for example, varies a lot depending on whether or not you are hiring for a data scientist or a mainframe specialist.

For our top skills, we had the "importance" dimension, but we still needed the "abundance" dimension. We considered purchasing IT survey data that could tell us how many IT professionals had a

particular skill, but we couldn't find a source with enough breadth and detail. We considered conducting a survey of our own, but that would be expensive and time consuming. Instead, we decided to take a step back and perform an agile experiment.

Our goal was to find the relative number of technical professionals with a certain skill. Perhaps we could estimate that number based on activity within a technical community. It seemed reasonable to assume that the more people who have a skill, the more you will see helpful posts in communities like Stack Overflow. For example, if there are twice as many Java programmers as Python programmers, you should see about twice as many helpful Java programmer posts as Python programmer posts. Which led us to a hypothesis:

> *You can predict the relative number of technical professionals with a certain IT skill based on the relative number of helpful contributors in a technical community.*

We looked for the fastest, cheapest way to test the hypothesis. We took a handful of important programming skills and counted the number of unique contributors with posts rated above a certain threshold. We ran this query in the Stack Overflow Data Explorer:

```
SELECT
Count(DISTINCT Users.Id),
Tags.TagName as Tag_Name
FROM
Users, Posts, PostTags, Tags
WHERE
Posts.OwnerUserId = Users.Id AND
PostTags.PostId = Posts.Id AND
Tags.Id = PostTags.TagId AND
Posts.Score > 15 AND
Posts.CreationDate BETWEEN '1/1/2012' AND '1/1/2015' AND
 Tags.TagName  IN  ('python',  'r',  'java',  'perl',  'sql',
'c#', 'c++')
GROUP BY
Tags.TagName
```

Which gave us these results:

Programming language	Unique contributors	Scaled value (0–1)
Java	2,276	1.00
C#	1,868	0.82
C++	1,529	0.67
Python	1,380	0.61
SQL	314	0.14
Perl	70	0.03

We converted the scores according to a linear scale with the top score mapped to 1 and the lowest score being 0. Considering a skill to be "plentiful" is a relative thing. We decided to use the skill with the highest population score as the standard. At first glance, these results seemed to match our intuition, but we needed a simple, objective way of cross-validating the results. We considered looking for a targeted IT professional survey, but decided to perform a simple LinkedIn people search instead. We went into LinkedIn, typed a programming language into the search box, and recorded the number of people with that skill:

Programming language	LinkedIn population (M)	Scaled value (0–1)
Java	5.2	1.00
C#	4.6	0.88
C++	3	0.58
Python	1.7	0.33
SQL	1	0.19
Perl	0.5	0.10

Some of the experiment's results matched the cross-validation, but some were way off. The Java and C++ population scores predicted by the experiment matched pretty closely with the validation. But the experiment predicted that SQL would be one of the rarest skills, while the LinkedIn search told us that it is the most plentiful. This discrepancy makes sense. Foundational skills, such as SQL, that have been around a while will have a lot of practitioners, but are unlikely to be a hot topic of discussion. By the way, adjusting the allowable post creation dates made little difference to the relative outcome.

We couldn't confirm the hypothesis, but we learned something valuable. Why not just use the number of people that show up in the LinkedIn search as the measure of our population with the particular skill? We have to build the population list by hand, but that kind of grunt work is the cost of doing business in data science. Combining the results of LinkedIn searches with our previous analysis of skills importance, we can categorize programming language skills for the company, as shown in Figure 1-2.

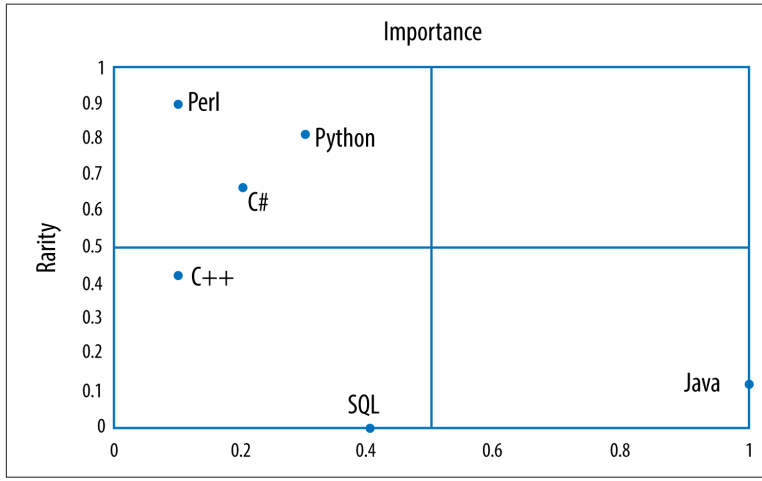


Figure 1-2. Programming language skill categorization (image courtesy of Jerry Overton)

Lessons Learned from a Minimum Viable Experiment

The entire experiment, from hypothesis to conclusion, took just three hours to complete. Along the way, there were concerns about which Stack Overflow contributors to include, how to define a helpful post, and the allowable sizes of technical communities—the list of possible pitfalls went on and on. But we were able to slice through the noise and stay focused on what mattered by sticking to a basic hypothesis and a minimum viable experiment.

Using simple tests and minimum viable experiments, we learned enough to deliver real value to our stakeholders in a very short amount of time. No one is getting hired or fired based on these results, but we can now recommend to our stakeholders strategies for getting the most out of our skills. We can recommend targets for recruiting and strategies for prioritizing talent development efforts.

Best of all, I think, we can tell our stakeholders how these priorities should change depending on the technology domain.

CHAPTER 2
Data Science

The term "data science" connotes opportunity and excitement. Organizations across the globe are rushing to build data science teams. The 2015 version of the Data Science Salary Survey reveals that usage of Spark and Scala has skyrocketed since 2014, and their users tend to earn more. Similarly, organizations are investing heavily in a variety of tools for their data science toolkit, including Hadoop, Spark, Kafka, Cassandra, D3, and Tableau—and the list keeps growing. Machine learning is also an area of tremendous innovation in data science—see Alice Zheng's report "Evaluating Machine Learning Models," which outlines the basics of model evaluation, and also dives into evaluation metrics and A/B testing.

So, where are we going? In a keynote talk at Strata + Hadoop World San Jose, US Chief Data Scientist DJ Patil provides a unique perspective of the future of data science in terms of the federal government's three areas of immediate focus: using medical and genomic data to accelerate discovery and improve treatments, building "game changing" data products on top of thousands of open data sets, and working in an ethical manner to ensure data science protects privacy.

This chapter's collection of blog posts reflects some hot topics related to the present and the future of data science. First, Jerry Overton takes a look at what it means to be a professional data science programmer, and explores best practices and commonly used tools. Russell Jurney then surveys a series of networks, including LinkedIn InMaps, and discusses what can be inferred when visualizing data in networks. Finally, Ben Lorica observes the reasons why

tensors are generating interest—speed, accuracy, scalability—and details recent improvements in parallel and distributed computing systems.

What It Means to "Go Pro" in Data Science

by Jerry Overton

You can read this post on oreilly.com here.

My experience of being a data scientist is not at all like what I've read in books and blogs. I've read about data scientists working for digital superstar companies. They sound like heroes writing automated (near sentient) algorithms constantly churning out insights. I've read about MacGyver-like data scientist hackers who save the day by cobbling together data products from whatever raw material they have around.

The data products my team creates are not important enough to justify huge enterprise-wide infrastructures. It's just not worth it to invest in hyper-efficient automation and production control. On the other hand, our data products influence important decisions in the enterprise, and it's important that our efforts scale. We can't afford to do things manually all the time, and we need efficient ways of sharing results with tens of thousands of people.

There are a lot of us out there—the "regular" data scientists; we're more organized than hackers but with no need for a superhero-style data science lair. A group of us met and held a speed ideation event, where we brainstormed on the best practices we need to write solid code. This article is a summary of the conversation and an attempt to collect our knowledge, distill it, and present it in one place.

Going Pro

Data scientists need software engineering skills—just not all the skills a professional software engineer needs. I call data scientists with essential data product engineering skills "professional" data science programmers. Professionalism isn't a possession like a certification or hours of experience; I'm talking about professionalism as an approach. Professional data science programmers are self-correcting in their creation of data products. They have general strategies for recognizing where their work sucks and correcting the problem.

The professional data science programmer has to turn a hypothesis into software capable of testing that hypothesis. Data science programming is unique in software engineering because of the types of problems data scientists tackle. The big challenge is that the nature of data science is experimental. The challenges are often difficult, and the data is messy. For many of these problems, there is no known solution strategy, the path toward a solution is not known ahead of time, and possible solutions are best explored in small steps. In what follows, I describe general strategies for a disciplined, productive trial and error: breaking problems into small steps, trying solutions, and making corrections along the way.

Think Like a Pro

To be a professional data science programmer, you have to know more than how the systems are structured. You have to know how to design a solution, you have to be able to recognize when you have a solution, and you have to be able to recognize when you don't fully understand your solution. That last point is essential to being self-correcting. When you recognize the conceptual gaps in your approach, you can fill them in yourself. To design a data science solution in a way that you can be self-correcting, I've found it useful to follow the basic process of look, see, imagine, and show:

Step 1: Look
: Start by scanning the environment. Do background research and become aware of all the pieces that might be related to the problem you are trying to solve. Look at your problem in as much breadth as you can. Get visibility to as much of your situation as you can and collect disparate pieces of information.

Step 2: See
: Take the disparate pieces you discovered and chunk them into abstractions that correspond to elements of the blackboard pattern. At this stage, you are casting elements of the problem into meaningful, technical concepts. Seeing the problem is a critical step for laying the groundwork for creating a viable design.

Step 3: Imagine
: Given the technical concepts you see, imagine some implementation that moves you from the present to your target state. If you can't imagine an implementation, then you probably missed something when you looked at the problem.

Step 4: Show

Explain your solution first to yourself, then to a peer, then to your boss, and finally to a target user. Each of these explanations need only be just formal enough to get your point across: a water-cooler conversation, an email, a 15-minute walk-through. *This is the most important regular practice in becoming a self-correcting professional data science programmer*. If there are any holes in your approach, they'll most likely come to light when you try to explain it. Take the time to fill in the gaps and make sure you can properly explain the problem and its solution.

Design Like a Pro

The activities of creating and releasing a data product are varied and complex, but, typically, what you do will fall somewhere in what Alistair Croll describes as *the big data supply chain* (see Figure 2-1).

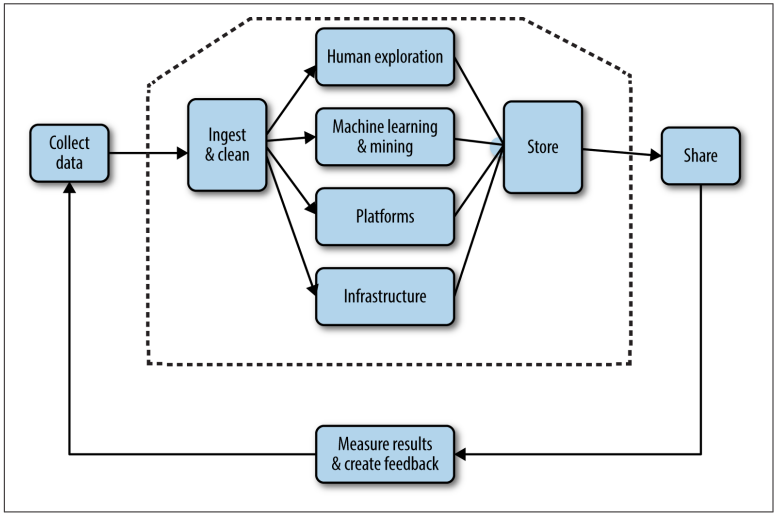


Figure 2-1. The big data supply chain (image courtesy of Jerry Overton)

Because data products execute according to a paradigm (real time, batch mode, or some hybrid of the two), you will likely find yourself participating in a combination of data supply chain activity and a data-product paradigm: ingesting and cleaning batch-updated data, building an algorithm to analyze real-time data, sharing the results

of a batch process, and so on. Fortunately, the blackboard architectural pattern gives us a basic blueprint for good software engineering in any of these scenarios (see Figure 2-2).

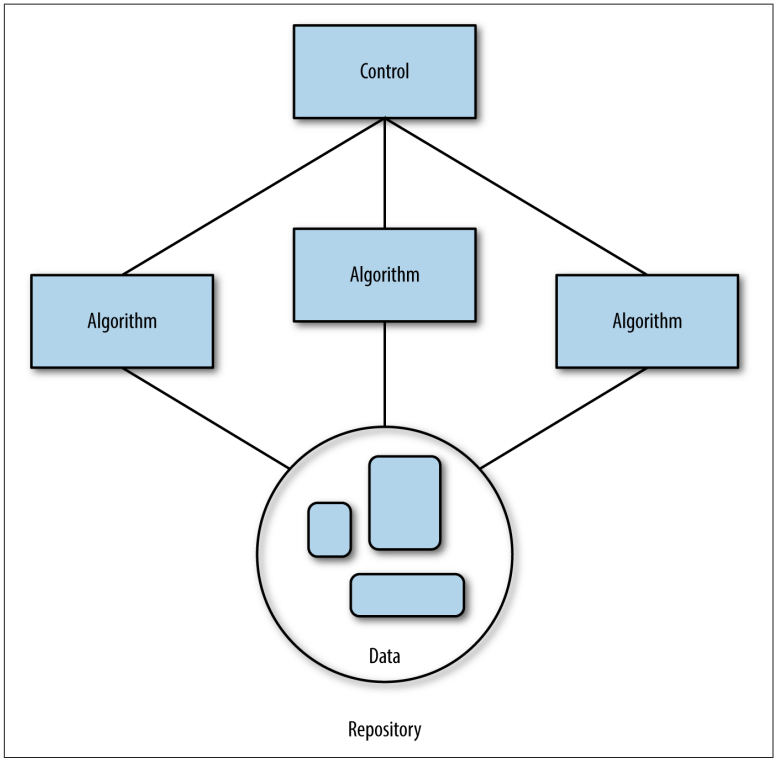


Figure 2-2. The blackboard architectural pattern (image courtesy of Jerry Overton)

The blackboard pattern tells us to solve problems by dividing the overall task of finding a solution into a set of smaller, self-contained subtasks. Each subtask transforms your hypothesis into one that's easier to solve or a hypothesis whose solution is already known. Each task gradually improves the solution and leads, hopefully, to a viable resolution.

Data science is awash in tools, each with its own unique virtues. Productivity is a big deal, and I like letting my team choose whatever tools they are most familiar with. Using the blackboard pattern makes it OK to build data products from a collection of different technologies. Cooperation between algorithms happens through a shared repository. Each algorithm can access data, process it as

input, and deliver the results back to the repository for some other algorithm to use as input.

Last, the algorithms are all coordinated using a single control component that represents the heuristic used to solve the problem. The control is the implementation of the strategy you've chosen to solve the problem. This is the highest level of abstraction and understanding of the problem, and it's implemented by a technology that can interface with and determine the order of all the other algorithms. The control can be something automated (e.g., a cron job, script), or it can be manual (e.g., a person that executes the different steps in the proper order). But overall, it's the total strategy for solving the problem. It's the one place you can go to see the solution to the problem from start to finish.

This basic approach has proven useful in constructing software systems that have to solve uncertain, hypothetical problems using incomplete data. The best part is that it lets us make progress to an uncertain problem using certain, deterministic pieces. Unfortunately, there is no guarantee that your efforts will actually solve the problem. It's better to know sooner rather than later if you are going down a path that won't work. You do this using the order in which you implement the system.

Build Like a Pro

You don't have to build the elements of a data product in a set order (i.e., build the repository first, then the algorithms, then the controller; see Figure 2-3). The professional approach is to build in the order of *highest technical risk*. Start with the riskiest element first, and go from there. An element can be technically risky for a lot of reasons. The riskiest part may be the one that has the highest workload or the part you understand the least.

You can build out components in any order by focusing on a single element and stubbing out the rest (see Figure 2-4). If you decide, for example, to start by building an algorithm, dummy up the input data and define a temporary spot to write the algorithm's output.

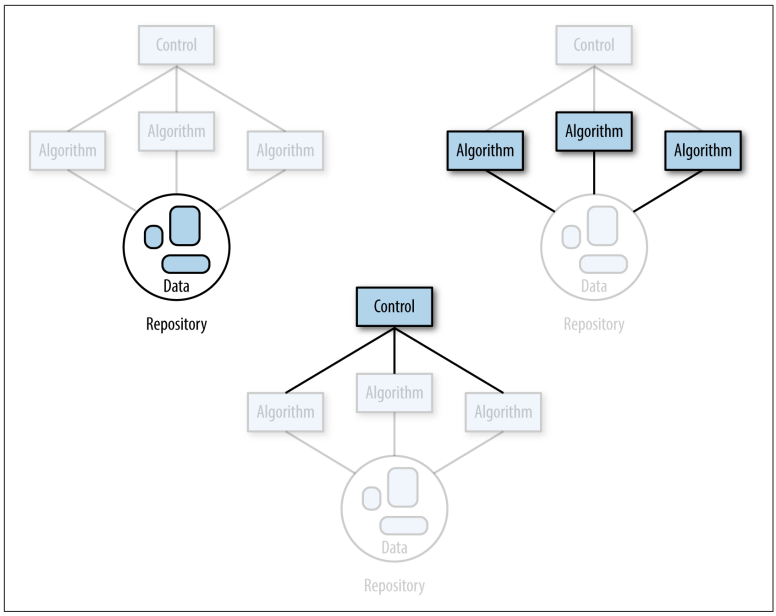


Figure 2-3. Sample 1 approach to building a data product (image courtesy of Jerry Overton)

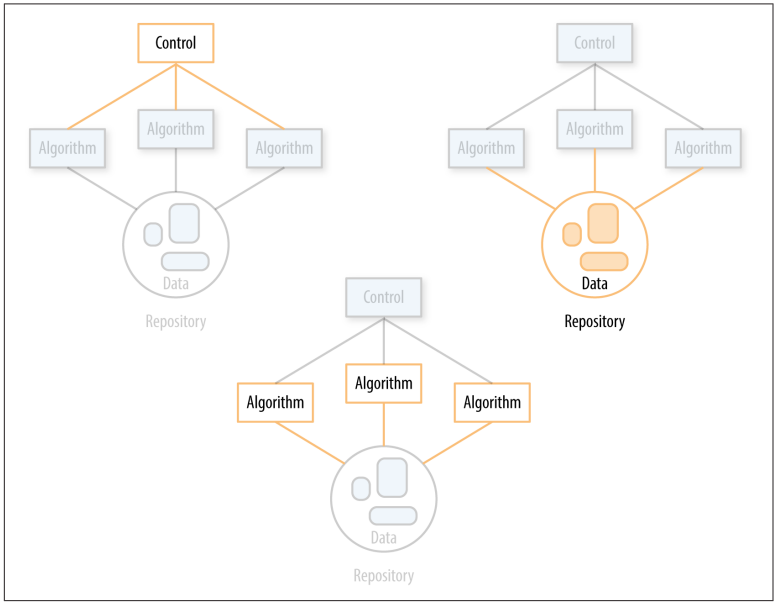


Figure 2-4. Sample 2 approach to building a data product (image courtesy of Jerry Overton)

Then, *implement* a data product in the order of technical risk, putting the riskiest elements first. Focus on a particular element, stub out the rest, replace the stubs later.

The key is to build and run in small pieces: write algorithms in small steps that you understand, build the repository one data source at a time, and build your control one algorithm execution step at a time. The goal is to have a working data product at all times—it just won't be fully functioning until the end.

Tools of the Pro

Every pro needs quality tools. There are a lot of choices available. These are some of the most commonly used tools, organized by topic:

Visualization

D3.js
: D3.js (or just D3, for data-driven documents) is a JavaScript library for producing dynamic, interactive data visualizations in web browsers. It makes use of the widely implemented SVG, HTML5, and CSS standards.

Version control

GitHub
: GitHub is a web-based Git repository hosting service that offers all of the distributed revision control and source code management (SCM) functionality of Git as well as adding its own features. GitHub provides a web-based graphical interface and desktop as well as mobile integration.

Programming languages

R
: R is a programming language and software environment for statistical computing and graphics. The R language is widely used among statisticians and data miners for developing statistical software and data analysis.

Python
: Python is a widely used general-purpose, high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express

concepts in fewer lines of code than would be possible in languages such as C++ or Java.

Scala
: Scala is an object-functional programming language for general software applications. Scala has full support for functional programming and a very strong static type system. This allows programs written in Scala to be very concise and thus smaller in size than other general-purpose programming languages.

Java
: Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA).

The Hadoop ecosystem

Hadoop
: Hadoop is an open source software framework written in Java for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware.

Pig
: Pig is a high-level platform for creating MapReduce programs used with Hadoop.

Hive
: Hive is a data warehouse infrastructure built on top of Hadoop for providing data summarization, query, and analysis.

Spark
: Spark's in-memory primitives provide performance up to 100 times faster for certain applications.

Epilogue: How This Article Came About

This article started out as a discussion of occasional productivity problems we were having on my team. We eventually traced the issues back to the technical platform and our software engineering knowledge. We needed to plug holes in our software engineering

practices, but every available course was either too abstract or too detailed (meant for professional software developers). I'm a big fan of the outside-in approach to data science and decided to hold an open CrowdChat discussion on the matter.

We got great participation: 179 posts in 30 minutes; 600 views, and 28K+ reached. I took the discussion and summarized the findings based on the most influential answers, then I took the summary and used it as the basis for this article. I want to thank all those who participated in the process and take the time to acknowledge their contributions.

The O'Reilly Data Show Podcast

Topic Models: Past, Present, and Future
An interview with David Blei

> "My understanding when I speak to people at different startup companies and other more established companies is that a lot of technology companies are using topic modeling to generate this representation of documents in terms of the discovered topics, and then using that representation in other algorithms for things like classification or other things."
>
> —David Blei, Columbia University

Listen to the full interview with David Blei here.

Graphs in the World: Modeling Systems as Networks

by Russell Jurney

You can read this post on oreilly.com here.

Networks of all kinds drive the modern world. You can build a network from nearly any kind of data set, which is probably why network structures characterize some aspects of most phenomena. And yet, many people can't see the networks underlying different systems. In this post, we're going to survey a series of networks that model different systems in order to understand various ways networks help us understand the world around us.

We'll explore how to see, extract, and create value with networks. We'll look at four examples where I used networks to model different phenomena, starting with startup ecosystems and ending in network-driven marketing.

Networks and Markets

Commerce is one person or company selling to another, which is inherently a network phenomenon. Analyzing networks in markets can help us understand how market economies operate.

Strength of weak ties

Mark Granovetter famously researched job hunting and discovered the strength of weak ties, illustrated in Figure 2-5.

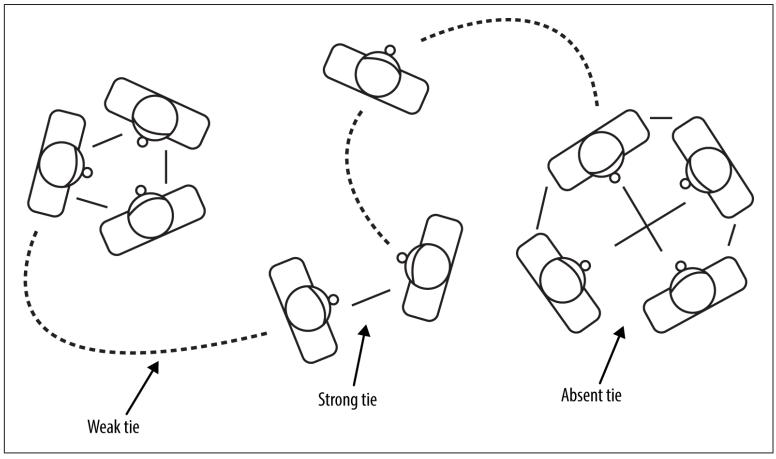


Figure 2-5. The strength of weak ties (image via Wikimedia Commons)

Granovetter's paper is one of the most influential in social network analysis, and it says something counterintuitive: Loosely connected professionals (weak ties) tend to be the best sources of job tips because they have access to more novel and different information than closer connections (strong ties). The weak tie hypothesis has been applied to understanding numerous areas.

In Granovetter's day, social network analysis was limited in that data collection usually involved a clipboard and good walking shoes. The modern Web contains numerous social networking websites and apps, and the Web itself can be understood as a large graph of web

pages with links between them. In light of this, a backlog of techniques from social network analysis are available to us to understand networks that we collect and analyze with software, rather than pen and paper. Social network analysis is driving innovation on the social web.

Networks of success

There are other ways to use networks to understand markets. Figure 2-6 shows a map of the security sector of the startup ecosystem in Atlanta as of 2010.

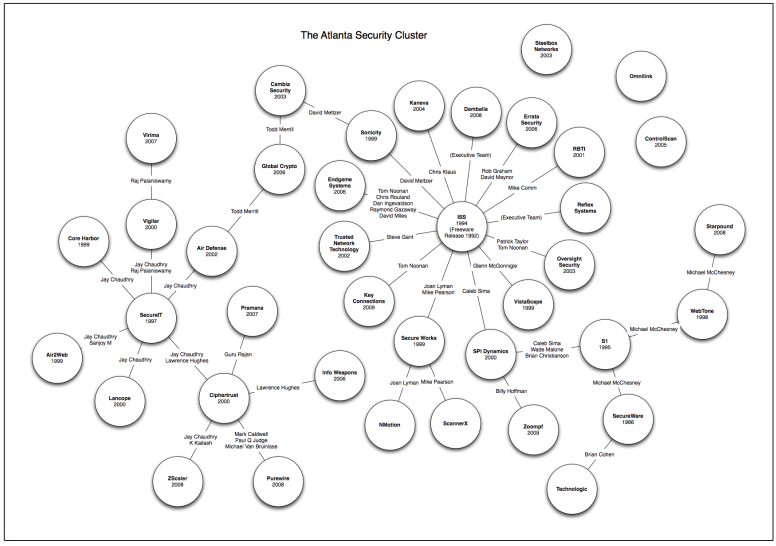


Figure 2-6. The Atlanta security startup map (image courtesy of Russell Jurney, used with permission); click here for larger version

I created this map with the help of the startup community in Atlanta, and LinkedIn and Google. Each node (circle) is a company. Each link between nodes represents a founder who worked at the originating company and went on to found the destination company. Look carefully and you will see that Internet Security Systems (ISS) and SecureIT (which sold the Internet Scanner by ISS) spawned most of the other companies in the cluster.

This simple chart illustrates the network-centric process underlying the emergence of startup ecosystems. Groups of companies emerge together via "networks of success"—groups of individuals who work together and develop an abundance of skills, social capital, and cash.

This network is similar to others that are better known, like the PayPal Mafia or the Fairchildren.

This was my first venture into social network research—a domain typically limited to social scientists and Ph.D. candidates. And when I say *social network*, I don't mean Facebook; I mean *social network* as in social network analysis.

The Atlanta security startup map shows the importance of apprenticeship in building startups and ecosystems. Participating in a solid IPO is equivalent to seed funding for every early employee. This is what is missing from startup ecosystems in provincial places: Collectively, there isn't enough success and capital for the employees of successful companies to have enough skills and capital to start their own ventures.

Once that tipping point occurs, though, where startups beget startups, startup ecosystems self-sustain—they grow on their own. Older generations of entrepreneurs invest in and mentor younger entrepreneurs, with each cohort becoming increasingly wealthy and well connected. Atlanta has a cycle of wealth occurring in the security sector, making it a great place to start a security company.

My hope with this map was to affect policy—to encourage the state of Georgia to redirect stimulus money toward economic clusters that *work* as this one does. The return on this investment would dwarf others the state makes because the market *wants* Atlanta to be a security startup mecca. This remains a hope.

