O'REILLY®

AI and Analytics in Production

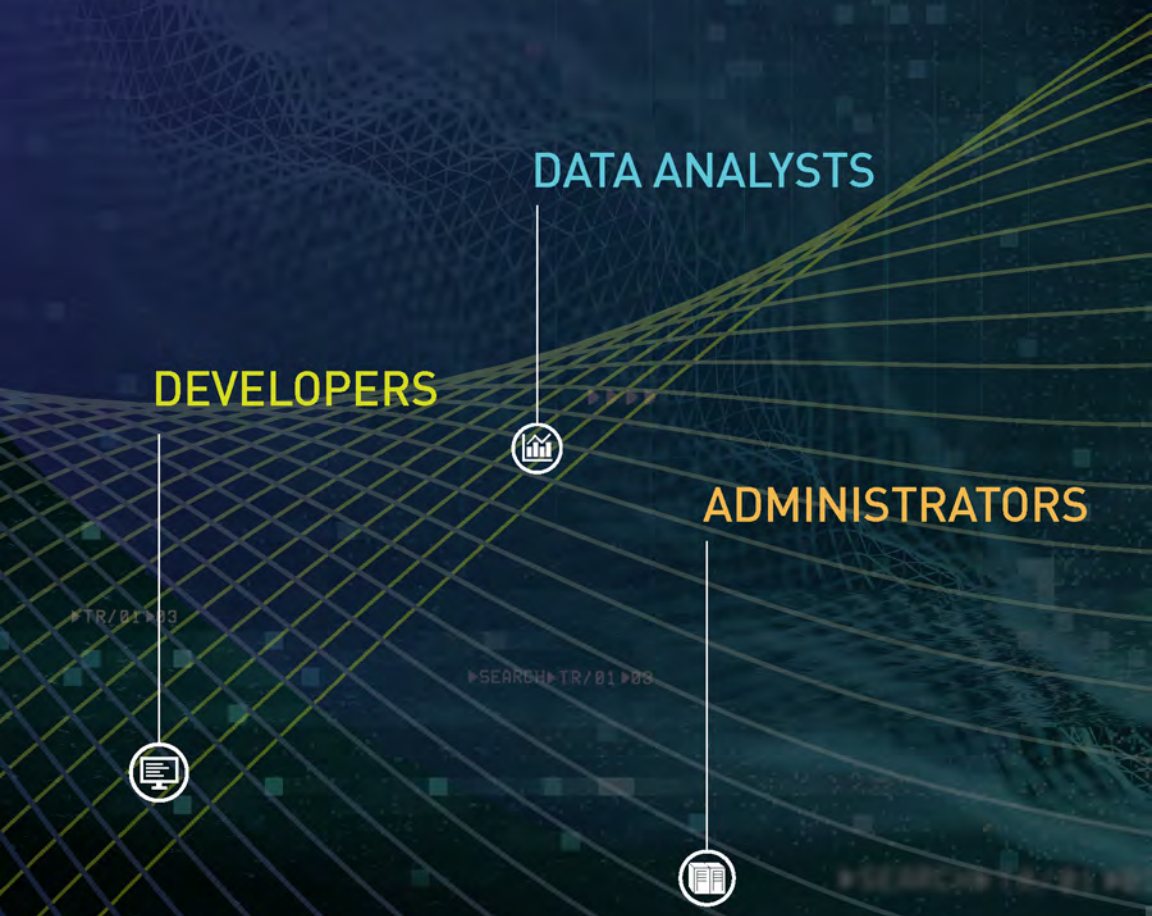
How to Make It Work

Ted Dunning & Ellen Friedman

training

MAPR.

MAPR INSTRUCTOR-LED & ON-DEMAND TRAINING LEADS TO GREAT THINGS



Start today at mapr.com/training

AI and Analytics in Production

How to Make It Work

Ted Dunning and Ellen Friedman

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

AI and Analytics in Production
by Ted Dunning and Ellen Friedman

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://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jonathan Hassell
Editor: Jeff Bleiel
Production Editor: Nicholas Adams
Copyeditor: Octal Publishing, Inc.

Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Ted Dunning

August 2018: First Edition

Revision History for the First Edition
2018-08-10: First Release

This work is part of a collaboration between O'Reilly and MapR. See our *statement of editorial independence*.

978-1-492-04408-6

[LSI]

Table of Contents

Preface

If you are in the process of deploying large-scale data systems into production or if you are using large-scale data in production now, this book is for you. In it we address the difference in big data hype versus serious large-scale projects that bring real value in a wide variety of enterprises. Whether this is your first large-scale data project or you are a seasoned user, you will find helpful content that should reinforce your chances for success.

Here, we speak to business team leaders; CIOs, CDOs, and CTOs; business analysts; machine learning and artificial intelligence (AI) experts; and technical developers to explain in practical terms how to take big data analytics and machine learning/AI into production successfully. We address why this is challenging and offer ways to tackle those challenges. We provide suggestions for best practice, but the book is intended as neither a technical reference nor a comprehensive guide to how to use big data technologies. You can understand it regardless of whether you have a deep technical background. That said, we think that you'll also benefit if you're technically adept, not so much from a review of tools as from fundamental ideas about how to make your work easier and more effective.

The book is based on our experience and observations of real-world use cases so that you can gain from what has made others successful.

How to Use This Book

Use the first two chapters to gain an understanding of the goals and challenges and some of the potential pitfalls of deploying to production (Chapter 1) and for guidance on how to best approach the

design, planning, and execution of large data systems for production (Chapter 2). You will learn how to reduce risk while maintaining innovative approaches and flexibility. We offer a pragmatic approach, taking into account that winning systems must be cost effective and make sense as sustainable, practical, and profitable business solutions.

From there, the book digs into specific examples, based on real-world experience with customers who are successfully using big data in production. Chapter 3 focuses on the special case of machine learning and AI in production, given that this topic is gaining in widespread popularity. Chapter 4 describes an example technology of a data platform with the necessary technical capabilities to support emerging trends for large-scale data in production.

With this foundational knowledge in hand, you'll be set in the last part of the book to explore in Chapter 5 a range of design patterns that are working well for the customers in production we see across various sectors. You can customize these patterns to fit your own needs as you build and adapt production systems. Chapter 6 offers a variety of specific tips for best practice and how to avoid "gotchas" as you move to production.

We hope you find this content makes production easier and more effective in your own business setting.

—Ted Dunning and Ellen Friedman
September 2018

CHAPTER 1
Is It Production-Ready?

The future is already here—it's just not evenly distributed.
—William Gibson

Big data has grown up. Many people are already harvesting huge value from large-scale data via data-intensive applications in production. If you're not yet doing that or not doing it successfully, you're missing out. This book aims to help you design and build production-ready systems that deliver value from large-scale data. We offer practical advice on how to do this based on what we've observed across a wide range of industries.

The first thing to keep in mind is that finding value isn't just about collecting and storing a lot of data, although that is an essential part of it. Value comes from acting on that data, through data-intensive applications that connect to real business goals. And this means that you need to identify practical actions that can be taken in response to the insights revealed by these data-driven applications. A report by itself is not an action; instead, you need a way to connect the results to value-based business goals, whether internal or customer facing. For this to work in production, the entire pipeline—from data ingestion, through processing and analytic applications to action—must be doable in a predictable, dependable, and cost-effective way.

NOTE

Big data isn't just big. It's much more than just an increase in data volume. When used to full advantage, big data offers qualitative changes as well as quantitative. In aggregate, data often has more value than just the sum of the parts. You often can ask—and, if you're lucky, answer—questions that could not have been addressed previously.

Value in big data can be based on building more efficient ways of doing core business processes, or it might be found through new lines of business. Either way, it can involve working not only at new levels of scale in terms of data volume but also at new speeds. The world is changing: data-intensive applications and the business goals they address need to match the new microcycles that modern businesses often require. It's no longer just a matter of generating reports at yearly, quarterly, monthly, weekly, or even daily cycles. Modern businesses move at a new rhythm, often needing to respond to events in seconds or even subseconds. When decisions are needed at very low latency, especially at large scale, they usually require automation. This is a common goal of modern systems: to build applications that automate essential processes.

Another change in modern enterprises has to do with the way applications are designed, developed, and deployed: for your organization to take full advantage of innovative new approaches, you need to work on a foundation and in a style that can allow applications to be developed over a number of iterations.

These are just a few examples of the new issues that modern businesses working with large-scale systems face. We're going to delve into the goals and challenges of big data in production and how you can get the most out of the applications and systems you build, but first, we want to make one thing clear: the possibilities are enormous and well worth pursuing, as depicted in Figure 1-1. Don't fall for doom-and-gloom blogs that claim big data has failed because some early technologies for big data have not performed well in production. If you do, you'll miss out on some great opportunities. The business of getting value from large-scale data is alive and well and growing rapidly. You just have to know how to do it right.

Figure 1-1. Successful production projects harvest the potential value in large-scale data in businesses as diverse as financial services, agritech, and transportation. Data-intensive applications are specific for each industry, but there are many similarities in basic goals and challenges for production.

Production brings its own challenges as compared to work in development or experimentation. These challenges are, for some, seemingly a barrier, but they don't need to be. The first step is to clearly recognize the challenges and pitfalls that you might encounter as you move into production so that you can have a clear and well-considered plan in advance on how to avoid or address them. In this chapter, we talk not only about the goals of production, but also the challenges and offer some hints about how you can recognize a system that is in fact ready for production.

There is no magic formula for success in production, however. Success requires making good choices about how to design a production-capable architecture, how to handle data and build effective applications, and what the right technologies and organizational culture are to fit your particular business. There are several themes that stand out among those organizations that are doing this successfully: what "production" really is, why multitenancy matters, the importance and power of simplicity and the value of flexibility. It's not a detailed or exhaustive list, but we think these ideas make a big difference as you go to production, so we touch on them in this chapter and then dig deeper throughout the rest of the book as to how you can best address them.

Let's begin by taking a look at what production is. We have a bit of a different view than what is traditionally meant by production. This new view can help you be better prepared as you tackle the challenge of production in large-scale, modern systems.

What Does Production Really Mean?

What do we mean by "in production"? The first thing that you might have in mind is to assume that production systems are applications that are customer facing. Although that is often true, it's not the only important characteristic. For one thing, there are internal systems that are mainstream processes and are critical to business success. The fact that business deliverables depend on such systems makes them be in production, as well.

There's a better way to think about what production really means. If a process truly matters to your business, consider it as being in production and plan for it accordingly. We take that a step further: *being in production means making promises you must keep*. These promises are about connecting to real business value and meeting goals in a reasonable time frame. They also have to do with collecting and providing access to the right data, being able to survive a disaster, and more.

NOTE

"In production" means making and keeping value-oriented promises. These promises are made and kept because they are about the stuff that matters to somebody.

The key is to correctly identify what really matters, to document (formalize) the promises you are making to address these issues, and to have a way to monitor whether the promises are met. This somewhat different view of production—the making and keeping of promises for processes essential to your business—helps to ensure that you take into account all aspects of what matters for production to be successful across a complete pipeline rather than focusing on just one step. This view also helps you to future-proof your systems so that you can take advantage of new opportunities in a practical, timely, and cost-effective way. We have more to say about that later, but first, think about the idea that "in production" is about much more than just the deployment of applications.

Data and Production

The idea of what is meant by in production *also should extend to data*. With data-driven business, keep in mind that data is different than code: Data is, importantly, in production sooner. In fact, you might say that data has a longer memory than code. Developers work through multiple iterations of code as an application evolves, but data can have a role in production from the time it is ingested and for decades after, and so it must be treated with the same care as any production system.

There are several scenarios that can cause data to need to be considered in production earlier than traditionally thought, and, of course, that will depend on the particular situation. For instance, it's an unfortunate fact that messing up your data can cause you problems much longer than messing up your code ever could. The problem, of course, comes from the fact that you can fix code and deploy a new version. Problem sorted. But if you mess up archival data, you often can't fix the problem at all. If you build a broken model, version control will give you the code you used, but what about the data? Or, what about when you use an archive of nonproduction data to build that model? Is that nonproduction data suddenly promoted retrospectively? In fact, data often winds up effectively in production long before your code is ready, and it can wind up in production without you even knowing at the time.

Another example is the need for compliance. Increasingly, businesses are being held responsible to be able to document what was known and when it was known for key processes and decisions,

whether manual or automated. With new regulations, the situations that require this sort of promise regarding data are expanding.

Newly ingested, or so-called "raw," data also surprisingly might need to be treated as production-grade even if data at all known subsequent steps in processing and Extract, Transform, and Load (ETL) for a particular application do not need to be. Here's why. Newly developed applications might come to need particular features of the raw data that were discarded by the original application. To be prepared for that possibility, you would need to preserve raw data reliably as a valuable asset for future production systems even though much of the data currently seems useless.

We don't mean to imply that all data at all stages of a workflow be treated as production grade. But one way to recognize whether you're building production-ready systems is to have a proactive approach to planning data integrity across multiple applications and lines of business. This kind of commonality in planning is a strength in preparing for production. The issue of how to deal with when to consider data as "in production" and how to treat it is difficult but important. Another useful approach is to securely archive raw data or partially raw data and treat that storage process as a production process even if downstream use is not (yet) for production. Then, document the boundary. We provide some suggestions in Chapter 6 that should help.

Do You Have the Right Data and Right Question?

The goal of producing real value through analytics often comes down to asking the right question. But which questions you can actually ask may be severely limited by how much data you keep and how you can analyze it. Inherently, you have more degrees of freedom in terms of which questions and what analyses are possible if you retain the original data in a form closer to how events in the real world happened.

Let's take a simplified example as an illustration of this. Assume for the moment that we have sent out three emails and want to determine which is the most effective at getting the response that we want. This example isn't really limited to emails, of course, but could be all kinds of problems involving customer response or even physical effects like how a manufacturing process responds to various changes.

Which email is the best performer? Look at the dashboard showing the number of responses per hour in Figure 1-2. You can see that it makes option C appear to be the best by far. Moreover, if all we have is the number of responses in the most recent hour, this is the only question we can ask and the only answer we can get. But it is really misleading. It is only telling us which email performs best at t_{now} and that's not what we want to know.

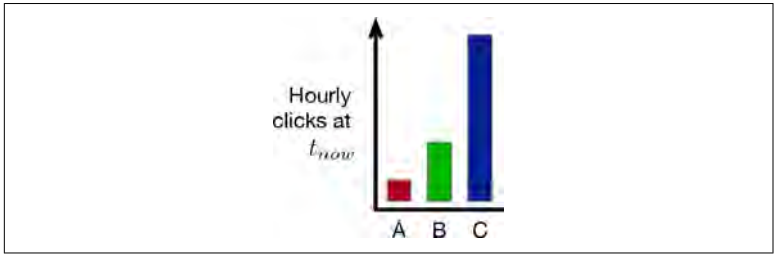


Figure 1-2. A dashboard shows current click rates. In this graph, option C seems to be doing better than either A or B. Such a dashboard can be dangerously misleading because it shows no history.

There is a lot more to the story. Plotting the response rate against time gives us a very different view, as shown in the top graph in Figure 1-3. Now we see that each email was sent at different times, which means that the instantaneous response rate at t_{now} is mostly just a measure of which email was most recently sent. Accumulating total responses instead of instantaneous response rate doesn't fix things, because that just gives a big advantage to the email that was sent first instead of most recently.

In contrast to comparing instantaneous rates as in the upper panel of Figure 1-3, by aligning these response curves according to their launch times we get a much better picture of what is happening, as shown in the lower panel. Doing this requires that we retain a history of click rates as well as record the events corresponding to each email's launch.

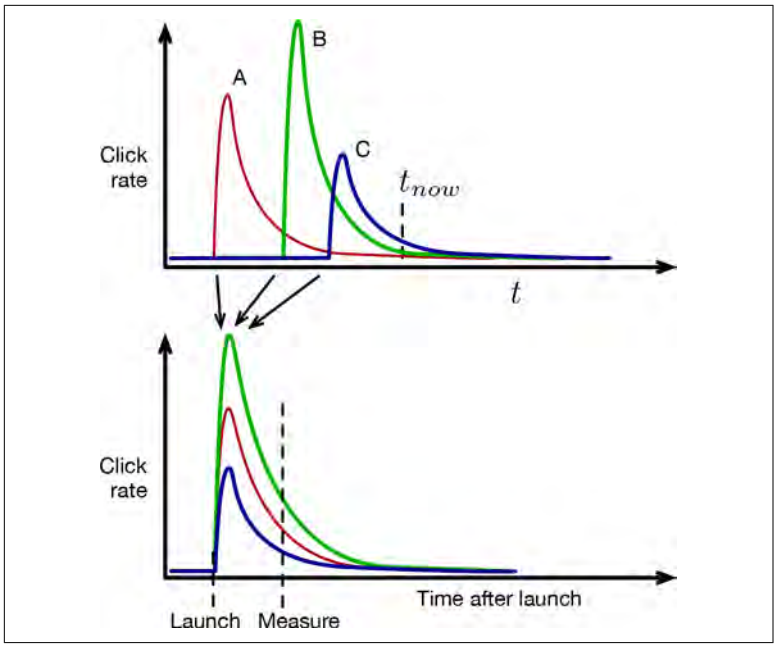


Figure 1-3. Raw click data is graphed in the upper graph. Three email options (A, B and C) were launched at different times, which makes comparing their short-term click rate at t_{now} very misleading. In contrast, the lower graph shows responses aligned at their launch times. Here the response is compared at a fixed time after launch. With this data, it's clear that option B (green) is actually the best performer.

But what if we want to do some kind of analysis that depends on which time zone the recipient was in? Our aggregates are unlikely to make this distinction. At some point, the only viable approach is to record all the information we have about each response to every email as a separate business event. Recording just the count of events that fit into particular preknown categories (like A, B, or C) takes up a lot less space but vastly inhibits what we can understand about what is actually happening.

What technology we use to record these events is not nearly as important as the simple fact that we *do* record them (we have suggestions on how to do this in Chapter 5). Getting this wrong by summarizing event data too soon and too much has led some people to conclude that big data technologies are of no use to them. Often, however, this conclusion is based on using these new technologies to

do the same analysis on the same summarized data as they had always done and getting results that are no different. But the alternative approach of recording masses of detailed events inevitably results in a lot more data. That is, often, big data.

Does Your System Fit Your Business?

Production promises built into business goals define the Service Level Agreements (SLAs) for data-intensive applications. Among the most common criteria to be met are speed, scale, reliability, and sustainability.