In any case, that's a lot to learn from a simple map, but that's the kind of insight you can obtain from collecting and analyzing social networks.

LinkedIn InMaps

Ali Imam invented LinkedIn's InMaps as a side project. InMaps were a hit: People went crazy for them. Ali was backlogged using a step-by-step, manual process to create the maps. I was called in to turn the one-off process into a product. The product was cool, but more than that, we wanted to prove that anyone at LinkedIn could come up with a good idea and we could take it from an idea to a production application (which we did).

Snowball sampling and 1.5-hop networks

InMaps was a great example of the utility of snowball samples and 1.5-hop networks. A snowball sample is a sample that starts with one or more persons, and grows like a snowball as we recruit their friends, and then their friend's friends, until we get a large enough sample to make inferences. 1.5-hop networks are local neighborhoods centered on one entity or ego. They let us look at a limited section of larger graphs, making even massive graphs browsable.

With InMaps, we started with one person, and then added their connections, and finally added the connections between them. This is a "1.5-hop network." If we only looked at a person and their friends, we would have a "1-hop network." If we included the person, their friends, as well as all connections of the friends, as opposed to just connections between friends, we would have a "2-hop network."

Viral visualization

My favorite thing about InMaps is a bug that became a feature. We hadn't completed the part of the project where we would determine the name of each cluster of LinkedIn users. At the same time, we weren't able to get placement for the application on the site. So how would users learn about InMaps?

We had several large-scale printers, so I printed my brother's InMap as a test case. We met so I could give him his map, and we ended up labeling the clusters by hand right there in the coffee shop. He was excited by his map, but once he labeled it, he was ecstatic. It was "his" art, and it represented his entire career. He *had* to have it. Ali created my brother's InMap, shown in Figure 2-7, and I hand labeled it in Photoshop.

So, we'd found our distribution: virality. Users would create their own InMaps, label the clusters, and then share their personalized InMap via social media. Others would see the InMap, and want one of their own—creating a viral loop that would get the app in front of users.

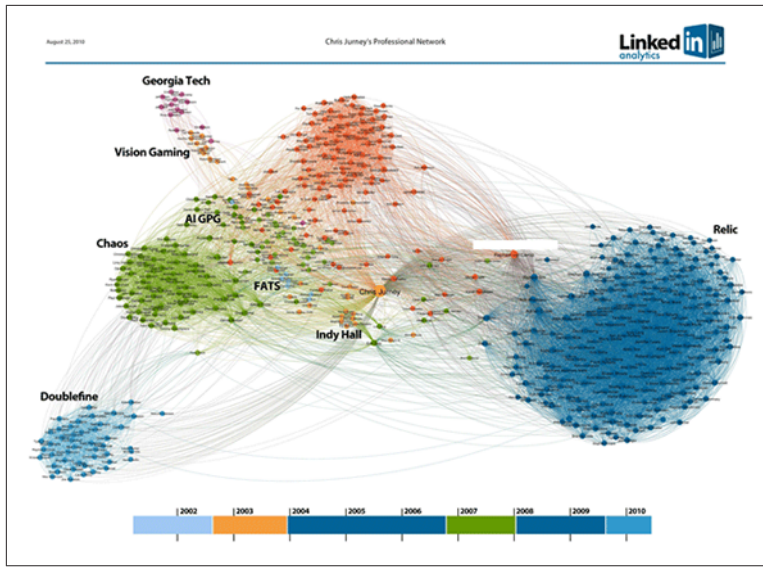


Figure 2-7. Chris Jurney's InMap (photo courtesy of Ali Imam and Russell Jurney, used with permission)

Inbox Networks

After I left LinkedIn, I missed its data. I needed a new data set to play with, and Chris Diehl told me about his work on the Enron data set.

About half a gigabyte of emails that surfaced during the investigation into the collapse of Enron have become a standard data set against which researchers discover and develop a variety of statistical software and systems.

After playing with the Enron data set, I wanted something more personal. I wrote a script that downloads your Gmail inbox into Avro format. After all, if it's your data, then you can really gauge insight.

Taking a cue from InMaps, I rendered maps of my inbox and labeled the clusters (see Figure 2-8).

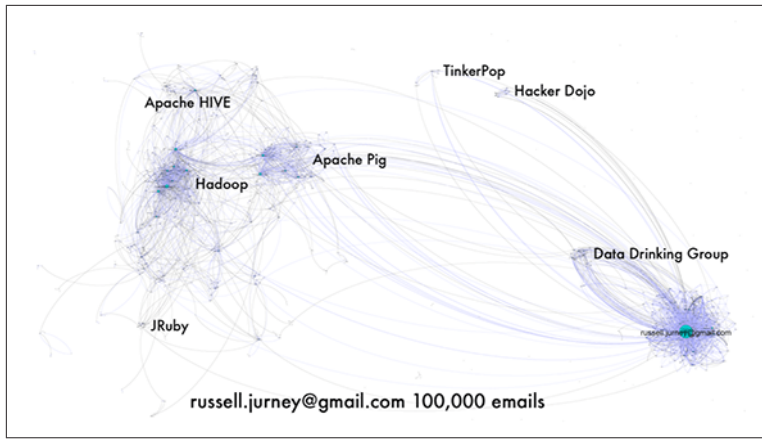


Figure 2-8. Map of Russell Jurney's Gmail inbox showing labeled clusters (image courtesy of Russell Jurney, used with permission)

Inbox ego networks

These maps showed the different groups I belonged to, mailing lists, etc. From there, it was possible to create an ego network of senders of emails, and to map users to groups and organizations. Inbox ego networks are a big deal: This is the technology behind RelateIQ, which was acquired in 2014 for $392 million. RelateIQ's killer feature is that it reduces the amount of data entry required, as it automatically identifies companies you're emailing by their domain and creates customer relationship management (CRM) entries for each email you send or receive.

Agile data science

I founded Kontexa to create a collaborative, semantic inbox. I used graph visualization to inspect the results of my data processing and created my own simple graph database on top of Voldemort to allow the combination of different inboxes at a semantic level. Figure 2-9 shows a visualization of my inbox unioned with my brother's.

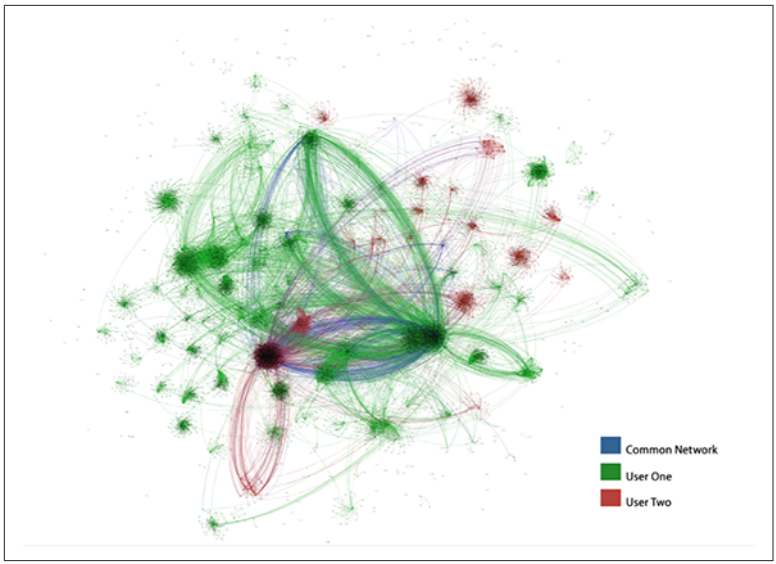


Figure 2-9. Graphical representation of inbox combining Russell Jurney and Chris Jurney's data (image courtesy of Russell Jurney, used with permission)

This work became the foundation for my first book, *Agile Data Science*. In the book, users download their own inboxes and then we analyze these Avro records in Apache Pig and Python.

Customer Relationship Management Analytics

During a nine-month stint as data scientist in residence at The Hive, I helped launch the startup E8 Security, acting as the first engineer on the team (E8 went on to raise a $10 million series A). As my time at E8 came to a close, I once again found myself needing a new data set to analyze.

Former Hiver Karl Rumelhart introduced me to CRM data. CRM databases can be worth many millions of dollars, so it's a great type of data to work with. Karl posed a challenge: Could I cluster CRM databases into groups that we could then use to target different sectors in marketing automation?

We wanted to know if segmenting markets was possible before we asked any prospective customers for their CRM databases. So, as a test case, we decided to look at the big data market. Specifically, we

focused on the four major Hadoop vendors: Cloudera, Hortonworks, MapR, and Pivotal.

In the absence of a CRM database, how would I link one company to another? The answer: partnership pages. Most companies in the big data space have partnership pages, which list other companies a given company works with in providing its products or services. I created a hybrid machine/turk system that gathered the partnerships of the four Hadoop vendors. Then I gathered the partnerships of these partners to create a "second degree network" of partnerships.

Once clustered, the initial data looked like Figure 2-10.

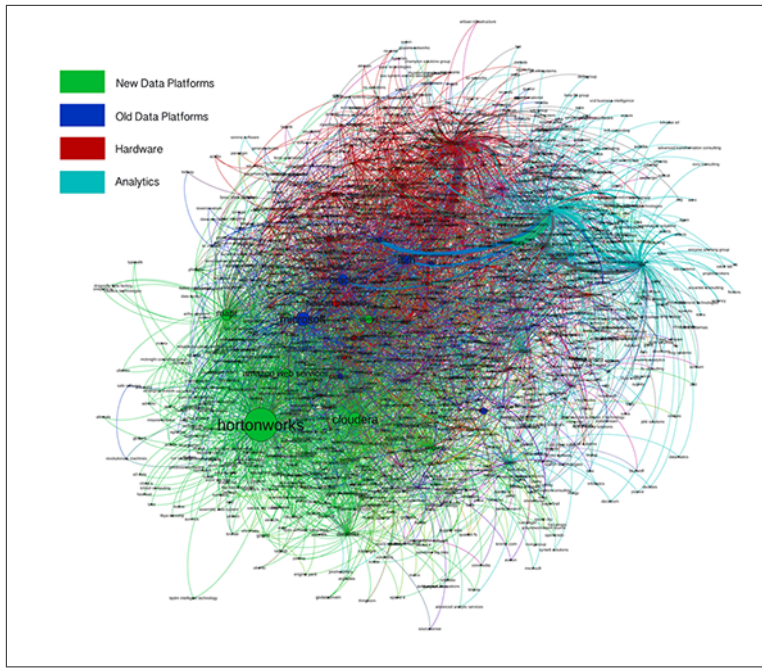


Figure 2-10. Graphical representation of corporate partnerships among four Hadoop vendors (image courtesy of Russell Jurney, used with permission)

Taking a cue from InMaps once again, I hand labeled the clusters. We were pleased to find that they corresponded roughly with sectors in the big data market—new/old data platforms, and hardware and analytic software companies. An idea we've been playing with is to create these clusters, then classify new leads into its cluster, and use

this cluster field in marketing automation. This would allow better targeting with cluster-specific content.

Market reports

At this point, I really thought I was onto something. Something worth exploring fully. *What if we mapped entire markets, indeed the entire economy, in terms of relationships between companies?* What could we do with this data? I believe that with a scope into how the economy works, we could make markets more efficient.

Early in 2015, I founded Relato with this goal in mind: *improve sales, marketing, and strategy by mapping the economy*. Working on the company full time since January, we've partnered with O'Reilly to extend the initial work on the big data space to create an in-depth report: "Mapping Big Data: A Data-Driven Market Report." The report includes an analysis of data we've collected about companies in the big data space, along with expert commentary. This is a new kind of market report that you'll be seeing more of in the future.

Conclusion

We've shown how networks are the structure behind many different phenomena. When you next encounter a new data set, you should ask yourself: Is this a network? What would understanding this data as a network allow me to do?

Let's Build Open Source Tensor Libraries for Data Science

by Ben Lorica

You can read this post on oreilly.com here.

Data scientists frequently find themselves dealing with high-dimensional feature spaces. As an example, text mining usually involves vocabularies comprised of 10,000+ different words. Many analytic problems involve linear algebra, particularly 2D matrix factorization techniques, for which several open source implementations are available. Anyone working on implementing machine learning algorithms ends up needing a good library for matrix analysis and operations.

But why stop at 2D representations? In a Strata + Hadoop World San Jose presentation, UC Irvine professor Anima Anandkumar described how techniques developed for higher-dimensional arrays can be applied to machine learning. *Tensors* are generalizations of matrices that let you look beyond pairwise relationships to higher-dimensional models (a matrix is a second-order tensor). For instance, one can examine patterns between any three (or more) dimensions in data sets. In a text mining application, this leads to models that incorporate the co-occurrence of three or more words, and in social networks, you can use tensors to encode arbitrary degrees of influence (e.g., "friend of friend of friend" of a user).

Being able to capture higher-order relationships proves to be quite useful. In her talk, Anandkumar described applications to latent variable models, including text mining (topic models), information science (social network analysis), recommender systems, and deep neural networks. A natural entry point for applications is to look at generalizations of matrix (2D) techniques to higher-dimensional arrays. For example, Figure 2-11 illustrates one form of eigen decomposition.

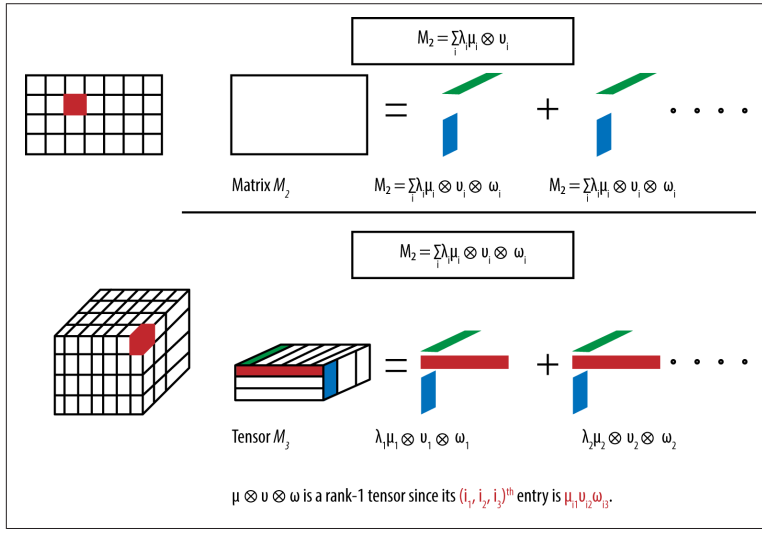
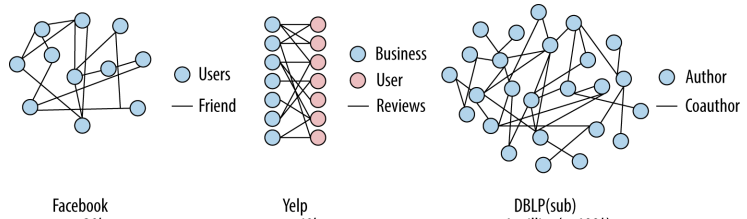


Figure 2-11. Spectral decomposition of tensors (image courtesy of Anima Anandkumar, used with permission)

Tensor Methods Are Accurate and Embarrassingly Parallel

Latent variable models and deep neural networks can be solved using other methods, including maximum likelihood and local search techniques (gradient descent, variational inference, EM). So, why use tensors at all? Unlike variational inference and EM, tensor methods produce global and not local optima, under reasonable conditions. In her talk, Anandkumar described some recent examples—topic models and social network analysis—where tensor methods proved to be *faster* and *more accurate* than other methods (see Figure 2-12).



Error (ξ) and Recovery ratio (*R*)

Dataset	$\hat{k}$	Method	Running Time	ξ	*R*
Facebook(k=360)	500	tensor	468	0.0175	100%
Facebook(k=360)	500	variational	86,808	0.0308	100%
Yelp(k=159)	100	tensor	287	0.046	86%
Yelp(k=159)	100	variational	N/A	N/A	N/A
DBLP sub(k=250)	500	tensor	10,157	0.139	89%
DBLP sub(k=250)	500	variational	558,723	16.38	99%
DBLP(k=6000)	100	tensor	5407	0.105	95%

Figure 2-12. Error rates and recovery ratios from recent community detection experiments (running time measured in seconds; image courtesy of Anima Anandkumar, used with permission)

Scalability is another important reason why tensors are generating interest. Tensor decomposition algorithms have been parallelized using GPUs, and more recently using Apache REEF (a *distributed* framework originally developed by Microsoft). To summarize, early results are promising (in terms of speed and accuracy), and implementations in distributed systems lead to algorithms that scale to extremely large data sets (see Figure 2-13).

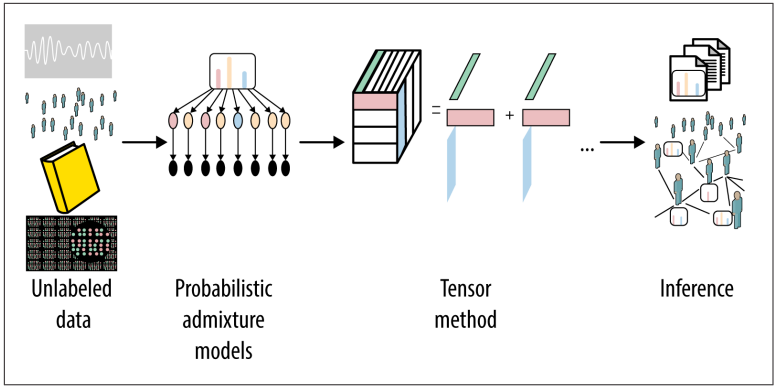


Figure 2-13. General framework (image courtesy of Anima Anandkumar, used with permission)

Hierarchical Decomposition Models

Their ability to model multiway relationships makes tensor methods particularly useful for uncovering hierarchical structures in high-dimensional data sets. In a recent paper, Anandkumar and her collaborators automatically found patterns and "concepts reflecting co-occurrences of particular diagnoses in patients in outpatient and intensive care settings."

Why Aren't Tensors More Popular?

If they're faster, more accurate, and embarrassingly parallel, why haven't tensor methods become more common? It comes down to libraries. Just as matrix libraries are needed to implement many machine learning algorithms, open source libraries for tensor analysis need to become more common. While it's true that tensor computations are more demanding than matrix algorithms, recent improvements in parallel and distributed computing systems have made tensor techniques feasible.

There are some early libraries for tensor analysis in MATLAB, Python, TH++ from Facebook, and many others from the scientific computing community. For applications to machine learning, software tools that include tensor *decomposition* methods are essential. As a first step, Anandkumar and her UC Irvine colleagues have released code for tensor methods for topic modeling and social network modeling that run on single servers.

But for data scientists to embrace these techniques, we'll need well-developed libraries accessible from the languages (Python, R, Java, Scala) and frameworks (Apache Spark) we're already familiar with. (Coincidentally, Spark developers just recently introduced *distributed* matrices.)

It's fun to see a tool that I first encountered in math and physics courses having an impact in machine learning. But the primary reason I'm writing this post is to get readers excited enough to build open source tensor (decomposition) libraries. Once these basic libraries are in place, tensor-based algorithms become easier to implement. Anandkumar and her collaborators are in the early stages of porting some of their code to Apache Spark, and I'm hoping other groups will jump into the fray.

The O'Reilly Data Show Podcast

The Tensor Renaissance in Data Science
An interview with Anima Anandkumar

> "The latest set of results we have been looking at is the use of tensors for feature learning as a general concept. The idea of feature learning is to look at transformations of the input data that can be classified more accurately using simpler classifiers. This is now an emerging area in machine learning that has seen a lot of interest, and our latest analysis is to ask how can tensors be employed for such feature learning. What we established is you can learn recursively better features by employing tensor decompositions repeatedly, mimicking deep learning that's being seen."
>
> —Anima Anandkumar, UC Irvine

Listen to the full interview with Anima Anandkumar here.

CHAPTER 3
Data Pipelines

Engineering and optimizing data pipelines continues to be an area of particular interest, as researchers attempt to improve efficiency so they can scale to very large data sets. Workflow tools that enable users to build pipelines have also become more common—these days, such tools exist for data engineers, data scientists, and even business analysts. In this chapter, we present a collection of blog posts and podcasts that cover the latest thinking in the realm of data pipelines.

First, Ben Lorica explains why interactions between parts of a pipeline are an area of active research, and why we need tools to enable users to build certifiable machine learning pipelines. Michael Li then explores three best practices for building successful pipelines—reproducibility, consistency, and productionizability. Next, Kiyoto Tamura explores the ideal frameworks for collecting, parsing, and archiving logs, and also outlines the value of JSON as a unifying format. Finally, Gwen Shapira discusses how to simplify backend A/B testing using Kafka.

Building and Deploying Large-Scale Machine Learning Pipelines

by Ben Lorica

You can read this post on oreilly.com here.

There are many algorithms with implementations that scale to large data sets (this list includes matrix factorization, SVM, logistic regression, LASSO, and many others). In fact, machine learning experts are fond of pointing out: If you can pose your problem as a simple optimization problem then you're almost done.

Of course, in practice, most machine learning projects can't be reduced to simple optimization problems. Data scientists have to manage and maintain complex data projects, and the analytic problems they need to tackle usually involve specialized machine learning pipelines. Decisions at one stage affect things that happen downstream, so interactions between parts of a pipeline are an area of active research (see Figure 3-1).

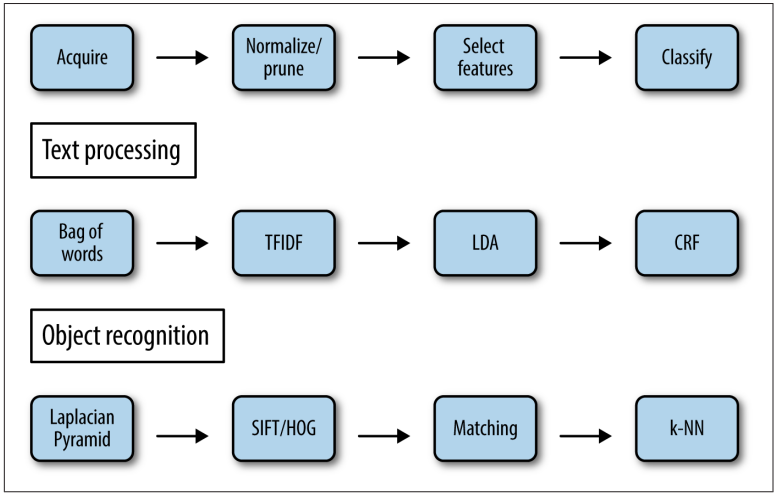


Figure 3-1. Some common machine learning pipelines (image courtesy of Ben Recht, used with permission)

In his 2014 Strata + Hadoop World New York presentation, UC Berkeley professor Ben Recht described new UC Berkeley AMPLab projects for building and managing large-scale machine learning pipelines. Given AMPLab's ties to the Spark community, some of the ideas from their projects are starting to appear in Apache Spark.

Identify and Build Primitives

The first step is to create building blocks. A pipeline is typically represented by a graph, and AMPLab researchers have been focused on scaling and optimizing *nodes* (primitives) that can scale to very large data sets (see Figure 3-2). Some of these primitives might be specific to particular domains and data types (text, images, video, audio, spatiotemporal) or more general purpose (statistics, machine learning). A recent example would be ml-matrix, a distributed matrix library that runs on top of Apache Spark.

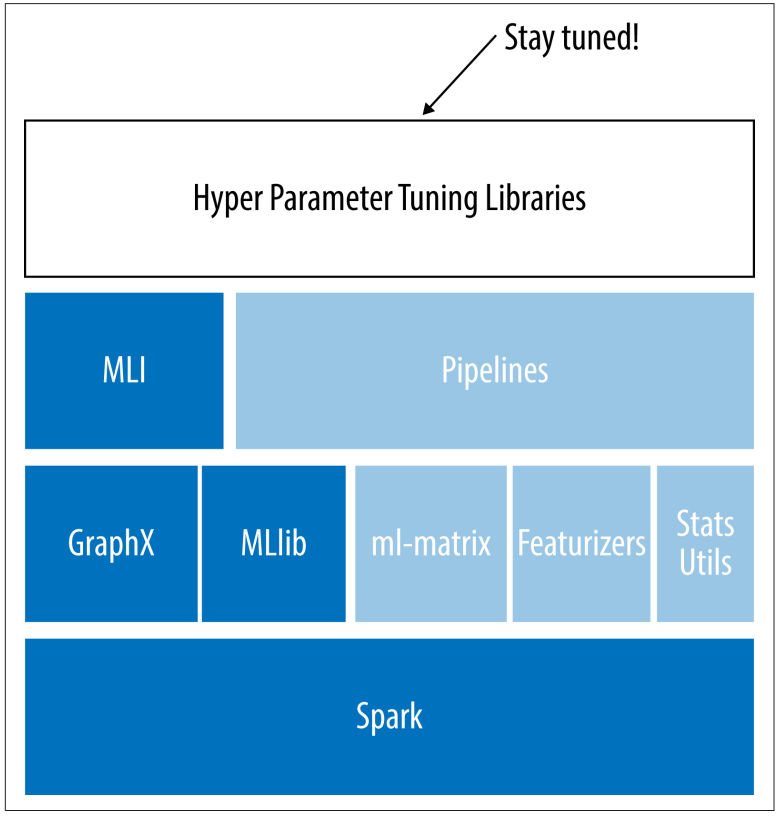


Figure 3-2. AMPLab advanced analytics stack (image courtesy of Evan Sparks, used with permission)

Casting machine learning models in terms of primitives makes these systems more accessible. To the extent that the nodes of your pipeline are "interpretable," resulting machine learning systems are more transparent and explainable than methods relying on black boxes.

Make Machine Learning Modular: Simplifying Pipeline Synthesis

While primitives can serve as building blocks, we still need tools that enable users to build pipelines. Workflow tools have become more common, and these days, such tools exist for data engineers, data scientists, and even business analysts (Alteryx, RapidMiner, Alpine Data, Dataiku).

As I noted in a recent post, we'll see more data analysis tools that combine an elegant interface with a simple DSL that non-programmers can edit. At some point, DSLs to encode graphs that represent these pipelines will become common. The latest release of Apache Spark (version 1.2) comes with an API for building machine learning pipelines (if you squint hard enough, it has the makings of a DSL for pipelines).

Do Some Error Analysis

Figure 3-3 supports Ben Recht's following statement:

> We're trying to put (machine learning systems) in self-driving cars, power networks … If we want machine learning models to actually have an impact in everyday experience, we'd better come out with the same guarantees as one of these complicated airplane designs.
>
> —Ben Recht

Can we bound approximation errors and convergence rates for layered pipelines? Assuming we can compute error bars for nodes, the next step would be to have a mechanism for extracting error bars for these pipelines. In practical terms, what we need are tools to *certify* that a pipeline will work (when deployed in production) and that can provide some measure of the size of errors to expect.