The need for speed

There often is time-value to large-scale data. Examples occur across many industries. Real-time traffic and navigation insights are more valuable when the commuter is en route to their destination rather than hearing about a traffic jam that occurred yesterday. Data for market reports, predictive analytics, utilities or telecommunications usage levels, or recommendations for an ecommerce site all have a time-based value. You build low latency data-intensive applications because your business needs to know what's happening in the real world fast enough to be able to respond.

That said, *it's not always true that faster is better.* Just making an application or model run faster might not have any real advantage if the timing of that process is already faster than reasonable requirements. Make the design fit the business goal; otherwise, you're wasting effort and possibly resources. The thing that motivates the need for speed (your SLA) is getting value from data, not bragging rights.

NOTE

Does it fit? Fit your design and technology to the needs particular to specific business goals, anticipating what will be required for production and planning accordingly. This is an overarching lesson, not just about speed. Each situation defines its own requirements. A key to success is to recognize those requirements and address them appropriately.

In other words, don't pick a solution before you understand the problem.

Scale Is More Than Just Data Volume

Much of the value of big data lies in its scale. But scale—in terms of processing and storage of very large data volumes of many terabytes or petabytes—can be challenging for production, especially if your systems and processes have been tested at only modest scale. In addition, do you look beyond your current data volume requirements to be ready to scale up when needed? This change can sometimes need to happen quickly depending on your business model and timeline, and of course it should be doable in a cost-effective way and without unwanted disruption. A key characteristic of organizations that deploy into production successfully is being able to handle large volume and velocity of data for known projects but also being prepared for growth without having to completely rebuild their system.

A different twist on the challenge of scale isn't just about data volume. It can also be about *the number of files you need to handle,* especially if the files are small. This might sound like a simple challenge but it can be a show-stopper. We know of a financial institution that needed to track all incoming and outgoing texts, chats, and emails for compliance reasons. This was a production-grade promise that absolutely had to be kept. In planning for this critical goal, these customers realized that they would need to store and be able to retrieve billions of small files and large files and run a complex set of applications including legacy code. From their previous experience with a Hadoop Distributed File System (HDFS)–based Apache Hadoop system, the company knew that this would likely be very difficult to do using Hadoop and would require complicated workarounds and dozens of name nodes to meet stringent requirements for long-term data safety. They also knew that the size would make conventional storage systems implausibly expensive. They avoided the problem in this particular situation by building and deploying the project on technology designed to handle large numbers of small as well as large files and to have legacy applications directly access the files. (We discuss that technology, a modern big data platform, in Chapter 4). The point is, this financial company was successful in keeping its promises because potential problems were recognized in advance and planned for accordingly. These customers made certain that their SLAs fit their critical business needs and, clearly understanding the problem, found a solution to fit the needs.

Additional issues to consider in production planning are the range of applications that you'll want to run and how you can do this reliably and without resulting in cluster sprawl or a nightmare of administration. We touch on these challenges in the sections on multitenancy and simplicity in this chapter as well as with the solutions introduced in Chapter 2.

Reliability Is a Must

Reliability is important even during development stages of a project in order to make efficient use of developer time and resources, but obviously pressures change as work goes into production. This change is especially true for reliability. One way to think of the difference between a production-ready project and one that is not ready is to compare the behavior of a professional musician to an amateur. The amateur musician practices a song until they can play it through without a mistake. In contrast, the professional musician practices until they cannot play it wrong. It's the same with data and software. Development is the process of getting software to work. Production is the process of setting up a system so that it (almost) never fails.

Issues of reliability for Hadoop-based systems built on HDFS might have left some people thinking that big data systems are not suitable for serious production deployments, especially for mission-critical processes, but this should not be generalized to all big data systems. That's a key point to keep in mind: *big data does not equal Hadoop*. Reliability is not the only issue that separates these systems, but it is an important one. Well-designed big data systems can be relied on with extreme confidence. Here's an example for which reliability and extreme availability are absolutely required.

Aadhaar: reliability brings success to an essential big data system

An example of when it matters to get things right is an impressive project in which data has been used to change society in India. The project is the Aadhaar project run by the Unique Identification Authority of India (UIDAI). The basic idea of the project is to provide a unique, randomly chosen 12-digit government-issued identification number to every resident of India and to provide a biometric data base so that anybody with an Aadhaar number can prove their identity. The biometric data includes an iris scan of both eyes plus the fingerprint of all ten fingers, as suggested by the illus-

tration in Figure 1-4. This record-scale biometric system requires reliability, low latency, and complete availability 24/7 from anywhere in India.

Figure 1-4. UIDAI runs the Aadhaar project whose goal is to provide a unique 12-digit identification number plus biometric data for authentication to every one of the roughly 1.2 billion people in India. (Figure based on image by Christian Als/Panos Pictures.)

Previously in India, most of the population lacked a passport or any other identification documents, and most documents that were available were easily forged. Without adequately verifiable identification, it was difficult or impossible for many citizens to set up a bank account or otherwise participate in a modern economy, and there was also a huge amount of so-called "leakage" of government aid that disappeared to apparent fraud. Aadhaar is helping to change that.

The Aadhaar data base can be used to authenticate identities for every citizen, even in rural villages where a wide range of mobile devices from cell phones to microscanners are used to authenticate identities when a transaction is requested. Aadhaar ID authentication is also used to verify qualification for government aid programs such as food deliveries for the poor or pension payments for the elderly. Implementation of this massive digital identification system has spurred economic growth and saved a huge amount of money by thwarting fraud.

From a technical point of view, what are the requirements for such an impressive big data project? For this project to be successful in production, reliability and availability are a must. Aadhaar must meet strict SLAs for availability of the authentication service every day, at any time, across India. The authentication process, which involves a profile look-up, supports thousands of concurrent transactions with end-to-end response times on the order of 100 milliseconds. The authentication system was originally designed to run on Apache Hadoop and Apache HBase, but the system was neither fast enough nor reliable enough, even with multiple redundant datacenters. Late in 2014, the authentication service was moved to a MapR platform to make use of MapR-DB, a NoSQL data base that supports the HBase API but avoids compaction delays. Since then, there has been no downtime, and India has reaped the benefits of this successful big data project in production.

Predictability and Repeatability

Predictability and repeatability also are key factors for business and for engineering. If you don't have confidence in those qualities, it's not a business; it's a lottery—it's not engineering; it's a lucky accident.

These qualities are especially important in the relationship between test environments and production settings. Whether it's a matter of scale, meeting latency requirements or running in a very specific environment, it's important for test conditions to accurately reflect what will happen in production. You don't want surprises. Just observing that an application worked in a test setting is not in itself sufficient to determine that it is production ready. You must examine the gap between test conditions and what you expect for real-world production settings and, as much as is feasible, have them match, or at least you should understand the implication of their differences. How do you get better predictability and repeatability? In Chapter 2, we explain several approaches that help with this, including running containerized applications and using Kubernetes as an orchestration layer. This is also one of the ways in which data should be considered in production from early stages because it's important to preserve enough data to replay operations. We discuss that further in the design patterns presented in Chapter 5.

Security On-Premises, in Cloud, and Multicloud

Like reliability, data and system security are a must. You should address them from the start, not as an add-on afterthought when you are ready to deploy to production. People who are highly experienced with security know that it is a process of design, data handling, management, and good technology rather than a fancy tool you plug in and forget. Security should extend from on-premises deployments across multiple datacenters and to cloud and multicloud systems, as well.

Depending solely on perimeter security implemented in user-level software is not a viable approach for production systems unless it is part of a layered defense that extends all the way down to the data itself.

Risk Versus Potential: Pressures Change in Production

Pressures change as you move from development into production, partly because the goals are different and partly because the scale or SLAs change. Also, the people who handle requirements might not be the same in production as they were in development.

First, consider the difference in goals. *In development, the goal is to maximize potential.* You will likely explore a range of possibilities for a given business goal, experimenting to see what approach will provide the best performance for the predetermined goal. It's smart to keep in mind the SLAs your application will need to meet in production, but in development your need to meet these promises right away is more relaxed. In development, you can better afford some risk because the impact of a failure is less serious.

The balance between potential and risk changes as you move into production. *In production, the goal is to minimize risk,* or at least to keep it to acceptable levels. The potentially broad goals that you had in development become narrower: you know what needs to be done, and it must be delivered in a predictable, reproducible, cost-effective, and reliable way, without requiring an army for effective administration. Possible sources of risk come from the pressure of scale and from speed; ironically these are two of the same characteristics that are often the basis for value.

Let's look for a moment at the consequences of unreliable systems. As we stated earlier, outages in development systems can have con-

sequences for lost time and lost morale when systems are down, but these pale in comparison to the consequences for unreliability in production because of the more immediate impact on critical business function. Reliability also applies to data safety, and, as we mentioned earlier, data can have in-production pressures much sooner than code does. Because the focus in production shifts to minimizing risk, it's often good to consider the so-called "blast radius" or impact of a failure. The blast radius is generally much more limited for application failures than for an underlying platform failure, so the requirements for stability are higher for the platform.

Furthermore, the potential blast radius is larger for multitenant systems, but this can have a paradoxical effect on overall business risk. It might seem that the simple solution here is to avoid multitenancy to minimize blast radius, but that's not the best answer. If you don't make use of multitenancy, you are missing out on some substantial advantages of a modern big data system. The trick is to pick a very-high-reliability data platform to set up an overall multitenant design but to logically isolate systems at the application level, as we explain later in this chapter.

Should You Separate Development from Production?

With a well-designed system and the right data and analytics platform capabilities, it is possible to run development and production applications on the same cluster, but generally we feel it is better to keep these at least logically separated. That separation can be physical so that development and production run on separate clusters, and data is stored separately, as well, but it does not need to be so. Mainly, the *impact* of development and production applications should be separated. To do that requires that the system you use lets you exert this control with reasonable effort rather than inflicting a large burden for administration. In Chapters 2 and 4, we describe techniques that help with either physical separation or separation of impact.

There are additional implications when production data is stored on a separate cluster from development. As more and more processes depend on real data, it is becoming increasingly difficult to do serious development without access to production data. Again, there is more than one way to deal with this issue, but it is important to recognize in advance whether this is needed in your projects and to thus plan accordingly.

A different issue arises over data that comes from development applications. *Development-grade processes should not produce production data.* To do otherwise would introduce an obligation to live up to promises that you aren't ready to keep. We have already stated that any essential data pipeline should be treated as being in production. This means that you should consider all of the data sources for that pipeline and all of its components as production status. Development-stage processes can still read production data, but any output produced as a result will not be production grade.

Your system should also make it possible for an entire data flow to be versioned and permission controlled, easily and efficiently. Surprisingly, even in a system with strong separation of development and production, you probably still need multitenancy. Here's why.

Why Multitenancy Matters

Multitenancy refers to an assignment of resources such that multiple applications, users, and user groups and multiple datasets all share the same cluster. This approach requires the ability to strictly and securely insulate separate tenants as appropriate while still being able to allow shared access to data when desired. Multitenancy should be one of the core goals of a well-designed large data system because it helps support large-scale analytics and machine learning systems both in development and in production. Multitenancy is valuable in part because it makes these systems more cost effective. Sharing resources among applications, for instance, results in resource optimization, keeping CPUs busy and having fewer underused disks. Well-designed and executed multitenancy offers better optimization for specialized hardware such as Graphics Processing Units (GPUs), as well. You could provide one GPU machine to each of 10 separate data scientists, but that gives each one only limited compute power. In contrast, with multitenancy you can give each data scientist shared access to a larger, more powerful, shared GPU cluster for bursts of heavy computation. This approach uses the same number or less of GPUs yet delivers much more effective resources for data-intensive applications.

There are also long-term reasons that multitenancy is a desirable goal. Properly done, multitenancy can substantially reduce administrative costs by allowing a single platform to be managed independent of how many applications are using it. In addition, multitenancy

makes collaboration more effective while helping to keep overall architectures simple. A well-designed multitenant system is also better positioned to support development and deployment of your second (and third, and so on) big data project by taking advantage of sunk costs. That is, you can do all of this if your platform makes multitenancy safe and practical. Some large platforms don't have robust controls over access or might not properly isolate resource-hungry applications from one another or from delay-sensitive applications. The ability to control data placement is also an important requirement of a data platform suitable for multitenancy.

Multitenancy also serves as a key strategy because many high-value applications are also the ones that pose the highest development risk. Taking advantage of the sunk costs of a platform intended for current production or development by using it for speculative projects allows high-risk/high-reward projects to proceed to a go/no-go decision without large upfront costs. That means experimentation with new ideas is easier because projects can fail fast and cheap. Multitenancy also allows much less data duplication, thus driving down amortized cost, which again allows more experimentation.

Putting lots of applications onto a much smaller single cluster instead of a number of larger clusters can pose an obvious risk, as well. That is, outage in a cluster that supports a large number of applications can be very serious because all of those applications are subject to failure if the platform fails. It will also be possible (no matter the system) for some applications to choke off access to critical resources unless you have suitable operational controls and platform-level controls. This means that you should not simply put lots of applications on a single cluster without considering the increased reliability required of a shared platform. We explain how to deal with this risk in Chapter 2.

If you are thinking it's too risky or too complicated to use a truly multitenant system, look more closely at your design and the capabilities of your underlying platform and other tools: multitenancy is practical to achieve, and it's definitely worth it, but it won't happen by accident. We talk more about how to achieve it in later chapters.

Simplicity Is Golden

Multitenancy is just one aspect of an efficient and reliable production system. You need to be able maintain performance, data locality, manage computation and storage resources, and deploy into a predictable and controlled environment when new applications are launched—and you should be able to do all this without requiring an army of administrators. Otherwise, systems could be too expensive and too complicated to be sustainable in the long run.

We have experience with a very large retail company that maintains a mixed collection of hundreds of critical business processes running on a small number of production clusters with very effective multitenancy. In this case the entire set of clusters is managed by a very small team of administrators. They are able to share resources across multiple applications and can deploy experimental programs even in these essential systems. This retailer found its big data platform was so reliable and easy to manage that it didn't even need a war room for this data platform during the Christmas lockdown months. These systems are returning large amounts of traceable incremental revenues with modest ongoing overhead costs. The savings have gone into developing new applications and exploring new opportunities.

The lesson here is that *simplicity is a strength*. Again, keep in mind that big data does not equal Hadoop. HDFS is a write once/read-only distributed file system that is difficult to access for legacy software or a variety of machine learning tools. These traits can make HDFS be a barrier to streamlined design. Having to copy data out of HDFS systems to do data processing or machine learning on it and then copy it back is an unnecessary complication. That is just one example of how your choices in design and technology make a difference to how easily you can make a system be production ready.

NOTE

Big data systems don't need to be cumbersome. If you find your design has a lot of workarounds, that's a flag to warn you that you might not have the best architecture and technology to support large-scale data-intensive applications in production.

For a system to be sustainable in production it must be cost effective, both in terms of infrastructure and administrative costs. A

well-designed big data system does not take an army of administrators to maintain.

Flexibility: Are You Ready to Adapt?

The best success comes from systems that can expand to larger scale or broaden to include new data and new applications—even make use of new technologies—without having to be rebuilt from scratch. A data and analytics platform should have the capabilities to support a range of data types, storage structures and access APIs, legacy, and new code, in order to give you the flexibility needed for modern data-driven work. Even a system running very well in production today will need to be able to change in a facile manner in future because the world doesn't stay the same—you need a system that delivers reliability now but makes it possible for you to adapt to changing conditions and new opportunities.

If you are working in a system design and with a platform that gives you the flexibility to add new applications or new data sources, you are in an excellent position to be able to capture *targets of opportunity*. These are opportunities to capture value to that can arise on a nearly spur-of-the-moment basis, either through insights that have come from data exploration, through experimentation with new design patterns, or simply because the right two people sat together at lunch and came up with a great idea. With a system that gives you flexibility, you can take advantage of these situations, maybe even leading to a new line of business.

Although flexible systems and architectures are critical, it is equally important that your organization has a flexible culture. You won't get the full advantage of the big data tools and data-intensive approaches you've adopted if you are stuck in a rigid style of managing human resources. We recently asked Terry McCann, a consultant with Adatis, a company that helps people get their big data systems into production, what he thought was one of the most important issues for production. Somewhat surprisingly, McCann said the lack of a DevOps and DataOps approach is one of the biggest issues because that can make the execution of everything else so much more difficult. That observation is in line with a 2017 survey by New Vantage Partners that highlights the major challenge for success in big data projects as the difficulty of organizational and cultural change around big data.

Formula for Success

We have discussed goals important for big data systems if they are to be production ready. You will want to see that there is a clear connection to business value and a clearly defined way to act on the results of data-intensive applications. Data may need to be treated with production care as soon as it is ingested if later it could be required as a system of record or to serve as critical input for a production process, either now or in future. Reliability is an essential requirement for production as well as the ability to handle scale and speed as appropriate for your SLAs in production. Effective systems take advantage of multitenancy but are not cumbersome to maintain. They also should provide a good degree of flexibility, making it easy to adapt to changing conditions and to take advantage of new opportunities.

These are desirable goals, but how do you get there? We said earlier there is no magic formula for getting value from big data in production, and that is true. We should clarify, however: *there is a formula for success—it's just not magic.*

That's what we show you in Chapter 2: what you can do to deploy into production successfully.

CHAPTER 2

Successful Habits for Production

Success in production starts long before an application is deployed to a production setting. The key is having a system with the simplicity and flexibility to support a wide range of applications using a wide variety of data types and sources reliably and affordably at scale. In short, when you're ready for production, the system is ready for you.

No doubt doing all this at scale is challenging, but it's doable, it's worth it and you can get there from where you are. We're not talking about having a separate, isolated big data project but, instead, a centralized design and big data system that is *core to your business*. We also are not promoting the opposite extreme: to entirely do away with traditional solutions such as Relational Database Management Systems (RDBMS) and legacy code and applications. When done correctly, big data systems can work together with traditional solutions. The flexibility and agility possible in modern technologies make it appropriate for the big data system to be the backbone of your enterprise. These qualities of flexibility and agility also make it easier to make the transition to using big data approaches in your mainstream organization.

To plan and build such a system for your business, it can help to see what others have done that led them to success. In our "day jobs" we work with a lot of people who have done this and have overcome the difficulties. Almost all of the customers we deal with at MapR Technologies—more than 90% of them—are in production. This is in fairly stark contrast to the levels we see reported in surveys. For

example, a 2018 Gartner report stated that only 17% of Hadoop-based systems were deployed successfully into production, according to the customers surveyed ("Enabling Essential Data Governance for Successful Big Data Architecture"). This made us curious: Why are some people in production successfully and others are not? In this chapter, we report the basic habits we observe in organizations that are winning in production.