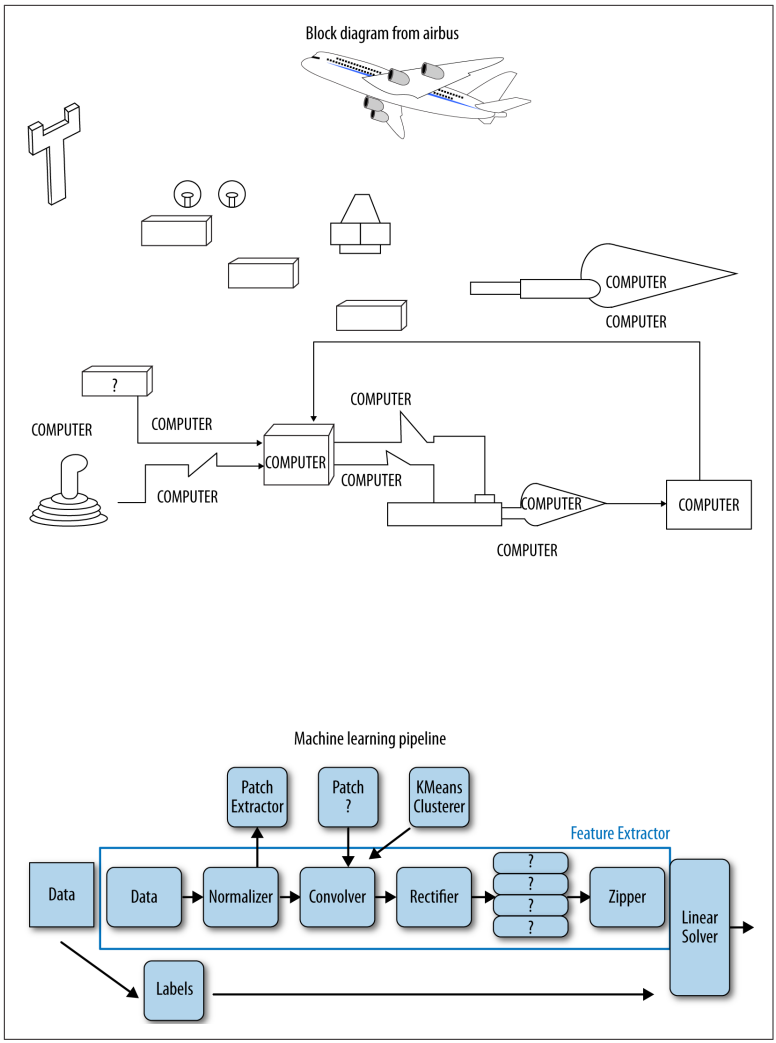


Figure 3-3. ML pipelines are beginning to resemble the intricacy of block diagrams from airplanes (image courtesy of Ben Recht, used with permission); click for a larger view

To that end, Laurent Lessard, Ben Recht, and Andrew Packard have been using techniques from control theory to automatically generate verification certificates for machine learning pipelines. Their methods can analyze many of the most popular techniques for machine learning on large data sets. And their longer term goal is to be able to derive performance characteristics and analyze the robustness of

complex, distributed software systems directly from pseudocode. (A related AMPLab project Velox provides a framework for managing models in production.)

As algorithms become even more pervasive, we need better tools for building complex yet robust and stable machine learning systems. While other systems like scikit-learn and GraphLab support pipelines, a popular *distributed* framework like Apache Spark takes these ideas to extremely large data sets and a wider audience. Early results look promising: AMPLab researchers have built large-scale pipelines that match some state-of-the-art results in vision, speech, and text processing.

The O'Reilly Data Show Podcast

6 Reasons Why I love KeystoneML
An interview with Ben Recht

> "The other thing that we're hoping to be able to do are *systems optimizations*: meaning that we don't want you to load a thousand-node cluster because you made an incorrect decision in caching. We'd like to be able to actually make these things where we can smartly allocate memory and other resources to be able to run to more compact clusters."
>
> —Ben Recht, UC Berkeley

Listen to the full interview with Ben Recht here.

Three Best Practices for Building Successful Data Pipelines

by Michael Li

You can read this post on oreilly.com here.

Building a good data pipeline can be technically tricky. As a data scientist who has worked at Foursquare and Google, I can honestly say that one of our biggest headaches was locking down our Extract, Transform, and Load (ETL) process.

At The Data Incubator, our team has trained more than 100 talented Ph.D. data science fellows who are now data scientists at a wide range of companies, including Capital One, *The New York Times*, AIG, and Palantir. We commonly hear from Data Incubator alumni

and hiring managers that one of their biggest challenges is also implementing their own ETL pipelines.

Drawn from their experiences and my own, I've identified three key areas that are often overlooked in data pipelines. These include ensuring your analysis is:

- Reproducible
- Consistent
- Productionizable

While these areas alone cannot guarantee good data science, getting these three technical aspects of your data pipeline right helps ensure that your data and research results are both reliable and useful to an organization.

Ensuring Reproducibility by Providing a Reliable Audit Trail

To ensure the reproducibility of your data analysis, there are three dependencies that need to be locked down: analysis code, data sources, and algorithmic randomness (see Figure 3-4).

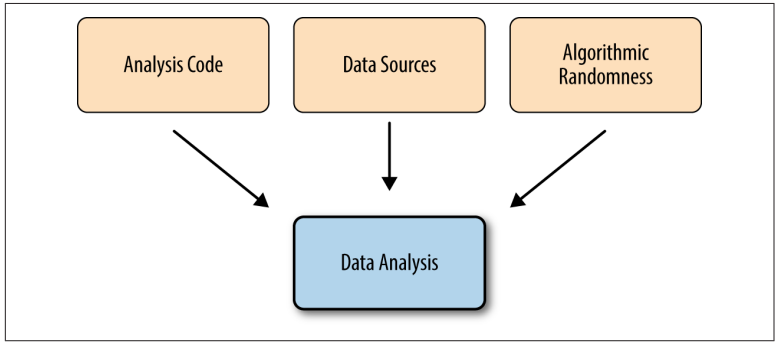


Figure 3-4. Three dependencies that ensure reproducible data analysis (image courtesy of Michael Li)

Science that cannot be reproduced by an external third party is just not science—and this does apply to data *science*. One of the benefits of working in data science is the ability to apply the existing tools from software engineering. These tools let you isolate all the dependencies of your analyses and make them reproducible.

Dependencies fall into three categories:

Analysis code
: All analysis code should be checked into source control. Every analysis consists of innumerable assumptions made during both ETL and modeling. These are impossible to document exhaustively, and the rapid iteration cycles enabled by faster tools means the results and documented processes are often out of sync. Like with soft engineering, what was run is what was in code, and having that timestamped and recorded in a version control system makes analysis much more reproducible.

Data sources
: While the analysis code needs to be locked down, so does the data source. (You've likely encountered analyses that open up a mysterious *sales.dat* file that no one can find.) Even if you *can* locate your data files, you might find a version that's encoded differently than the one originally used (e.g., using JSON rather than XML), thus making your otherwise carefully source-controlled analysis code much less useful. Locking down the exact file is the first step in data consistency.

Algorithmic randomness
: Is your data being randomly sub-sampled? Even seemingly deterministic analyses can have hidden randomness. For example, check whether your gradient descent starts at a random initial condition. These techniques all depend on a random number generator. The problem with algorithmic randomness is that even if we freeze the input data and analysis code, the outputs will still vary randomly. This eliminates reproducibility because another person trying to reproduce the result can never answer the question *Is the new answer different because I did something wrong or because of algorithmic randomness?* To promote reproducibility, set the random number generator's "seed" to an arbitrary, but fixed, value (that's checked into source control) before running the analysis so that results are consistent and discrepancies can be attributed to code or data.

By locking down analysis code, data sources, and algorithmic randomness, we can ensure that the entire analysis can be re-run by anyone. The benefits, however, go beyond scientific reproducibility; making the analysis fully documented provides a reliable audit trail, which is critical for data-driven decision making.

Establishing Consistency in Data

How we establish the consistency of data sources is a bit trickier than establishing reproducibility. Establishing consistency is of fundamental importance because data science obeys the maxim "garbage in, garbage out," and as we have discussed, access to the correctly formatted data is just as important as access to the right analysis code.

There are generally two ways of establishing the consistency of data sources. The first is by checking all code *and data* into a single revision control repository. The second method is to reserve source control for code and build a pipeline that explicitly depends on external data being in a stable, consistent format and location.

Checking data into version control is generally considered verboten for production software engineers, but it has a place in data analysis. For one thing, it makes your analysis very portable by isolating all dependencies into source control. Here are some conditions under which it makes sense to have both code and data in source control:

Small data sets
: A lot of data analysis either fully or partially depends on a few small data sets. Even if you are performing an analysis on a large amount of data, sub-sampling to a smaller data set can be sufficient. In this case, it may make sense to keep your data checked into source control rather than building an expensive pipeline to manage it. Obviously, if your data set is large, it becomes impractical to check it into source control.

Regular analytics
: A lot of data analytics are simply one-off reporting, and it may not make much sense to set up a pipeline simply for the sake of reproducibility. On the other hand, if this is an analysis that needs to occur daily, it doesn't make sense to check in each day's data to an ever-growing version control repository.

Fixed source
: If you control the data source or know that it will likely remain fixed, this might be a good candidate for setting up a full pipeline. On the other hand, if your data source is managed by an external party and subject to frequent changes, it may not be worth setting up and maintaining a consistent pipeline. For example, if your data includes files that need to be downloaded

from a government website, it may make sense to check those into source control, rather than maintaining a custom script that downloads the files each time.

While source-control systems are considered forbidden on the production end, source-control systems like Git can easily handle tens of megabytes of data without significant performance lag. Services like GitHub's large file storage system promise to make handling large data in source control even easier in the future. Whether you choose to check all code and data into a single revision control repository or just put your code under source control and lock down your data sources in an external pipeline, securing your data sources is the key to consistent data and reproducible analysis.

Productionizability: Developing a Common ETL

Of course data science that isn't deployed is useless, and the ability to productionize results is always a central concern of good data scientists. From a data pipelining perspective, one of the key concerns is the development of a common ETL process shared by production and research.

As a simple example, we may want to join-in user data and purchase data to feed into a recommender model for a website. This join would need to happen in two environments: in research (where we need the data for training), and in production (where we need the data for predictions and recommendations). Ideally, the ETL code that joins these two chunks of data (and the myriad of data normalization decisions embedded in this code) would be shared between the two environments. In practice, this faces two major challenges:

Common data format
: In practice, there are a number of constraints to establishing a common data format. You'll have to select a data format that plays well with production and Hadoop (or whatever backend data store you use). Being able to use the same data formats reduces the number of unnecessary data transformations, and helps prevent introducing bugs that contaminate data purity. Using the same data formats also reduces the number of data schemas and formats that your data team has to learn and maintain.

Isolating library dependencies

You will want to isolate library dependencies used by your ETL in production. On most research environments, library dependencies are either packaged with the ETL code (e.g., Hadoop) or provisioned on each cluster node (e.g., mrjob). Reducing these dependencies reduces the overhead of running an ETL pipeline. However, production code typically supports a much larger set of functionality (HTML templating, web frameworks, task queues) that are not used in research, and vice versa. Data engineers and scientists need to be careful about isolating the ETL production code from the rest of the codebase to keep the dependencies isolated so that the code can be run efficiently on both the frontend and backend.

While sharing ETL code between production and research introduces some complexities, it also greatly reduces the potential for errors, by helping guarantee that the transformed data used for model training is the same as the transformed data the models will use in production. Even after locking down your data sources and ensuring data consistency, having separate ETL code can lead to differences in modeling input data that renders the outputs of models completely useless. Common ETL code can go a long way to ensuring reproducible results in your analysis (see Figure 3-5).

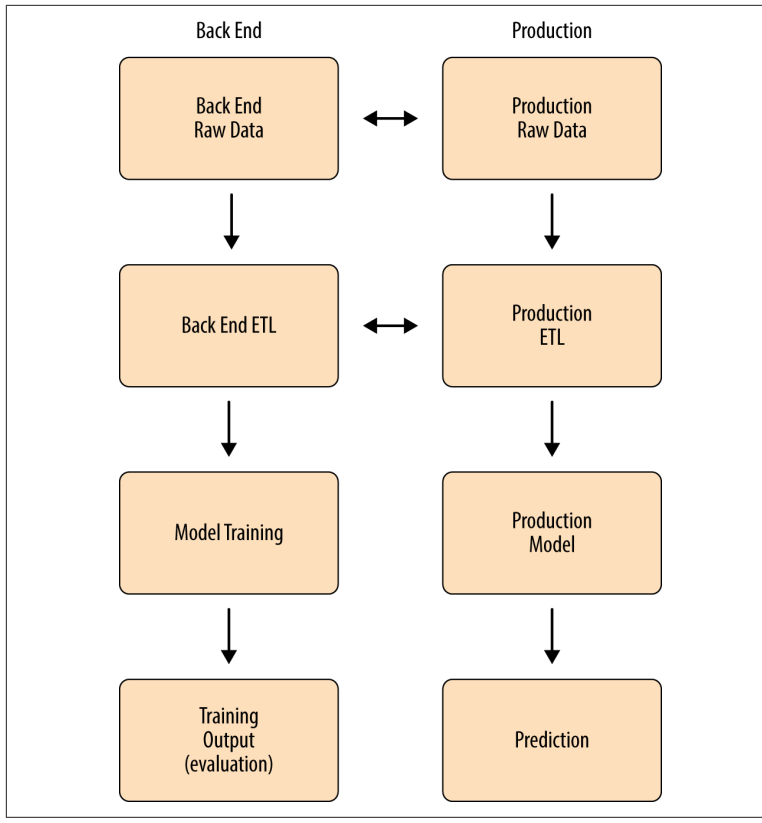


Figure 3-5. Backend and production sharing data formats and ETL code make research rapidly productionizable (image courtesy of Michael Li)

Focusing on the Science

With the high cost of data science, managers need to ensure that their data analytics are both sound and useful. Ultimately, achieving this depends on external factors, including the quality of the team and the quality of the underlying data, which may be outside of the manager's control. Data analysis is hard enough without having to worry about the correctness of your underlying data or its future ability to be productionizable. By employing these engineering best practices of making your data analysis reproducible, consistent, and productionizable, data scientists can focus on science, instead of worrying about data management.

The Log: The Lifeblood of Your Data Pipeline

by Kiyoto Tamura

You can read this post on oreilly.com here.

The value of log data for business is unimpeachable. On every level of the organization, the question *How are we doing?* is answered, ultimately, by log data. Error logs tell developers what went wrong in their applications. User event logs give product managers insights on usage. If the CEO has a question about the next quarter's revenue forecast, the answer ultimately comes from payment/CRM logs. In this post, I explore the ideal frameworks for collecting and parsing logs.

Apache Kafka architect Jay Kreps wrote a wonderfully crisp survey on log data. He begins with the simple question of *What is the log?* and elucidates its key role in thinking about data pipelines. Jay's piece focuses mostly on storing and processing log data. Here, I focus on the steps before storing and processing.

Changing the Way We Think About Log Data

Over the past decade, the primary consumer of log data shifted from humans to machines (see Figure 3-6).

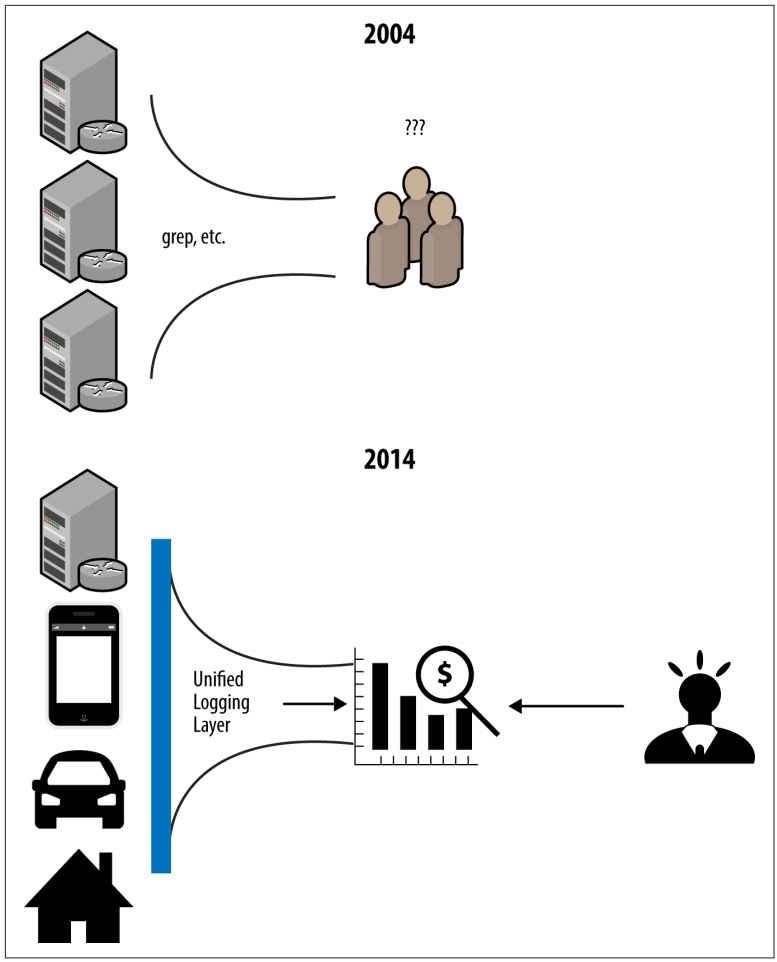


Figure 3-6. The old paradigm, machines to humans, compared to the new paradigm, machines to machines (image courtesy of Kiyoto Tamura)

Software engineers still read logs, especially when their software behaves in an unexpected manner. However, in terms of "bytes processed," humans account for a tiny fraction of the total consumption. Much of today's "big data" is some form of log data, and businesses run tens of thousands of servers to parse and mine these logs to gain competitive edge.

This shift brought about three fundamental changes in how we ought to think about collecting log data. Specifically, we need to do the following:

Make logs consumable for machines first, humans second
: Humans are good at parsing unstructured text, but they read very slowly, whereas machines are the exact opposite—they are terrible at guessing the hidden structure of unstructured text but read structured data very, very quickly. The log format of today must serve machines' needs first and humans' second.

  Several formats are strong candidates (Protocol Buffer, MessagePack, Thrift, etc.). However, JSON seems to be the clear winner for one critical reason: It is human readable. At the end of the day, humans also need to read the logs occasionally to perform sanity checks, and JSON is easy to read for both machines and humans. This is one decided advantage that JSON has over binary alternatives like Protocol Buffer. While human reading is secondary to machine intelligibility, it is, after all is said and done, still a requirement for quality logging.

Transport logs reliably
: The amount of log data produced today is staggering. It's not unheard of for a young startup to amass millions of users, who in turn produce millions of user events and billions of lines of logs. Product managers and engineers revel at the opportunity to analyze all these logs to understand their customers and make their software better.

  But doing so presents a challenge: Logs need to be transported from where they are produced (mobile devices, sensors, or plain old web servers) to where they can be archived cost-effectively (HDFS, Amazon S3, Google Cloud Storage, etc.).

  Transporting terabytes and petabytes of logs over a network creates an interesting technical challenge. At minimum, the transport mechanism must be able to cope with network failures and not lose any data. Ideally, it should be able to prevent data duplication. Achieving both—or the "exactly once" semantics in data infrastructure parlance—is the holy grail of distributed computing (see Figure 3-7).

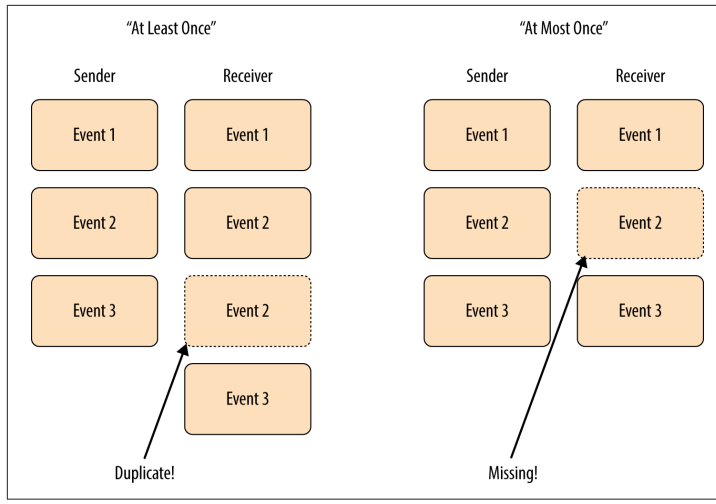


Figure 3-7. Errors in log transporting over networks (image courtesy of Kiyoto Tamura)

Support many data inputs and outputs

Today, collecting and storing logs is more complex than ever. On the data input side, we have more devices producing logs in a wide range of formats. On the data output side, it looks like there is a new database or storage engine coming out every month (see Figure 3-8). How can one hope to maintain logging pipelines with so many data inputs and outputs?

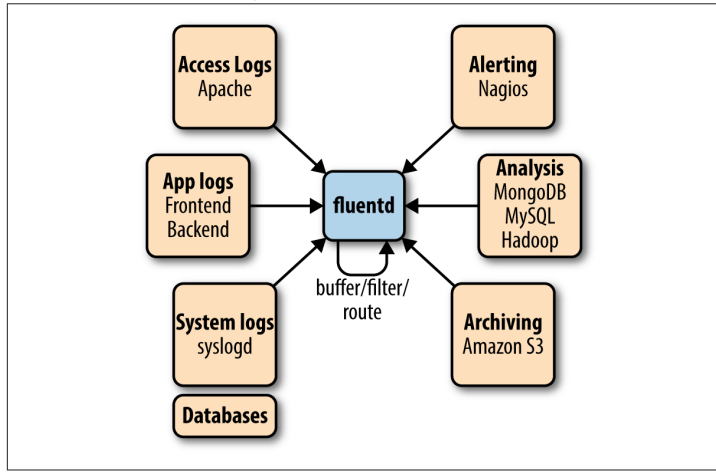


Figure 3-8. Many-to-many log routing (image courtesy of Kiyoto Tamura)

The Need for the Unified Logging Layer

The answer is pluggable architecture. Every single platform that has managed to address a growing set of use cases, from WordPress through jQuery to Chrome, has a mechanism that allows individual users to extend the core functionality by contributing and sharing their work with other users as a plug-in.

This means the user should be able to add his or her own data inputs and outputs as plug-ins. If a new data input plug-in is created, it should be compatible with all existing output plug-ins and vice versa.

I've been calling the software system that meets the three criteria just discussed the *Unified Logging Layer* (*ULL*). The ULL should be part of any modern logging infrastructure, as it helps the organization to collect more logs faster, more reliably, and scalably.

Fluentd

Fluentd is an open source log collector that implements the ULL and addresses each of its three requirements:

JSON as the unifying format
: Fluentd embraces JSON as the data exchange format. When logs come into Fluentd, it parses them as a sequence of JSON objects. Because JSON has become the de facto standard for data exchange while remaining human readable, it's a great candidate to satisfy the "make logs consumable machine-first, human-second" requirement of the ULL.

Reliability through file-based buffering and failover
: Fluentd ensures reliable transport by implementing a configurable buffering mechanism. Fluentd buffers log data by persisting it to disk and keeps retrying until the payload is successfully sent to the desired output system. It also implements failover so that the user can send data to a safe secondary output until the connection to the primary output is recovered.

Pluggable inputs, outputs, and more
: Fluentd implements all inputs and outputs as plug-ins. Today, the ecosystem boasts hundreds of input and output plug-ins contributed by the community, making it easy to send data from anywhere to anywhere.

Figure 3-9 illustrates these points.

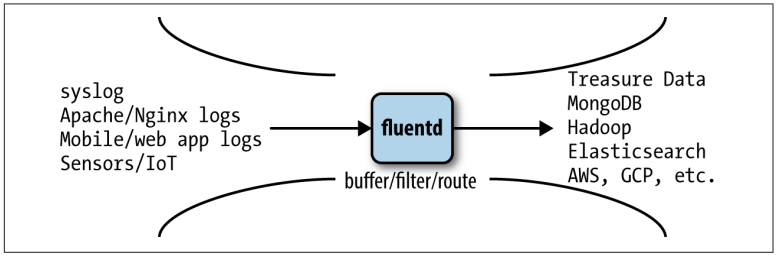


Figure 3-9. Fluentd: The Unified Logging Layer (image courtesy of Kiyoto Tamura)

What's more, Fluentd's parser (parsing incoming logs), filter (filtering parsed logs), buffer (how data is buffered), and formatter (formatting outgoing logs) are all pluggable, making it very easy to create and manage flexible and reliable logging pipelines.

You can learn more about the ULL and Fluentd on our website as well as via our GitHub repository.

Appendix: The Duality of Kafka and Fluentd

Over the last few years, I have given several talks on Fluentd, and many people have asked me how Fluentd is different from Apache Kafka. The difference is quite clear, actually. Fluentd is one of the data inputs or outputs for Kafka, and Kafka is one of the data inputs or outputs for Fluentd (see Figure 3-10).

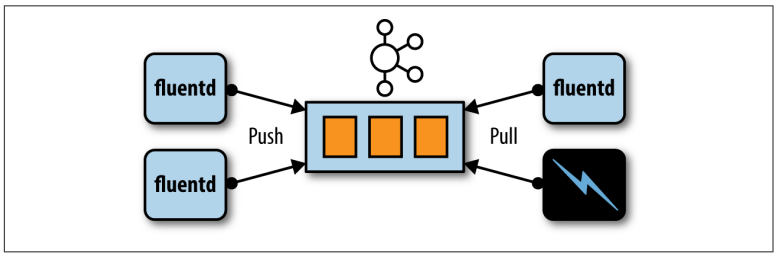


Figure 3-10. Kafka and Fluentd complement each other (image courtesy of Kiyoto Tamura)

Kafka is primarily related to *holding* log data rather than *moving* log data. Thus, Kafka producers need to write the code to put data in Kafka, and Kafka consumers need to write the code to pull data out of Kafka. Fluentd has both input and output plug-ins for Kafka so

that data engineers can write less code to get data in and out of Kafka. We have many users that use Fluentd as a Kafka producer and/or consumer.

Validating Data Models with Kafka-Based Pipelines

by Gwen Shapira

You can read this post on oreilly.com here.

A/B testing is a popular method of using business intelligence data to assess possible changes to websites. In the past, when a business wanted to update its website in an attempt to drive more sales, decisions on the specific changes to make were driven by guesses; intuition; focus groups; and ultimately, which executive yelled louder. These days, the data-driven solution is to set up multiple copies of the website, direct users randomly to the different variations, and measure which design improves sales the most. There are a lot of details to get right, but this is the gist of things.

When it comes to backend systems, however, we are still living in the stone age. Suppose your business grows significantly and you notice that your existing MySQL database is becoming less responsive as the load increases. You might consider moving to a NoSQL system, but you'll need to decide which NoSQL solution to select—there are a lot of options: Cassandra, MongoDB, Couchbase, or even Hadoop. There are also many possible data models: normalized, wide tables, narrow tables, nested data structures, and so on.

A/B Testing Multiple Data Stores and Data Models in Parallel

It is surprising how often companies choose solutions based on intuition or even which architect yelled louder. Rather than making a decision based on facts and numbers regarding capacity, scale, throughput, and data-processing patterns, the backend architecture decisions are made with fuzzy reasoning. In that scenario, what usually happens is that a data store and a data model are somehow chosen, and the entire development team will dive into a six-month project to move their entire backend system to the new thing. This project will inevitably take 12 months, and about 9 months in,

everyone will suspect that this was a bad idea, but it's way too late to do anything about it.

Note how this approach is anti-agile. Even though the effort is often done by scrum teams with stories, points, sprints, and all the other agile trappings, a methodology in which you are taking on a project without knowing if early design decisions were correct or not for six months is inherently a waterfall methodology. You can't correct course based on data because by the time you have data, you've invested too much in the project already. This model is far too inflexible and too risky compared to a model of choosing few reasonable options, testing them out for few weeks, collecting data, and proceeding based on the results.

The reason smart companies that should know better still develop data backends using this waterfall model is that the backend system is useless without data in it. Migrating data and the associated data pipelines is by far the most challenging component in testing out a new backend system. Companies do six-month backend migration projects because they have no way of testing a backend system in a two-week spike.

But what if you could? What if you could easily "split" all of your data pipelines to go through two backends instead of one? This will allow you to kick the new system a bit with your real data and check out how to generate reports, how to integrate existing software, and to find out how stable the new database is under various failure conditions. For a growing number of organizations, this is not just an interesting possibility; this is a reality.