Whatever Happened to Hadoop?

For years many people equated "big data" with Hadoop, but we're talking about systems that go way beyond what Hadoop was intended to do. Apache Hadoop was a great pioneer, popularizing the idea that very large data sets could be stored and analyzed by distributing data across multiple networked machines (a cluster) making very-large-scale computing affordable and feasible for a broader audience. But it always had limitations, especially as a file system, for real time, and for non-Java programs. Legacy software and common tools like R and Python could not easily access distributed data stored on Hadoop Distributed File System (HDFS). It's as though there were a wall between the HDFS cluster and everything else: you have to copy data out elsewhere to be processed and then copy results back. That's just too limited and too cumbersome to be the best choice, especially for production systems and systems that have real-time needs.

Nowadays, Hadoop can be viewed as just one of many workloads on a modern data platform, along with Spark and many others. Hadoop is not what we have in mind as the central technology for building large-scale, efficient systems.

Build a Global Data Fabric

One of the most powerful habits of successful data-driven businesses is to have an effective large data system—we call it a data fabric—that spans their organization. A data fabric is not a product you buy. Instead it is the system you assemble in order to make data from many sources available to a wide range of applications developed and managed by multiple users. The data fabric is what houses and delivers data to your applications. This may exist across a single cluster or it may span multiple data centers, on-premises or in a cloud or multicloud, as suggested by Figure 2-1.

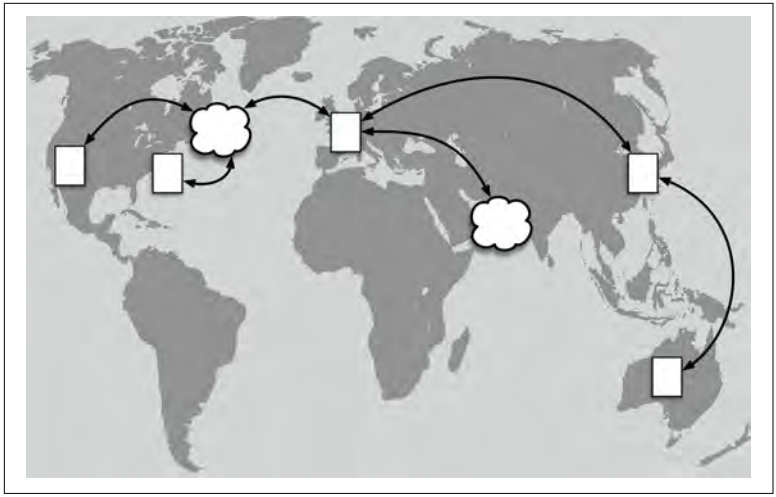


Figure 2-1. Building a unified data fabric across a single cluster or across geo-distributed data centers on-premises or in cloud (or multi-cloud or a hybrid of cloud/on-premises) is a huge advantage in getting production grade performance from applications in a cost-effective way.

As with a piece of cloth, you can identify or isolate any single fiber of many, and yet together they act as one thing. A data fabric works best when it is under one system of administration and the same security mechanisms.

Edge Computing

Increasingly, it is important to be able to handle large volumes of Internet of Things (IoT) data from many sensors being generated at a high rate. In some cases, this data needs to be moved to data centers; in other cases, it's desirable to be able to do some *edge processing* near the data source and then move the results or partially processed data to other data centers. We find customers who need to do this in industries like telecommunications, oil and gas exploration, mining, manufacturing, utilities companies, medical technology, transportation and shipping, online global services, and even smart shops in physical retail locations where real-time interactive engagement with shoppers is the goal.

NOTE

Because data can come from many small data sources such as IoT sensors, a data fabric needs to extend to the edge, to handle raw data and to support edge applications.

Data Fabric Versus Data Lake

We've all heard of the promises of the so-called "data lake": the idea that we would build a comprehensive and centralized collection of data. The term "data hub" was often used when the focus was on self-service analytics. *A data fabric is a new and larger idea, not just a new term for a data lake or data hub.* Here's why that matters.

The power of a data lake was to preserve raw data—structured and unstructured data—and, most important, to see across data silos, but there was a danger of the data lake becoming yet one more huge silo itself. People often assumed that the data lake needed to be built on Hadoop (HDFS), but that automatically limited what you could do, setting up barriers to widespread and facile use, thus increasing the risk of siloing. Due to lack of standard interfaces with HDFS, data must be copied in and out for many processing or machine learning tasks, and users are forced to learn Hadoop skills. A data lake that is difficult to access can fall into disuse or misuse, either way becoming a data swamp. There's also a need to extend the lake beyond one cluster, to move computing to the edge, and to support real-time and interactive tasks, as we've already mentioned.

In contrast, building a data fabric across multiple clusters on-premises or in the cloud and extending it to edge computing gives you the benefits of comprehensive data without forcing everything to a centralized location. A data fabric built with technologies that allow familiar kinds of access (not limited to Hadoop APIs) plus multiple data structures (i.e., files, tables, and streams) encourages use. This new approach is especially attractive for low-latency needs of modern business microcycles. There's another advantage: The use of a modern data fabric helps because part of the formula for success in a comprehensive data design is social rather than technical. You need to get buy-in from users, and a simpler design with familiar access styles makes that much more likely to happen. A data fabric that is easy to maintain helps you focus on tasks that address the business itself.

This is important, too: your data fabric should let you distribute objects such as streams or tables across multiple clusters. Consider this scenario. A developer in one location builds an application that draws on data from a topic in a message stream. It is important for the developer to be able to focus on goals and Service Level Agreements (SLAs) for the application rather than having to deal with the logistics of where the data originated or is located or how to transport it. It's a great advantage if stream replication can let you stream live both locally and in a data center somewhere else *without having to build that in at the application level*, as depicted in Figure 2-2. It's the same experience for the application developer whether the data is local or in another location. We talk about a data platform with this capability in Chapter 4.

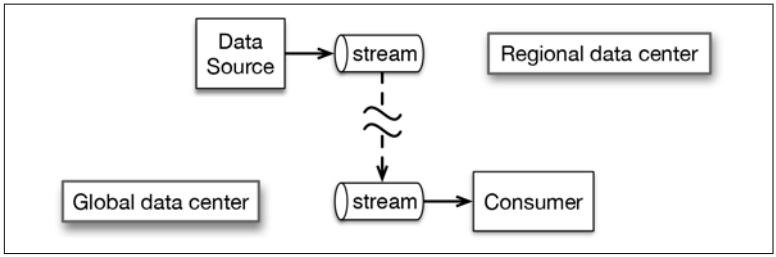


Figure 2-2. With efficient geo-distributed stream replication, the developer can focus on the goals of the application and the insights to be drawn from data rather than having to worry about where data is located. The data source thinks of the stream as local and so does the consumer, even though there may be an ocean between them. A similar advantage could be gained through table replication in a global data fabric.

This *separation of concerns* between developers and data scientists on the one hand versus operations and systems administrators on the other is a huge benefit, making all parties more productive. The head of a data science team at one of our customers found this to be extremely valuable. His business deals with a globally distributed online service, and his team's applications were responsible for collecting metrics and making the data available for business-critical billing. Having taken advantage of stream replication (and thus not having to build these logistics in at the application level), he commented that after just a few months he felt he had "a year of developer time in the bank." That's a good situation to be in!

Understand Why the Data Platform Matters

A modern data platform plays an enormous role in either setting up for success in production or, if it lacks the right capabilities, creating barriers. There are a number of things the data platform in a successful production deployment does, but, surprisingly, one of the most important is for the platform to fade from view. What we mean is that a good platform does what needs to be done without getting in the way. Consider the analogy of a professional tennis player about to hit the ball. The player needs to have all attention focused on the ball and on their shot rather than thinking about the racket: has it been strung properly? Is the tension correct? Can the strings withstand the impact? A top player's racquet is, no doubt, an excellent one, but during a shot, it is like an unconscious extension of the hand and arm. Similarly, if your job is to build and run applications or models or to draw insights, you shouldn't need to spend a lot of effort and time on dealing with platform issues or with building steps into applications that should have been handled by the platform.

Just as there are a number of common capabilities that essentially all services need for managing containers, network names, and such, there are also common capabilities that almost all services need from a data platform.

The idea that all of container orchestration would be implemented once by Kubernetes is becoming a bog standard design. The same argument applies to factoring out data capabilities into a single data platform that serves as a kind of uber-service for all other services. In contrast, having services implement their own data layer causes a large amount of unnecessary and repetitive effort just as does having every service implement its own container management.

What other roles does the data platform need to fill to build a data fabric that supports development and production? One very important one is ease of data access, for multiple types of data. We have pointed out that one thing that turns a data lake into a swamp is inaccessibility. For example, it is a huge advantage to be able to talk about files, tables, or streams by name and to organize them together in directories. This gives you a coherent way to refer to them in applications, between applications, or between users. Simplicity such as this lessens the chance for errors and reduces dupli-

cated effort. In Chapter 4 we describe a data platform with this capability.

Another key requirement of data access is to *not* have to copy everything out of the data platform for processing and then copy results back in. That was one of the big problems with Hadoop-based systems, especially for production. It was as though a big wall stood between the data and whatever you wanted to do with it—you had to keep copying it out over the wall to use it and then throw the results back over the wall. That barrier stood in the way of using legacy applications and familiar tools. It required Hadoop specialized skills just to access data. And Hadoop systems couldn't meet the requirements of real-time or low-latency applications that are so widely needed for the pace of modern business cycles. The ideal production-ready big data platform should be directly accessible via traditional as well as new APIs, be available for development and production applications simultaneously, and support low-latency or real-time applications.

The effective data platform is ubiquitous, letting you use the same platform across data centers on-premises, from IoT edge to center, and to cloud or multi cloud. This ubiquity streamlines your overall organization and architectures, and, as we mentioned in Chapter 1, simplicity such as this reduces risk and generally improves performance. And of course, a good platform is reliable, cost-effective for highly distributed very-large-scale data, supports multitenancy, gives you confidence about security, and provides adequate mechanisms for disaster recovery.

These are the high-level ways in which a data platform supports building a data fabric and helps you get your applications production ready. Keep in mind that as your projects expand or you add new projects or new lines of business, your data platform should support what you are doing without a lot of workarounds or sprawl. Returning to our tennis metaphor, if you find yourself thinking about your racquet too much, that's a red flag: you should rethink your platform and your architecture, because overly complicated is not inevitable in large-scale systems.

Capabilities and Traits Required by the Data Platform

Happily, there is a pretty strong consensus about what a platform should do and how those capabilities should be accessed. So, here are the first three requirements for a modern data platform:

- A data platform must support common data persistence patterns, including files, tables, and streams
- The platform must support standard APIs including:
  - POSIX file access for conventional program access to shared files
  - HDFS API access for Spark, Hadoop, and Hive
  - Document-oriented NoSQL
  - Apache Kafka API for streaming
- Performance must be very high, especially for large-scale distributed programs. Input/output (I/O) rates ranging from hundreds of GB/s up to tens of TB/s are not uncommon. Message rates of millions to hundreds of millions per second are not uncommon.

Note that POSIX file access is effectively the same as allowing conventional databases because Postgres, MariaDB, mySQL, Vertica, Sybase, Oracle, Hana, and many others all run on POSIX files.

It is also common that companies have distributed operations that are distant from one another. Even if operations are not geo-distributed, it is common to have a combination of on-premises compute plus cloud compute or to require cloud vendor neutrality. Some companies have a need for edge computing near the source of data on a factory line, on trucks or ships, or in the field. All of these require that the data platform grow beyond a single cluster at a single location to support a bona fide data fabric.

This leads to the next requirement:

- Put data where you want it but allow computation where you need it. This includes supporting data structures that span multiple clusters with simple consistency models.

If a data platform is to provide common services to multiple teams, avoiding duplication of effort, and improving access to data, there

has to be substantial reuse of the platform. This means that the data platform is going to have to support many applications simultaneously. In large enterprises, that can reach into the hundreds of applications. To do this, it is important to:

- Enforce security at the data level using standard user identity management systems and familiar and consistent permission schemes
- Support multitenancy by automatically isolating load where possible and providing manual management capabilities to allow additional administrative control over data placement and use of high-performance storage
- Operate reliably for years at a time

One fact of life for modern businesses is that they need to explore and (as soon as possible) exploit machine learning and AI to help automate business decisions, particularly high-volume decisions that are difficult for humans to make. Machine learning systems require access to data from a variety of sources and have many particularly stressful I/O patterns. Essentially all emerging machine learning frameworks use standard file access APIs, but they often use certain capabilities such as randomly reading data from very large files that ordinary I/O systems find difficult to support.

- Support machine learning and AI development by transparent access to a wide variety of data from other systems and supporting standard POSIX APIs at very high levels of random reads.

Finally, it is common to overlook the fact that there is a lot of ancillary data associated with the analysis of large distributed data. For instance, there are container images, configuration files, processing status tables, core dumps, log files and much more. Using the same data platform for data and for housekeeping makes total sense and simplifies operations tremendously. All that is needed to support this are POSIX APIs, so no bullet point is needed, but a reminder definitely is.

In summary, the data platform plays an essential role in building the data fabric that you will need to move seamlessly to production deployments. It's not just the need to store data (in a variety of data structures) at scale in a cost-effective way that matters. The role of the effective modern data platform also includes supporting a wide

variety of access modes including by legacy applications and using familiar processing tools. And that leads our thinking back to the running of applications and how that can be managed at a production-ready level.

Orchestrate Containers with Kubernetes

When you build a production analytic system, there are two major halves of the problem. One half is the data, for which issues of scale, redundancy, and locality should be handled in a data platform as we have described. But the other half of the problem is the analytics: actually running the software to analyze the data. Analytics applications require a lot of coordination. One reason is that because the data involved is large, analyzing it often requires many computers working together in a well-coordinated fashion. There are probably many small parts of the computation, too. There are tons of non-analytical but data-dependent processes that need to be run as well. Getting large programs and small, I/O-intensive or CPU-intensive, low-latency, and batch to all run together is daunting. You really need to not only develop ways of efficiently orchestrating your analytics and AI/machine learning applications but also of having them coexist with other processes that are essentially "running in the background."

All of this implies that we need to have an analog of the data platform, what you might call a process platform, to coordinate pieces of applications. Given the increasingly widespread containerization of applications, it's essential to have a way to coordinate processes running in containers. The good news is that based on their experience building the world's biggest and most distributed big data production system out there, Google decided to open source Kubernetes. Kubernetes coordinates the execution of programs across a cluster of machines in pretty much the way that is needed for large-scale data analysis.

The way that Kubernetes works is that large-scale services are broken down into subservices that are ultimately composed of tightly bound groups of processes that run inside containers. Containers provide the ability to control most of the environment so that programs with different environmental requirements (such as library versions and such) can be run on the same machine without con-

flicts. Kubernetes orchestrates the execution of containers arranged as systems across a pool of computers.

Kubernetes purposely does not manage data persistence. Its role is to act as a container orchestration system that only manages where containers run and limits how much memory and CPU is available to a container. You really don't want to persist state in normal containers. The container is intended to be ephemeral, which is a key aspect of its flexibility; *data should (must) outlive the container in order to be useful*. The management of persistent data, whether in the form of files, databases, or message streams, is essential for many applications (stateful applications) that will be run in containers. The issue arises because applications generally need to store data to access later or so that other applications can access it later. Purely stateless applications are of only very limited utility. Kubernetes provides methods for connecting to different data storage systems, but some sort of data platform needs to be supplied in addition to Kubernetes itself.

NOTE

The combination of a modern data platform working in concert with Kubernetes for container orchestration is a powerful duo that offers substantial advantages for production deployments.

Containers managed by Kubernetes can have access to local data storage, but that storage is generally not accessible to other containers. Moreover, if a large amount of data is stored in these local data areas, the container itself becomes very difficult to reprovision on different machines. This makes it difficult to run applications with large amounts of state under Kubernetes, and it is now generally viewed as an anti-pattern to store large amounts of state in local container storage.

There is an alternative. Figure 2-3 shows a diagram of how several applications might interact. Application 1 reads a file and uses a direct remote procedure call (RPC) to send data to Application 2. Alternatively, Application 2 could read data that Application 1 writes to a message stream. Application 2 produces a log file that is analyzed by Application 3 after Application 2 has exited. If these applications all run in containers, storing these files and the stream locally is problematic.

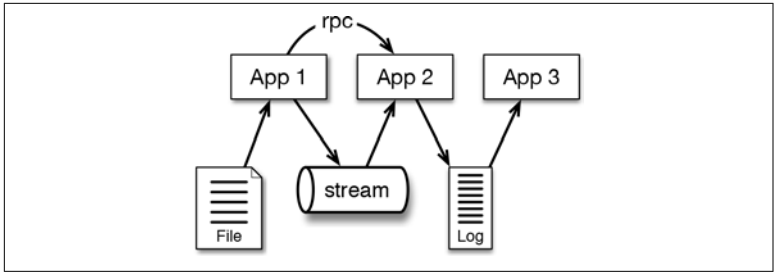


Figure 2-3. Three applications that communicate using files and a stream. Each of these applications needs to persist data, but none of them should persist it inside their own container.

Figure 2-4 demonstrates a better way to manage this situation. The applications themselves are managed by Kubernetes, whereas the data that the applications create or use is managed by a data platform, preferably one that can handle not only files, but also streams and tables, in a uniformly addressable single namespace.

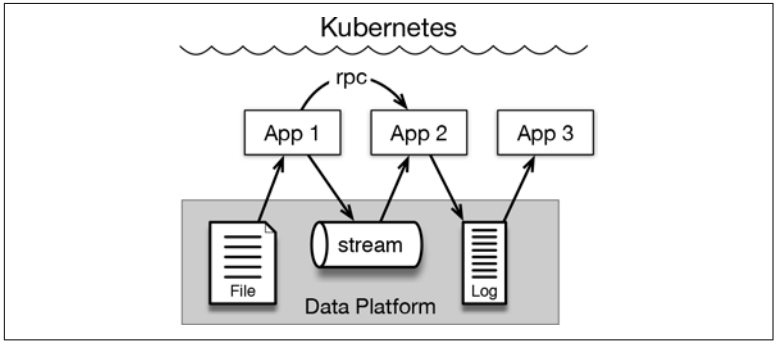


Figure 2-4. The applications are now run in containers managed by Kubernetes, but the data they operate on is managed by a data platform. This is the preferred way to manage state with Kubernetes.

Note that the output Application 1 writes to a stream can be used not only by Application 2 but also by some other as yet unknown application running on the same platform.