Kafka's Place in the "Which Data Store Do We Choose" Debate

Data bus is a central component of all modern data architectures: in both Lambda or Kappa architectures, the first step in the data-processing pipeline is collecting all events in Apache Kafka. From Kafka, data can be processed in streams or batches, and because Kafka scales so well when you store more data or add more brokers, most organizations store months of data within Kafka.

If you follow the principles of agile data architectures, you will have most of your critical data organized in different topics in Kafka—not just the raw input data, but data in different stages of processing

—cleaned, validated, aggregated, and enriched. Populating a new backend with the data for a proof of concept is no longer a question of connecting many existing systems to the new backend and re-creating huge number of pipelines. With Kafka, populating a back-end system is a relatively simple matter of choosing which topics should be used and writing a consumer that reads data from these topics and inserts them into the new backend.

This means the new backend will receive data without changing a single thing about the existing systems. The data sources and the old backends simply continue on from the existing pipeline, unaware that there is a new consumer somewhere populating a new backend data store. The data architecture is truly decoupled in a way that allows you to experiment without risking existing systems.

Once the new data store and its data model is populated, you can validate things like throughput, performance, ease of querying, and anything else that will come up in the "which data store do we choose?" debate. You can even A/B test multiple data stores and multiple data models in parallel, so the whole decision process can take less time.

Next time your team debates between three different data models, there will still be opinions, intuition, and possibly even yelling, but there will also be data to help guide the decision.

CHAPTER 4

Big Data Architecture and Infrastructure

As noted in O'Reilly's 2015 Data Science Salary Survey, the same four tools—SQL, Excel, R, and Python—continue to be the most widely used in data science for the third year in a row. Spark also continues to be one of the most active projects in big data, seeing a 17% increase in users over the past 12 months. Matei Zaharia, creator of Spark, outlined in his keynote at Strata + Hadoop San Jose two new goals Spark was pursuing in 2015. The first goal was to make distributed processing tools accessible to a wide range of users, beyond big data engineers. An example of this is seen in the new DataFrames API, inspired by R and Python data frames. The second goal was to enhance integration—to allow Spark to interact efficiently in different environments, from NoSQL stores to traditional data warehouses.

In many ways, the two goals for Spark in 2015—greater accessibility for a wider user base and greater integration of tools/environments—are consistent with the changes we're seeing in architecture and infrastructure across the entire big data landscape. In this chapter, we present a collection of blog posts that reflect these changes.

Ben Lorica documents what startups like Tamr and Trifacta have learned about opening up data analysis to non-programmers. Benjamin Hindman laments the fact that we still don't have an operating system that abstracts and manages hardware resources in the data center. Jim Scott discusses his use of Myriad to

enable Mesos and YARN to work better together (but notes improvements are still needed). Yanpei Chen chronicles what happened when his team at Cloudera used SSDs instead of HDDs to assess their impact on big data (sneak preview: SSDs again offer up to 40% shorter job duration, and 70% higher performance). Finally, Shaoshan Liu discusses how to use Baidu and Tachyon with Spark SQL to increase data processing speed 30-fold.

Lessons from Next-Generation Data-Wrangling Tools

by Ben Lorica

You can read this post on oreilly.com here.

One of the trends we're following is the rise of applications that combine big data, algorithms, and efficient user interfaces. As I noted in "Big Data's Big Ideas," our interest stems from both consumer apps as well as tools that democratize data analysis. It's no surprise that one of the areas where "cognitive augmentation" is playing out is in data preparation and curation. Data scientists continue to spend a lot of their time on data wrangling, and the increasing number of (public and internal) data sources paves the way for tools that can increase productivity in this critical area.

At Strata + Hadoop World New York, two presentations from academic spinoff startups focused on data preparation and curation:

- Tamr: Mike Stonebraker's "Three Approaches to Scalable Data Curation"
- Trifacta: Joe Hellerstein and Sean Kandel's "Advantages of a Domain-Specific Language Approach to Data Transformation"

While data wrangling is just one component of a data science pipeline, and granted we're still in the early days of productivity tools in data science, some of the lessons these companies have learned extend beyond data preparation.

Scalability ~ Data Variety and Size

Not only are enterprises faced with many data stores and spreadsheets, data scientists have many more (public and internal) data sources they want to incorporate. The absence of a global data

model means integrating data silos, and data sources require tools for consolidating schemas.

Random samples are great for working through the initial phases, particularly while you're still familiarizing yourself with a new data set. Trifacta lets users work with samples while they're developing data-wrangling "scripts" that can be used on full data sets.

Empower Domain Experts

In many instances, you need subject area experts to explain specific data sets that you're not familiar with. These experts can place data in context and are usually critical in helping you clean and consolidate variables. Trifacta has tools that enable non-programmers to take on data-wrangling tasks that previously required a fair amount of scripting.

Consider DSLs and Visual Interfaces

> Programs written in a [domain specific language] (DSL) also have one other important characteristic: they can often be written by non-programmers…a user immersed in a domain already knows the domain semantics. All the DSL designer needs to do is provide a notation to express that semantics.
>
> —Paul Hudak, 1997

I've often used regular expressions for data wrangling, only to come back later unable to read the code I wrote (Joe Hellerstein describes regex as "meant for writing and never reading again"). Programs written in DSLs are concise, easier to maintain, and can often be written by non-programmers.

Trifacta designed a "readable" DSL for data wrangling but goes one step further: Its users "live in visualizations, not code." Its elegant visual interface is designed to accomplish most data-wrangling tasks, but it also lets users access and modify accompanying scripts written in Trifacta's DSL (power users can also use regular expressions).

These ideas go beyond data wrangling. Combining DSLs with visual interfaces can open up other aspects of data analysis to non-programmers.

Intelligence and Automation

If you're dealing with thousands of data sources, then you'll need tools that can automate routine steps. Tamr's next-generation extract, transform, load (ETL) platform uses machine learning in a variety of ways, including schema consolidation and expert (crowd) sourcing.

Many data analysis tasks involve a handful of data sources that require painstaking data wrangling along the way. Scripts to automate data preparation are needed for replication and maintenance. Trifacta looks at user behavior and context to produce "utterances" of its DSL, which users can then edit or modify.

Don't Forget About Replication

If you believe the adage that data wrangling consumes a lot of time and resources, then it goes without saying that tools like Tamr and Trifacta should produce reusable scripts and track lineage. Other aspects of data science—for example, model building, deployment, and maintenance—need tools with similar capabilities.

Why the Data Center Needs an Operating System

by Benjamin Hindman

You can read this post on oreilly.com here.

Developers today are building a new class of applications. These applications no longer fit on a single server, but instead run across a fleet of servers in a data center. Examples include analytics frameworks like Apache Hadoop and Apache Spark, message brokers like Apache Kafka, key/value stores like Apache Cassandra, as well as customer-facing applications such as those run by Twitter and Netflix.

These new applications are more than applications—they are distributed systems. Just as it became commonplace for developers to build multithreaded applications for single machines, it's now becoming commonplace for developers to build distributed systems for data centers.

But it's difficult for developers to build distributed systems, and it's difficult for operators to run distributed systems. Why? Because we expose the wrong level of abstraction to both developers and operators: machines.

Machines Are the Wrong Abstraction

Machines are the wrong level of abstraction for building and running distributed applications. Exposing machines as the abstraction to developers unnecessarily complicates the engineering, causing developers to build software constrained by machine-specific characteristics, like IP addresses and local storage. This makes moving and resizing applications difficult if not impossible, forcing maintenance in data centers to be a highly involved and painful procedure.

With machines as the abstraction, operators deploy applications in anticipation of machine loss, usually by taking the easiest and most conservative approach of deploying one application per machine. This almost always means machines go underutilized, as we rarely buy our machines (virtual or physical) to exactly fit our applications, or size our applications to exactly fit our machines.

By running only one application per machine, we end up dividing our data center into highly static, highly inflexible partitions of machines, one for each distributed application. We end up with a partition that runs analytics, another that runs the databases, another that runs the web servers, another that runs the message queues, and so on. And the number of partitions is only bound to increase as companies replace monolithic architectures with service-oriented architectures and build more software based on microservices.

What happens when a machine dies in one of these static partitions? Let's hope we over-provisioned sufficiently (wasting money), or can re-provision another machine quickly (wasting effort). What about when the web traffic dips to its daily low? With static partitions, we allocate for peak capacity, which means when traffic is at its lowest, all of that excess capacity is wasted. This is why a typical data center runs at only 8%–15% efficiency. And don't be fooled just because you're running in the cloud: You're still being charged for the resources your application is not using on each virtual machine (someone is benefiting—it's just your cloud provider, not you).

And finally, with machines as the abstraction, organizations must employ armies of people to manually configure and maintain each individual application on each individual machine. People become the bottleneck for trying to run new applications, even when there are ample resources already provisioned that are not being utilized.

If My Laptop Were a Data Center

Imagine if we ran applications on our laptops the same way we run applications in our data centers. Each time we launched a web browser or text editor, we'd have to specify which CPU to use, which memory modules are addressable, which caches are available, and so on. Thankfully, our laptops have an operating system that abstracts us away from the complexities of manual resource management.

In fact, we have operating systems for our workstations, servers, mainframes, supercomputers, and mobile devices, each optimized for their unique capabilities and form factors.

We've already started treating the data center itself as one massive warehouse-scale computer. Yet, we still don't have an operating system that abstracts and manages the hardware resources in the data center just like an operating system does on our laptops.

It's Time for the Data Center OS

What would an operating system for the data center look like?

From an operator's perspective, it would span all of the machines in a data center (or cloud) and aggregate them into one giant pool of resources on which applications would be run. You would no longer configure specific machines for specific applications; all applications would be capable of running on any available resources from any machine, even if there are other applications already running on those machines.

From a developer's perspective, the data center operating system would act as an intermediary between applications and machines, providing common primitives to facilitate and simplify building distributed applications.

The data center operating system would not need to replace Linux or any other host operating systems we use in our data centers today. The data center operating system would provide a software stack on top of the host operating system. Continuing to use the host operat-

ing system to provide standard execution environments is critical to immediately supporting existing applications.

The data center operating system would provide functionality for the data center that is analogous to what a host operating system provides on a single machine today: namely, resource management and process isolation. Just like with a host operating system, a data center operating system would enable multiple users to execute multiple applications (made up of multiple processes) concurrently, across a shared collection of resources, with explicit isolation between those applications.

An API for the Data Center

Perhaps the defining characteristic of a data center operating system is that it provides a software interface for building distributed applications. Analogous to the system call interface for a host operating system, the data center operating system API would enable distributed applications to allocate and deallocate resources, launch, monitor, and destroy processes, and more. The API would provide primitives that implement common functionality that all distributed systems need. Thus, developers would no longer need to independently re-implement fundamental distributed systems primitives (and inevitably, independently suffer from the same bugs and performance issues).

Centralizing common functionality within the API primitives would enable developers to build new distributed applications more easily, more safely, and more quickly. This is reminiscent of when virtual memory was added to host operating systems. In fact, one of the virtual memory pioneers wrote that "it was pretty obvious to the designers of operating systems in the early 1960s that automatic storage allocation could significantly simplify programming."

Example Primitives

Two primitives specific to a data center operating system that would immediately simplify building distributed applications are service discovery and coordination. Unlike on a single host where very few applications need to discover other applications running on the same host, discovery is the norm for distributed applications. Likewise, most distributed applications achieve high availability and fault tolerance through some means of coordination and/or consen-

sus, which is notoriously hard to implement correctly and efficiently.

Developers today are forced to choose between existing tools for service discovery and coordination, such as Apache ZooKeeper and CoreOS' etcd. This forces organizations to deploy multiple tools for different applications, significantly increasing operational complexity and maintainability.

Having the data center operating system provide primitives for discovery and coordination not only simplifies development, it also enables application portability. Organizations can change the underlying implementations without rewriting the applications, much like you can choose between different filesystem implementations on a host operating system today.

A New Way to Deploy Applications

With a data center operating system, a software interface replaces the human interface that developers typically interact with when trying to deploy their applications today; rather than a developer asking a person to provision and configure machines to run their applications, developers launch their applications using the data center operating system (e.g., via a CLI or GUI), and the application executes using the data center operating system's API.

This supports a clean separation of concerns between operators and users: Operators specify the amount of resources allocatable to each user, and users launch whatever applications they want, using whatever resources are available to them. Because an operator now specifies *how much* of any type of resource is available, but not which *specific* resource, a data center operating system, and the distributed applications running on top, can be more intelligent about which resources to use in order to execute more efficiently and better handle failures. Because most distributed applications have complex scheduling requirements (think Apache Hadoop) and specific needs for failure recovery (think of a database), empowering software to make decisions instead of humans is critical for operating efficiently at data-center scale.

The "Cloud" Is Not an Operating System

Why do we need a new operating system? Didn't Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) already solve these problems?

IaaS doesn't solve our problems because it's still focused on machines. It isn't designed with a software interface intended for applications to use in order to execute. IaaS is designed for humans to consume, in order to provision virtual machines that other humans can use to deploy applications; IaaS turns machines into more (virtual) machines, but does not provide any primitives that make it easier for a developer to build distributed applications on top of those machines.

PaaS, on the other hand, abstracts away the machines, but is still designed first and foremost to be consumed by a human. Many PaaS solutions do include numerous tangential services and integrations that make building a distributed application easier, but not in a way that's portable across other PaaS solutions.

Apache Mesos: The Distributed Systems Kernel

Distributed computing is now the norm, not the exception, and we need a data center operating system that delivers a layer of abstraction and a portable API for distributed applications. Not having one is hindering our industry. Developers should be able to build distributed applications without having to reimplement common functionality. Distributed applications built in one organization should be capable of being run in another organization easily.

Existing cloud computing solutions and APIs are not sufficient. Moreover, the data center operating system API must be built, like Linux, in an open and collaborative manner. Proprietary APIs force lock-in, deterring a healthy and innovative ecosystem from growing. It's time we created the POSIX for distributed computing: a portable API for distributed systems running in a data center or on a cloud.

The open source Apache Mesos project, of which I am one of the co-creators and the project chair, is a step in that direction. Apache Mesos aims to be a distributed systems kernel that provides a portable API upon which distributed applications can be built and run.

Many popular distributed systems have already been built directly on top of Mesos, including Apache Spark, Apache Aurora, Airbnb's Chronos, and Mesosphere's Marathon. Other popular distributed systems have been ported to run on top of Mesos, including Apache Hadoop, Apache Storm, and Google's Kubernetes, to list a few.

Chronos is a compelling example of the value of building on top of Mesos. Chronos, a distributed system that provides highly available and fault-tolerant cron, was built on top of Mesos in only a few thousand lines of code and without having to do any explicit socket programming for network communication.

Companies like Twitter and Airbnb are already using Mesos to help run their data centers, while companies like Google have been using in-house solutions they built almost a decade ago. In fact, just like Google's MapReduce spurred an industry around Apache Hadoop, Google's in-house data center solutions have had close ties with the evolution of Mesos.

While not a complete data center operating system, Mesos, along with some of the distributed applications running on top, provide some of the essential building blocks from which a full data center operating system can be built: the kernel (Mesos), a distributed init.d (Marathon/Aurora), cron (Chronos), and more.

A Tale of Two Clusters: Mesos and YARN

by Jim Scott

You can read this post on oreilly.com here.

This is a tale of two siloed clusters. The first cluster is an Apache Hadoop cluster. This is an island whose resources are completely isolated to Hadoop and its processes. The second cluster is the description I give to all resources that are not a part of the Hadoop cluster. I break them up this way because Hadoop manages its own resources with Apache YARN (Yet Another Resource Negotiator). Although this is nice for Hadoop, all too often those resources are underutilized when there are no big data workloads in the queue. And then when a big data job comes in, those resources are stretched to the limit, and they are likely in need of more resources. That can be tough when you are on an island.

Hadoop was meant to tear down walls—albeit, data silo walls—but walls, nonetheless. What has happened is that while tearing some walls down, other types of walls have gone up in their place.

Another technology, Apache Mesos, is also meant to tear down walls—but Mesos has often been positioned to manage the "second cluster," which are all of those other, non-Hadoop workloads.

This is where the story really starts, with these two silos of Mesos and YARN (see Figure 4-1). They are often pitted against each other, as if they were incompatible. It turns out they work together, and therein lies my tale.

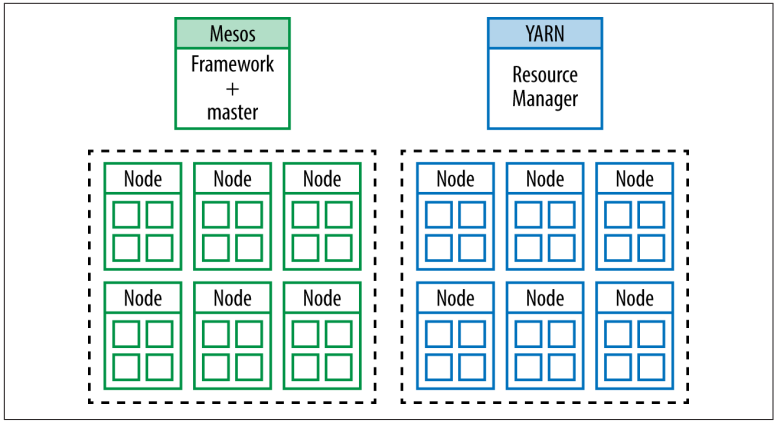


Figure 4-1. Isolated clusters (image courtesy of Mesosphere and MapR, used with permission)

Brief Explanation of Mesos and YARN

The primary difference between Mesos and YARN is around their design priorities and how they approach scheduling work. Mesos was built to be a scalable global resource manager for the entire data center. It was designed at UC Berkeley in 2007 and hardened in production at companies like Twitter and Airbnb. YARN was created out of the necessity to scale Hadoop. Prior to YARN, resource management was embedded in Hadoop MapReduce V1, and it had to be removed in order to help MapReduce scale. The MapReduce 1 JobTracker wouldn't practically scale beyond a couple thousand machines. The creation of YARN was essential to the next iteration of Hadoop's lifecycle, primarily around scaling.

Mesos Scheduling

Mesos determines which resources are available, and it makes offers back to an application scheduler (the application scheduler and its executor is called a "framework"). Those offers can be accepted or rejected by the framework. This model is considered a non-monolithic model because it is a "two-level" scheduler, where scheduling algorithms are pluggable. Mesos allows an infinite number of schedule algorithms to be developed, each with its own strategy for which offers to accept or decline, and can accommodate thousands of these schedulers running multitenant on the same cluster.

The two-level scheduling model of Mesos allows each framework to decide which algorithms it wants to use for scheduling the jobs that it needs to run. Mesos plays the arbiter, allocating resources across multiple schedulers, resolving conflicts, and making sure resources are fairly distributed based on business strategy. Offers come in, and the framework can then execute a task that consumes those offered resources. Or the framework has the option to decline the offer and wait for another offer to come in. This model is very similar to how multiple apps all run simultaneously on a laptop or smartphone, in that they spawn new threads or request more memory as they need it, and the operating system arbitrates among all of the requests. One of the nice things about this model is that it is based on years of operating system and distributed systems research and is very scalable. This is a model that Google and Twitter have proven at scale.

YARN Scheduling

Now, let's look at what happens over on the YARN side. When a job request comes into the YARN resource manager, YARN evaluates all the resources available, and it places the job. It's the one making the decision where jobs should go; thus, it is modeled in a monolithic way. It is important to reiterate that YARN was created as a necessity for the evolutionary step of the MapReduce framework. YARN took the resource-management model out of the MapReduce 1 JobTracker, generalized it, and moved it into its own separate ResourceManager component, largely motivated by the need to scale Hadoop jobs.

YARN is optimized for scheduling Hadoop jobs, which are historically (and still typically) batch jobs with long run times. This means that YARN was not designed for long-running services, nor for

short-lived interactive queries (like small and fast Spark jobs), and while it's possible to have it schedule other kinds of workloads, this is not an ideal model. The resource demands, execution model, and architectural demands of MapReduce are very different from those of long-running services, such as web servers or SOA applications, or real-time workloads like those of Spark or Storm. Also, YARN was designed for stateless batch jobs that can be restarted easily if they fail. It does not handle running stateful services like distributed filesystems or databases. While YARN's monolithic scheduler could theoretically evolve to handle different types of workloads (by merging new algorithms upstream into the scheduling code), this is not a lightweight model to support a growing number of current and future scheduling algorithms.

Is It YARN Versus Mesos?

When comparing YARN and Mesos, it is important to understand the general scaling capabilities and why someone might choose one technology over the other. While some might argue that YARN and Mesos are competing for the same space, they really are not. The people who put these models in place had different intentions from the start, and that's OK. There is nothing explicitly wrong with either model, but each approach will yield different long-term results. I believe this is the key between when to use one, the other, or both. Mesos was built at the same time as Google's Omega. Ben Hindman and the Berkeley AMPlab team worked closely with the team at Google designing Omega so that they both could learn from the lessons of Google's Borg and build a better nonmonolithic scheduler.

When you evaluate how to manage your data center as a whole, on one side you've got Mesos (which can manage all the resources in your data center), and on the other, you have YARN (which can safely manage Hadoop jobs, but is not capable of managing your entire data center). Data center operators tend to solve for these two use cases by partitioning their clusters into Hadoop and non-Hadoop worlds.

Using Mesos and YARN in the same data center, to benefit from both resource managers, currently requires that you create two static partitions. Using both would mean that certain resources would be dedicated to Hadoop for YARN to manage and Mesos would get the

rest. It might be oversimplifying it, but that is effectively what we are talking about here. Fundamentally, this is the issue we want to avoid.

Introducing Project Myriad

This leads us to the question: Can we make YARN and Mesos work together? Can we make them work harmoniously for the benefit of the enterprise and the data center? The answer is "yes." A few well-known companies—eBay, MapR, and Mesosphere—collaborated on a project called Myriad.

This open source software project is both a Mesos framework and a YARN scheduler that enables Mesos to manage YARN resource requests. When a job comes into YARN, it will schedule it via the Myriad Scheduler, which will match the request to incoming Mesos resource offers. Mesos, in turn, will pass it on to the Mesos worker nodes. The Mesos nodes will then communicate the request to a Myriad executor which is running the YARN node manager. Myriad launches YARN node managers on Mesos resources, which then communicate to the YARN resource manager what resources are available to them. YARN can then consume the resources as it sees fit. As shown in Figure 4-2, Myriad provides a seamless bridge from the pool of resources available in Mesos to the YARN tasks that want those resources.

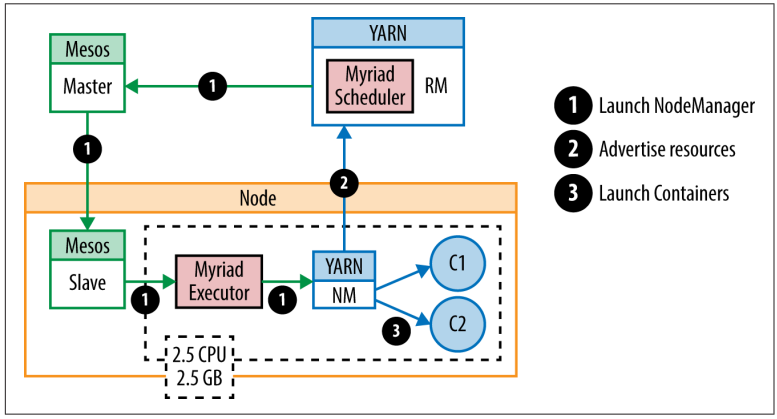


Figure 4-2. How Myriad works (image courtesy of Mesosphere and MapR, used with permission)

The beauty of this approach is that not only does it allow you to elastically run YARN workloads on a shared cluster, but it actually

makes YARN more dynamic and elastic than it was originally designed to be. This approach also makes it easy for a data center operations team to expand resources given to YARN (or, take them away, as the case might be) without ever having to reconfigure the YARN cluster. It becomes very easy to dynamically control your entire data center. This model also provides an easy way to run and manage multiple YARN implementations, even different versions of YARN on the same cluster.

Myriad blends the best of both the YARN and Mesos worlds. By utilizing Myriad, Mesos and YARN can collaborate, and you can achieve an as-it-happens business. Data analytics can be performed in-place on the same hardware that runs your production services. No longer will you face the resource constraints (and low utilization) caused by static partitions. Resources can be elastically reconfigured to meet the demands of the business as it happens (see Figure 4-3).

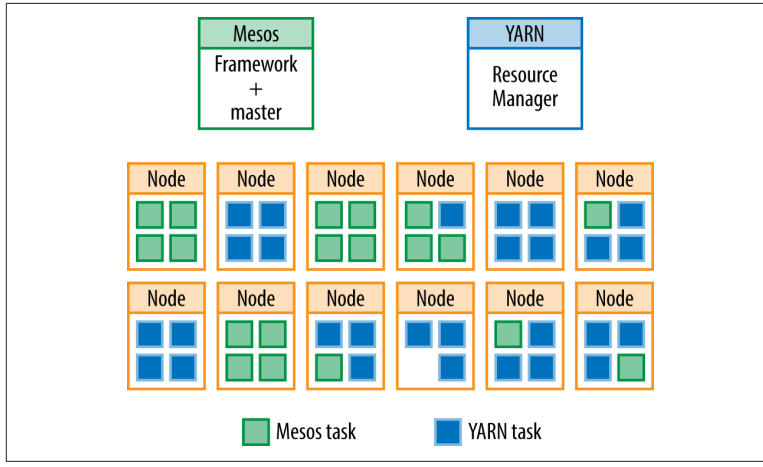


Figure 4-3. Resource sharing (image courtesy of Mesosphere and MapR, used with permission)

Final Thoughts

To make sure people understand where I am coming from here, I feel that both Mesos and YARN are very good at what they were built to achieve, yet both have room for improvement. Both resource managers can improve in the area of security; security support is paramount to enterprise adoption.

Mesos needs an end-to-end security architecture, and I personally would not draw the line at Kerberos for security support, as my personal experience with it is not what I would call "fun." The other area for improvement in Mesos—which can be extremely complicated to get right—is what I will characterize as resource revocation and preemption. Imagine the use case where all resources in a business are allocated and then the need arises to have the single most important "thing" that your business depends on run—even if this task only requires minutes of time to complete, you are out of luck if the resources are not available. Resource preemption and/or revocation could solve that problem. There are currently ways around this in Mesos today, but I look forward to the work the Mesos committers are doing to solve this problem with Dynamic Reservations and Optimistic (Revocable) Resources Offers.

Myriad is an enabling technology that can be used to take advantage of leveraging all of the resources in a data center or cloud as a single pool of resources. Myriad enables businesses to tear down the walls between isolated clusters, just as Hadoop enabled businesses to tear down the walls between data silos. With Myriad, developers will be able to focus on the data and applications on which the business depends, while operations will be able to manage compute resources for maximum agility. This opens the door to being able to focus on data instead of constantly worrying about infrastructure. With Myriad, the constraints on the storage network and coordination between compute and data access are the last-mile concern to achieve full flexibility, agility, and scale.

Project Myriad is hosted on GitHub and is available for download. There's documentation there that provides more in-depth explanations of how it works. You'll even see some nice diagrams. Go out, explore, and give it a try.

The Truth About MapReduce Performance on SSDs

by Yanpei Chen (with Karthik Kambatla)

You can read this post on oreilly.com here.

It is a well-known fact that solid-state drives (SSDs) are fast and expensive. But exactly how much faster—and more expensive—are they than the hard disk drives (HDDs) they're supposed to replace? And does anything change for big data?

I work on the performance engineering team at Cloudera, a data management vendor. It is my job to understand performance implications across customers and across evolving technology trends. The convergence of SSDs and big data does have the potential to broadly impact future data center architectures. When one of our hardware partners loaned us a number of SSDs with the mandate to "find something interesting," we jumped on the opportunity. This post shares our findings.

As a starting point, we decided to focus on MapReduce. We chose MapReduce because it enjoys wide deployment across many industry verticals—even as other big data frameworks such as SQL-on-Hadoop, free text search, machine learning, and NoSQL gain prominence.

We considered two scenarios: first, when setting up a new cluster, we explored whether SSDs or HDDs, of equal aggregate bandwidth, are superior; second, we explored how cluster operators should configure SSDs, when upgrading an HDDs-only cluster.

SSDs Versus HDDs of Equal Aggregate Bandwidth

For our measurements, we used the storage configuration in the following table (the machines were Intel Xeon 2-socket, 8-core, 16-thread systems, with 10 GBps Ethernet and 48 GB RA):

Setup	Storage	Capacity	Sequential R/W bandwidth	Price
HDD	11 HDDs	22 TB	1300 MBps	$4,400
SSD	1 SSD	1.3 TB	1300 MBps	$14,000

By their physical design, SSDs avoid the large seek overhead of small, random I/O for HDDs. SSDs do perform much better for shuffle-heavy MapReduce jobs. In the graph shown in Figure 4-4, "terasort" is a common benchmark with 1:1:1 ratio between input:shuffle:output sizes; "shuffle" is a shuffle-only job that we wrote in-house to purposefully stress only the shuffle part of MapReduce. SSDs offer as much as 40% lower job duration, which translates to 70% higher performance. A common but incomplete mental model assumes that MapReduce contains only large, sequential read and writes. MapReduce does exhibit large, sequential I/O when reading input from and writing output to HDFS. The intermediate shuffle stage, in contrast, involves smaller read and writes. The output of each map task is partitioned across many reducers in the job, and each reduce task fetches only its own data. In our customer workloads, this led to each reduce task accessing as little as a few MBs from each map task.

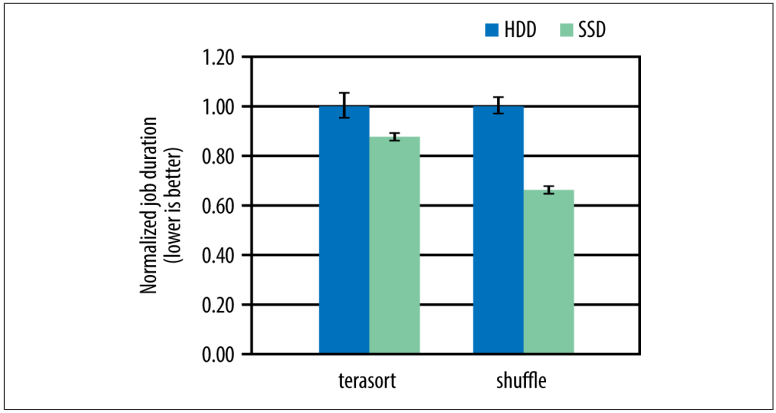


Figure 4-4. Terasort and shuffle job durations, using HDDs and SSDs (graph courtesy of Yanpei Chen)

To our initial surprise, we learned that SSDs also benefit MapReduce jobs that involve only HDFS reads and writes, despite HDDs having the same aggregate sequential bandwidth according to hardware specs. In the graph shown in Figure 4-5, "teragen" writes data to HDFS with three-fold replication, "teravalidate" reads the output of terasort and checks if they are in sorted order, and "hdfs data write" is a job we wrote in-house and writes data to HDFS with single-fold replication. SSDs again offer up to 40% lower job duration, equating to 70% higher performance.

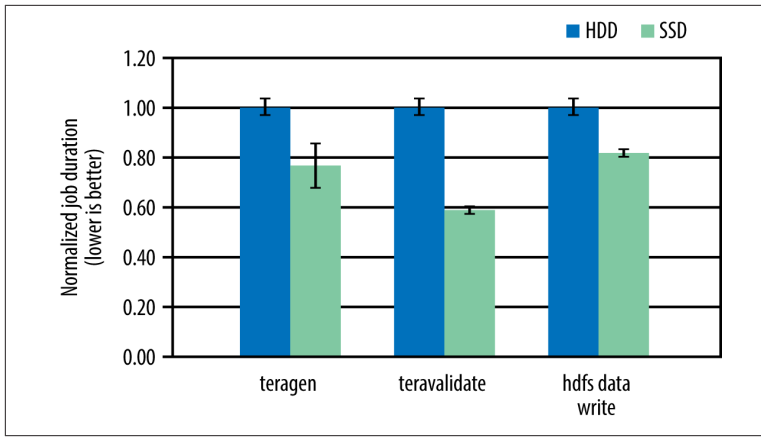


Figure 4-5. Teragen, teravalidate, and HDFS data write job durations using HDDs and SSDs (graph courtesy of Yanpei Chen)

It turns out that our SSDs have an advantage for sequential workloads because they deliver higher sequential I/O size—2x larger than the HDDs in our test setup. To write the same amount of data, SSDs incur half the number of I/Os. This difference may be a vendor-specific characteristic, as other SSDs or HDDs likely offer different default configurations for sequential I/O sizes.

There is another kind of MapReduce job—one that is dominated by compute rather than I/O. When the resource bottleneck is not the I/O subsystem, the choice of storage media makes no difference. In the graph shown in Figure 4-6, "wordcount" is a job that involves high CPU load parsing text and counting word frequencies; "shuffle compressed" is the shuffle-only job from earlier, except with MapReduce shuffle compression enabled. Enabling this configuration shifts load from I/O to CPU. The advantage from SSDs decreases considerably compared with the uncompressed "shuffle" from earlier.

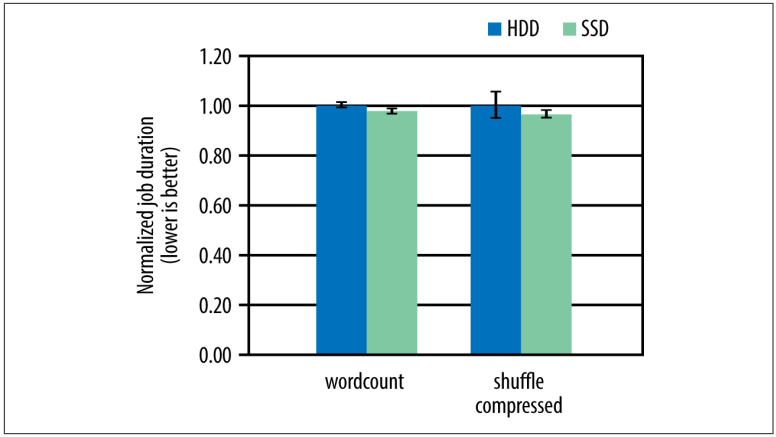


Figure 4-6. Wordcount and shuffle compressed job durations using HDDs and SSDs (graph courtesy of Yanpei Chen)

Ultimately, we learned that SSDs offer considerable performance benefit for some workloads, and at worst do no harm. The decision on whether to use SSDs would then depend on any premium cost to obtain higher performance. We'll return to that discussion later.

Configuring a Hybrid HDD-SSD Cluster

Almost all existing MapReduce clusters use HDDs. There are two ways to introduce SSDs: (1) buy a new SSD-only cluster, or (2) add SSDs to existing HDD-only machines (some customers may prefer the latter option for cost and logistical reasons). Therefore, we found it meaningful to figure out a good way to configure a hybrid HDD-SSD cluster.

We set up clusters with the following storage configurations:

Setup	Storage	Capacity	Sequential R/W bandwidth	Price
HDD-baseline	6 HDDs	12 TB	720 MBps	$2,400
HDD-11	11 HDDs	22 TB	1300 MBps	$4,400
Hybrid	6 HDDs + 1 SSD	13.3 TB	2020 MBps	$16,400

We started with a low-I/O-bandwidth cluster of six HDDs. With default configurations, adding a single SSD leads to higher performance, about the same improvement we get by adding five HDDs. This is an undesirable result, because the single additional SSD has

double the bandwidth than the additional five HDDs (see Figure 4-7).

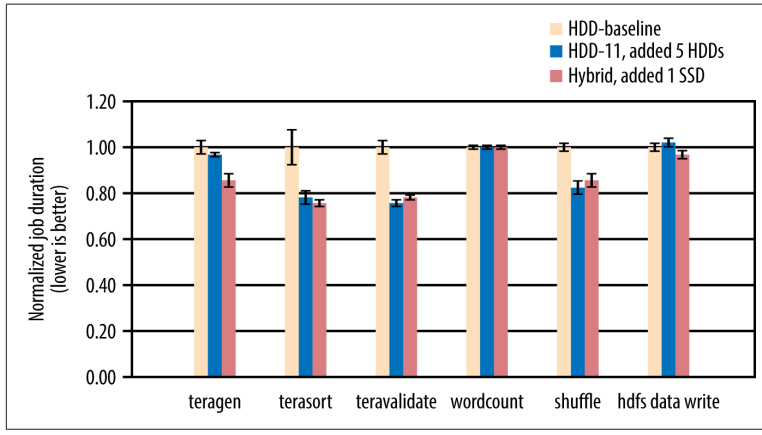


Figure 4-7. Job durations across six HDD clusters (graph courtesy of Yanpei Chen)

A closer look at HDFS and MapReduce implementations reveals a critical insight: Both the HDFS DataNode and the MapReduce NodeManager write to local directories in a round-robin fashion. A typical setup would mount each piece of storage hardware as a separate directory—for example, */mnt/disk-1*, */mnt/disk-2*, */mnt/ssd-1*. With each of these directories mounted as an HDFS and MapReduce local directory, they each receive the same amount of data. Faster progress on the SSD does not accelerate slower progress on the HDDs.

So, to fully utilize the SSD, we need to split the SSD into multiple directories to maintain equal bandwidth per local directory. In our case, SSDs should be split into 10 directories. The SSDs would then receive 10x the data directed at each HDD, written at 10x the speed, and complete in the same amount of time. When the SSD capacity accommodates the 10x data size written, performance is much better than the default setup (see Figure 4-8).

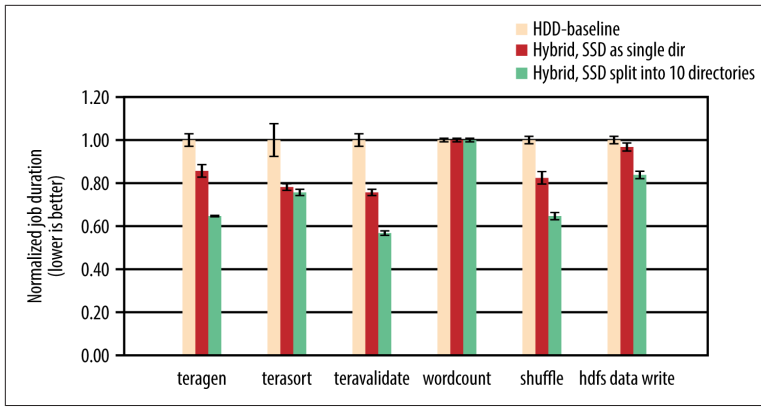


Figure 4-8. Job durations across six hybrid clusters (graph courtesy of Yanpei Chen)

Price Per Performance Versus Price Per Capacity

We found that for our tests and hardware, SSDs delivered up to 70% higher performance, for 2.5x higher $-per-performance (average performance divided by cost). Each customer can decide whether the higher performance is worth the premium cost. This decision employs the $-per-performance metric, which differs from the $-per-capacity metric that storage vendors more frequently track. The SSDs we used hold a 50x premium for $-per-capacity—a gap far larger than the 2.5x premium for $-per-performance.

The primary benefit of SSD is high performance, rather than high capacity. Storage vendors and customers should also consider $-per-performance, and develop architectures to work around capacity constraints.

The following table compares the $-per-performance and $-per-capacity between HDDs and SSDs (we also include some updated data we received from different hardware partners earlier this year; the $-per-performance gap is approaching parity even as the $-per-capacity gap remains wide).

Setup	Unit cost	Capacity	Unit BW	US$ per TB	US$ per MBps
HDD circa 2013	$400	2 TB	120 MBps	200 (2013 baseline)	3.3 (2013 baseline)
SSD circa 2013	$14,000	1.3 TB	1300 MBps	10,769 (54x 2013 baseline)	10.8 (2.5x 2013 baseline)
HDD circa 2015	$250	4 TB	120 MBps	62.5 (2015 baseline)	2.1 (2015 baseline)
SSD circa 2015	$6,400	2 TB	2000 MBps	3,200 (51x 2015 baseline)	3.2 (1.5x 2015 baseline)

SSD Economics—Exploring the Trade-Offs

Overall, SSD economics involve the interplay between ever-improving software and hardware as well as ever-evolving customer workloads. The precise trade-off between SSDs, HDDs, and memory deserves regular reexamination over time.

We encourage members of the community to extend our work and explore how SSDs benefit SQL-on-Hadoop, free text search, machine learning, NoSQL, and other big data frameworks.

More extended versions of this work appeared on the Cloudera Engineering Blog and at the Large Installation System Administration Conference (LISA) 2014.

Accelerating Big Data Analytics Workloads with Tachyon

by Shaoshan Liu

You can read this post on oreilly.com here.

As an early adopter of Tachyon, I can testify that it lives up to its description as "a memory-centric distributed storage system, enabling reliable data sharing at memory-speed, across cluster frameworks." Besides being reliable and having memory-speed, Tachyon also provides a means to expand *beyond memory* to provide enough storage capacity.

As a senior architect at Baidu USA, I've spent the past nine months incorporating Tachyon into Baidu's big data infrastructure. Since then, we've seen a 30-fold increase in speed in our big data analytics

workloads. In this post, I'll share our experiences and the lessons we've learned during our journey of adopting and scaling Tachyon.

Creating an Ad-Hoc Query Engine

Baidu is the biggest search engine in China, and the second biggest search engine in the world. Put simply, we have a lot of data. How to manage the scale of this data, and quickly extract meaningful information, has always been a challenge.

To give you an example, product managers at Baidu need to track top queries that are submitted to Baidu daily. They take the top 10 queries, and drill down to find out which province of China contributes the most information to the top queries. Product managers then analyze the resulting data to extract meaningful business intelligence.

Due to the sheer volume of data, however, each query would take tens of minutes, to hours, just to finish—leaving product managers waiting hours before they could enter the next query. Even more frustrating was that modifying a query would require running the whole process all over again. About a year ago, we realized the need for an ad-hoc query engine. To get started, we came up with a high level of specification: The query engine would need to manage petabytes of data and finish 95% of queries within 30 seconds.

From Hive, to Spark SQL, to Tachyon

With this specification in mind, we took a close look at our original query system, which ran on Hive (a query engine on top of Hadoop). Hive was able to handle a large amount of data and provided very high throughput. The problem was that Hive is a batch system and is not suitable for interactive queries.

So, why not just change the engine?

We switched to Spark SQL as our query engine (many use cases have demonstrated its superiority over Hadoop Map Reduce in terms of latency). We were excited and expected Spark SQL to drop the average query time to within a few minutes. Still, it did not quite get us all the way. While Spark SQL did help us achieve a 4-fold increase in the speed of our average query, each query still took around 10 minutes to complete.

So, we took a second look and dug into more details. It turned out that the issue was not CPU—rather, the queries were stressing the *network*. Since the data was distributed over multiple data centers, there was a high probability that a query would hit a remote data center in order to pull data over to the compute center—this is what caused the biggest delay when a user ran a query. With different hardware specifications for the storage nodes and the compute nodes, the answer was not as simple as moving the compute nodes to the data center. We decided we could solve the problem with a cache layer that buffered the frequently used data, so that most of the queries would hit the cache layer without leaving the data center.

A High-Performance, Reliable Cache Layer

We needed a cache layer that could provide high performance and reliability, and manage a petabyte-scale of data. We developed a query system that used Spark SQL as its compute engine, and Tachyon as a cache layer, and we stress tested for a month. For our test, we used a standard query within Baidu, which pulled 6 TB of data from a remote data center, and then we ran additional analysis on top of the data.

The performance was amazing. With Spark SQL alone, it took 100–150 seconds to finish a query; using Tachyon, where data may hit local or remote Tachyon nodes, it took 10–15 seconds. And if all of the data was stored in Tachyon local nodes, it took about 5 seconds, flat—a 30-fold increase in speed. Based on these results, and the system's reliability, we built a full system around Tachyon and Spark SQL.

Tachyon and Spark SQL

The anatomy of the system, as shown in Figure 4-9:

Operation manager
: A persistent Spark application that wraps Spark SQL. It accepts queries from query UI, and performs query parsing and optimization.

View manager
: Manages cache metadata and handles query requests from the operation manager.

Tachyon
: Serves as a cache layer in the system and buffers the frequently used data.

Data warehouse
: The remote data center that stores the data in HDFS-based systems.

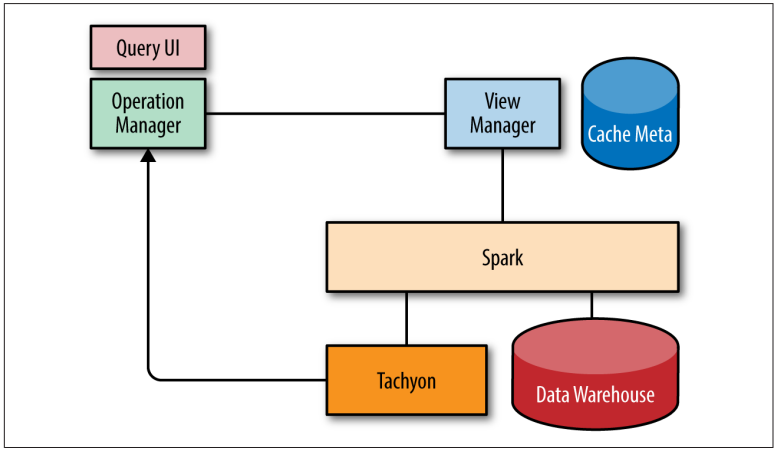


Figure 4-9. Overview of the Tachyon and Spark SQL system (courtesy of Shaoshan Liu)

Now, let's discuss the physiology of the system:

1. A query gets submitted. The operation manager analyzes the query and asks the view manager if the data is already in Tachyon.
2. If the data is already in Tachyon, the operation manager grabs the data from Tachyon and performs the analysis on it.
3. If data is not in Tachyon, then it is a cache miss, and the operation manager requests data directly from the data warehouse. Meanwhile, the view manager initiates another job to request the same data from the data warehouse and stores the data in Tachyon. This way, the next time the same query gets submitted, the data is already in Tachyon.

How to Get Data from Tachyon

After the Spark SQL query analyzer (Catalyst) performs analysis on the query, it sends a physical plan, which contains HiveTableScan statements (see Figure 4-10). These statements specify the address of the requested data in the data warehouse. The HiveTableScan statements identify the table name, attribute names, and partition keys. Note that the cache metadata is a key/value store, the <table name, partition keys, attribute names> tuple is the key to the cache metadata, and the value is the name of the Tachyon file. So, if the requested <table name, partition keys, attribute names> combination is already in cache, we can simply replace these values in the HiveTableScan with a Tachyon filename, and then the query statement knows to pull data from Tachyon instead of the data warehouse.

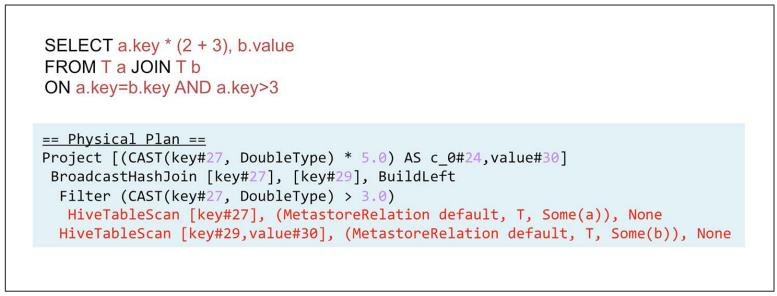


Figure 4-10. Physical plan showing HiveTableScan statements (courtesy of Shaoshan Liu)

Performance and Deployment

With the system deployed, we also measured its performance using a typical Baidu query. Using the original Hive system, it took more than 1,000 seconds to finish a typical query (see Figure 4-11). With the Spark SQL-only system, it took 150, and using our new Tachyon + Spark SQL system, it took about 20 seconds. We achieved a 50-fold increase in speed and met the interactive query requirements we set out for the project.

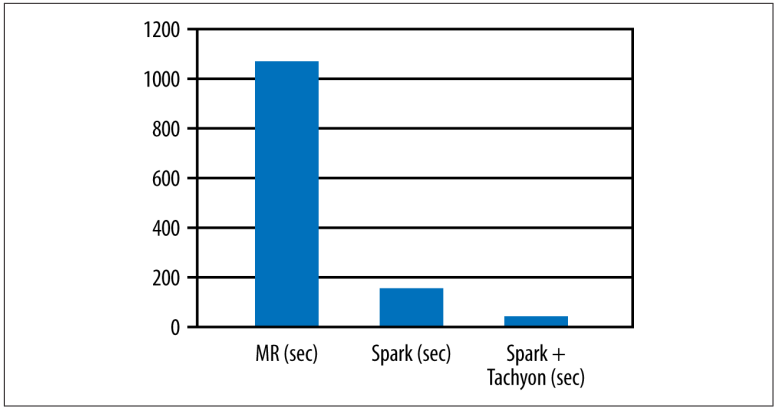


Figure 4-11. Query performance times using Hive, Spark SQL, and Tachyon + Spark SQL systems (image courtesy of Shaoshan Liu)

Today, the system is deployed in a cluster with more than 100 nodes, providing more than two petabytes of cache space, using an advanced feature (tiered storage) in Tachyon. This feature allows us to use memory as the top-level cache, SSD as the second-level cache, and HDD as the last-level cache; with all of these storage mediums combined, we are able to provide two petabytes of storage space.

Problems Encountered in Practice

Cache

The first time we used Tachyon, we were shocked—it was not able to cache anything! What we discovered was that Tachyon would only cache a block if the whole block was read into Tachyon. For example, if the block size was 512 MB, and you read 511 MB, the block would not cache. Once we understood the mechanism, we developed a workaround. The Tachyon community is also developing a page-based solution so that the cache granularity is 4 KB instead of the block size.

Locality

The second problem we encountered was when we launched a Spark job, the Spark UI told us that the data was node-local, meaning that we should not have to pull data from remote Tachyon nodes. However, when we ran the query, it fetched a lot of data from remote nodes. We expected the local cache hit rate to be 100%, but when the actual hit rate was about 33%, we were puzzled.

Digging into the raw data, we found it was because we used an outdated HDFS InputFormat, meaning that if we requested block 2 for the computation, it would pull a line from block 1 and a line from block 3, even though you didn't need any data from block 1 or 3. So, if block 2 was in the local Tachyon node, then blocks 1 and 3 may be in remote Tachyon nodes—leading to a local cache hit rate of 33% instead of 100%. Once we updated our InputFormat, this problem was resolved.

SIGBUS

Sometimes we would get a SIGBUS error, and the Tachyon process would crash. Not only Tachyon, but Spark had the same problem, too. The Spark community actually has a workaround for this problem that uses fewer memory-mapped files, but that was not the real solution. The root cause of the problem was that we were using Java 6 with the CompressedOOP feature, which compressed 64-bit pointers to 32-bit pointers, to reduce memory usage. However, there was a bug in Java 6 that allowed the compressed pointer to point to an invalid location, leading to a SIGBUS error. We solved this problem by either not using CompressedOOP in Java 6, or simply updating to Java 7.

Time-to-Live Feature and What's Next for Tachyon

In our next stage of development, we plan to expand Tachyon's functionalities and performance. For instance, we recently developed a Time-to-Live (TTL) feature in Tachyon. This feature helps us reduce Tachyon cache space usage automatically.

For the cache layer, we only want to keep data for two weeks (since people rarely query data beyond this point). With the TTL feature, Tachyon is able to keep track of the data and delete it once it expires —leaving enough space for fresh data. Also, there are efforts to use Tachyon as a parameter server for deep learning (Adatao is leading the effort in this direction). As memory becomes cheaper, I expect to see the universal usage of Tachyon in the near future.

CHAPTER 5

The Internet of Things and Real Time

The Internet of Things (IoT) is hot, and as Alistair Croll predicts in "The Internet of Things Has Four Big Data Problems," it's here to stay. The abundance of smart devices on the market is generating new questions about how to handle the real-time event data produced downstream. Apache Kafka has emerged as a leader among stream-processing frameworks, with its high throughput, built-in partitioning, replication, and fault tolerance. Numerous other Apache projects, such as Flume and Cassandra, have cropped up and are being used alongside Kafka to effectively collect and store real-time data. Stream processing and data management continues to be an area of intense activity and interest. In this chapter, we recap some of the most exciting advancements in IoT and real time over the past year.

Ben Lorica reviews a few of the more popular components in stream-processing stacks and combinations that are on the rise for collecting, storing, and analyzing event data. John Piekos explores some of the challenges with lambda architecture, and offers an alternative architecture using a fast in-memory scalable relational database that can simplify and extend the capabilities of lambda. Ben Lorica explores how intelligent data platforms are powering smart cities, specifically in Singapore, and highlights the intersection of communities and technologies that power our future cities. Alistair Croll explains why the IoT needs more practical data, less specialization, and more context.

A Real-Time Processing Revival

by Ben Lorica

You can read this post on oreilly.com here.

Editor's note: Ben Lorica is an advisor to Databricks and Graphistry.

There's renewed interest in stream processing and analytics. I write this based on some data points (attendance in webcasts and conference sessions; a recent meetup), and many conversations with technologists, startup founders, and investors. Certainly, applications are driving this recent resurgence. I've written previously about systems that come from IT operations as well as how the rise of cheap sensors are producing stream mining solutions from wearables (mostly health-related apps) and the IoT (consumer, industrial, and municipal settings). In this post, I'll provide a short update on some of the systems that are being built to handle large amounts of *event data*.

Apache projects (Kafka, Storm, Spark Streaming, Flume) continue to be popular components in stream processing-stacks (I'm not yet hearing much about Samza). Over the past year, many more engineers started deploying Kafka alongside one of the two leading distributed stream-processing frameworks (Storm or Spark Streaming). Among the major Hadoop vendors, Hortonworks has been promoting Storm, Cloudera supports Spark Streaming, and MapR supports both. Kafka is a high-throughput distributed pub/sub system that provides a layer of indirection between "producers" that write to it and "consumers" that take data out of it. A new startup (Confluent) founded by the creators of Kafka should further accelerate the development of this already very popular system. Apache Flume is used to collect, aggregate, and move large amounts of streaming data, and is frequently used with Kafka (*Flafka* or Flume + Kafka). Spark Streaming continues to be one of the more popular components within the Spark ecosystem, and its creators have been adding features at a rapid pace (most recently Kafka integration, a Python API, and zero data loss).

Apache HBase, Apache Cassandra, and Elasticsearch are popular open source options for *storing* event data (the *Team Apache stack* of Cassandra, Kafka, Spark Streaming is an increasingly common combination). Time-series databases built on top of open source NoSQL data stores—OpenTSDB (HBase) and Kairos—continue to have their share of users. The organizers of HBaseCon recently told me

that OpenTSDB remains popular in their community. Advanced technical sessions on the ELK stack (Elasticsearch, Logstash, Kibana) were among the best-attended presentations at the recent Elasticon conference.

Database software vendors have taken note of the growing interest in systems for handling large amounts of real-time event data. The use of memory and SSDs have given Aerospike, memsql, VoltDB, and other vendors interesting products in this space.

At the end of the day, companies are interested in using these software components to build data products or improve their decision making. Various combinations of the components I've just listed appear in the stream-processing platforms of many companies. Big Internet firms such as Netflix, Twitter, and LinkedIn describe their homegrown (streaming) data platforms to packed audiences at conferences such as Strata + Hadoop World.