Key requirements of the data platform are that it must do the following:

- Abstract away file location so that containers need not worry about where their data is. In particular, there must be a consistent namespace so that containers can access files, streams, or

tables without knowing anything about which machines host the actual bits.

- Provide sufficient performance so that the applications are the only limiting factor. This might even require collocating data with some containers that demand the highest levels of performance.
- Allow access to data no matter where a container is running, even if a container is moved by Kubernetes.
- Provide reliability consistent with use by multiple applications.

Pairing a data platform with a container orchestration system like Kubernetes is a great example of success through separation of concerns. Kubernetes does almost nothing with respect to data platforms other than provide hooks by which the data platform can be bound to containers. This works because the orchestration of computation and the orchestration of data storage are highly complementary tasks that can be managed independently.

Extend Applications to Clouds and Edges

The combination of Kubernetes to orchestrate containerized applications and a data platform that can work with Kubernetes to persist state for those applications in multiple structures (i.e., files, tables, streams) is extremely powerful, but it's not enough if it is able to do so for only one data center. That's true whether that one location is in the cloud or on-premises. The business goals of many modern organizations drive a need to extend applications across multiple data centers—to take advantage of cloud deployments or multicloud architecture—and to go between IoT edge and data centers. We see organizations in the area of oil and gas production, mining, manufacturing, telecommunications, online video services, and even smart home appliances that need to have computation that is geographically distributed to match how their data is produced and how it needs to be used. Something as fundamental as having multiple data centers as part of advance planning for disaster recovery underlines this need. What's the best way to handle the scale and complexity associated with multiple required locations and to do so in a cost-effective way?

Let's break this down into several issues. The first is to realize that the journey to the cloud need not be all at once. It may be preferable

for systems to function partly on-premises and partly in cloud deployments, even if the eventual intent is a fully cloud-based system. A second point to keep in mind is that cloud deployments are not automatically the most cost-effective approach. Look at the situations shown in Figure 2-5. A hybrid on-premises plus cloud architecture can have benefits for being able to best optimize compute resources for different workloads in a cost-effective manner.

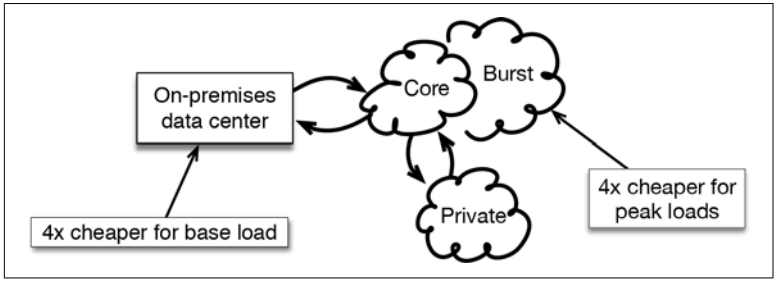


Figure 2-5. Although cloud deployments may be cost-effective for bursts of computation, base load often is better off in an on-premises data center. For this reason, an efficient hybrid architecture may be optimal. Similarly, optimization sometimes calls for having computation and data in edge locations as well as multiple clouds and on-premises data centers.

What you need to handle all these situations, including edge deployments, is to make sure your data platform is designed to handle geo-distribution of data and computation in a cost-effective way that is reliable and meets SLAs. That means having effective and easily administered ways to mirror data (files, tables and streams) and preferably to have efficient geo-distributed stream and table replication as well. The same platform should extend across multiple centers and be able to interact with Kubernetes across these geo-distributed locations. It's not just about data storage but also about built-in capabilities to coordinate data functions and computation across these locations securely. This use of edge computing is a basic idea that is described in more detail as one of the design patterns presented in Chapter 5.

The good news is that even though widespread use of Kubernetes in on-premises data centers is a relatively new, it is spreading rapidly. And for cloud deployments, Kubernetes already is, as of this writing, the native language of two of the three major cloud vendors, so using it in the cloud isn't a problem. The issue is to plan in advance

for these cross-location deployments both in your architecture and in the capabilities of your data platform. Even if you start with a single location, this geo-distribution of applications is likely what you will need as you go forward, so be ready for it from the start.

Use Streaming Architecture and Streaming Microservices

Often, the first reason people turn to event-by-event message streaming is for some particular low-latency application such as updates to a real-time dashboard. It makes good sense to put some sort of message queue ahead of this, as a sort of safety measure, so that you don't drop message data if there is an interruption in the application. But with the right stream transport technology having the right capabilities—as do Apache Kafka or MapR Streams—stream transport offers much more. Streaming can form the heart of an overall architecture that provides widespread advantages including flexibility and new uses for data along with fast response. Figure 2-6 shows how stream transport of this style can support several classes of use cases from real time to very long persistence. We've discussed these advantages at length in the short book *Streaming Architecture* (O'Reilly, 2016).

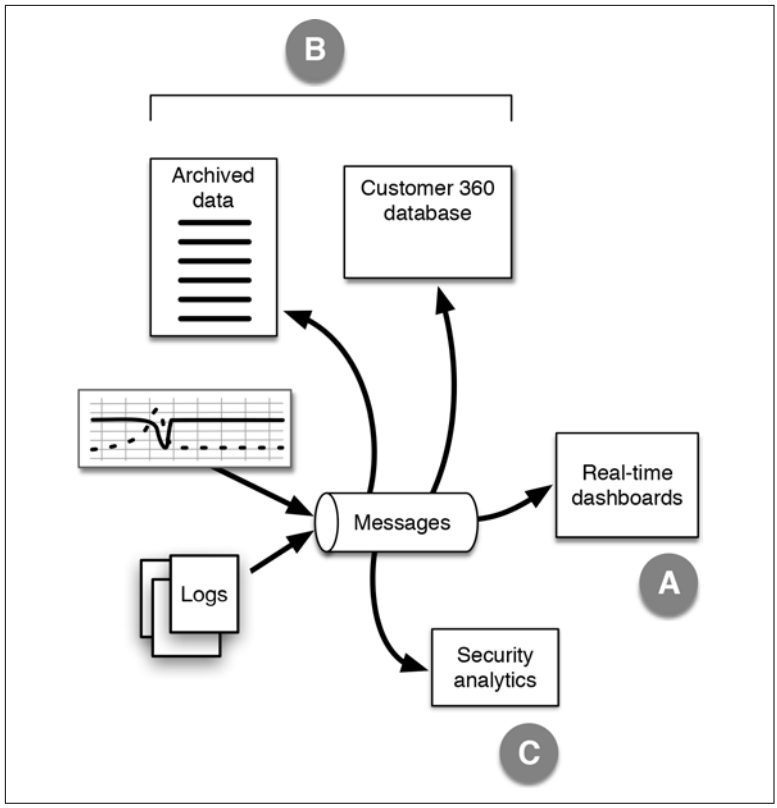


Figure 2-6. Stream-first architecture for a web-activity-based business. A message stream transport such as Apache Kafka or MapR Streams (shown here as a horizontal cylinder) lies at the heart of an overall streaming architecture that supports a variety of use case types. Group A is the obvious use, for a real-time dashboard. Group B, for archived web activity or Customer 360, is also important. Applications like security analytics (group C) become possible with long-term persistence.

Streaming is not the only way to build a data fabric, but stream-based architectures are a very natural fit for a fabric, especially with efficient stream replication, as mentioned in the previous section. In addition, message stream transport can provide the lightweight connection needed between services of a microservices style of work (see also "Streaming Microservices" by Dunning and Friedman in Springer's *Encyclopedia of Big Data Technologies* [2018]). The role of the connector, in this case a message stream, is to provide isolation

between microservices so that these applications are highly independent. This in turn leads to a flexible way to work, making it much easier to add or replace a service as well as support agile development. The producers in Figure 2-7, for instance, don't need to run at the same time as the consumers, thus allowing temporal isolation.

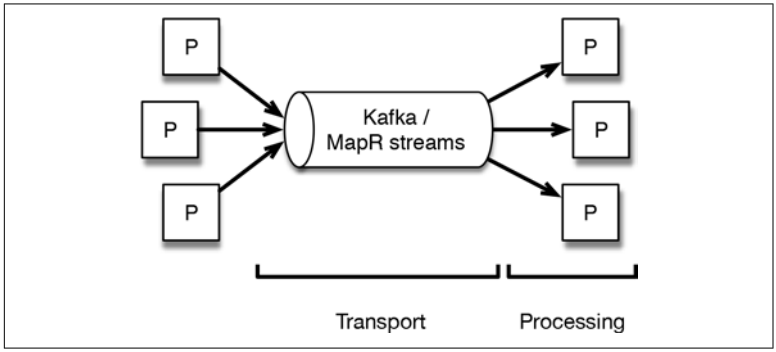


Figure 2-7. Stream transport technologies in the style of Apache Kafka exhibit high performance at scale even with message persistence. As a result, they can help decouple multiple data producers from multiple consumers. This behavior makes stream transport a good connector between microservices.

A microservices style is a good design for deployment to production as well as making development easier and agile. We've talked about the importance of flexibility in production, making it easier to make changes and respond to new situations. Microservices is also valuable because being able to isolate services means that you isolate issues. A problem in one service does not necessarily propagate to other services, so the risk (and blast radius) is minimized.

Commonality Versus Isolation (Independence): You Need Both

It may sound like a contradiction to say that there's a strength in the commonality offered by a global data fabric yet you still want the isolation and independence offered by microservices, but in a successful production design, you want both, just at different levels. Commonality is important for a comprehensive view of data, avoiding duplicated effort, easy use of multiple data structures, and deploying applications where you need them from edge to center to

cloud, under the same administration and security. Isolation at the level of microservices means that teams can work well and independently, that they can organize and protect their data and that new services can be introduced quickly with the least disturbance to other systems.

Commonality and isolation, each at the right level, provide a balanced, stable, and flexible system.

The idea of stream-first architecture is being put to work in real-world situations. We know a customer, for example, who built a stream-based platform used in production to handle the pipeline of processes for finance decisions. This stream-based approach was used mainly because of the need for fast response and agility. But keep in mind that there are many reasons, as we have mentioned, other than just latency to motivate choosing a streaming architecture.

Cultivate a Production-Ready Culture

Using new technologies with old thinking often limits the benefits you get from big data and can create difficulties in production. Here are some pointers on how to have good social habits for success in production.

DataOps

DataOps extends the flexibility of a DevOps approach by adding data-intensive skills such as data science and data engineering—skills that are needed not only for development but also to deploy and maintain data-intensive applications in production. The driving idea behind DataOps, as with DevOps, is simple but powerful: if you assemble cross-skill teams and get them focused on a shared goal (such as a microservice), you get much better cooperation and collaboration, which in turn leads to more effective use of everyone's time. DataOps cuts across "skill guilds" to get people working together in a more goal-focused and agile manner for data-intensive projects. Figure 2-8 schematically illustrates the organization of DataOps teams.

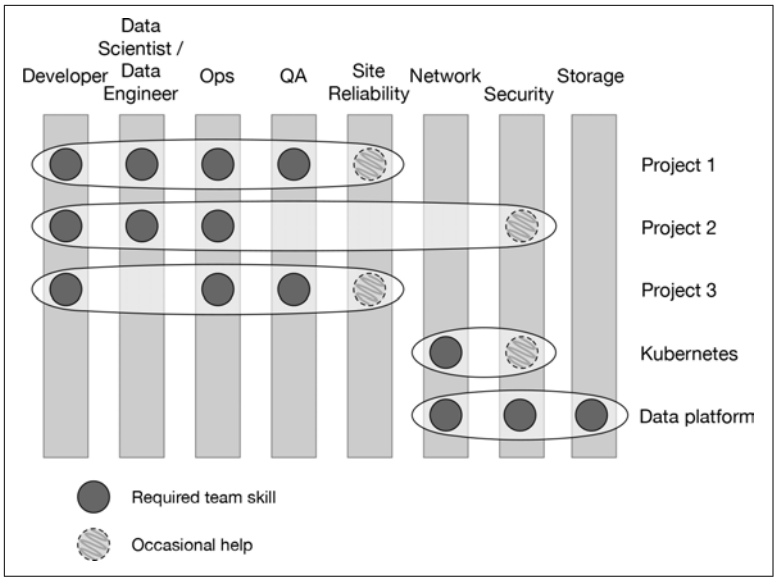


Figure 2-8. A DataOps style of work provides cross-skill teams (horizontal ovals), each focused on a shared goal. This diagram depicts teams for different applications (teams for Projects 1–3) as well as the separate teams for platform-level services (Kubernetes and the data platform). Note how the platform teams have very different skill mix than the other teams.

The key ingredients for DataOps are the right mix of skills, the concept of pulling together toward the common goal as opposed to strict reporting based on departmental divides and much better communication. Better communication is natural in this design because it changes the normal lines of communication and avoids great walls between skills that can be caused by slow-downs waiting for interdepartmental decisions.

A DataOps style of work does not mean having to hire lots of new people for each role. Generally, it involves rearranging assignments to better focus people in cross-functional teams. In some cases, people with particular data skills (such as data scientists or machine learning experts) might be temporarily embedded in an operations/engineering team. The mix of roles will naturally evolve as you go from development teams to production teams. Generally, there are fewer data scientists in a production team than in a development or exploration team.

Making Room for Innovation

A well-designed organizational culture not only provides for agility and flexibility. People also must be free to explore and experiment. A no-fail policy is a no-innovation policy. It's usually not practical to do unbounded experimentation, but to support innovation you do need to recognize that deliverables can be about more than just products: experience with new technologies and advance testing of new lines of business can be targeted as deliverables.

Remember: IT Does Not Have a Magic Wand

We have seen a pattern of failure where a development group gets the DevOps religion, builds a complex service, and puts it "into production" by slapping an SLA onto it and launching the service but leaves hazily defined critical tasks to the IT department. Commonly, the next thing that happens is that the service turns out to be less reliable than expected, and an IT group gets involved in running it (but that doesn't help much).

Obviously, one problem here is that you should declare SLAs *before* you build something if you actually want them to be met, but that isn't really the ultimate cause of failure here. Some people will respond to these stories by blaming the basic idea of a dev and ops team being fused together into DevOps, but that doesn't really describe what happened, either.

There is often a deeper issue in these cases. What happens is that when the original team starts building its system, it winds up taking on responsibility for core platform components such as storage, database, or message queuing on top of building the service that it is supposed to focus on. This sounds like the soup-to-nuts kinds of things that DevOps teams do, after all. As the project progresses, however, that core focus becomes more and more elusive because managing a platform as a side hustle doesn't really work very well. As deadlines loom, more and more development effort is siphoned into operational issues resulting in the design, implementation, and testing of the core service getting short-changed. By the time the service is delivered, the resulting problem is practically inevitable.

The core problem here is that DevOps and DataOps teams shouldn't be duplicating efforts on building duplicative platforms. This idea is indicated in Figure 2-8. The real lesson of the cloud revolution is

that it is a waste of time for application teams to be solving the same platform problems over and over (and doing it poorly because it isn't their focus). Public clouds have been successful precisely because they draw a sharp line between infrastructure and application. Due to limits of available technology, that line was drawn a tiny bit lower in the stack than it should have been, at least initially. The balance that Google struck with services like storage and tables being part of the platform appears to be where things really ought to be.

The good news is that this is exactly what you can do with modern systems.

Putting It All Together: Common Questions

Now that you've seen the ingredients of a formula for success in production—the commonality of a global data fabric, the role of the data platform and technologies like containerization and Kubernetes, the flexibility of streaming microservices and of DataOps—here's a few thoughts about how to put those concepts to work in the form of some common questions and answers about how to get big data into production.

Can You Manage End-to-End Workloads?

Managing means measuring and controlling. In real-world production systems, however, multitenancy adds the complication that there are lots of workloads active at once, so we need be able to detect and control interactions. Real-world systems also have to deal with ongoing hardware failures and the background loads due to recovery efforts. This means that just managing individual workloads in isolation isn't sufficient. Realistically, in production you also need to manage these implied workloads at the same time. For instance, if there is a disk or controller failure outage, a competent system will have built-in ways to re-replicate any lost data. That's great, but that re-replication will cause a load on your system that competes with your applications. You have to plan for both kinds of load and you have to measure and manage both as well as interactions between loads. In an advanced system, the implied loads will be managed automatically; less advanced systems require manual recovery (Kafka, Gluster, Ceph) or lack throttling of recovery loads (HDFS).

Kubernetes helps you orchestrate when and where applications run, the implied use of containerization helps with this by isolating environments in terms of dependencies. Kubernetes is mostly a control mechanism, however, and is limited to controlling the computation side of things.

You also need to have comparable control on the data side of things. Effectively, what you need is Kubernetes, but for data. So, although Kubernetes allows you to control where processes run, avoid too much over-subscription of compute and memory resources and to isolate network access, your data platform should let you control where your data actually is, to avoid oversubscribing storage space and I/O bandwidth and logically isolate data using uniform permission schemes.

But all that control is nearly worthless if you are flying blind. As such, you need to be able to measure. Obviously, you need to know how much data you have and where it is, but you should also be able to get comprehensive data about I/O transfer rates and latency distributions per node, per application and per storage object.

How Do You Migrate from Test to Production?

If you've designed and built your organization according to the guidelines we've described, you laid the groundwork for this migration long before your application was ready to deploy: you've treated data appropriately for in-production status from early on (where that matters) and you are operating within a global data fabric, with much of the logistics handled by your platform.

This means that you have what you need to be able to migrate to production. Keep in mind that this migration is not just a matter of unit testing but also of *integration testing*—making certain that the various parts of a particular application or interactive processes will work together properly. One way to do that is to have a system that lets you easily light up a production-like environment for testing: Kubernetes, containers, and a data platform that can persist state for applications help with this, particularly if they are all part of the same system that you will use in production. Using a stream as a system-of-record also makes it easier to pretest and stage migration to production because it is so much easier to replay realistic operations for testing.

Can You Find Bottlenecks?

We've argued for simplicity in both architecture and technology, for a unified data fabric, and for a unified data platform with the capabilities to handle a lot of the burden of logistics at the platform level rather than application level. This approach helps with finding (and fixing) bottlenecks. A simple system is easier to inspect, and an inspectable system is easier to simplify. Good visibility is important for determining when SLAs are missed and a good historical record should allow you to pinpoint causes. Was it bad design that caused hot-spotting in access to a single table? Was it transient interference with another application that should be avoided by better scheduling or moving loads around? This is important because you should operate with the philosophy that the default state of all software is "broken" and it's up to you to prove it's working.

You need to record as much of the function of your system as you can so that you can demonstrate SLA compliance and correct functioning. Your data and execution platforms should help you with this by providing metrics about what might have been slowing you down or interfering with your processes.

To meet this pressure, not only is a simpler system useful but so is having a comprehensive system of record. Having a persisted event stream, change data capture (CDC) for a database, and an audit system for files are various ways to do this. Some of the most successful uses of big data that we have seen record enormous amounts of data that can used to find problems and to test new versions as they are developed.

CHAPTER 3

Artificial Intelligence and Machine Learning in Production

Machine learning and artificial intelligence (AI) have become mainstream processes for many businesses because of their considerable potential to unlock value from big data. In light of this fairly new and increasing interest, we focus this chapter on AI and machine learning as a special example of big data in production. That is not to say that these processes stand apart entirely from what matters for all data-intensive applications in production. The same concerns, goals, and habits of best practice that we've already presented apply here, as well, but in several ways AI and machine learning shine a stronger light on some aspects of production. And in some cases, there are special requirements imposed by the special needs of AI and machine learning applications. Our goal is to give you practical pointers that can help make these systems a success in production applications.

AI and machine learning are favorites of ours. We've cowritten four books on aspects of these topics (three of them for O'Reilly), and one of us (Ted) has built a number of machine learning systems at several different companies that ranged from recommenders for music, video, or retail sales to detection and prevention of identity theft. We've also seen how many MapR customers are using machine learning and AI to extract value from their data in production deployments. From all of this, we've figured out some of the things that make a big difference for those people who are getting these systems to work well, and that's what we highlight in this chapter. It's

neither a checklist of how to build an AI or machine learning system nor a detailed lesson in data science. It is a collection of key insights that you could put to advantage in your own situation. Hopefully you'll find something useful whether you're a seasoned data scientist or someone just starting out.

Often people think of the algorithm used for learning as the thing that matters most for a successful machine learning system. But for serious AI and machine learning systems in production, the logistics of machine learning—managing data at all stages and multiple models—has far more impact on success than does the specific choice of learning algorithm or the model itself. Put another way, if you train the best possible model, but you cannot handle the logistics in a reliable, agile, and predictable way, you will be worse off than if you have a mediocre model and good logistics.

> *90% of the effort in successful machine learning is not about the algorithm or the model or the learning. It's about the logistics.*
>
> —From *Machine Learning Logistics* by Dunning and Friedman (O'Reilly, 2017)

Our data science friends tend to keep at least four or five favorite AI and machine learning tools in their tool kit because no single specialty tool or algorithm is the best fit for every situation. In addition, an organization likely will have more than one machine learning project in play at a given time. But the need to effectively handle logistics cuts across all these choices, and the logistical issues are surprisingly similar. That makes the data platform itself, working in concert with Kubernetes, the best tool for AI and machine learning overall. That's good news. You don't need to handle all this at the application level or build a separate system for each project. This, in turn, frees up data scientists and data engineers to focus more on the goals of the AI or machine learning itself. (For more on this topic, see the article "TensorFlow, MXNet, Caffe, H2O: Which ML Tool is Best?"). In addition to the platform and application orchestration technologies, you will need an architectural design that simplifies logistics, supports multiple models and multiple teams easily, and gives you agility to respond quickly as the world (and data) changes, as indeed it will.

Logistics are not the only issue that matters for success. Connecting AI and machine learning projects to real business value is of huge importance. And the social structure of your organization makes a

big difference, as well. Throughout the first part of this chapter, we examine these and other things (beyond the algorithms) that matter for success in AI and machine learning systems. In the second section of the chapter, we suggest specific new methods to deal with them.

What Matters Most for AI and Machine Learning in Production?

Machine learning and particularly AI are not well-defined terms, and there is considerable overlap. In both cases, models are usually built to make decisions and, in most cases, learn from training data to perform their desired task. Sometimes, the decisions represent tasks that humans can do. Other times, a system built using machine learning goes beyond what is feasible for humans due to the complexity of the problem, or the scale of data that must be considered during training, or the required speed. AI is sometimes used to refer distinctively to machine learning systems that mimic complex and sophisticated human-like performance, such as classifying objects in images, speech recognition, or the interactive real-time processes required for autonomous cars to interact safely with their environment. For ease of discussion, from here on we will just use the general term "machine learning" unless there's a specific reason to distinguish AI from machine learning.

Getting Real Value from AI and Machine Learning

What about value? Big data and new technologies have opened a whole new spectrum of opportunities using machine learning in mainstream business processes that previously were not practical.

The excitement around machine learning has people looking for new opportunities to use it to do new things, either by opening up a new line of business or by enhancing an existing line in previously impossible ways. For example, we have large industrial customers who use machine learning to identify complex patterns in Internet of Things (IoT) sensor data that can serve as potential "failure signatures"—patterns that signal an impending equipment failure sometimes weeks before the event. This process is known as *predictive maintenance*. This wasn't possible prior to the use of machine learning with large-scale data, and it decreases unplanned outages and decreases the cost of planned maintenance.

Another example of doing something new using machine learning is one we observed with one of our large financial customers. Its data science team built a recommendation system based on customer transactions in order to provide personalized offers for its credit card customers. This recommendation service has proven very popular and valuable for the company, providing significant net new revenue. Other customers breaking new ground include automobile manufacturers that use very sophisticated machine learning to build autonomous vehicles. At the same time, they are building better monitoring and even some predictive maintenance capabilities into vehicles that will be available soon.

In addition to the wide-open opportunities for doing new things, some of the best value per effort from machine learning can come from applying it to things your organization already does. Machine-based decisions can be faster or more consistent than human decisions, and this can result in incremental value especially for monotonous processes. These opportunities arise across all types of enterprises, even those that also need machine learning for new processes. We have seen a large industrial customer use machine learning to improve the boring but essential process of labeling accounting charges for parts and services more accurately so that they are correctly handled with regard to billing, reimbursements, and tax reporting. This optimization might not sound exciting, but it can save the company millions of dollars right now. An effective data scientist knows to look for those places in existing business processes where automation or optimization via machine learning can offer a big return for small effort.

Whether machine learning is used for doing new things or to automate preexisting processes and decisions, the model needs to fit the requirements of the situation. In some cases, this means building a complex model, but often a very simple model is plenty powerful. We have customers in the medical field, for instance, using very sophisticated models to help physicians analyze medical imaging results. Similarly, the real-time interactive models such as those in autonomous cars must be very complex. In contrast, the recommendation system we mentioned earlier, although very accurate, uses a very simple approach. (see Chapter 5 for more on this, or we invite you to read an earlier book of ours, *Practical Machine Learning: Innovations in Recommendation* (O'Reilly, 2014).

Not only can simple be powerful, it is intrinsically desirable. Premature complexity in building a machine learning system can obscure what is really happening and wastes time that you should be spending on getting the infrastructure and logistics right. Getting complicated too early can cause you to miss opportunities for value by getting bogged down on one problem while neglecting another. The point is to recognize that simplicity is a virtue in and of itself; more complex approaches should be used only when really needed.

NOTE

The value to be derived from machine learning is not proportional to the complexity or sophistication of the model you use. Make the model as simple as it can be and still do the job, but no simpler.

One thing that is surprisingly easy to forget when you get into the details of making a model work is that *a machine learning system must connect to practical action that supports business goals*. The model might come up with a clever answer, but what action will you take based on those insights? Or better, what action will be taken automatically? Remember: a report is not an action. A machine learning model that does not connect to practical actions for the business is like a well-tuned motor spinning out of gear and thus ineffective.

We close this discussion on getting value from machine learning with an example that shows that there can be more than one level of insight from a machine learning system. If you ask the right question and have access to the right data to support it, you could find more valuable insights than you originally expect.

Fraud detection example: when a time machine would help

Fraud detection is a classic example where low latency decisions are needed. This might be fraud in the form of a false credit card transaction or in the form of a breached online account. The shorter the time between the event when fraud occurs and the time that you detect it, the sooner you can limit loss and perhaps find the perpetrator. This time interval is depicted in part A of Figure 3-1. Fast detection can certainly be valuable.

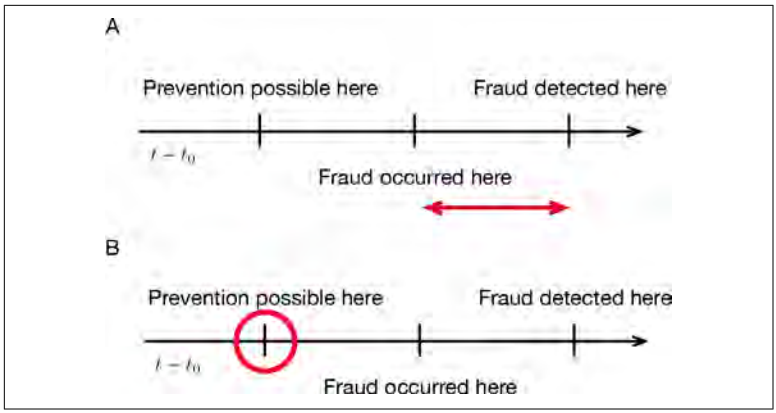


Figure 3-1. Do you use machine learning just to detect fraud or to go further—to learn how to recognize an earlier time when the fraud might have been prevented? Detection is nice, but taking action to prevent it is better.

But how much more valuable is it to *prevent* the fraud from taking place at all? Often (in a manner analogous to the preventive maintenance example), there are clues in behavior in advance of a fraud attempt, perhaps seconds or minutes before the fraud event or during the transaction itself. With the right model, you might be able to recognize a significant pattern in the window of time *prior* to the fraud attempt, back when you could prevent the fraud and thus prevent any loss, as suggested by part B of Figure 3-1. To build a prevention model, you need to correlate what you knew at the moment of prevention with the later outcome. But that is difficult, particularly because you can't know to collect the data before you see the fraud.

This situation happens commonly in the real world and is even more challenging with many users and frauds at different times. This means that the prevention windows all happen at different times, as shown in Figure 3-2.

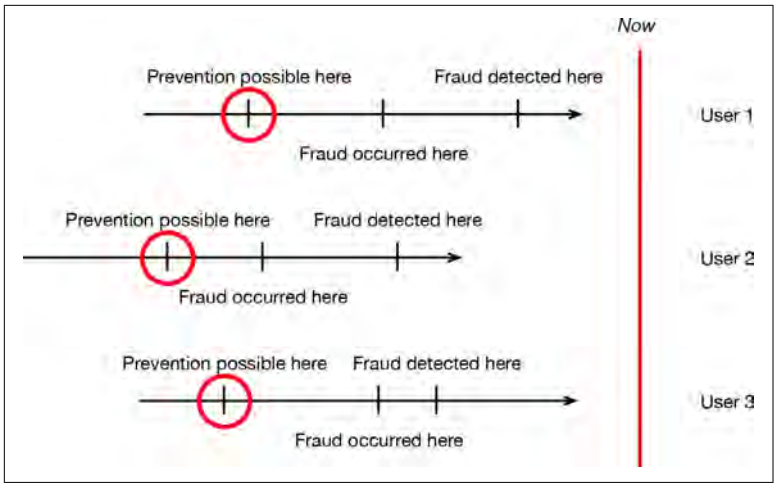


Figure 3-2. In building a fraud prevention model that tracks multiple users, you need to sample data from many different times when fraud could have been predicted and prevented. Those times can only be determined retrospectively, so you must collect this data in advance of knowing the exact time points of interest.

The approach that gives you the best option in situations such as these is to have retained event data even in advance of detecting fraud. You need a comprehensive system of record that stores everything you knew and could have used. With a data platform having capabilities such as we described in Chapter 2, this is surprisingly doable. You can use something like change data capture (CDC) to record changes in a table or, even better, a streaming system of record. We describe both of these design patterns in Chapter 5. But the key lesson here is that you need to have arranged to collect data without knowing in advance exactly when data will be required. If you don't do this, you must extrapolate back to each prediction point from the current time, which inevitably results in poorer results.

NOTE

The data you retain (and have access to) changes the questions that you can ask.

As we've said in earlier chapters, it's important to recognize that data might need to be considered "in production" much earlier than people often think. The use of *data time machines* to retrospectively find out which data is important is a great example of how this happens.

Data at Different Stages

A data-oriented business looks to data as a key decision-making tool to inspire, validate, or contradict intuitions. Machine learning models are creatures of data, and for that reason, a data-oriented organization will find machine learning a relatively natural process. As we described in Chapter 2, this type of organization can benefit from a good data platform that works well with Kubernetes and containerization of data-intensive applications.

On the other hand, a business that isn't data oriented will typically have a very difficult time getting good value out of machine learning. This happens partly because an organization that does not consider data to be a key business asset will tend not to retain data in the first place, nor will it put a priority on getting data right. Such an organization won't typically see as much value in a data platform.

Following are data characteristics that are needed in machine learning:

- Comprehensive view of data
- Data from many sources and of different formats
- Flexibility to add new data sources easily
- Data accessibility for a wide variety of applications
- Large data volume
- Access to data *as it was* (both raw data and training data)

To expand on some of these points, consider this: data used to train models (training data) generally is derived from, but is significantly different from, raw data. Data scientists experiment with extracting different features and evaluating how models perform with these different options. Along the way, they learn about which data is not relevant or useful for building a model, retaining the most useful parts and discarding those parts that aren't useful. Training data also can be processed, aggregated, and combined with other data so that it can be used as input to train a model. The same process happens

with input data consumed by models in production. What data do you need to preserve after you've deployed models to production?

The answer is that *you should preserve data at different processing stages, particularly raw data, even after the system is in production.* There are two good reasons for this. First, raw data has less possibility for corruption due to buggy processing. Second, raw data is valuable in part because later you might want to use features you would have thrown away if you preserved only what you currently know to be valuable. This situation is not unique to machine learning applications, but given the way that machine learning systems must adapt to a changing world and given how new machine learning techniques are often able to find new value in surprising features, being able to go back to raw data to extract additional features is important.

In addition to having ongoing access to raw data, it's also useful to preserve *training data exactly as it was* for every model that at some point could be destined to go into production. Preserving training data this way is the exact analog to controlling source-code versions. Being able to rebuild a model exactly is important for compliance but just as valuable for debugging. When a model misbehaves in some new way, you need to be able to answer whether to blame new data or new code.

NOTE

It's important to have access to data as it was, not as we think it was! Raw data should be recorded as it was and training data should be frozen and tracked at the moment of training.

It's not usually possible to save absolutely all raw data, but there are technologies that make it reasonable and affordable to save large amounts of historical data, including files or streams, and make that data accessible to multiple users. To make it truly accessible, the data platform should make it easy to document and locate data versions and the organization needs to cultivate good data habits.

The Life Cycle of Machine Learning Models

One of the things that may surprise people who are new to machine learning is the fact that in real-world machine learning systems, the life cycle of a model isn't just one successful turn through planning,

training, evaluation, and deployment. Put simply, the process isn't "won and done," but instead is more like "a little better each and every day." Iteration is the rule.

Not only will you have multiple successive versions of a particular model to manage over the life cycle from development to production, you'll also have different models for different purposes, possibly hundreds, all being evaluated, modified, deployed, and reevaluated, all at once. Most experienced data science teams try many different approaches for the same goal, using multiple tools and algorithms to develop their models. It makes sense to try many models and tools because no single approach or tool is best for every situation. In addition, even after high-performing models have been deployed, you'll still need an update. It's not a matter of *if* the outside world will change, it's just a matter of *when*.

Discard the myth of the unitary model: real-world machine learning systems involve large numbers of models at every stage of the system. It's almost never just one.

Effective model management, as with data management, depends on having good technology support for multiple scoring applications and learning frameworks all working together.

Being effective at model management also requires that you collect a lot of metrics about model performance, both in development and in production. Is a model even running? Is it producing results with acceptable latency? Is the general level and shape of input and output data what is expected? There are also the questions of how one model's performance compares to other models, in terms of accuracy or usefulness of insights. To manage all this across many models not only needs lots of metrics but also a history of decisions.

You'll want to retain additional information about models—a kind of tribal history. The features that work well in one model often are good for others even if the models don't do the same thing. Keeping a "famous feature" list is a good practice, especially if it can spread across multiple data science or DataOps teams. Code and data sharing are critical to foster this.

A final but very important issue in the life cycle of models is that you should hold in reserve the ability to deploy new models very

quickly as external conditions change, particularly if you are in an adversarial situation against, say, fraudsters. Obviously, you need lots of monitoring to be able to determine when external conditions have changed, but for high-value cases, you might want to keep some alternative models handy that were developed using very different strategies than your primary models. These alternatives should be ready to deploy at a moment's notice. These models would have already performed well on evaluation and just be "waiting in the wings" to be used when needed. The point is that if attackers find a hole in one model's behavior, a different model can throw them off balance by having different vulnerabilities.

In addition to continuous deployment, you also might consider speculative execution. In this case, you not only have models ready to deploy, you actually do deploy them and have them evaluate all (or most) incoming requests. This lets you get some experience with these alternative models in a completely realistic environment. Because they are already producing results, you can switch over to the secondary models at any time, even in the middle of a request. This approach isn't appropriate for all situations but in situations for which it is reasonable, it can offer big advantages in agility and resilience. We talk later about the rendezvous architecture that takes this to the limit.

Specialized Hardware: GPUs

If you take a cross section of all different kinds of businesses, there are almost no really widespread business applications for Graphics Processing Units (GPUs)—almost no application, that is, other than machine learning. The reason is that what GPUs really do well is numerical computing on a really massive scale. Outside of, say, Hollywood render farms and large-scale physical simulations of structures, airflow, molecules, or drugs, there aren't a lot of business applications that require that level of numerical capability. Mostly, GPUs are a science kind of thing. Those non-machine learning applications that do tend to use GPUs are found in very particular industries. Not so with machine learning, which is being used more and more across a huge swath of industry segments and businesses.

But with machine learning, the advantage of GPUs is not at all a hard-and-fast rule. For each problem, there is a real question about whether they actually will help. Surprisingly, even though GPUs have enormous potential speed for mathematical operations, there

are also substantial overheads, a sort of mathematical inertia. Computations also need to be arranged in a data-parallel way and have limited amount of input/output (I/O), especially network I/O, per unit of mathematical computation. You could even say that the problem of machine learning logistics applies all the way down to the hardware level.

Some machine learning algorithms, however, make very good use of a GPU, and there can be an advantage of 10 times or more in terms of number of machines required to train a model. However, for other computations, ordinary processors can actually be faster than GPUs for training models. The latter is particularly true if you are training relatively simple models based on large amounts of training data.

Training is one thing, though, and using a model is quite another. Training is typically far more compute intensive than using a model for inference, so it may be optimal to do training on GPUs. Using a model, on the other hand, commonly, but not always, shifts the advantage back to general purpose processors even if the GPU has a substantial advantage for training.