Designing and deploying data platforms based on distributed, open source software components requires data engineers who know how to *evaluate and administer* many pieces of technology. I have been noticing that small- to medium-sized companies are becoming much more receptive to working with cloud providers: Amazon and Google have components that mirror popular open source projects used for stream processing and analysis, Databricks Cloud is an option that quite a few startups that use Spark are turning to.

As far as focused solutions, I've always felt that some of the best systems will emerge from IT operations and data centers. There are many solutions for collecting, storing, and analyzing event data ("logs") from IT operations. Some companies piece together popular open source components and add proprietary technologies that address various elements of the *streaming pipeline* (move, refine, store, analyze, visualize, reprocess, streaming "joins").

Companies such as Splunk, SumoLogic, ScalingData, and CloudPhysics use some of these open source software components in their IT monitoring products. Some users of Graphite have turned to startups such as Anodot or SignalFx because they need to scale to larger data sizes. Another set of startups such as New Relic provide SaaS for monitoring software app performance ("software analytics").

While IT operations and data centers are the areas I'm most familiar with, note that there are interesting stream processing and analysis software systems that power solutions in other domains (we devoted a whole day to them at Strata + Hadoop World in NYC in 2014). As an example, GE's Predix is used in industrial settings, and I've seen presentations on similar systems for smart cities, agriculture, health care, transportation, and logistics.

One topic that I've been hearing more about lately is the use of graph mining techniques. Companies such as Graphistry, SumoLogic, and Anodot have exploited the fact that log file entries are related to one another, and these relationships can be represented as network graphs. Thus network visualization and analysis tools can be brought to bear on some types of event data ("from time-series to graphs").

Stream *processing and data management* continues to be an area of intense activity and interest. Over the next year, I'll be monitoring progress on the stream *mining and analytics* front. Most of the tools and solutions remain focused on simple (approximate) counting algorithms (such as identifying heavy hitters). Companies such as Numenta are tackling real-time pattern recognition and machine learning. I'd like to see similar efforts built on top of the popular distributed, open source frameworks data engineers have come to embrace. The good news is the leaders of key open source stream processing projects plan to tack on more analytic capabilities.

Improving on the Lambda Architecture for Streaming Analysis

by John Piekos

You can read this post on oreilly.com here.

Modern organizations have started pushing their big data initiatives beyond historical analysis. Fast data creates big data, and applications are being developed that capture value, specifically real-time analytics, the moment fast data arrives. The need for real-time analysis of streaming data for real-time analytics, alerting, customer engagement, or other on-the-spot decision making, is converging on a layered software setup called the Lambda Architecture.

The Lambda Architecture, a collection of both big and fast data software components, is a software paradigm designed to capture value, specifically analytics, from not only historical data, but also from data that is streaming into the system.

In this article, I'll explain the challenges that this architecture currently presents and explore some of the weaknesses. I'll also discuss an alternative architecture using an in-memory database that can simplify and extend the capabilities of Lambda. Some of the enhancements to the Lambda Architecture that will be discussed are:

- The ability to return real-time, low-latency responses back to the originating system for immediate actions, such as customer-tailored responses. Data doesn't have to only flow one way, into the system.
- The addition of a transactional, consistent (ACID) data store. Data entering the system can be transactional and operated upon in a consistent manner.
- The addition of SQL support to the speed layer, providing support for ad hoc analytics as well as support for standard SQL report tooling.

What Is Lambda?

The Lambda Architecture is an emerging big data architecture designed to ingest, process, and compute analytics on both fresh (real-time) and historical (batch) data together. In his book *Big Data—Principles and Best Practices of Scalable Realtime Data Systems*, Nathan Marz introduces the Lambda Architecture and states that:

> The Lambda Architecture…provides a general-purpose approach to implementing an arbitrary function on an arbitrary data set and having the function return its results with low latency.

Marz further defines three key layers in the Lambda Architecture:

Batch layer
: This is the historical archive used to hold all of the data ever collected. This is usually a "data lake" system, such as Hadoop, though it could also be an online analytical processing (OLAP) data warehouse like Vertica or Netezza. The batch layer supports batch queries, which compute historical predefined and *ad hoc* analytics.

Speed layer
: The speed layer supports computing real-time analytics on fast-moving data as it enters the system. This layer is a combination of queuing, streaming, and operational data stores.

  Like the batch layer, the speed layer computes analytics—except that the speed layer runs computations in real time, on fresh data, as the data enters the system. Its purpose is to compute these analytics quickly, at low latency. The analytics the batch layer calculates, for example, are performed over a larger, slightly older data set and take significantly longer to compute. If you relied solely on the batch layer to ingest and compute analytics, the speed at which the batch layer computes the results would mean that the results would likely be minutes to an hour old. It is the speed layer's responsibility to calculate real-time analytics based on fast-moving data, such as data that is zero to one hour old. Thus, when you query the system, you can get a complete view of analytics across the most recent data and all historical data.

Serving layer
: This layer caches results from batch-layer computations so they are immediately available to answer queries. Computing batch layer queries can take time. Periodically, these analytics are re-computed and the cached results are refreshed in the serving layer.

To summarize, Lambda defines a big data architecture that allows predefined and arbitrary queries and computations on both fast-moving data and historical data.

Common Lambda Applications

New applications for the Lambda Architecture are emerging seemingly weekly. Some of the more common use cases of Lambda-based applications revolve around log ingestion and analytics on those log messages. "Logs" in this context could be general server log messages, website clickstream logging, VPN access logs, or the popular practice of collecting analytics on Twitter streams.

The architecture improves on present-day architectures by being able to capture analytics on fast-moving data as it enters the system. This data, which is immutable, is ingested by both Lambda's speed layer and batch layer, usually in parallel, by way of message queues

and streaming systems, such as Kafka and Storm. The ingestion of each log message does not require a response to the entity that delivered the data—it is a one-way data pipeline.

A log message's final resting place is the data lake, where batch metrics are (re)computed. The speed layer computes similar results for the most recent "window," staying ahead of the Hadoop/batch layer. Thus, the Lambda Architecture allows applications to take recent data into account but supports the same basic applications as batch analytics—not real-time decision making, such as determining which ad or promotion to serve to a visitor.

Analytics at the speed and batch layer can be predefined or *ad hoc*. Should new analytics be desired in the Lambda Architecture, the application could rerun the entire data set, from the data lake or from the original log files, to recompute the new metrics. For example, analytics for website click logs could count page hits and page popularity. For Tweet streams, it could compute trending topics.

Limitations of the Lambda Architecture

Although it represents an advance in data analysis and exploits many modern tools well, Lambda falls short in a few ways:

One-way data flow
: In Lambda, immutable data flows in one direction: into the system. The architecture's main goal is to execute OLAP-type processing faster—in essence, reducing the time required to consult column-stored data from a couple of seconds to about 100 ms.

  Therefore, the Lambda Architecture doesn't achieve some of the potentially valuable applications of real-time analytics, such as user segmentation and scoring, fraud detection, detecting denial of service attacks, and calculating consumer policies and billing. Lambda doesn't transact and make per-event decisions on the streaming data, nor does it respond immediately to the events coming in.

Eventual consistency
: Although adequate for popular consumer applications such as displaying status messages, the eventual consistency that solves the well-known CAP dilemma is less robust than the transactions offered by relational databases and some NoSQL products. More important, reliance on eventual consistency makes it

impossible to feed data quickly back into the batch layer and alter analytics on the fly.

NoSQL
: Most tools require custom coding to their unique APIs instead of allowing well-understood SQL queries or the use of common tools, such as business intelligence.

Complexity
: The Lambda Architecture is currently composed of many disparate components passing messages from one to the next. This complexity gets in the way of making instant decisions on real-time data. An oft-cited blog posting from last year explains this weakness.

One company attempted to solve a streaming data problem by implementing the Lambda Architecture as follows:

- The speed layer enlisted Kafka for ingestion, Storm for processing, Cassandra for state, and Zookeeper for distributed coordination.
- The batch layer loaded tuples in batches into S3, then processed the data with Cascading and Amazon Elastic MapReduce.
- The serving layer employed a key/value store such as ElephantDB.

Each component required at least three nodes; the speed layer alone needed 12 nodes. For a situation requiring high speed and accuracy, the company implemented a fragile, complex infrastructure.

In-memory databases can be designed to fill the gaps left by the Lambda Architecture. I'll finish this article by looking at a solution involving in-memory databases, using VoltDB as a model.

Simplifying the Lambda Architecture

The Lambda Architecture can be simplified, preserving its key virtues while enabling missing functionality by replacing the complex speed layer and part of the batch layer with a suitable distributed in-memory database.

VoltDB, for instance, is a clustered, in-memory, relational database that supports the fast ingest of data, real-time *ad hoc* analytics, and the rapid export of data to downstream systems such as Hadoop and OLAP offerings. A fast relational database fits squarely and solidly

in the Lambda Architecture's speed layer—provided the database is fast enough. Like popular streaming systems, VoltDB is horizontally scalable, highly available, and fault tolerant, all while sustaining transactional ingestion speeds of hundreds of thousands to millions of events per second. In the standard Lambda Architecture, the inclusion of this single component greatly simplifies the speed layer by replacing both its streaming and operational data store portions.

In this revised architecture, a queuing system such as Kafka feeds both VoltDB and Hadoop, or the database directly, which would then in turn immediately export the event to the data lake.

Applications That Make Use of In-Memory Capabilities

As defined today, the Lambda Architecture is very focused on fast data collection and read-only queries on both fast and historical data. In Lambda, data is immutable. External systems make use of the Lambda-based environment to query the computed analytics. These analytics are then used for alerts (should metric thresholds be crossed), or harvested, for example in the case of Twitter trending topics.

When considering improvements to the Lambda Architecture, what if you could react, per event, to the incoming data stream? In essence, you'd have the ability to take action based on the incoming feed, in addition to performing analytics.

Many developers are building streaming, fast data applications using the clustered, in-memory, relational database approach suggested by VoltDB. These systems ingest events from sources such as log files, the Internet of Things (IoT), user clickstreams, online game play, and financial applications. While some of these applications passively ingest events and provide real-time analytics and alerting on the data streams (in typical Lambda style), many have begun interacting with the stream, adding per-event decision making and transactions in addition to real-time analytics.

Additionally, in these systems, the speed layer's analytics can differ from the batch layer's analytics. Often, the data lake is used to mine intelligence via exploratory queries. This intelligence, when identified, is then fed to the speed layer as input to the per-event decisions. In this revised architecture:

- Data arrives at a high rate and is ingested. It is immediately exported to the batch layer.
- Historical intelligence can be mined from the batch layer and the aggregate "intelligence" can be delivered to the speed layer for per-event real-time decision making (e.g., to determine which ad to display for a segmented/categorized web browser/user).
- Fast data is either passively ingested, or a response can be computed by the new decision-making layer, using both real-time data and historical "mined" intelligence.

A blog posting from VoltDB offers an overview and example of this fully interactive Lambda-like approach. Another VoltDB resource offers code for a working speed layer.

Conclusion

The Lambda Architecture is a powerful big data analytics framework that serves queries from both fast and historical data. However, the architecture emerged from a need to execute OLAP-type processing faster, without considering a new class of applications that require real-time, per-event decision making. In its current form, Lambda is limited: Immutable data flows in one direction, into the system, for analytics harvesting.

Using a fast in-memory scalable relational database in the Lambda Architecture greatly simplifies the speed layer by reducing the number of components needed.

Lambda's shortcoming is the inability to build responsive, event-oriented applications. In addition to simplifying the architecture, an in-memory, scale-out relational database lets organizations execute transactions and per-event decisions on fast data as it arrives. In contrast to the one-way streaming system feeding events into the speed layer, using a fast database as an ingestion engine provides developers with the ability to place applications in front of the event stream. This lets applications capture value the moment the event arrives, rather than capturing value at some point after the event arrives on an aggregate-basis.

This approach improves the Lambda Architecture by:

- Reducing the number of moving pieces—the products and components. Specifically, major components of the speed layer

can be replaced by a single component. Further, the database can be used as a data store for the serving layer.
- Letting an application make per-event decision making and transactional behavior.
- Providing the traditional relational database interaction model, with both *ad hoc* SQL capabilities and Java on fast data. Applications can use familiar standard SQL, providing agility to their query needs without requiring complex programming logic. Applications can also use standard analytics tooling, such as Tableau, MicroStrategy, and Actuate BIRT, on top of fast data.

This post is part of a collaboration between O'Reilly and VoltDB. See our statement of editorial independence.

How Intelligent Data Platforms Are Powering Smart Cities

by Ben Lorica

You can read this post on oreilly.com here.

According to a 2014 UN report, 54% of the world's population resides in urban areas, with further urbanization projected to push that share up to 66% by the year 2050. This projected surge in population has encouraged local and national leaders throughout the world to rally around "smart cities"—a collection of digital and information technology initiatives designed to make urban areas more livable, agile, and sustainable.

Smart cities depend on a collection of enabling technologies that we've been highlighting at Strata + Hadoop World and in our publications: sensors, mobile computing, social media, high-speed communication networks, and intelligent data platforms. Early applications of smart city technologies are seen in transportation and logistics, local government services, utilities, health care, and education. Previous Strata + Hadoop World sessions have outlined the use of machine learning and big data technologies to understand and predict vehicular traffic and congestion patterns, as well the use of wearables in large-scale healthcare data platforms.

As we put together the program for the upcoming Strata + Hadoop World in Singapore, we have been cognizant of the growing interest in our host country's Smart Nation program. And more generally,

we are mindful that large infrastructure investments throughout the Asia-Pacific region have engaged local leaders in smart city initiatives. For readers comfortable with large-scale streaming platforms, many of the key technologies for enabling smart cities will already be familiar.

Data Collection and Transport

In smart cities, Internet of Things and industrial Internet applications, proper instrumentation, and data collection depend on sensors, mobile devices, and high-speed communication networks. Much of the private infrastructure belongs to and is operated by large telecommunication companies, and many of the interesting early applications and platforms are originating from telcos and network equipment providers.

Data Processing, Storage, and Real-Time Reports

As I noted in an earlier article, recent advances in distributed computing and hardware have produced high-throughput engines capable of handling bounded and unbounded data-processing workloads. Examples of this include cloud computing platforms (e.g., AWS, Google, Microsoft) and homegrown data platforms comprised of popular open source components. At the most basic level, these data platforms provide near real-time reports (business intelligence) on massive data streams, as shown in Figure 5-1:

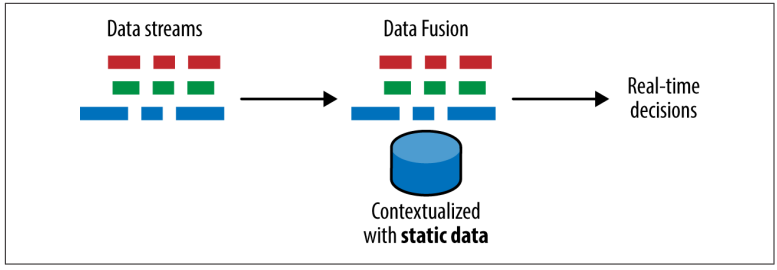


Figure 5-1. Basic function of data streams, leading to real-time decisions (image courtesy of Ben Lorica)

Intelligent Data Applications

Beyond simple counts and anomaly detection, the use of advanced techniques in machine learning and statistics opens up novel real-time applications (machine-to-machine) with no humans in the

loop. Popular examples of such applications include systems that power environments like data centers, buildings and public spaces, and manufacturing (industrial Internet). Recognizing that future smart city applications will rely on disparate data—including event data (metrics from logs and time-series), unstructured data (images, audio, text), and geospatial data sources—we have planned sessions at Strata + Hadoop World Singapore that will cover advanced analytic techniques targeting these data types.

Smart city platforms represent some of the more exciting and impactful applications of real-time, intelligent big data systems. These platforms will confront many of the same challenges faced by applications in the commercial sector, including security, ethics, and governance.

The Internet of Things Has Four Big Data Problems

by Alistair Croll

You can read this post on oreilly.com here.

The Internet of Things (IoT) has a data problem. Well, four data problems. Walking the halls of the 2015 CES in Las Vegas, it's abundantly clear that the IoT is hot. Everyone is claiming to be the world's smartest something. But that sprawl of devices, lacking context, with fragmented user groups, is a huge challenge for the burgeoning industry.

What the IoT needs is data. Big data and the IoT are two sides of the same coin. The IoT collects data from myriad sensors; that data is classified, organized, and used to make automated decisions; and the IoT, in turn, acts on it. It's precisely this ever-accelerating feedback loop that makes the coin as a whole so compelling.

Nowhere are the IoT's data problems more obvious than with that darling of the connected tomorrow known as the wearable. Yet, few people seem to want to discuss these problems.

Problem #1: Nobody Will Wear 50 Devices

If there's one lesson today's IoT startups have learned from their failed science project predecessors, it's that things need to be simple and turnkey. As a result, devices are designed to do one thing really

well. A corollary of this is that there's far too much specialization happening—a device specifically, narrowly designed to measure sleep, or eating speed, or knee health.

Unfortunately, nobody's going to charge, manage, and wear 50 devices, looking like a demented garage-sale cyborg. VentureBeat's Harrison Weber managed to try on 56 different wearables at CES.

With this many competitors, the industry will crash. Wearables today are a digital quilt, a strange patchwork of point solutions trying to blanket a human life. To achieve simplicity, companies have over-focused on a single problem, or a single use case, deluding themselves that their beach-head is actually a sustainable market. The aisles of CES were littered with digital yoga mats, smart sun sensors, epilepsy detectors, and instrumented snowboard bindings.

Problem #2: More Inference, Less Sensing

Consider the aforementioned sun sensor. Do you really need a wristband that senses how much sunlight you've been exposed to? Or can your smartphone instead measure light levels periodically (which it does to determine screen brightness anyway), decide whether you're outside, and check the UV index? The latter is inference, rather than sensing, and it's probably good enough.

When the IoT sprawl finally triggers a mass extinction, only a few companies will survive. Many of the survivors will be the ones that can discover more information by inference, and that means teams that have a data science background.

Early versions of Jawbone's wearable, for example, asked wearers to log their activity manually. More recent versions are smarter: The device notices a period of activity, guesses at what that activity was by comparing it to known patterns—*were you playing basketball for a half hour?*—and uses your response to either reinforce its guess, or to update its collective understanding of what basketball feels like.

Problem #3: Datamandering

This sprawl of devices also means a sprawl of data. Unless you're one of the big wearable players—Jawbone, Fitbit, Withings, and a handful of others—you probably don't have enough user data to make significant breakthrough discoveries about your users' lives. This gives the big players a strong first-mover advantage.

When the wearables sector inevitably consolidates, all the data that failed companies collected will be lost. There's little sharing of information across product lines, and export is seldom more than a comma-separated file.

Consider that one of the strongest reasons people don't switch from Apple to Android is the familiarity of the user experience and the content in iTunes. Similarly, in the IoT world, interfaces and data discourage switching. Unfortunately, this means constant wars over data formats in a strange kind of digital jerrymandering—call it datamandering—as each vendor jockeys for position, trying to be the central hub of our health, parenting, home, or finances.

As Samsung CEO BK Yoon said in his CES keynote, "I've heard people say they want to create a single operating system for the Internet of Things, but these people only work with their own devices."

Walking CES, you see hundreds of manufacturers from Shenzhen promoting the building blocks of the IoT. Technologies like fabric sensors—which only a short time ago were freshly released from secret university labs and lauded on tech blogs—can now be had at scale from China. Barriers to entry crumble fast. What remains for IoT companies are attention, adoption, and data.

When technical advances erode quickly, companies have little reason to cooperate on the data they collect. There's no data lake in wearables, just myriad jealously guarded streams.

Problem #4: Context Is Everything

If data doesn't change your behavior, why bother collecting it? Perhaps the biggest data problem the IoT faces is correlating the data it collects with actions you can take. Consider V1bes, which calls itself a "mind app." It measures stress levels and brain activity. Sociometric Solutions does the same thing by listening to the tone of the user's voice, and can predict my stress levels accurately.

That sounds useful: It'd be great to see how stressed I was at a particular time, or when my brain was most active. But unless I can see the person to whom I was talking, or hear the words I was thinking about, at that time, it's hard to do anything about it. The data tells me I'm stressed; it doesn't tell me who's triggering my chronic depression or who makes my eyes light up.

There might be hope here. If I had a photostream of every day, and with it a voice recorder, I might be able to see who I was with (and whom to avoid). Startups such as Narrative Clip, which constantly logs my life by taking a photo every 30 seconds and using algorithms to decide which of those photos are interesting, might give me a clue about what triggered my stress. And portable recorders like Kapture can record conversations with timestamps; their transcripts, analyzed, could help me understand how I react to certain topics.

Ultimately, it's clear that the Internet of Things is here to stay. We're in the midst of an explosion of ideas, but many of them are stillborn, either too specific or too disconnected from the context of our lives to have true meaning. The Internet of Things and big data are two sides of the same coin, and building one without considering the other is a recipe for doom.

CHAPTER 6
Applications of Big Data

Because the big data space stretches across many industries, organizations, and users, there are a wide wide variety of use cases and applications. Mike Hadley, senior director of emerging products and technology at Autodesk, delivered a keynote at Strata + Hadoop World London that described his company's use of an unsupervised machine learning system that operates against 3D models to discover patterns and identify taxonomies among various hardware products. The goal is to create a "living catalog" of each user's creations that will inform future designs. During her Strata + Hadoop World London keynote, Cait O'Riordan, VP of product, music, and platforms at Shazam, discussed how her company is using big data to predict Billboard music hits. This chapter's collection of blog posts conveys big data's utility and power through current, real-world applications.

First, Gerhard Kress discusses how railways are at the intersection of Internet and industry. Max Neunhöffer then discusses multimodel databases and describes one area where the flexibility of a multimodel database is extremely well suited—the management of large amounts of hierarchical data, such as in an aircraft fleet. Liza Kindred then explains how the fashion industry is embracing algorithms, natural language processing, and visual search. Finally, Timothy McGovern summarizes how oil and gas exploration have long been at the forefront of data collection and analysis.

How Trains Are Becoming Data Driven

by Gerhard Kress

You can read this post on oreilly.com here.

Trains and public transport are, for many of us, a vital part of our daily lives. Large cities are particularly dependent on an efficient public transport system, and if disruption occurs, it usually affects many passengers while spreading across the transport network. But our requirements as passengers are growing and maturing. Safety is paramount, but we also care about timeliness, comfort, Internet access, and other amenities. With strong competition for regional and long-distance trains, providing an attractive service has become critical for many rail operators today.

The railway industry is an old industry. For the past 150 years, this industry was built around mechanical systems maintained throughout a lifetime of 30 years, mostly through reactive or preventive maintenance. But this is not enough anymore to deliver the type of service we all want and expect to experience.

Deriving Insight from the Data of Trains

Over the past few years, the rail industry has been transforming itself, embracing IT, digitalization, big data, and the related changes in business models. This change is driven both by the railway operating companies demanding higher vehicle and infrastructure availability, and, increasingly, wanting to transition their operational risk to suppliers. In parallel, the thought leaders among maintenance providers have embraced the technology opportunities to radically improve their offerings and help their customers deliver better value.

At the core of all these changes is the ability to derive insights and value from the data of trains, rail infrastructure, and operations. In essence, this means automatically gathering and transmitting data from rail vehicles and rail infrastructure, providing the rail operator with an up-to-date view of the fleet, and using data to improve maintenance processes and predict upcoming failures. When data is used to its full extent, the availability of rail assets can be substantially improved, while the costs of maintenance are reduced significantly. This can allow rail operators to create new offerings for customers.

A good example is Spain's high-speed train from Madrid to Barcelona. The rail operating company, Renfe, is successfully competing with airline flight services on this route. The train service brings passengers from city center to city center in 2.5 hours, compared to a pure flight time of 1 hour 20 minutes. Part of what makes this service so competitive is the reliability of the trains. Renfe actually promises passengers a full refund for a delay of more than 15 minutes. This performance is ensured by a highly professional service organization between Siemens and Renfe (in a joint venture called Nertus), which uses sophisticated data analytics to detect upcoming failures and prevent any disruptions to the scheduled service (full disclosure: I'm the director of mobility data services at Siemens).

Requirements of Industrial Data

In my role on the mobility data services team, I focus on creating the elements of a viable data-enabled business. Since the summer of 2014, we have built a dedicated team for data-driven services, a functioning remote diagnostic platform, and a set of analytical models. The team consists of 10 data scientists, supported by platform architects, software developers, and implementation managers.

The architecture of the remote diagnostic platform is derived from the popular Lambda Architecture, but adapted to the requirements of industrial data that needs to be stored for long periods of time. This platform connects to vehicles and infrastructure in the field, displays the current status to customers, and stores the lifetime data in a data lake built on a Teradata system and a large Hadoop installation. On top of the data lake, there are a variety of analytics workbenches to help our data scientists identify patterns and derive predictive models for operational deployment. The data volumes might not be as large as the click streams from popular websites, for example, but they still require a sophisticated platform for analysis. A typical fleet of regional trains would generate a few terabytes of data and around 100 billion data points per year. And now, imagine that such data needs to be stored and accessed for 10–15 years and you see the challenge.

The target of a large set of the analytical models is the prediction of upcoming component failures. However, such a prediction is not an easy task to perform if you only rely on classical data mining approaches. First of all, rail assets are usually very reliable and do

not fail very often. Preventive maintenance strategies make sure that failures are avoided, and safety for the passengers is secured. This all leads to a skewed distribution in the data set. Furthermore, the data usually contains significantly more possible features than failures, making it a challenge to avoid overfitting. Also, vehicle fleets are often rather small, with anything from 20 to 100 vehicles per fleet. But to create usable prediction models, a high prediction accuracy is required, and especially, a very low ratio of false failure predictions.

We have worked extensively on these topics in my team and were able to create a first set of robust predictive models that are now being introduced to the market. Understanding the data provided by rail vehicles lies fully at the core of these prediction models. Such data can be in the form of error messages, log files, sensor data snapshots, or true time-series sensor data. The semantics of this data needs to be well understood, and the interpretation of the data often depends on the situation of the vehicle at that time. Elements that need to be considered include: whether multiple vehicles were coupled together, if the vehicle was loaded, the topography of the track, and which direction the vehicle was heading. All of these aspects may have visible influence on the interpretation of the data and help to separate the signal from the noise.

Many of the models we have developed also take into account physical processes in the assets we are examining. When it comes to predicting the failure of an air conditioning system, for example, some type of failure mode analysis is required, together with an analysis of how the physical system should behave (i.e., in terms of air pressure, air flow, temperatures, etc.).

How Machine Learning Fits In

All of the approaches mentioned here rely heavily on deep interactions with rail engineering departments. Engineering models are often assessed in order to better separate normal from noteworthy behavior of the system under observation. These insights are used to define the structure of the machine learning system that is trying to predict an upcoming failure. This can be done by supplementing the defined features that describe the system or by providing additional boundary conditions for a support vector machine, for example. Very often, all of this cannot be mapped into a single model, but results in an ensemble of models working together to predict a failure and to avoid false failure predictions.

Our mobility data services team is still a young one, but we have been able to create and successfully apply some models already. We see strong interest from our customers, who are pushing to get ever more insights from the data coming from their vehicles or infrastructure. The value this data can provide is, of course, not only limited to failure prediction—many of them are now trying to identify how they can use this data to improve their own processes, their operations, or how they can change business models for improving their relations with passengers or freight customers. These developments are only the beginning, and there is so much more to come—using data to improve mobility services is at the forefront of mobility technology.

Multimodel Database Case Study: Aircraft Fleet Maintenance

by Max Neunhöffer

You can read this post on oreilly.com here.

Editor's note: Full disclosure—the author is a developer and software architect at ArangoDB GmbH, which leads the development of the open source multimodel database ArangoDB.

In recent years, the idea of "polyglot persistence" has emerged and become popular—for example, see Martin Fowler's excellent blog post. Fowler's basic idea is that it is beneficial to use a variety of appropriate data models for different parts of the persistence layer of larger software architectures. According to this, one would, for example, use a relational database to persist structured, tabular data; a document store for unstructured, object-like data; a key/value store for a hash table; and a graph database for highly linked referential data. Traditionally, this means that one has to use multiple databases in the same project (as shown in Figure 6-1), which leads to some operational friction (more complicated deployment, more frequent upgrades) as well as data consistency and duplication issues.

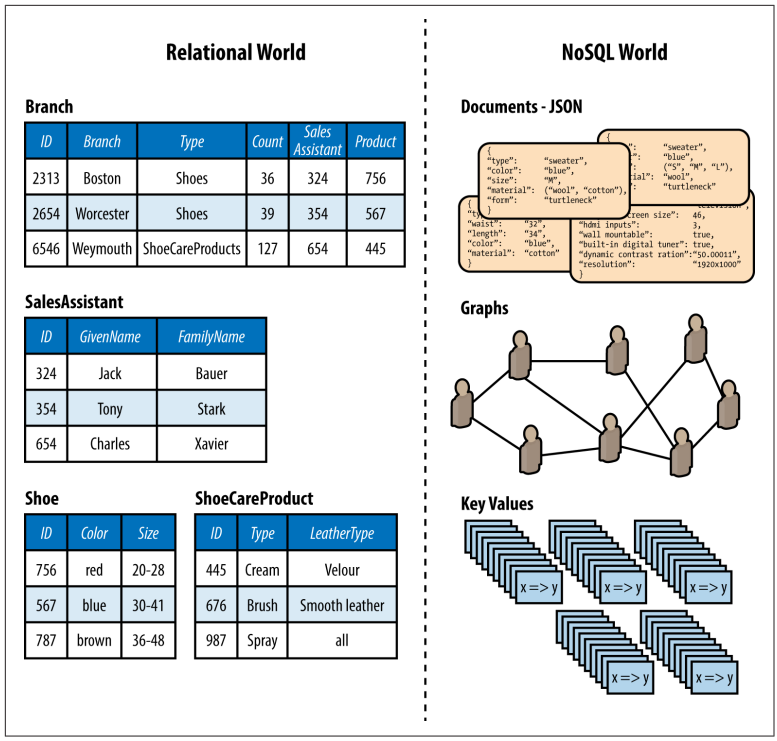


Figure 6-1. Tables, documents, graphs, and key/value pairs: different data models (image courtesy of Max Neunhöffer)

This is the calamity that a multimodel database addresses. You can solve this problem by using a multimodel database that consists of a document store (JSON documents), a key/value store, and a graph database, all in one database engine and with a unifying query language and API that cover all three data models and even allow for mixing them in a single query. Without getting into too much technical detail, these three data models are specially chosen because an architecture like this can successfully compete with more specialized solutions on their own turf, both with respect to query performance and memory usage. The column-oriented data model has, for example, been left out intentionally. Nevertheless, this combination allows you—to a certain extent—to follow the polyglot persistence approach without the need for multiple data stores.

At first glance, the concept of a multimodel database might be a bit hard to swallow, so let me explain this idea briefly. Documents in a document collection usually have a unique primary key that encodes

document identity, which makes a document store naturally into a key/value store, in which the keys are strings and the values are JSON documents. In the absence of secondary indexes, the fact that the values are JSON does not really impose a performance penalty and offers a good amount of flexibility. The graph data model can be implemented by storing a JSON document for each vertex and a JSON document for each edge. The edges are kept in special edge collections that ensure that every edge has "from" and "to" attributes that reference the starting and ending vertices of the edge, respectively. Having unified the data for the three data models in this way, it only remains to devise and implement a common query language that allows users to express document queries, key/value lookups, "graphy queries," and arbitrary mixtures of these. By "graphy queries," I mean queries that involve the particular connectivity features coming from the edges—for example, "ShortestPath," "GraphTraversal," and "Neighbors."

Aircraft Fleet Maintenance: A Case Study

One area where the flexibility of a multimodel database is extremely well suited is the management of large amounts of hierarchical data, such as in an aircraft fleet. Aircraft fleets consists of several aircraft, and a typical aircraft consists of several million parts, which form subcomponents, larger and smaller components, such that we get a whole hierarchy of "items." To organize the maintenance of such a fleet, one has to store a multitude of data at different levels of this hierarchy. There are names of parts or components, serial numbers, manufacturer information, maintenance intervals, maintenance dates, information about subcontractors, links to manuals and documentation, contact persons, warranty and service contract information, to name but a few. Every single piece of data is usually attached to a specific item in this hierarchy.

This data is tracked in order to provide information and answer questions. Questions can include but are not limited to the following examples:

- What are all the parts in a given component?
- Given a (broken) part, what is the smallest component of the aircraft that contains the part and for which there is a maintenance procedure?
- Which parts of this aircraft need maintenance next week?

A data model for an aircraft fleet

So, how do we model the data about our aircraft fleet if we have a multimodel database at our disposal?

There are probably several possibilities, but one good option here is the following (because it allows us to execute all required queries quickly): There is a JSON document for each item in our hierarchy. Due to the flexibility and recursive nature of JSON, we can store nearly arbitrary information about each item, and since the document store is schemaless, it is no problem that the data about an aircraft is completely different from the data about an engine or a small screw. Furthermore, we store containment as a graph structure. That is, the fleet vertex has an edge to every single aircraft vertex, an aircraft vertex has an edge to every top-level component it consists of, component vertices have edges to the subcomponents they are made of, and so on, until a small component has edges to every single individual part it contains. The graph that is formed in this way is in fact a directed tree (see Figure 6-2).

We can either put all items in a single (vertex) collection or sort them into different ones—for example, grouping aircraft, components, and individual parts, respectively. For the graph, this does not matter, but when it comes to defining secondary indexes, multiple collections are probably better. We can ask the database for exactly those secondary indexes we need, such that the particular queries for our application are efficient.

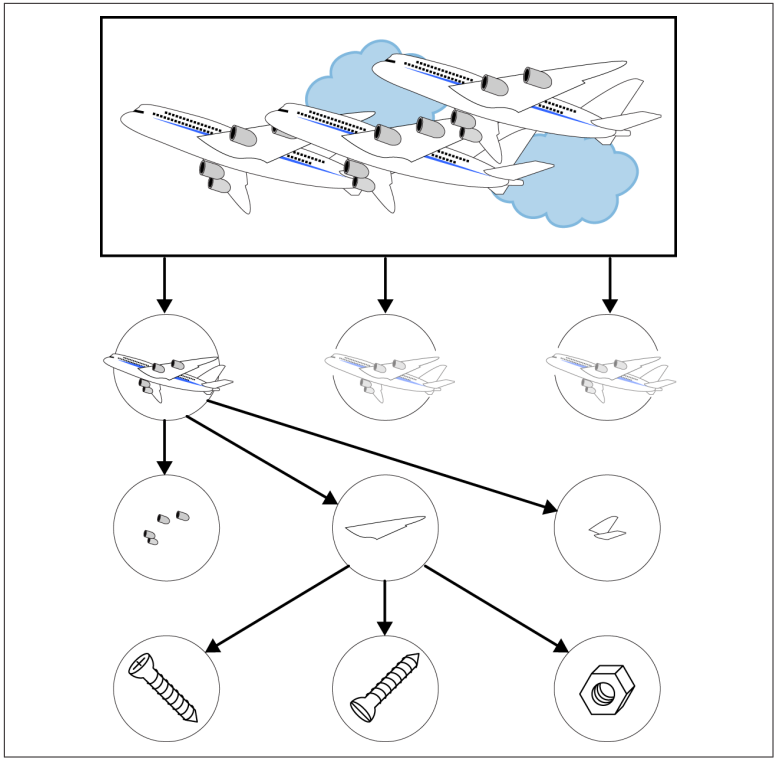


Figure 6-2. A tree of items (image courtesy of Max Neunhöffer)

Queries for Aircraft Fleet Maintenance

We now come back to the typical questions we might ask of the data, and discuss which kinds of queries they might require. We will also look at concrete code examples for these queries using the ArangoDB Query Language (AQL).

What are all the parts in a given component?

This involves starting at a particular vertex in the graph and finding all vertices "below"—that is, all vertices that can be reached by following edges in the forward directions (see Figure 6-3). This is a graph traversal, which is a typical graphy query.

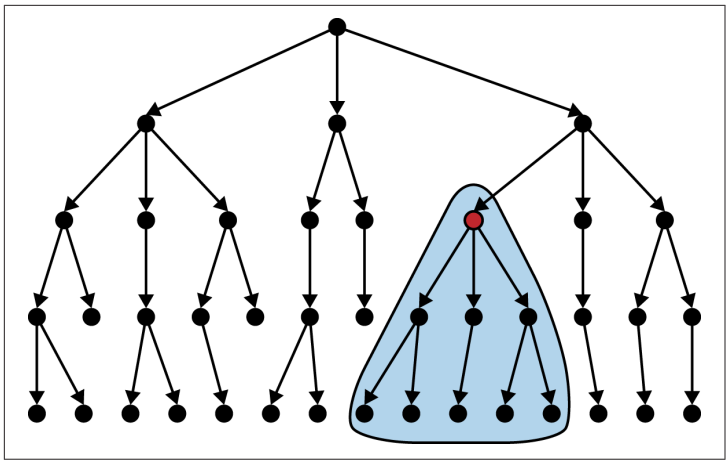


Figure 6-3. Finding all parts in a component (image courtesy of Max Neunhöffer)

Here is an example of this type of query, which finds all vertices that can be reached from `"components/Engine765"` by doing a graph traversal:

```
1 RETURN GRAPH_TRAVERSAL("FleetGraph",
2 "components/Engine765",
3 "outbound")
```

In ArangoDB, one can define graphs by giving them a name and by specifying which document collections contain the vertices and which edge collections contain the edges. Documents, regardless of whether they are vertices or edges, are uniquely identified by their `_id` attribute, which is a string that consists of the collection name, a slash character (/), and then the primary key. The call to `GRAPH_TRAVERSAL` thus only needs the graph name `"FleetGraph"`, the starting vertex, and `"outbound"` for the direction of the edges to be followed. You can specify further options, but that is not relevant here. AQL directly supports this type of graphy query.

Given a (broken) part, what is the smallest component of the aircraft that contains the part and for which there is a maintenance procedure?

This involves starting at a leaf vertex and searching upward in the tree until a component is found for which there is a maintenance procedure, which can be read off the corresponding JSON document. This is again a typical graphy query since the

number of steps to go is not known *a priori*. This particular case is relatively easy, as there is always a unique edge going upward (see Figure 6-4).

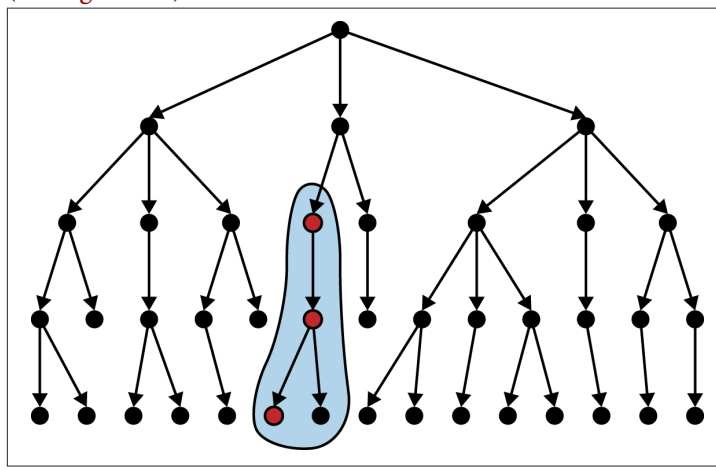


Figure 6-4. Finding the smallest maintainable component (image courtesy of Max Neunhöffer)

For example, the following is an AQL query that finds the shortest path from `"parts/Screw56744"` to a vertex whose `isMaintainable` attribute has the boolean value true, following the edges in the `"inbound"` direction:

```
1 RETURN GRAPH_SHORTEST_PATH("FleetGraph",
2       "parts/Screw56744",
3       {isMaintainable: true},
4       {direction: "inbound",
5       stopAtFirstMatch: true})
```

Note that here, we specify the graph name, the `_id` of the start vertex, and a *pattern* for the target vertex. We could have given a concrete `_id` instead, or could have given further options in addition to the direction of travel in the last argument. We see again that AQL directly supports this type of graphy query.

Which parts of this aircraft need maintenance next week?

This is a query that does not involve the graph structure at all: rather, the result tends to be nearly orthogonal to the graph structure (see Figure 6-5). Nevertheless, the document data model with the right secondary index is a perfect fit for this query.

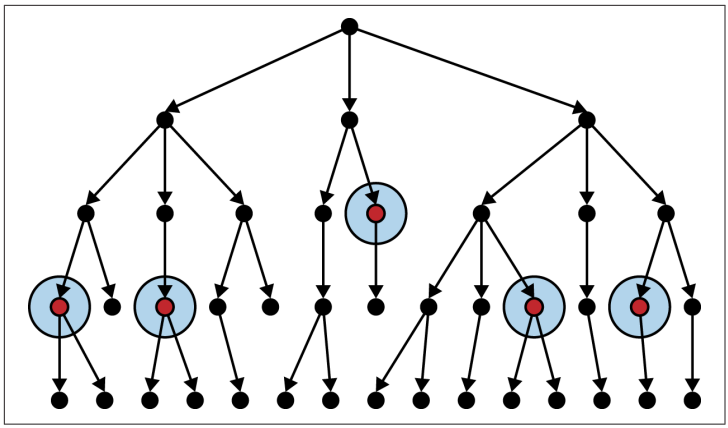


Figure 6-5. Query whose result is orthogonal to the graph structure (image courtesy of Max Neunhöffer)

With a pure graph database, we would be in trouble rather quickly for such a query. That is because we cannot use the graph structure in any sensible way, so we have to rely on secondary indexes—here, for example, on the attribute storing the date of the next maintenance. Obviously, a graph database could implement secondary indexes on its vertex data, but then it would essentially become a multimodel database.

To get our answer, we turn to a document query, which does not consider the graph structure. Here is one that finds the components that are due for maintenance:

```
1 FOR c IN components
2  FILTER c.nextMaintenance <= "2015-05-15"
3  RETURN {id: c._id,
4     nextMaintenance: c.nextMaintenance}
```

What looks like a loop is AQL's way of describing an iteration over the `components` collection. The query optimizer recognizes the presence of a secondary index for the `nextMaintenance` attribute such that the execution engine does not have to perform a full collection scan to satisfy the `FILTER` condition. Note AQL's way of specifying projections by simply forming a new JSON document in the `RETURN` statement from known data. We see that the very same language supports queries usually found in a document store.

Using multimodel querying

To illustrate the potential of the multimodel approach, I'll finally present an AQL query that mixes the three data models. The following query starts by finding parts with maintenance due, runs the preceding shortest path computation for each of them, and then performs a join operation with the `contacts` collection to add concrete contact information to the result:

```
1 FOR p IN parts
2       FILTER p.nextMaintenance <= "2015-05-15"
3       LET path = GRAPH_SHORTEST_PATH("FleetGraph", p._id,
4       {isMaintainable: true},
5       {direction: "inbound",
6       stopAtFirstMatch: true})
7       LET pathverts = path[0].vertices
8       LET c = DOCUMENT(pathverts[LENGTH(pathverts)-1])
9       FOR person IN contacts
10  FILTER person._key == c.contact
11      RETURN {part: p._id, component: c, contact: person}
```

In AQL, the `DOCUMENT` function call performs a key/value lookup via the provided `_id` attribute; this is done for each vertex found as a target of the shortest path computation. Finally, we can see AQL's formulation for a join. The second `FOR` statement brings the `contacts` collection into play, and the query optimizer recognizes that the `FILTER` statement can be satisfied best by doing a join, which in turn is very efficient because it can use the primary index of the `contacts` collection for a fast hash lookup.

This is a prime example for the potential of the multimodel approach. The query needs all three data models: documents with secondary indexes, graphy queries, and a join powered by fast key/value lookup. Imagine the hoops through which we would have to jump if the three data models would *not* reside in the same database engine, or if it would not be possible to mix them in the same query.

Even more importantly, this case study shows that the three different data models were indeed necessary to achieve good performance for all queries arising from the application. Without a graph database, the queries of a graphy nature with path lengths, which are not *a priori* known, notoriously lead to nasty, inefficient multiple join operations. However, a pure graph database cannot satisfy our needs for the document queries that we got efficiently by using the right secondary indexes. The efficient key/value lookups complement the picture by allowing interesting join operations that give us further

flexibility in the data modeling. For example, in the preceding situation, we did not have to embed the whole contact information with every single path, simply because we could perform the join operation in the last query.

Lessons Learned for Data Modeling

The case study of aircraft fleet maintenance reveals several important points about data modeling and multimodel databases:

JSON is very versatile for unstructured and structured data
: The recursive nature of JSON allows embedding of subdocuments and variable length lists. Additionally, you can even store the rows of a table as JSON documents, and modern datastores are so good at compressing data that there is essentially no memory overhead in comparison to relational databases. For structured data, schema validation can be implemented as needed using an extensible HTTP API.

Graphs are a good data model for relations
: In many real-world cases, a graph is a very natural data model. It captures relations and can hold label information with each edge and with each vertex. JSON documents are a natural fit to store this type of vertex and edge data.

A graph database is particularly good for graphy queries
: The crucial thing here is that the query language must implement routines like "shortest path" and "graph traversal," and the fundamental capability for these is to access the list of all outgoing or incoming edges of a vertex rapidly.

A multimodel database can compete with specialized solutions
: The particular choice of the three data models—documents, key/value, and graph—allows us to combine them in a coherent engine. This combination is no compromise. It can, as a document store, be as efficient as a specialized solution; and it can, as a graph database, be as efficient as a specialized solution (see this blog post for some benchmarks).

A multimodel database allows you to choose different data models with less operational overhead
: Having multiple data models available in a single database engine alleviates some of the challenges of using different data models at the same time, because it means less operational over-

head and less data synchronization, and therefore allows for a huge leap in data modeling flexibility. You suddenly have the option to keep related data together in the same data store, even if it needs different data models. Being able to mix the different data models within a single query increases the options for application design and performance optimizations. And if you choose to split the persistence layer into several different database instances (even if they use the same data model), you still have the benefit of only having to deploy a single technology. Furthermore, a data model lock-in is prevented.

Multimodel has a larger solution space than relational
: Considering all these possibilities for queries, the additional flexibility in data modeling and the benefits of polyglot persistence without the usually ensuing friction, the multimodel approach covers a solution space that is even larger than that of the relational model. This is all-the-more astonishing, as the relational model has dominated the database market as well as the database research for decades.

Additional Use Cases for Multimodel Databases

Here are a few more use cases for which multimodel is well suited or even outright necessary:

- **Workflow management software** often models the dependencies between tasks with a graph; some queries need these dependencies, others ignore them and only look at the remaining data.
- **Knowledge graphs** are enormous data collections; most queries from expert systems use only the edges and graphy queries, but you often need "orthogonal" queries that only consider the vertex data.
- **E-commerce systems** need to store customer and product data (JSON), shopping carts (key/value), orders and sales (JSON or graph), and data for recommendations (graph), and need a multitude of queries featuring all of these data items.
- **Enterprise hierarchies** come naturally as graph data and **rights management** typically needs a mixture of graphy and document queries.
- **Social networks** are the prime example for large, highly connected graphs and typical queries are graphy; nevertheless,

actual applications need additional queries which totally ignore the social relationship and thus need secondary indexes and possibly joins with key lookups.
- **Version management applications** usually work with a directed acyclic graph, but also need graphy queries and others.
- Any application that deals with **complex, user-defined data structures** benefits dramatically from the flexibility of a document store and has often good applications for graph data as well.

The Future of Multimodel Databases

Currently there are only two products that are multimodel in the sense used here, making use of JSON, key/value, and graphs: ArangoDB and OrientDB. A few others are marketed under the term "multimodel" (for a complete overview, see the ranking at DB-engines), which support multiple data models, but none of them has graphs and targets the operational domain.

Other players, like MongoDB or Datastax, who have traditionally concentrated on a single data model, show signs of broadening their scope. MongoDB, which is a pure document store, made its storage engine pluggable with the 3.0 release in March 2015. Datastax, a company that produces a commercial product based on the column-based store Apache Cassandra, has recently acquired Aurelius, the company behind the distributed graph database TitanDB. Apple just acquired FoundationDB, a distributed key/value store with multiple "personalities" for different data models layered on top.

The arrival of the new players, as well as the moves of the more established ones, constitute a rather recent trend toward support for multiple data models. At the same time, more and more NoSQL solutions are appearing that rediscover the traditional virtues of relational databases, such as ACID transactions, joins, and relatively strong consistency guarantees.

These are golden times for data modelers and software architects. Stay tuned—watch the exciting new developments in the database market and enjoy the benefits of an unprecedented amount of choice.

Big Data Is Changing the Face of Fashion

by Liza Kindred

You can read this post on oreilly.com here.

Fashion is an industry that struggles for respect—despite its enormous size globally, it is often viewed as frivolous or unnecessary.

And it's true—fashion can be spectacularly silly and wildly extraneous. But somewhere between the glitzy, million-dollar runway shows and the ever-shifting hemlines, a very big business can be found. One industry profile of the global textiles, apparel, and luxury goods market reported that fashion had total revenues of $3.05 trillion in 2011, and is projected to create $3.75 trillion in revenues in 2016.

Solutions for a Unique Business Problem

The majority of clothing purchases are made not out of necessity, but out of a desire for *self-expression* and *identity*—two remarkably difficult things to quantify and define. Yet, established brands and startups throughout the industry are finding clever ways to use big data to turn fashion into "bits and bytes," as much as threads and buttons.

In the updated O'Reilly report *Fashioning Data: A 2015 Update, Data Innovations from the Fashion Industry*, we explore applications of big data that carry lessons for industries of all types. Topics range from predictive algorithms to visual search—capturing structured data from photographs—to natural language processing, with specific examples from complex lifecycles and new startups; this report reveals how different companies are merging human input with machine learning.

Using Data to Drive Big Sales

Don't like to shop? Don't worry. The report encompasses the essence of how fashion brands and startups are using data to drive big sales—and how their techniques relate to what businesses in other industries can do as well.

As we found in our research for the report, one of the things that fashion has always done well is to have two-way conversations with customers. In the report, we interview Eric Colson, who spent six

years at Netflix before becoming the chief algorithms officer at Stitch Fix, a personalized online shopping and styling service for women. Colson explains a unique model from the fashion industry:

> Most companies—Google, Yahoo!, Netflix—use what they call "inferred attributes:" they guess. We don't guess; we ask.

The Original Big Data Industry

by Timothy McGovern

You can read this post on oreilly.com here.

Petroleum extraction is an industry marked by price volatility and high capital exposure in new ventures. Big data is reducing risk, not just to capital, but to workers and the environment as well, as Dan Cowles explores in the free report *Oil, Gas, and Data*.

At the Global Petroleum Show in Calgary, exhibiting alongside massive drill heads, chemical analysts, and the latest in valves and pipes are companies with a decidedly more virtual product: data. IBM's Aspera, Abacus Datagraphics, Fujitsu, and Oracle's Front Porch Digital are pitching data intake, analysis, and storage services to the oil industry, and industry stalwarts such as Halliburton, Lockheed Martin, and BP have been developing these capacities in-house.

The primary benefits of big data occur at the upstream end of petroleum production: exploration, discovery, and drilling. Better analysis of seismic and other geological data allows for drilling in more productive locations, and continual monitoring of equipment results in more uptime and better safety for both workers and environment. These marginal gains can be enough to keep an entire region competitive: The trio of cheap sensors, fast networks, and distributed computation that we've so often seen in other industries is the difference-maker keeping the North Sea oilfields productive in a sub-$100/barrel market.

Beyond productivity (though intimately tied up with safety), is the role of data in petroleum security. While automation and monitoring can detect leaks, threats, and potential points of failure, it can also introduce a new weak point for malicious actors, such as the perpetrators of the Shamoon virus, or plausibly, nongovernmental organizations like those responsible for Stuxnet.

The petroleum industry may be unique in the scale of investment that needs to be committed to a new venture ($10 million or more for a new well), but it provides a model for using data for more efficiency, safety, and security in a well-established industry.

CHAPTER 7

Security, Ethics, and Governance

Conversations around big data, and particularly, the Internet of Things, often steer quickly in the direction of security, ethics, and data governance. At this year's Strata + Hadoop World London, ProPublica journalist and author Julia Angwin delivered a keynote where she speculated whether privacy is becoming a luxury good. The ethical and responsible handling of personal data has been, and will continue to be, an important topic of discussion in the big data space. There's much to discuss: What kinds of policies are organizations implementing to ensure data privacy and restrict user access? How can organizations *use* data to develop data governance policies? Will the "data for good movement" gain speed and traction? The collection of blog posts in this chapter address these, and other, questions.

Andy Oram first discusses building access policies into data stores and how security by design can work in a Hadoop environment. Ben Lorica then explains how comprehensive metadata collection and analysis can pave the way for many interesting applications. Citing a use case from the healthcare industry, Andy Oram returns to explain how federal authentication and authorization could provide security solutions for the Internet of Things. Gilad Rosner suggests the best of European and American data privacy initiatives—such as the US Privacy Principles for Vehicle Technology & Services and the European Data Protection Directive—can come together for the betterment of all. Finally, Jake Porway details how to go from well-intentioned efforts to lasting impact with five principles for applying

data science for social good—a topic he focused on during his keynote at Strata + Hadoop World New York.

The Security Infusion

by Andy Oram

You can read this post on oreilly.com here.

Hadoop jobs reflect the same security demands as other programming tasks. Corporate and regulatory requirements create complex rules concerning who has access to different fields in data sets; sensitive fields must be protected from internal users as well as external threats, and multiple applications run on the same data and must treat different users with different access rights. The modern world of virtualization and containers adds security at the software level, but tears away the hardware protection formerly offered by network segments, firewalls, and DMZs.

Furthermore, security involves more than saying "yes" or "no" to a user running a Hadoop job. There are rules for archiving or backing up data on the one hand, and expiring or deleting it on the other. Audit logs are a must, both to track down possible breaches and to conform to regulation.

Best practices for managing data in these complex, sensitive environments implement the well-known principle of *security by design*. According to this principle, you can't design a database or application in a totally open manner and then layer security on top if you expect it to be robust. Instead, security must be infused throughout the system and built in from the start. *Defense in depth* is a related principle that urges the use of many layers of security, so that an intruder breaking through one layer may be frustrated by the next.

In this article, I'll describe how security by design can work in a Hadoop environment. I interviewed the staff of PHEMI for the article and will refer to their product PHEMI Central to illustrate many of the concepts. But the principles are general ones with long-standing roots in computer security.

The core of a security-by-design approach is a *policy enforcement engine* that intervenes and checks access rights before any data enters or leaves the data store. The use of such an engine makes it easier for an organization to guarantee consistent and robust restric-

tions on its data, while simplifying application development by taking policy enforcement off the shoulders of the developers.

Combining Metadata into Policies

Security is a cross between two sets of criteria: the traits that make data sensitive and the traits of the people who can have access to it.

Sometimes you can simply label a column as sensitive because it contains private data (an address, a salary, a Social Security number). So, column names in databases, tags in XML, and keys in JSON represent the first level of metadata on which you can filter access. But you might want to take several other criteria into account, particularly when you add data retention and archiving to the mix. Thus, you can add any metadata you can extract during data ingestion, such as filenames, timestamps, and network addresses. Your users may also add other keywords or tags to the system.