This strong dichotomy about which kind of system is faster for training or inferencing is even more tricky if you look at total cost per useful computation rather than number of computers or elapsed time.

The real lesson of all of this is that if you are serious about machine learning, you should get access to some GPU horsepower (at least for testing), but you probably won't want to run everything on GPU machines by a long stretch. Having a cluster where you have a choice of hardware is the ideal situation.

Practically speaking, this implies that you need to account for special purpose machines in otherwise vanilla clusters. This, in turn, puts demands on your data platform to support the ability to position data near (or far from) GPUs as well as to force programs to run on GPU machines or to avoid them. The systems that we recommend elsewhere in this book can generally meet these requirements, but there are a number of big data systems that really don't deal well with these requirements because they don't have any workable concept of configurable data locality, nor can they deal easily with clusters composed of non-uniform machines.

Social and Teams

Much of what leads to successful deployment of machine learning and artificial intelligence systems into production is really more about people, communication, and organization than it is about the technical side of things. This isn't surprising given how much of the problem really comes down to understanding what your data can offer and the questions to ask as well as handling logistics. After all, logistics problems are usually solved by teams with good communication paths. There are, however, some surprising aspects of how machine learning systems work that can make some of these communication issues even more important than they would be in a more conventional application.

DataOps enhances business awareness

The biggest communications issue we have seen in machine learning is that data scientists need to have a direct line to business value stakeholders. This is exactly the kind of communication that a DataOps approach is intended to foster. Without this line of communication, it is incredibly easy for data scientists to solve the wrong problem. It might sound stupid, but there lots of opportunities for this, and pretty much every experienced data scientist has stories like this (even if they don't like to talk about them). From our own experience, for instance, there was a case in which we built a fraud detection model that was producing awesomely good results. Although that seemed like really good news, it turned out that the model was predicting the contents of the column labeled "Fraud" which was wasn't actually fraud at all but was instead an indicator of which fraud analyst had worked the case. The correct target had a name something like "dummy53" because it was one of a number of columns reserved for expansion in the schema of the database on initial deployment. This situation was embarrassing, but it could have been avoided if we had been part of a DataOps team. In that case, the error likely would have been noticed early on by another team member with a different knowledge base during model reviews and early discussions. It was the relative isolation of our analytics team from the business and data owners that allowed the error to go undetected until after we delivered a nearly useless model.

A DataOps team organization doesn't automagically give you the communications you need, however. It is important to enhance that communication with other tools such as model design reviews in

which the builder of a proposed model walks through the planned inputs, proposed features, the expected output, training plan and the expected complexity of the model. Data engineers are very good sounding boards for this kind of information given that they know what data is available and typically have a good idea about how model outputs can be used. Often it is helpful to use a tool such as Jupyter or Zeppelin notebooks to facilitate these reviews. Having active code available helps avoid the problem of slideware that doesn't match any actual code. These code reviews can also help to get an earlier start on the development of pipeline steps that will provide required inputs.

Remembering history

The process of building a new model is often highly iterative, which often results in problems if the development history of a model is not well documented. The issue is that problems in models are often quite subtle and that can make them difficult to pin down. For instance, there can be a bug in feature extraction code that gets "fixed" in a way that makes the feature less useful to a model. Or training data from a particular time period might be polluted by a bot gaming your product. This is particularly problematic when there are lots of design iterations. Unless you keep careful track so that you can go back and test for bad assumptions, you can spend a lot of time in the weeds. Is the training data collection broken? Is feature extraction behaving wrongly? Is the training process being stopped too soon? Or too late? Being able to exactly replicate past work is incredibly important. It's good practice to preserve your code and snapshot your training data.

Methods to Manage AI and Machine Learning Logistics

Model and data management is an essential and ubiquitous need across all types of machine learning and AI. There's more than one way to handle it, but handle it well you must. Keep in mind some of the most important challenges: to make a large volume of data from a wide variety of data sources available to multiple models, to have a way to go back to raw data and to know exactly what training data a model saw, to manage many models running in predictable but different environments without interfering with each other and to be

able to quickly and easily roll new models into production (or roll them back). You need an architecture and a data platform that let you handle much of the boring but essential work of managing logistics at the platform level rather than having to build it in to each application. The good news is that with a good overall approach, you don't need to change the architecture and underlying technology you use to handle logistics with each different AI or machine learning tool and each different project.

One new method for doing this easily for many types of AI and machine learning systems is called the *rendezvous architecture*. It's not only a useful and innovative approach to managing logistics, it's also a good example of the basic principles of good architecture for production. We describe key features of rendezvous in some detail in the next section. After that, we provide a brief overview of other emerging methodologies for managing logistics.

Rendezvous Architecture

Rendezvous architecture is a fairly new streaming microservices architecture that takes advantage of key capabilities in a modern data platform combined with containerization and Kubernetes to provide a framework for deploying models. This design was introduced in 2017 and already has been implemented (and in some cases with parts in production) by several groups we are aware of, including teams at Lightbend, Adatis, and a large US financial company. Rendezvous has been described in detail in our short book *Machine Learning Logistics: Model Management in the Real World*, (O'Reilly, 2017) and "Rendezvous Architecture" is an entry in *The Encyclopedia of Big Data Technologies*, Sherif Sakr and Albert Zomaya (eds.) (Springer International Publishing, 2018).

Rendezvous helps deploy models that solve synchronous decisioning problems. The key characteristic is that the amount of request, context, and state information for each request is relatively small, as is the result. Systems of this type include credit risk analysis for loans, flagging potential fraud in medical claims, marketing response prediction, customer churn prediction, detection of energy theft from smart meters, and systems to classify images through deep learning models. Autonomous driving is a good example of a problem that is not appropriate for rendezvous because the context is large (the entire recent history, the physical state of the car, and the contents of memory) and the responses are not bounded. Cer-

tain kinds of ad-targeting systems are not appropriate because of the cost of speculative execution.

The most important point of the rendezvous architecture is to make it easy to manage and evaluate multiple models at the same time and to have some known-good models already running and whose results are ready to be used easily and quickly when needed. This is done by having all live models evaluate all requests and then having a component known as the rendezvous server choose which of the results to return. This choice is done independently for every request by prioritizing the models against response time. Thus, a preferred model might be given a substantial amount of time to respond, but a response from a faster but less accurate backup model would be kept handy in case and used only if the preferred model doesn't come through in time. This trade-off of preference against time is encoded in the so-called rendezvous schedule.

With this system, roll out of a new model is easy. Start the model so that it starts reading requests from the input stream and generating results. Monitor it until you are confident that it is warmed up, stable and accurate. Then, update the rendezvous schedule so that the rendezvous server quits ignoring the new model. Figure 3-3 presents a simplified outline of the rendezvous architecture.

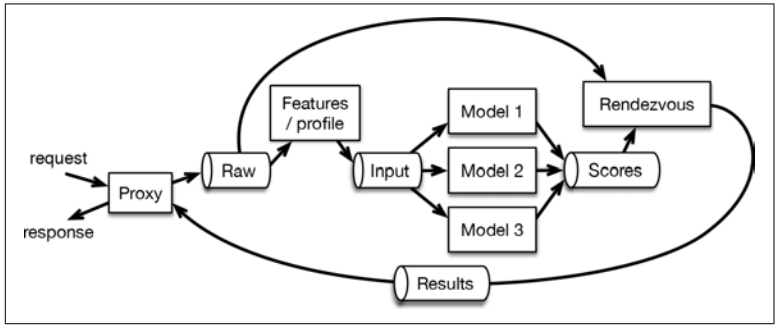


Figure 3-3. The stream-based rendezvous architecture uses message streams, represented here by horizontal cylinders. The upper curved arrow allows the rendezvous server to note request times as they are received by the proxy and put into the raw stream. The lower curved arrow shows how the rendezvous server publishes a single response back to the proxy for each request.

Of course, in real-world situations, there would be many models being compared and managed at any point, not just the few shown

in Figure 3-3. These additional models would be managed by additional instances of the rendezvous architecture, each isolated to a single problem. Notice how streams are used for raw data, for input data, for model scores, and for final results. Containerization of models with Kubernetes for orchestration is also very straightforward with rendezvous further providing non-interference of models running at the same time. New models can be started at any time without any reconfiguration or notification because they can begin reading requests and producing scores at any time. This streaming microservices architecture makes it possible for your machine learning system to respond in a flexible and agile manner, adapting quickly to changes as appropriate.

A rendezvous architecture helps with training data collection, as well. Figure 3-4 shows how you can deploy a decoy model (shaded) into a rendezvous service. This decoy model will receive exactly the same input data as any other model, but it won't produce results; instead, it will archive those inputs (which include request identifiers) for later correlation against model scores or ground truth information for model training. You might think that archiving input data is not necessary, but a perennial problem in machine learning is that reconstructed data is often not identical to real data, especially if it has been collected as a byproduct rather than an intentional and accurate record.

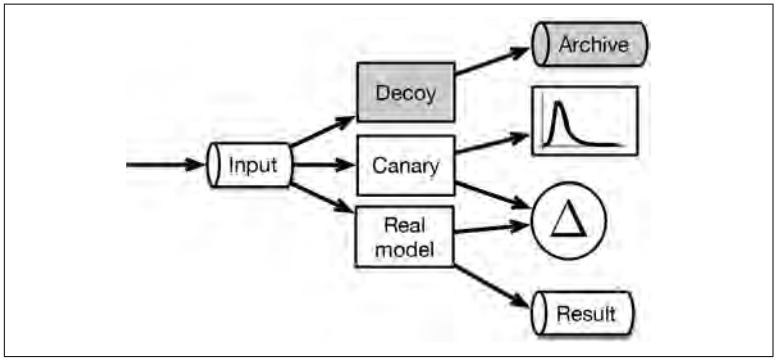


Figure 3-4. You can add a decoy and a canary model to a rendezvous architecture. The decoy archives input data for later use in training and debugging. The canary provides a useful real-time comparison for baseline behavior of both input data and other models.

Another handy trick that comes easily with a rendezvous architecture is a canary model, as depicted in Figure 3-4. The canary is not intended to provide usable results; the rendezvous service likely is configured to ignore the canary, in fact. Instead, the canary model is a model known to have reasonable and very stable performance, and, as such, it provides a scoring baseline to help you detect shifts in input data and as a benchmark for other models.

For detecting input shifts, you can compare between the distribution of outputs for the canary model and recent and older distributions. Details of how to do this are explained in the *Machine Learning Logistics* book, but the basic idea is that model score distribution is a very useful measure of semantic structure for the incoming data. If that distribution jumps in a surprising way, we can infer that the canary has detected a significant change in inputs.

Using the older canary as a benchmark for behavior of new models might seem surprising. Why not just compare new models to each other? As it turns out, the canary's age is a benefit—that's what makes it so useful as a benchmark for model performance. Over time, every new model will have been compared to the canary during the preproduction checkout and warm-up period. The DataOps team will have developed substantial experience in comparing models to the canary and will be better able to spot anomalies quickly. That can be an advantage over comparing new models to other new models because none of them will have much of a track record.

Other Systems for Managing Machine Learning

There are a number of systems being developed to help with the management of the machine learning process. Unfortunately, most are limited to the learning process itself. Very few are concerned with the cradle-to-grave life cycle of models. The systems that are available are under intense development and the selection of systems and advantages of each will change rapidly for the next few years.

Of these systems, the one with the most promise is likely to be Google's KubeFlow. Intended as a full-cycle management system, it is a bit rough at the moment, but most of the critical pieces are available, albeit from the command line and in limited form right now. If Google can follow up on the success of Kubernetes itself, Kubeflow is likely to become an overwhelming standard.

The Clipper system from the University of California Berkeley RISE lab does not take a comprehensive view of the problem. Instead, RISE looks at what it takes to deploy models into production and to serve results with very low latency. One of the interesting things that Clipper does is to integrate result caching into the framework to help lower latency.

There are a number of startups working in the area, each with a different focus. Valohai, for instance, is working on a flexible model as a service offering. Hydrosphere has, so far, taken a very advanced look at monitoring of production models. You can expect to hear from a large number of additional startups in this area, as well.

CHAPTER 4

Example Data Platform: MapR

You probably didn't wake up this morning thinking, "I'm gonna get me a global namespace!" But you likely do want the things it can make possible for your production systems. What makes this and other capabilities of a data platform interesting is knowing how they can help in getting systems into reliable production. This isn't always obvious, especially as new technologies offer features that are just that, new. In this chapter, we describe one example technology: the MapR data platform. Most of the insights we've discussed so far, plus the design patterns we describe in the next chapter, are based on watching how people build successful production systems, and this is the platform they used.

Only a few of the design patterns we describe in Chapter 5 absolutely require capabilities unique to MapR, but all of them demonstrate good practice in pushing platform-appropriate functions down to the platform rather than trying to implement them at the application level. That's an important general lesson regardless of the platform you use.

Understanding how the MapR data platform works will make it easier to see the key ideas in the design patterns. Once you see that, you'll understand how to adapt them to your own needs. We also think you might find this technology interesting in its own right. Toward that goal, we use the first half of the current chapter to give you a grounding in some key capabilities of the MapR platform in the context of production deployments and then go on in the second half of this chapter to describe the underlying technology.

NOTE

Fundamental features of the MapR platform—files, tables, and streams—are all engineered together into a single technology, part of the same code and able to run on the same cluster instead of being separate systems that work together via connectors.

A First Look at MapR: Access, Global Namespace, and Multitenancy

One of the key distinctions of MapR's data platform is that it provides a real-time, fully read-write filesystem. This capability means that you not only can interact with data stored on the cluster via Spark or Hadoop commands and applications, but you also can access the exact same data via more traditional methods and legacy applications. Any program in any language running on Linux or Windows system can access files in the MapR cluster using standard file input/output (I/O) mechanisms.

This broad compatibility with the MapR file system (MapR-FS) is possible primarily because MapR-FS allows access to files via a FUSE interface or via Network File System (NFS), both supporting a POSIX API. Additionally, files on a MapR system can also be accessed via the HDFS API and via the S3 API. This wide variety of access methods is very different from Hadoop Distributed File System (HDFS), the file system that Hadoop distributions use for distributed data storage, which supports only limited APIs and semantics. That's part of the reason for the sense you get of a wall between data in a Hadoop cluster and whatever non-Hadoop processing or modeling you want to do.

What are the implications of MapR-FS being a read/write file system? One effect is that *existing applications*—so called *legacy code*—*can access data (big or small) without needing to be rewritten using Spark or Hadoop.* This interoperability eliminates copy steps and saves considerable time. Fewer steps helps make success in production more likely.

Another surprising characteristic of MapR-FS is that *the file system is extended to include tables and streams on a par with files*. You can access MapR streams by using the Apache Kafka API, and tables can be accessed using the Apache HBase API or the OJAI document database API, but the data is actually stored in the MapR file system, in the same system that stores file data. We talk in more detail about

how that works in the second part of this chapter. The big idea is that having files, tables, and streams as objects in the same file system reduces complexity in infrastructure and architecture and often requires fewer servers because different services can share the same cluster. You can also use the same security and same administration for all three kinds of objects.

Equally surprising is that all of these data structures *live in the same directories and share a global namespace*, as you can see in Figure 4-1. Actually, we have known for decades that being able to name and organize files was a good thing that makes development easier and less error prone, so it makes sense that this is good for tables and streams. Directories and a global namespace simplify data management for administrators, as well.

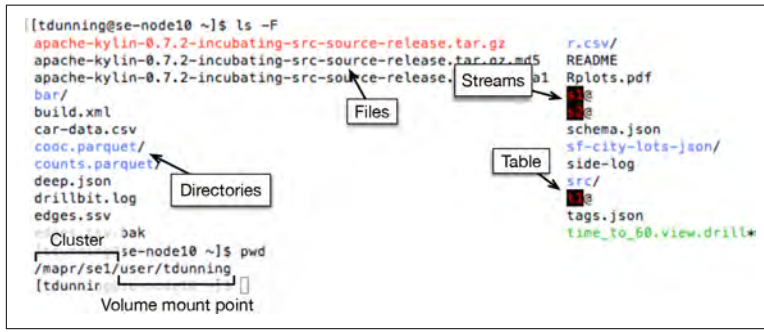


Figure 4-1. A screenshot of a user listing the contents of a directory on a MapR system. This directory contains conventional files, streams, and a table, as well as other directories. Note that conventional Linux utilities are used here even though the data is stored on a highly distributed system.

With an HDFS cluster, you have directories for organizing files but no analogous way to organize tables or streams short of proliferating lots of small special purpose clusters. If you use Apache Kafka for stream transport you can assign data to topics, but you don't have any good way to avoid topic name collision between applications.

You may also have noticed in Figure 4-1 that the path name is divided into a cluster name (the `/mapr/se1` part) and a volume mount point part (the `/user/tdunning` part). MapR-FS has something called a volume, which is essentially a directory with specialized management capabilities that enable scalability, disaster recovery, and multitenancy capabilities. To users and applications, volumes

appear to be ordinary directories. From an administrator's point of view, however, a volume has a number of additional capabilities that include data locality, snapshots, permissions, and mirrors.

Volumes allow you to position data explicitly within a cluster using a feature known as data locality. You can use this to prevent performance degradation for particular data sets due to heavy I/O loading from rogue jobs (i.e., almost anything developers or data scientists do without proper supervision). You also can use data locality to ensure that data being consumed by high-performance hardware such as Graphics Processing Units (GPUs) is close to the GPUs or is stored on high-performance storage devices such as flash memory. Regulatory and compliance issues may motivate a need to physically isolate some data in a cluster onto certain machines without impairing the ability to use that data.

Volumes also allow exact point-in-time snapshots to be taken manually or via automated scheduling, a valuable capability to protect against human error or for data versioning. Entire volumes can be atomically mirrored to other clusters for testing, development or disaster recovery purposes. These are additional ways that you can push many tasks to the platform for automatic execution rather than being a burden for the application developer.

MapR-FS is also automatically controls and prioritizes I/O loads caused by system functions such as data mirroring, recovery from hardware failures or moving data to cold storage. Correct prioritization of such loads is tricky because prioritizing application I/O loads can actually degrade reliability if lost replicas of data are not recreated quickly enough. Conversely, prioritizing recovery loads can interfere with applications unnecessarily. MapR-FS handles this variable prioritization by controlling I/O loads on an end-to-end basis.

Geo-Distribution and a Global Data Fabric

You can connect MapR clusters together into a single data fabric by near real-time replication of tables and streams. This means that an application developer or data scientist can just read or write data in a stream or table as if it were local, even if it decidedly is not. A program can write data into a stream in one data center and an analytics program could read it in another, but each application would not need to know where the other is. Data could have come from an instrument in front of you or across the ocean—it *feels* the same.