Each user, group, or department to which you grant access must be associated with some combination of metadata. For instance, a billing department might get access to a customer's address field and to billing data that's less than one year old.

Storing Policies with the Data

Additional security is provided by storing policies right with the raw data instead of leaving the policies in a separate database that might become detached from the system or out of sync with changing data. It's worth noting, in this regard, that several tools in the Hadoop family—Ranger, Falcon, and Knox—can check data against ACLs and enforce security, but they represent the older model of security as an afterthought. PHEMI Central exemplifies the newer security-by-design approach.

PHEMI Central stores a reference to each policy with the data in an Accumulo index. A policy can be applied to a row, a column, a field in XML or JSON, or even a particular cell. Multiple references to policies can be included without a performance problem, so that different users can have different access rights to the same data item. Performance hits are minimized through Accumulo's caching and through the use of locality groups. These cluster data according to the initial characters of the assigned keys and ensure that data with related keys are put on the same server. An administrator can also set up commonly used filters and aggregations such as min, max,

and average in advance, which gives a performance boost to users who need such filters.

The Policy Enforcement Engine

So far, we have treated data passively and talked only about its structure. Now we can turn to the active element of security: the software that stands between the user's query or job request and the data.

The policy enforcement engine retrieves the policy for each requested column, cell, or other data item and determines whether the user should be granted access. If access is granted, the data is sent to the user application. If access is denied, the effect on the user is just as if no such data existed at all. However, a sophisticated policy enforcement engine can also offer different types or levels of access. Suppose, for instance, that privacy rules prohibit researchers from seeing a client's birthdate, but that it's permissible to mask the birthdate and present the researcher with the year of birth. A policy enforcement engine can do this transformation. In other words, different users get different amounts of information based on access rights.

Note that many organizations duplicate data in order to grant quick access to users. For instance, they may remove data needed by analysts from the Hadoop environment and provide a data mart dedicated to those analysts. This requires extra servers and disk space, and leads to the risk of giving analysts outdated information. It truly undermines some of the reasons organizations moved to Hadoop in the first place.

In contrast, a system like PHEMI Central can provide each user with a view suited to his or her needs, without moving any data. The process is similar to views in relational databases, but more flexible.

Take as an example medical patient data, which is highly regulated, treated with great concern by the patients, and prized by the healthcare industry. A patient and the physicians treating the patient may have access to all data, including personal information, diagnosis, etc. A researcher with whom the data has been shared for research purposes must have access only to specific data items (e.g., blood glucose level) or the outcome of analyses performed on the data. A policy enforcement engine can offer these different views in a secure manner without making copies. Instead, the content is filtered based on access policies in force at the time of query.

Fraud detection is another common use case for filtering. For example, a financial institution has access to personal financial information for individuals. Certain patterns indicate fraud, such as access to a particular account from two countries at the same time. The institution could create a view containing only coarse-grained geographic information—such as state and country, along with date of access—and share that with an application run by a business partner to check for fraud.

Benefits of Centralizing Policy

In organizations without policy engines, each application developer has to build policies into the application. These are easy to get wrong, and take up precious developer time that should be focused on the business needs of the organization.

A policy enforcement engine can enforce flexible and sophisticated rules. For instance, HIPAA's privacy rules guard against the use or disclosure of an individual's identifying health information. These rules provide extensive guidelines on how individual data items must be de-identified for privacy purposes and can come into play, for example, when sharing patient data for research purposes. By capturing them as metadata associated with each data item, rules can be enforced at query time by the policy engine.

Another benefit of this type of system, as mentioned earlier, is that data can be massaged before being presented to the user. Thus, different users or applications see different views, but the underlying data is kept in a single place with no need to copy and store altered versions.

At the same time, the engine can enforce retention policies and automatically track data's provenance when the data enters the system. The engine logs all accesses to meet regulatory requirements and provides an audit trail when things go wrong.

Security by design is strongest when the metadata used for access is built right into the system. Applications, databases, and the policy enforcement engine can work together seamlessly to give users all the data they need while upholding organizational and regulatory requirements.

This post is a collaboration between O'Reilly and PHEMI. See our statement of editorial independence.

We Need Open and Vendor-Neutral Metadata Services

by Ben Lorica

You can read this post on oreilly.com here.

As I spoke with friends leading up to Strata + Hadoop World NYC, one topic continued to come up: metadata. It's a topic that data engineers and data management researchers have long thought about because it has significant effects on the systems they maintain and the services they offer. I've also been having more and more conversations about applications made possible by metadata collection and analysis.

At the recent Strata + Hadoop World, U.C. Berkeley professor and Trifacta co-founder Joe Hellerstein outlined the reasons why the broader data industry should rally to develop open and vendor-neutral metadata services. He made the case that improvements in metadata collection and sharing can lead to interesting applications and capabilities within the industry.

The following sections outline some of the reasons why Hellerstein believes the data industry should start focusing more on metadata.

Improved Data Analysis: Metadata on Use

> You will never know your data better than when you are wrangling and analyzing it.
>
> —Joe Hellerstein

A few years ago, I observed that context-switching—due to using multiple frameworks—created a lag in productivity. Today's tools have improved to the point that someone using a single framework like Apache Spark can get many of their data tasks done without having to employ other programming environments. But outside of tracking in detail the actions and choices analysts make, as well as the rationales behind them, today's tools still do a poor job of capturing how people interact and work with data.

Enhanced Interoperability: Standards on Use

If you've read the recent O'Reilly report "Mapping Big Data" or played with the accompanying demo, then you've seen the breadth of tools and platforms that data professionals have to contend with.

Re-creating a complex data pipeline means knowing the details (e.g., version, configuration parameters) of each component involved in a project. With a view to reproducibility, metadata in a persistent (stored) protocol that cuts across vendors and frameworks would come in handy.

Comprehensive Interpretation of Results

Behind every report and model (whether physical or quantitative) are assumptions, code, and parameters. The types of models used in a project determine what data will be gathered, and conversely, models depend heavily on the data that is used to build them. So, proper interpretation of results needs to be accompanied by metadata that focuses on factors that inform data collection and model building.

Reproducibility

As I noted earlier, the settings (version, configuration parameters) of each tool involved in a project are essential to the reproducibility of complex data pipelines. This usually means only documenting projects that yield a desired outcome. Using scientific research as an example, Hellerstein noted that having a comprehensive picture is often just as important. This entails gathering metadata for settings and actions in projects that succeeded as well as projects that failed.

Data Governance Policies by the People, for the People

Governance usually refers to policies that govern important items including the access, availability, and security of data. Rather than adhering to policies that are dictated from above, metadata can be used to develop a governance policy that is based on consensus and collective intelligence. A "sandbox" where users can explore and annotate data could be used to develop a governance policy that is "fueled by observing, learning, and iterating."

Time Travel and Simulations

Comprehensive metadata services lead to capabilities that many organizations aspire to have: The ability to quickly reproduce data pipelines opens the door to "what-if" scenarios. If the right metadata is collected and stored, then models and simulations can fill in any gaps where data was not captured, perform realistic re-

creations, and even conduct "alternate" histories (re-creations that use different settings).

What the IoT Can Learn from the Healthcare Industry

by Andy Oram (with Adrian Gropper)

You can read this post on oreilly.com here.

After a short period of excitement and rosy prospects in the movement we've come to call the Internet of Things (IoT), designers are coming to realize that it will survive or implode around the twin issues of security and user control: A few electrical failures could scare people away for decades, while a nagging sense that someone is exploiting our data without our consent could sour our enthusiasm. Early indicators already point to a heightened level of scrutiny —Senator Ed Markey's office, for example, recently put the automobile industry under the microscope for computer and network security.

In this context, what can the IoT draw from well-established technologies in federated trust? Federated trust in technologies as diverse as the Kerberos and SAML has allowed large groups of users to collaborate securely, never having to share passwords with people they don't trust. OpenID was probably the first truly mass-market application of federated trust.

OpenID and OAuth, which have proven their value on the Web, have an equally vital role in the exchange of data in health care. This task—often cast as the interoperability of electronic health records—can reasonably be described as the primary challenge facing the healthcare industry today, at least in the IT space. Reformers across the healthcare industry (and even Congress) have pressured the federal government to make data exchange the top priority, and the Office of the National Coordinator for Health Information Technology has declared it the centerpiece of upcoming regulations.

Furthermore, other industries can learn from health care. The Internet of Things deals not only with distributed data, but with distributed responsibility for maintaining the quality of that data and authorizing the sharing of data. The use case we'll discuss in this article, where an individual allows her medical device data to be

shared with a provider, can show a way forward for many other industries. For instance, it can steer a path toward better security and user control for the auto industry.

Health care, like other vertical industries, does best by exploiting general technologies that cross industries. When it depends on localized solutions designed for a single industry, the results usually cost a lot more, lock the users into proprietary vendors, and suffer from lower quality. In pursuit of a standard solution, a working group of the OpenID Foundation called Health Relationship Trust (HEART) is putting together a set of technologies that would:

- Keep patient control over data and allow her to determine precisely which providers have access.
- Cut out middlemen, such as expensive health information exchanges that have trouble identifying patients and keeping information up to date.
- Avoid the need for a patient and provider to share secrets. Each maintains their credentials with their own trusted service, and connect with each other without having to reveal passwords.
- Allow data transfers directly (or through a patient-controlled proxy app) from fitness or medical devices to the provider's electronic record, as specified by the patient.

Standard technologies used by HEART include the OpenID OAuth and OpenID Connect standards, and the Kantara Initiative's User-Managed Access (UMA) open standard.

A sophisticated use case developed by the HEART team describes two healthcare providers that are geographically remote from each other and do not know each other. The patient gets her routine care from one but needs treatment from the other during a trip. OAuth and OpenID Connect work here the way they do on countless popular websites: They extend the trust that a user invested in one site to cover another site with which the user wants to do business. The user has a password or credential with just a single trusted site; dedicated tokens (sometimes temporary) grant limited access to other sites.

Devices can also support OAuth and related technologies. The HEART use case suggests two hypothetical devices: one a consumer product and the other a more expensive, dedicated medical device. These become key links between the patient and her physicians. The

patient can authorize the device to send her vital signs independently to the physician of her choice.

OpenID Connect can relieve the patient of the need to enter a password every time she wants access to her records. For instance, the patient might want to use her cell phone to verify her identity. This is sometimes called *multisig technology* and is designed to avoid a catastrophic loss of control over data and avoid a single point of failure.

One could think of identity federation via OpenID Connect as promoting cybersecurity.

UMA extends the possibilities for secure data sharing. It can allow a single authorization server to control access to data on many resource servers. UMA can also enforce any policy set up by the authorization server on behalf of the patient. If the patient wants to release surgical records without releasing mental health records, or wants records released only during business hours as a security measure, UMA enables the authorization server to design arbitrarily defined rules to support such practices. One could think of identity federation via OpenID Connect as promoting cybersecurity by replacing many weak passwords with one strong credential. On top of that, UMA promotes privacy by replacing many consent portals with one patient-selected authorization agent.

For instance, the patient can tell her devices to release data in the future without requiring another request to the patient, and can specify what data is available to each provider, and even when it's available—if the patient is traveling, for example, and needs to see a doctor, she can tell the authentication server to shut off access to her data by that doctor on the day after she takes her flight back home. The patient could also require that anyone viewing her data submit credentials that demonstrate they have a certain medical degree.

Thus, low-cost services already in widespread use can cut the Gordian knot of information siloing in health care. There's no duplication of data, either—the patient maintains it in her records, and the provider has access to the data released to them by the patient. Gropper, who initiated work on the HEART use case cited earlier, calls this "an HIE of One." Federated authentication and authorization, with provision for direct user control over data sharing, provides the best security we currently know without the need to compromise private keys or share secrets, such as passwords.

There Is Room for Global Thinking in IoT Data Privacy Matters

by Gilad Rosner

You can read this post on oreilly.com here.

As devices become more intelligent and networked, the makers and vendors of those devices gain access to greater amounts of personal data. In the extreme case of the washing machine, the kind of data—for example, who uses cold versus warm water—is of little importance. But when the device collects biophysical information, location data, movement patterns, and other sensitive information, data collectors have both greater risk and responsibility in safeguarding it. The advantages of every company becoming a software company—enhanced customer analytics, streamlined processes, improved view of resources and impact—will be accompanied by new privacy challenges.

A key question emerges from the increasing intelligence of and monitoring by devices: Will the commercial practices that evolved in the Web be transferred to the Internet of Things? The amount of control users have over data about them is limited. The ubiquitous end-user license agreement tells people what will and won't happen to their data, but there is little choice. In most situations, you can either consent to have your data used or you can take a hike. We do not get to pick and choose how our data is used, except in some blunt cases where you can opt out of certain activities (which is often a condition forced by regulators). If you don't like how your data will be used, you can simply elect not to use the service. But what of the emerging world of ubiquitous sensors and physical devices? Will such a take-it-or-leave it attitude prevail?

In November 2014, the Alliance of Automobile Manufacturers and the Association of Global Automakers released a set of Privacy Principles for Vehicle Technologies and Services. Modeled largely on the White House's Consumer Privacy Bill of Rights, the automaker's privacy principles are certainly a step in the right direction, calling for transparency, choice, respect for context, data minimization, and accountability. Members of the two organizations that adopt the principles (which are by no means mandatory) commit to obtaining affirmative consent to use or share geolocation, biometrics, or driver behavior information. Such consent is not required, though, for

internal research or product development, nor is consent needed to collect the information in the first place. A cynical view of such an arrangement is that it perpetuates the existing power inequity between data collectors and users. One could reasonably argue that location, biometrics, and driver behavior are not necessary to the basic functioning of a car, so there should be an option to disable most or all of these monitoring functions. The automakers' principles do not include such a provision.

For many years, there have been three core security objectives for information systems: confidentiality, integrity, and availability—sometimes called the CIA triad. *Confidentiality* relates to preventing unauthorized access, *integrity* deals with authenticity and preventing improper modification, and *availability* is concerned with timely and reliable system access. These goals have been enshrined in multiple national and international standards, such as the US Federal Information Processing Standards Publication 199, the Common Criteria, and ISO 27002. More recently, we have seen the emergence of "Privacy by Design" (PbD) movements—quite simply the idea that privacy should be "baked in, not bolted on." And while the confidentiality part of the CIA triad implies privacy, the PbD discourse amplifies and extends privacy goals toward the maximum protection of personal data by default. European data protection experts have been seeking to complement the CIA triad with three additional goals:

- *Transparency* helps people understand who knows what about them—it's about awareness and comprehension. It explains whom data is shared with; how long it is held; how it is audited; and, importantly, defines the privacy risks.
- *Unlinkability* is about the separation of informational contexts, such as work, personal, family, citizen, and social. It's about breaking the links of one's online activity. Simply put, every website doesn't need to know every other website you've visited.
- *Intervenability* is the ability for users to intervene: the right to access, change, correct, block, revoke consent, and delete their personal data. The controversial "right to be forgotten" is a form of intervenability—a belief that people should have some control over the longevity of their data.

The majority of discussions of these goals happen in the field of identity management, but there is clear application within the

domain of connected devices and the Internet of Things. Transparency is specifically cited in the automakers' privacy principles, but the weakness of its consent principle can be seen as a failure to fully embrace intervenability. Unlinkability can be applied generally to the use of electronic services, irrespective of whether the interface is a screen or a device—for example, your Fitbit need not know where you drive. Indeed, the Article 29 Working Party, a European data protection watchdog, recently observed, "Full development of IoT capabilities might put a strain on the current possibilities of anonymous use of services and generally limit the possibility of remaining unnoticed."

The goals of transparency, unlinkability, and intervenability are ways to operationalize Privacy by Design principles and aid in user empowerment. While PbD is part of the forthcoming update to European data protection law, it's unlikely that these three goals will become mandatory or part of a regulatory regime. However, from the perspective of self-regulation, and in service of embedding a privacy ethos in the design of connected devices, makers and manufacturers have an opportunity to be proactive by embracing these goals. Some research points out that people are uncomfortable with the degree of surveillance and data gathering that the IoT portends. The three goals are a set of tools to address such discomfort and get ahead of regulator concerns, a way to lead the conversation on privacy.

Discussions about IoT and personal data are happening at the national level. The FTC just released a report on its inquiry into concerns and best practices for privacy and security in the IoT. The inquiry and its findings are predicated mainly on the Fair Information Practice Principles (FIPPs), the guiding principles that underpin American data protection rules in their various guises. The aforementioned White House Consumer Privacy Bill of Rights and the automakers' privacy principles draw heavily upon the FIPPs, and there is close kinship between them and the existing European Data Protection Directive.

Unlinkability and intervenability, however, are more modern goals that reflect a European sense of privacy protection. The FTC report, while drawing upon the Article 29 Working Party, has an arguably (and unsurprisingly) American flavor, relying on the "fairness" goals of the FIPPs rather than emphasizing an expanded set of privacy goals. There is some discussion of Privacy by Design principles, in

particular the de-identifying of data and the prevention of re-identification, as well as data minimization, which are both cousin to unlinkability.

Certainly, the FTC and the automakers' associations are to be applauded for taking privacy seriously as qualitative and quantitative changes occur in the software and hardware landscapes. Given the IoT's global character, there is room for global thinking on these matters. The best of European and American thought can be brought into the same conversation for the betterment of all. As hardware companies become software companies, they can delve into a broader set of privacy discussions to select design strategies that reflect a range of corporate goals, customer preference, regulatory imperative, and commercial priorities.

Five Principles for Applying Data Science for Social Good

by Jake Porway

You can read this post on oreilly.com here.

"We're making the world a better place." That line echoes from the parody of the Disrupt conference in the opening episode of HBO's *Silicon Valley*. It's a satirical take on our sector's occasional tendency to equate narrow tech solutions like "software-designed data centers for cloud computing" with historical improvements to the human condition.

Whether you take it as parody or not, there is a very real swell in organizations hoping to use "data for good." Every week, a data or technology company declares that it wants to "do good" and there are countless workshops hosted by major foundations musing on what "big data can do for society." Add to that a growing number of data-for-good programs from Data Science for Social Good's fantastic summer program to Bayes Impact's data science fellowships to DrivenData's data-science-for-good competitions, and you can see how quickly this idea of "data for good" is growing.

Yes, it's an exciting time to be exploring the ways new data sets, new techniques, and new scientists could be deployed to "make the world a better place." We've already seen deep learning applied to ocean health, satellite imagery used to estimate poverty levels,

and cellphone data used to elucidate Nairobi's hidden public transportation routes. And yet, for all this excitement about the potential of this "data for good movement," we are still desperately far from creating lasting impact. Many efforts will not only fall short of lasting impact—they will make no change at all.

At DataKind, we've spent the last three years teaming data scientists with social change organizations, to bring the same algorithms that companies use to boost profits to mission-driven organizations in order to boost their impact. It has become clear that using data science in the service of humanity requires much more than free software, free labor, and good intentions.

So how can these well-intentioned efforts reach their full potential for real impact? Embracing the following five principles can drastically accelerate a world in which we truly use data to serve humanity.

"Statistics" Is So Much More Than "Percentages"

We must convey what constitutes data, what it can be used for, and why it's valuable.

There was a packed house for the March 2015 release of the *No Ceilings Full Participation Report*. Hillary Clinton, Melinda Gates, and Chelsea Clinton stood on stage and lauded the report, the culmination of a year-long effort to aggregate and analyze new and existing global data, as the biggest, most comprehensive data collection effort about women and gender ever attempted. One of the most trumpeted parts of the effort was the release of the data in an open and easily accessible way.

I ran home and excitedly pulled up the data from the No Ceilings GitHub, giddy to use it for our DataKind projects. As I downloaded each file, my heart sunk. The 6 MB size of the entire global data set told me what I would find inside before I even opened the first file. Like a familiar ache, the first row of the spreadsheet said it all: "USA, 2009, 84.4%."

What I'd encountered was a common situation when it comes to data in the social sector: the prevalence of inert, aggregate data. Huge tomes of indicators, averages, and percentages fill the landscape of international development data. These data sets are sometimes cutely referred to as "massive passive" data, because they are

large, backward-looking, exceedingly coarse, and nearly impossible to make decisions from, much less actually perform any real statistical analysis upon.

The promise of a data-driven society lies in the sudden availability of more real-time, granular data, accessible as a resource for looking forward, not just a fossil record to look back upon. Mobile phone data, satellite data, even simple social media data or digitized documents can yield mountains of rich, insightful data from which we can build statistical models, create smarter systems, and adjust course to provide the most successful social interventions.

To affect social change, we must spread the idea beyond technologists that data is more than "spreadsheets" or "indicators." We must consider any digital information, of any kind, as a potential data source that could yield new information.

Finding Problems Can Be Harder Than Finding Solutions

We must scale the process of problem discovery through deeper collaboration between the problem holders, the data holders, and the skills holders.

In the immortal words of Henry Ford, "If I'd asked people what they wanted, they would have said a faster horse." Right now, the field of data science is in a similar position. Framing data solutions for organizations that don't realize how much is now possible can be a frustrating search for faster horses. If data cleaning is 80% of the hard work in data science, then problem discovery makes up nearly the remaining 20% when doing data science for good.

The plague here is one of education. Without a clear understanding that it is even possible to predict something from data, how can we expect someone to be able to articulate that need? Moreover, knowing what to optimize for is a crucial first step before even addressing how prediction could help you optimize it. This means that the organizations that can most easily take advantage of the data science fellowship programs and project-based work are those that are already fairly data savvy—they already understand what is possible, but may not have the skill set or resources to do the work on their own. As Nancy Lublin, founder of the very data savvy DoSomething.org and Crisis Text Line, put it so well at Data on Purpose—"data science is not overhead."

But there are many organizations doing tremendous work that still think of data science as overhead or don't think of it at all, yet their expertise is critical to moving the entire field forward. As data scientists, we need to find ways of illustrating the power and potential of data science to address social sector issues, so that organizations and their funders see this untapped powerful resource for what it is. Similarly, social actors need to find ways to expose themselves to this new technology so that they can become familiar with it.

We also need to create more opportunities for good old-fashioned conversation between issue area and data experts. It's in the very human process of rubbing elbows and getting to know one another that our individual expertise and skills can collide, uncovering the data challenges with the potential to create real impact in the world.

Communication Is More Important Than Technology

We must foster environments in which people can speak openly, honestly, and without judgment. We must be constantly curious about one another.

At the conclusion of one of our recent DataKind events, one of our partner nonprofit organizations lined up to hear the results from their volunteer team of data scientists. Everyone was all smiles—the nonprofit leaders had loved the project experience, the data scientists were excited with their results. The presentations began. "We used Amazon RedShift to store the data, which allowed us to quickly build a multinomial regression. The p-value of 0.002 shows ..." Eyes glazed over. The nonprofit leaders furrowed their brows in telegraphed concentration. *The jargon was standing in the way of understanding the true utility of the project's findings*. It was clear that, like so many other well-intentioned efforts, the project was at risk of gathering dust on a shelf if the team of volunteers couldn't help the organization understand what they had learned and how it could be integrated into the organization's ongoing work.

In many of our projects, we've seen telltale signs that people are talking past one another. Social change representatives may be afraid to speak up if they don't understand something, either because they feel intimidated by the volunteers or because they don't feel comfortable asking for things of volunteers who are so generously donating their time. Similarly, we often find volunteers who are excited to try out the most cutting-edge algorithms they can on

these new data sets, either because they've fallen in love with a certain model of Recurrent Neural Nets or because they want a data set to learn them with. This excitement can cloud their efforts and get lost in translation. It may be that a simple bar chart is all that is needed to spur action.

Lastly, some volunteers assume nonprofits have the resources to operate like the for-profit sector. Nonprofits are, more often than not, resource-constrained, understaffed, under appreciated, and trying to tackle the world's problems on a shoestring budget. Moreover, "free" technology and "pro bono" services often require an immense time investment on the nonprofit professionals' part to manage and be responsive to these projects. They may not have a monetary cost, but they are hardly free.

Socially minded data science competitions and fellowship models will continue to thrive, but we must build *empathy*—strong communication through which diverse parties gain a greater understanding of and respect for each other—into those frameworks. Otherwise we'll forever be "hacking" social change problems, creating tools that are "fun," but not "functional."

We Need Diverse Viewpoints

To tackle sector-wide challenges, we need a range of voices involved.

One of the most challenging aspects to making change at the sector level is the range of diverse viewpoints necessary to understand a problem in its entirety. In the business world, profit, revenue, or output can be valid metrics of success. Rarely, if ever, are metrics for social change so cleanly defined.

Moreover, any substantial social, political, or environmental problem quickly expands beyond its bounds. Take, for example, a seemingly innocuous challenge like "providing healthier school lunches." What initially appears to be a straightforward opportunity to improve the nutritional offerings available to schools quickly involves the complex educational budgeting system, which in turn is determined through even more politically fraught processes. As with most major humanitarian challenges, the central issue is like a string in a hairball wound around a nest of other related problems, and no single strand can be removed without tightening the whole mess. Oh, and halfway through you find out that the strings are actually snakes.

Challenging this paradigm requires diverse, or "collective impact," approaches to problem solving. The idea has been around for a while (h/t Chris Diehl), but has not yet been widely implemented due to the challenges in successful collective impact. Moreover, while there are many diverse collectives committed to social change, few have the voice of expert data scientists involved. DataKind is piloting a collective impact model called DataKind Labs, that seeks to bring together diverse problem holders, data holders, and data science experts to co-create solutions that can be applied across an entire sector-wide challenge. We just launched our first project with Microsoft to increase traffic safety and are hopeful that this effort will demonstrate how vital a role data science can play in a collective impact approach.

We Must Design for People

Data is not truth, and tech is not an answer in and of itself. Without designing for the humans on the other end, our work is in vain.

So many of the data projects making headlines—a new app for finding public services, a new probabilistic model for predicting weather patterns for subsistence farmers, a visualization of government spending—are great and interesting accomplishments, but don't seem to have an end user in mind. The current approach appears to be "get the tech geeks to hack on this problem, and we'll have cool new solutions!" I've opined that, though there are many benefits to hackathons, you can't just hack your way to social change.

A big part of that argument centers on the fact that the "data for good" solutions we build must be co-created with the people at the other end. We need to embrace human-centered design, to begin with the questions, not the data. *We have to build with the end in mind*. When we tap into the social issue expertise that already exists in many mission-driven organizations, there is a powerful opportunity to create solutions to make real change. However, we must make sure those solutions are sustainable given resource and data literacy constraints that social sector organizations face.

That means that we must design with people in mind, accounting for their habits, their data literacy level, and, most importantly, for what drives them. At DataKind, we start with the questions before we ever touch the data and strive to use human-centered design to create solutions that we feel confident our partners are going to use

before we even begin. In addition, we build all of our projects off of deep collaboration that takes the organization's needs into account, first and foremost.

These problems are daunting, but not insurmountable. Data science is new, exciting, and largely misunderstood, but we have an opportunity to align our efforts and proceed forward together. If we incorporate these five principles into our efforts, I believe data science will truly play a key role in making the world a better place for all of humanity.

What's Next

Almost three years ago, DataKind launched on the stage of Strata + Hadoop World NYC as Data Without Borders. True to its motto to "work on stuff that matters," O'Reilly has not only been a huge supporter of our work, but arguably one of the main reasons that our organization can carry on its mission today.

That's why we could think of no place more fitting to make our announcement that DataKind and O'Reilly are formally partnering to expand the ways we use data science in the service of humanity. Under this media partnership, we will be regularly contributing our findings to O'Reilly, bringing new and inspirational examples of data science across the social sector to our community, and giving you new opportunities to get involved with the cause, from volunteering on world-changing projects to simply lending your voice. We couldn't be more excited to be sharing this partnership with an organization that so closely embodies our values of community, social change, and ethical uses of technology.

We'll see you on the front lines!