You don't need to waste time at the application level making sure bytes move reliably; the platform takes care of this for you.

Figure 4-2 shows a highly simplified diagram of how a data fabric is built using the multimaster stream and table replication capabilities of the MapR platform. The figure shows two clusters within the data fabric. The first has two processes analyzing file, stream, and table data. The stream and table data from the first cluster are replicated to a second cluster where another program could analyze the data, as well.

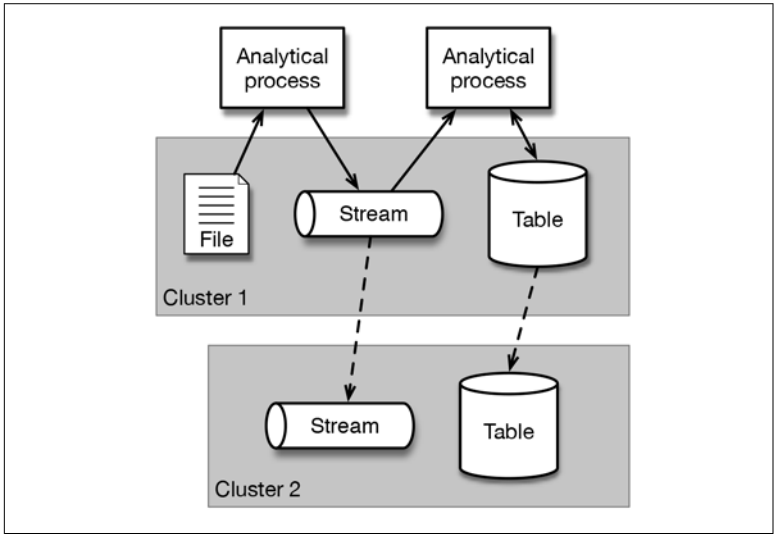


Figure 4-2. The streams and tables in one MapR cluster can be replicated to streams and tables in other clusters to form a coherent data fabric. This replication allows multimaster updates. Either or both clusters could be in an on-premises data center or hosted in a public cloud.

Data can also be moved between clusters in your data fabric using volume mirroring, and mirroring to a remote cluster sets you up for disaster recovery. We describe the details of how mirroring and stream/table replication work later.

You can connect small footprint MapR clusters to the data fabric forming an Internet of Things (IoT) edge. This makes it feasible to collect data from many distributed sources and, if desirable, do processing, aggregation, or data modeling at the edge rather than having to move all data back to a central cluster. *Edge Computing* is one of the basic design patterns that we describe in Chapter 5.

Implications for Streaming

Streaming architecture and streaming microservices require decoupling of data sources and data consumers and thus are based on stream transport technology in the style of Apache Kafka, which includes the MapR streams. (See Chapter 2 and Figure 2-7 for an explanation). Although MapR streams support the open source Kafka API, they are implemented very differently, and that gives MapR streams additional capabilities beyond those of Kafka. Both are useful as connectors between microservices, but MapR's stream replication, ability to handle a huge number of topics, and integration into the MapR file system set it apart from Kafka. Let's see how these capabilities play out for production applications.

For systems based on streaming microservices, MapR streams are handy because each stream bundles an independent set of topics. This avoids inadvertent topic name collision between services, a potential problem with Kafka for which all topics are in a single flat namespace. The scoping of topic names within MapR streams makes it practical, for example, to run multiple rendezvous frameworks for managing different machine learning systems the same MapR cluster without interference even if the frameworks use the same topic names (in their own separate streams).

One of the biggest differences between MapR streams and Kafka is that MapR streams are built into the file system, and that has many implications. For one thing, with MapR, adding streams or adding topics to existing streams (even hundreds of thousands or millions) does not affect the effort to administer the cluster, but adding a new Kafka broker cluster does increase the effort required to administer Kafka. This characteristic of MapR reduces cluster sprawl compared to Kafka, which has difficulty efficiently handling more than about 1,000 topic partitions per broker in the Kafka cluster. For MapR, policies such as stream replication, message time-to-live (TTL) and access control expressions (ACEs) are applied at the stream level for many topics together. These capabilities are important for design patterns such as the *IoT Data Web* and other situations for which consistent control of data from many sources is desirable. Figure 4-3 illustrates an industrial IoT use case inspired by an oil and gas exploration company.

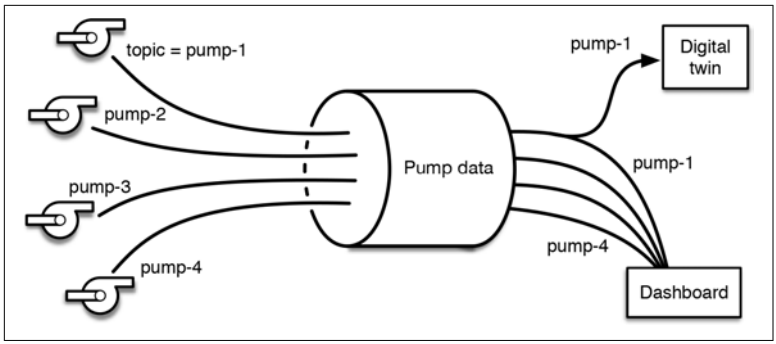


Figure 4-3. In this industrial IoT example, data from sensors on individual pumps are assigned to separate topics bundled together into a MapR stream, shown as a horizontal cylinder in the diagram. In reality, there could be thousands of topics handled by the stream or more. Here, the data for a digital twin is pulled from just one topic, pump-1. Another consumer, a dashboard, subscribes to topics pump-1 through pump-4.

The way that MapR streams are embedded in the MapR file system also means that pieces of topic partitions are distributed across the cluster. With Kafka, partition replicas are restricted to a single machine. That limitation makes a practical difference. With MapR streams it is reasonable to set the TTL to months, years, or even forever, thus providing a long-term, replayable, event-by-event history. This playback capability is needed for use cases that employ the design pattern we call *Streaming System of Record*. We mentioned this approach in the use cases described in Figure 2-6, where a message stream with a long TTL might serve as the system of record for security analytics.

Keep in mind that MapR streams being a first-class part of the file system means that you don't need a separate cluster just for stream transport as you do with Kafka. *Every stream in a MapR cluster is the equivalent of an entire cluster of Kafka brokers*. You can have as many streams on a MapR cluster as you like, each containing hundreds of thousands of topics or more. The more familiar a person is with Kafka, the harder it can be for them to realize that with MapR you don't need a separate cluster for stream transport; you stream messages on the same cluster where your Flink, Spark, or other applications are hosted. This reduces operational complexity by avoiding a network bottleneck. This difference was highlighted in a discussion

about benchmarking Apache Flink on MapR streams that appeared in the book *Introduction to Apache Flink* by Friedman and Tzoumas (O'Reilly, 2016; see Chapter 5).

A final implication of MapR streams being built in to the MapR file system is the role of the data platform working in conjunction with Kubernetes orchestration of containerized applications. Using the same platform across your entire data fabric, you can persist state from containerized applications in any form you like: as streams, files, or tables. This capability avoids the need for a swarm of containerized Kafka clusters, separate file stores, HDFS clusters, and databases.

Now you've seen how you can put to use some of the key capabilities of the MapR platform. In the rest of this chapter, we give an overview of the technology underlying these capabilities to show you how it works.

How This Works: Core MapR Technology

Internally, all data in MapR-FS is stored in database-like structures called *B-trees* that coordinate the storage of data arranged as 8-kiB blocks. Each tree lives on only a single machine, but each one can reference other trees on other machines. By choosing the key and the data in these trees, it's possible to implement an entire file system complete with directories and files. A tree can be a directory if the keys in the tree are file names and the values are references to files. These files are also trees whose keys are offsets and values are references to *chunks*. Each chunk is a tree with offsets for keys, and values are references to actual pages of storage.

These trees can also emulate other interesting kinds of data storage. A tree can become a document database where keys are database keys, and values are encoded rows. Or they can implement a message queue where keys are combinations of topic and message offset.

This technical generality is what allows MapR-FS to contain directories, files, streams, and tables. All of these objects share their underlying implementation in terms of trees. There are a few additional objects that help manage the distributed data, but the core principle is the representation of everything in terms of these primitive B-trees.

Comparison with Hadoop

Some people equate big data with Hadoop, but the needs of big data have migrated significantly over the past few years. Practically speaking, the MapR data platform has several differences relative to Hadoop-based storage using HDFS that change the scope for applications that run on MapR.

These differences include:

- HDFS has one or more special servers known as name nodes that hold all metadata for a cluster. Most commonly, a single name node is used, possibly with a backup (we talk about how name nodes can be federated later). The problem is that this name node is a bottleneck, a scalability limit, and a substantial reliability risk. Name node failure can (and we have heard of cases where it did) result in substantial data loss. In contrast, MapR has no name node; metadata is distributed across the cluster for higher reliability, higher performance, and less chance of bottlenecks.
- MapR-FS supports real-time updates to files, tables, and streams. In contrast, real-time use of HDFS is difficult or impractical. This limitation in HDFS comes about because every write to a file extends the length of the file when the write is committed. This requires a round-trip transaction to the name node, so programs try to commit writes only rarely. In fact, it is common for multiple blocks composed of hundreds of megabytes of data to be flushed all at once.
- HDFS supports only a write-once, read-many model in which files cannot be updated except by appending new data to the end, as soon as the file has been written.
- Files, tables, and streams in the MapR platform have full multi-reader/multiwriter semantics. HDFS allows only single-writer or multiple readers, which means that readers cannot read files while a writer still has them open for writing. This makes real-time processing nearly impossible.
- The content of files stored in MapR can be written in any order. This is required for many legacy systems. HDFS can't do out-of-order writes. Network protocols like NFS often result in out of order writes, as do databases such as Postgres, MySQL, or Sybase.

- The number of objects in a single MapR-FS cluster is practically unlimited; billions or trillions of files are viable. This makes the MapR data platform usable for many object storage applications, as mentioned in the design pattern called *Object Store*. With HDFS, the name node has to keep track of all of the blocks in all of the files in memory. That severely limits the number of blocks (and thus files) to a level several orders of magnitude too small for many applications.
- Recent support for name node federation in HDFS does not change scalability limits in HDFS significantly because scaling with federation increases the level of manual administrative effort super linearly. The federation structure is also visible in the file system names, which causes the structure to leak into service implementations. These issues mean that federation increases scalability by only a small factor at best; supporting 10 billion files would require several hundred primary and backup name nodes.

The simplistic design of HDFS made it much easier to implement originally, but it also made it much harder to work with, except with programs specially designed to accommodate its limitations. Spark and Hadoop MapReduce are two examples of programming frameworks that support this, but almost all conventional databases, for example, cannot run on HDFS even in principle.

Of course, the limitations of HDFS were well understood from the beginning. The design was a reasonable one for the limited original use case. The problem really comes with attempts to use HDFS for different kinds of workloads or where the limits on file count or metadata update rates are important.

Beyond Files: Tables, Streams, Audits, and Object Tiering

As mentioned earlier, MapR-FS supports a number of capabilities not normally found in file systems, distributed or not. These capabilities allow a wide range of applications to be built on a single cluster without needing additional clusters for databases or message streams. These capabilities also allow you to connect multiple clusters together to form a data fabric.

MapR DB Tables

One such extended capability is the ability to store tables in directories anywhere that you can store a file. These tables are accessible using table operations analogously to the way that a file is accessible using file operations. Looked at one way, tables in MapR-FS are an extension of a file system beyond what you can do with files alone, but from a different point of view, these tables could be seen as an extension to the idea of databases to allow tables to live in directories, allowing them to have path names and permissions just like files. Regardless of the perspective you choose, you can access tables in MapR-FS using either the Apache HBase API or using the OJAI API, an open source document-oriented table interface that supports nested data formats such as JSON. The HBase API provides binary access to keys and values, whereas the document-oriented OJAI is more in the style of Mongo and is generally preferred for new applications because of the built-in nested data structure.

Records in MapR tables are associated with a key and are kept in order according to the key. Each table is split into tablets with each tablet containing data corresponding to a particular range of keys. Each tablet is further divided into smaller units called segments that actually do the real work. As data is inserted, these tablets can grow larger than a configurable size. When this is detected, tablets are split by copying segments to a new copy of the tablet. An effort is made to avoid network traffic during this copy but still leave the new tablets on different machines in the cluster. The underlying implementation for MapR tables inherently avoids long delays due to compactions. The result of this and other technical characteristics is much better availability and reliability than Apache HBase, especially under heavy load. You can find an example of the practical impact in the Aadhaar project mentioned in Chapter 1, where the system meets strict latency guarantees.

Tables in MapR can have records describing all changes to be written to a message stream using a changed data capture (CDC) feature. Because message streams in MapR-FS are well ordered, you can use this data to reconstruct a table, possibly in another technology (such as a search engine) or in masked or aggregated form. The use of CDC is described in Chapter 5 in the *Table Transformation and Percolation* design pattern.

You also can configure tables so that all changes are replicated to a remote copy of the table. Table replication is near real time and can use multiple network connections to allow very high replication rates. Within any single cluster, updates to tables are strongly consistent, but replication is bidirectional and subject to delay on network partition with conflicting updates on rows resolved using timestamps.

To support multimaster updates, it is common to devote individual columns or document elements to individual source clusters. You could use this, for example, to support multimaster increments of counters by having a single counter column per cluster in the data fabric. Reading a row gives an atomic view of the most recent value for the counters for each cluster. Adding these together gives a good, if slightly out-of-date, estimate of the total count for all clusters. Local strong consistency means that the column increments will be safe, and because each column comes from only a single cluster, replication is safe. Similar techniques are easily implemented for a variety of similar tasks such as IoT reporting.

Message Streams

In MapR-FS, message streams are first-class objects, as are files and tables. You can write to and read from message streams using the Apache Kafka API, although the implementation of message streams in MapR-FS shares no code with Kafka itself.

Each stream logically contains of a number of topics, themselves divided, Kafka-style, into partitions. Messages within partitions are ordered. Topics in different streams are, however, completely independent. This allows two instances of the same application to run independently using different streams, even though they might use the same topic names. To achieve the same level of isolation while avoiding the risk of topic name collision, two Kafka-based applications would have to use separate Kafka clusters.

Physically, MapR streams are implemented on top of the primitive B-tree structures in MapR by using a combination of topic, partition, and message offset as the key for a batch of messages. The batching of messages allows very high write and read throughput, whereas the unification of the entire stream in a single distributed data structure means that the same low-level mechanisms for distributing table and file operations can be repurposed for streams.

This allows, for instance, all permission, security, encryption, and disaster recovery mechanisms to apply to streams with no special code required to make it so.

The MapR platform itself uses streams internally for a variety of purposes. The table CDC function uses streams to carry table updates. The audit system delivers audit messages using streams. System metrics that record operational volumes and latency summaries are also carried in streams.

You can replicate streams to remote clusters in a similar fashion as tables. The pattern of replication can be bidirectional, many-to-one, or even have loops. For consistency, you should write each topic in a set of stream replicants in only one cluster. Message offsets are preserved across all stream replicas so that messages can be read from different clusters interchangeably once they have arrived.

Auditing

In earlier chapters, we talked about the importance of having fine-grained control over data and applications and that requires knowing what is going on. Transparency into your systems helps with this in situations such as finding bottlenecks such as described at the end of Chapter 2.

With MapR, it is possible to turn on selective levels of audit information for any volumes in a system. With auditing enabled, information about object creation, deletion, update, or reads can be sent to a message stream in an easily parsed JSON format.

You can use audit information in a variety of ways, the most obvious being monitoring of data creation and access. This is very useful when combined with automated anomaly detection in which a predictive model examines access and creation of data for plausibility against historical patterns.

You can use audit information in other ways. For instance, by monitoring audit streams, a metadata extraction program can be notified about new files that must be analyzed or modified files whose metadata needs updating. You could imagine the creation of thumbnails for videos being triggered this way or the extraction of plain texts from PDF documents. Audit streams are also excellent ways to trigger indexing of files for search purposes.

Object Tiering

By default, all of the data in a MapR cluster is triplicated to provide resiliency against hardware failure and maximal performance. Replication applies to file, table, and stream data, as well as directory and volume metadata. It is common, however, that data is used intensively right after it is created, but usage drops off after that. Data that is rarely accessed may still need to be retained for a long time, however, possibly decades. Triplicating data in such cases gives no performance advantage but still costs space. It is important to differentially optimize how data is stored for speed, space, or cost.

Such quiescent data can be converted from triplicated form to a form known as *erasure coding* that takes up less than half the space but which has as good or better ability to survive multiple hardware failures. Conversion is controlled by administrative policies based on age and rate of access to data. As properties of volumes, these policies can vary from volume to volume.

Eventually, it may become desirable to absolutely minimize the cost of storing data to the point that speed of access to the data becomes an almost irrelevant consideration. In such cases, you can copy objects to a very low-cost object store such as Amazon Web Services' Simple Storage Service (Amazon S3) or the Microsoft Azure object store.

In any case, the data will still be accessible using the same path name and same operations as ever, no matter how data is stored. Data can also be recalled to the speed tier at any time if it becomes important to optimize performance again. By preserving the metadata and providing uniform access methods, MapR clusters even allow updates to files, tables, and streams that have been copied to immutable storage like Amazon S3.

CHAPTER 5

Design Patterns

Now that you have a good grounding in the habits that are characteristic of successful large-scale production deployments and an understanding of some of the capabilities to look for in data platform and containerization technologies with which you build your systems, it's useful to see how that comes to life in real-world situations. In this chapter, we present a collection of design patterns you can use across a wide range of use cases. These patterns are not industry specific and may be combined in a variety of ways to address particular business goals. Most important, these design patterns are not theoretical. We base them on what we see customers doing in successful large-scale production systems.

Your challenge is to figure out how you can put these powerful underlying patterns to use in your own situation. As you read this chapter, you may find it useful to skim the first paragraph of each design pattern to find those that relate to your needs.

Internet of Things Data Web

More and more businesses are building elements of intelligence, control, and reporting into physical products. These products can be cars, medical devices, shipping containers, or even kitchen appliances. As they build these features into their products, these businesses all have a common problem of moving data from these products to a central facility and pushing software updates back to the devices. Doing this requires secure transport of data between the things in the field back and an analytics system. One commonly requested

feature of such a system is to build what is known as a *digital shadow* of each thing for informational purposes or diagnostics. Other goals include computing aggregate statistics of product feature usage and distribution of product updates.

The most important constraints in this design pattern are data security, scale and reliability in a difficult and nearly uncontrolled working environment.

Figure 5-1 shows a basic design that is common for this pattern. The process starts at the device, of which there might be hundreds—or hundreds of millions. Each device records status and sensor data for central analysis but also needs to get software and configuration updates. These updates may be targeted to one particular device or class of devices. It is almost universal that the data produced by the device must be kept private and that device data and software updates must be unforgeable.

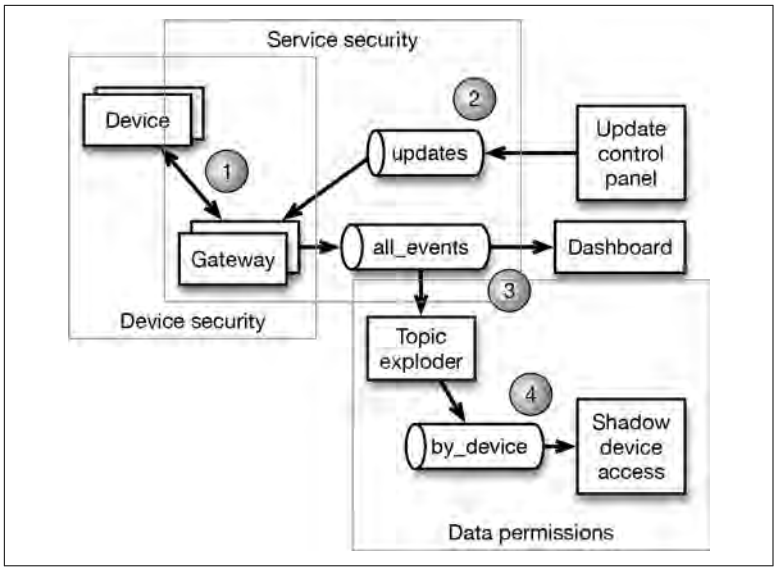


Figure 5-1. In the Internet of Things Data Web design pattern, data flows to and from devices across a link (1) secured by standard public-key cryptographic protocols. Both device and gateway identity are secured by certificates. Updates (2) to configuration and firmware can be sent to devices via this link. Data from devices is forwarded (3) to a comprehensive log for all devices so that it can be rendered into an overall dashboard. This same data is sent to a stream (4) where the topic is the device ID. This allows per device histories known as device shadows to be maintained by direct access to the by_device stream and without a database.

Locking Down the Data Link

Referring to Figure 5-1, step 1 is the data link between the device and a gateway. This first link must satisfy some key security requirements. To ensure security, each device has a unique digital certificate that is signed by a special signing authority controlled by the manufacturer of the device. When the device connects to a data gateway, this certificate is used to authenticate that the device is genuine, and a similar certificate for the gateway is used to validate that the gateway is genuine. These certificates are both signed by the manufacturer so that the device and the gateway can use those signatures to verify each other's authenticity. All data transferred on the link between the device and the gateway is also encrypted using standard protocols such as Transport Layer Security (TLS).

It is important that this link be implemented using a very widely used and standard protocol such as TLS in order to capitalize on the vast experience in defending the protocol against attackers. Using TLS with device and gateway certificates makes it difficult or impossible for attackers to create fake devices, to forge data from an existing device without invasive access to the device itself, or to forge software updates for devices. One key point with the TLS security is to strictly limit the list of signing authorities that are accepted for the certificates for the gateway and devices. The normally expansive list of signing authorities included in web browsers is inappropriate.

Just as there can be many devices, there can be multiple gateways to allow sufficient scale for the number of devices or to allow operation in multiple geographical regions. Typically, devices will connect to gateways using mobile networks, but if internet access is available directly, alternate channels can be used.

Messages that are downloaded from the gateway to the device can be addressed to the device by having multiple topics in the updates stream (item 2 in Figure 5-1). There can be one topic for all devices, a topic for each class of devices, and even a topic for each individual device. Each device will subscribe to all of the streams that might apply to it. Critical messages such as software updates should be independently cryptographically signed to prevent malware injection, even if the message stream security is compromised.

Device identity information must be sufficiently detailed to allow security constraints to be applied at the gateway. This identity information should be embedded in the device certificate to prevent device forgery. Some device information, on the other hand, is variable and should be outside the certificate but still included in messages sent from the device. Data coming from the device should be packaged by the gateway together with the device identity so any downstream consumers can unambiguously identify the device associated with each message. Stream permissions should be used so that only valid gateway processes can send device messages. This helps guarantee that all device messages are well formed and include all pertinent identity information. In Figure 5-1, such messages are in the `log_events` stream.

Dashboards For All or For Each

At the point labeled 3 in Figure 5-1, device messages can be aggregated for dashboards. Such dashboards are very handy for verifying overall system function, especially if device anomaly detection is used to highlight unusual behavior in the population of devices.

If you have a streaming system that can handle a large number of topics, it is a nice step to copy the events to a stream where the topic is equal to the device ID. This corresponds to item 4 in Figure 5-1 and allows you to have a real-time look at any single device very easily with no database overheads. If you are using a system that doesn't like high topic diversity, such as Apache Kafka, you will likely need to use a database for this, but this can be problematic if you have high update rates. Even if each device sends a message only rarely, the total rate can be pretty high. For instance, 100 million devices sending one message every 100 seconds results in a million messages per second. That is pretty easily handled by a well-partitioned streaming system, but it can be a bit more strenuous to handle that kind of update rate with a database. It isn't impossible, but you need to budget for it. Of course, if you have 1,000 devices reporting once per hour, the update rate isn't a big deal, and any technology will work.

The key steps in getting an IoT system of this kind into production is building out the gateway and first streaming layer. Getting security built in to the system correctly from the beginning is critical because almost everything else can be layered in after you are up and running.

Data Warehouse Optimization

One of the most straightforward ways to start with a big data system is to use it to optimize your use of a costly data warehouse. The goal of data warehouse optimization is to make the best use of your data warehouse (or relational database) resources in order to lower costs and keep your data warehouse working efficiently as your data scales up. One way to do this that offers a big payback is to move early Extract, Transform, and Load (ETL) processing and staging tables off of the data warehouse onto a cluster running Apache Spark or Apache Drill for processing. This approach is advantageous because these ETL steps often consume the majority of the processing power of the data warehouse, but they constitute only a much

smaller fraction of the total lines of code. Moreover, the staging tables that are inputs to these ETL steps are typically much larger than subsequent tables, so moving these tables to an external cluster can result in substantial space savings. You gain an advantage by relieving strain on the data warehouse at your current data volumes, plus you'll have set up a highly scalable system that will continue to work in a cost-effective way even as data volumes grow enormously. It makes sense to move each part of the process to the platform on which it works most efficiently. Often, initial data ingestion and ETL makes sense on Spark or Drill, whereas it may make sense to keep critical-path traditional processes on the data warehouse as before.

Figure 5-2 shows the evolution of a data warehouse system as data warehouse optimization proceeds. Initially, in the top panel, we see the traditional view of the process. Data is ingested by copying it into a networked storage system. This data is then imported into the staging tables on the actual data warehouse, and important data is extracted and transformed before loading (ETL) and final processing. This use of staging tables is broadly like what we have seen in actual customer installations. Significantly, the majority of computational resources are typically consumed in the ETL processing, but only a small minority of the code complexity is in this phase.

Data warehouse optimization works by moving staging tables and ETL processing to an external cluster, as shown in panels B and C. This change eliminates the need for a storage system to facilitate the transfer and removes considerable processing and storage load from the data warehouse.

You can change this process by using a data platform to optimize the system, as shown in the bottom two panels of Figure 5-2. In the middle panel of the figure, we see how this optimization works using a MapR cluster. Here, data is copied from the original source to a network-mounted file system exactly as before, but now the storage system has been replaced by a MapR cluster that holds the staging tables. All or some of the ETL process is run on the MapR cluster instead of the data warehouse, and then the work product of the ETL process is bulk loaded into the data warehouse using standard bulk import tools via another networked mount of the MapR cluster.

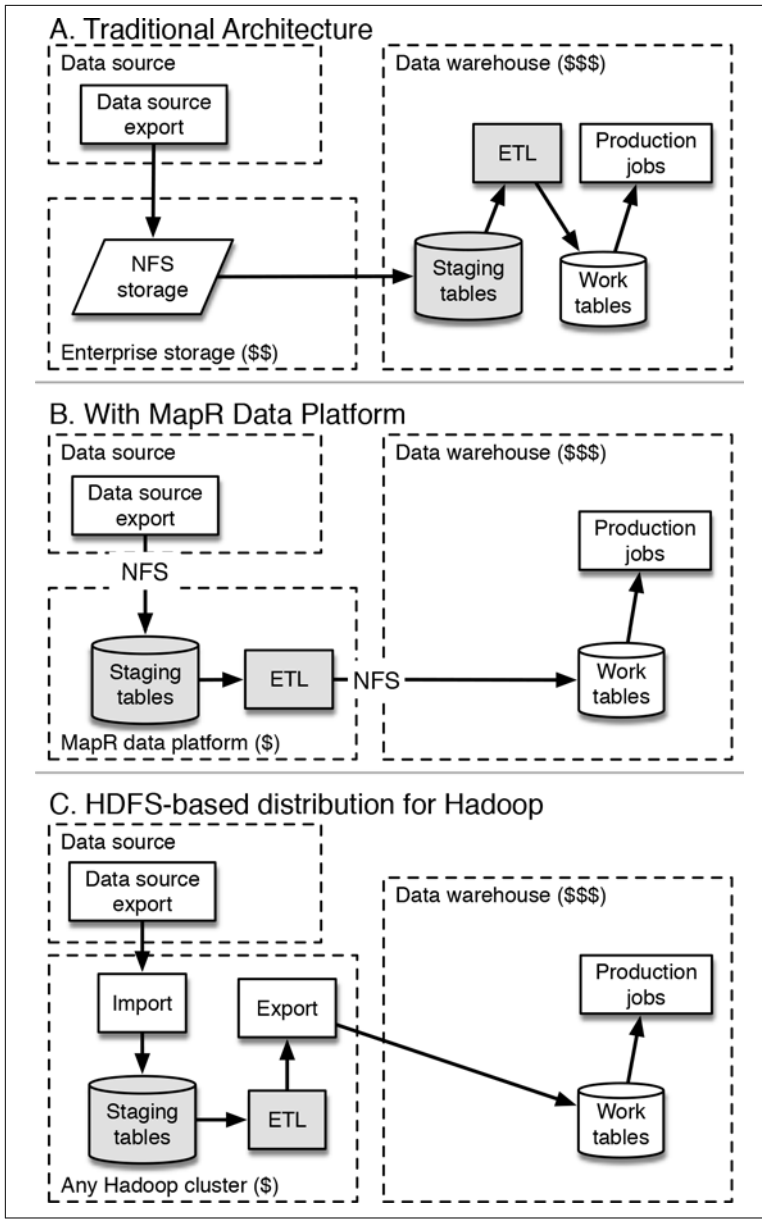


Figure 5-2. The evolution of a data warehouse system.

The lower panel of Figure 5-2 shows an alternative design for a non-MapR platforms. The goal is the same, but there are some variations in how the data is ingested for ETL and how the refined data is

exported to the data warehouse. The biggest difference is the use of specialized connectors to work around the lack of high-performance NFS access to the Hadoop Distributed Files System (HDFS) cluster.

Exactly how much of the ETL process is moved to the cluster depends on the exact trade-off of code size, performance, and natural modularity in the code on the data warehouse. Typically, true extract and transform code runs much more efficiently on a cluster than a data warehouse, while advanced reporting code may well run faster on the data warehouse. These speed trade-offs must be measured empirically by converting sample queries, and the benefits of conversion then need to be balanced against fixed conversion costs and the variable savings of running the process on the external cluster. The final reporting code on the data warehouse is often large enough, complex enough, and difficult enough to test that the trade-off is clearly on the side of leaving it in place, at least initially.

When data warehouse optimization is done with some kind of Hadoop cluster, special-purpose connectors are required, as shown in the bottom panel (c) of Figure 5-2. This increases the complexity of the overall solution and thus increases the management burden and decreases reliability. With the MapR data platform (b), the need for connectors is avoided by using standard bulk export and bulk import utilities on the data source and data warehouse systems respectively together with direct access to the MapR cluster using standard file API's.

The savings in using an external cluster for data warehouse optimization come from the displacement of the external storage and the substantial decrease in table space and ETL processing required on the data warehouse. This is offset slightly by the cost of the external cluster, but the net result is usually a substantial savings. In some cases, these savings are realized by the need for a smaller data warehouse, in others by a delay in having to upgrade or expand an existing data warehouse. In addition to a cost advantage, this style of cluster-based data warehouse optimization keeps all parts of the process running efficiently as your system grows. The move to an external cluster therefore future-proofs your architecture.

Extending to a Data Hub

A significant fraction of MapR customers name the centralization of data—sometimes called a *data hub* or *data lake*—as one of the most

important early use cases. More and more, however, they are looking beyond the data hub to building a data fabric. The term "data hub" is very loosely defined, but the centralization concept is fairly simple and very powerful: by bringing together data from a variety of sources and data types (structured, unstructured, or semi-structured, including nested data) into a centralized storage accessible by many different groups for various types of analysis or export to other systems, you widen the possibilities for what insights you can harvest.

The concept of an enterprise data hub was one of the most commonly cited use cases for Hadoop clusters a few years ago. This represented the fact that Hadoop clusters were becoming less specialized and more of a company-wide resource. There were commonly difficulties, however, in allowing multitenancy because different applications had a tendency to interfere with one another. It was intended that the centralization of data would help break down unwanted data silos. Some forms of analysis, including some valuable approaches to machine learning, are greatly improved by being able to combine insights from more than one data source.

The data hub is a natural evolution from the data warehouse optimization use case, as well. Because the early stages of ETL bring in and save raw data, that same data can be accessed for other purposes, which can lead organically to the construction of a data hub. The relatively low cost of large-scale storage on big data systems relative to dedicated storage devices makes this particularly attractive. In Chapter 6, "Tip #2: Shift Your Thinking" refers to the benefit of delaying some decisions about how you want to process and use data. The data hub fits that idea by building a central source in which data can be used in a variety of ways for many different internal customers, some currently of interest, others to be discovered in the future, as depicted in Figure 5-3.

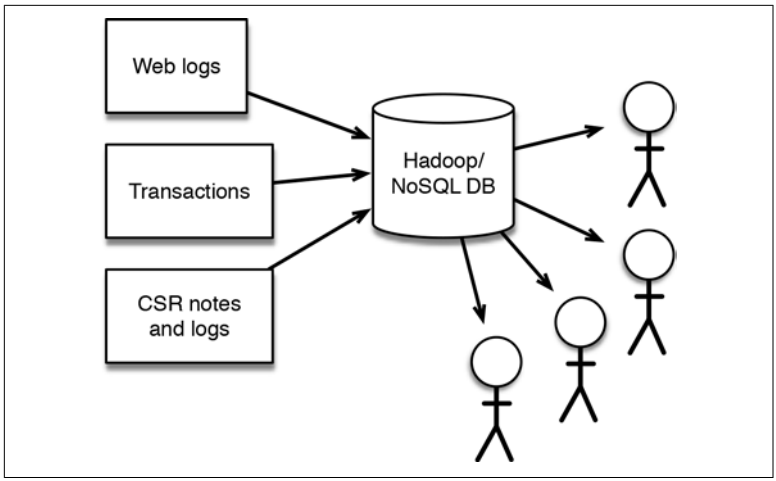


Figure 5-3. A data hub centralizes data from many sources and provides access to many users such as different groups of developers, data scientists, and business analysts. Here the reference database would be NoSQL HBase or MapR-DB. Having easy access to widely varied data makes new ideas and applications inevitable.

A data hub can also support development of a *Customer 360* database, as described in the next section, along with ETL for data warehouse optimization, analysis of log data, processing of streaming data to be visualized on a dashboard, complex anomaly detection, other machine learning projects, and more. The common theme is that these clusters have a lot going on on them in all kinds of ways.

At this point, however, a data hub *per se* is often difficult to bring into production unless you have very specific use cases to deliver. In addition, the early view that Hadoop was the best way to build a data hub has changed somewhat with the emergence of tools like Apache Spark and the increasing need to be able to have a big data system that can integrate direct support for business action (operational data) and the ability to analyze that action (analytics). To do that, you need to have persistent data structures that can support low-latency operation. Streams and tables are often the preferred choice for this.

Stream-Based Global Log Processing

Stream-based global log processing is probably the simplest example of a full-fledged data fabric. Large computing systems are composed of processes that transform data (as in ETL processes) or respond to queries (as with databases or web servers), but all of these programs typically record the actions they take or the anomalous conditions they encounter as so-called log events, which are stored in log files. There's a wealth of insights to be drawn from log file data, but up until recently, much of it has been overlooked and discarded. You can use logs to trigger alerts, monitor the current state of your machines, or to diagnose a problem shortly after it happens, but traditionally, the data has not been saved for more than a short period of time. Common uses of log data include security log analytics for analyzing network and computer intrusions, audience metrics and prediction, and the development of fraud prevention measures.

These log events often record a huge range of observations, such as records of performance or breakage. These records can capture the footprints of intruders or provide a detailed view of a customer's online behavior. Yet when system diagrams are drawn, these logs are rarely shown. In fact, some people refer to log files as "data exhaust" as though they are just expendable and unwanted pollution. Traditionally, logs were deleted shortly after being recorded, but even if retained, they were difficult to process due to their size. In addition, logs are produced on or near the machines that are doing the processing, making it hard to find all the log files that might need processing.

All that is changing. Modern data systems make it possible to store and process log data because it allows cheap and scalable data storage and the ability to process large amounts of data and message streams make it easy to bring back to a central point for analysis.

Traditionally, log processing has been done by arranging an intricate dance between the producers of these logs and the programs that analyze the logs. On the producer side, the tradition has been to "roll" the logs by closing one log file when it reaches an age or size constraint and then start writing to another. As log files are closed, they become available for transport to the analysis program. On the receiving side, this dance was mirrored by methods for signaling exactly which files were to be processed and when programs were to

run. Moreover, the output of one process typically was input to another, so this signaling dance cascaded.

This sort of processing can work well enough when all is well, but havoc reigns when something breaks or even when something is substantially delayed. Questions about whether old results had to be recomputed and which programs needed to be run late or run again were very difficult to answer.

More recently, a new pattern has emerged in which log processing is unified around a message-streaming system. The use of a message-streaming style dramatically simplifies the issues of what to do when and how to redo work that is affected by late-arriving data or system failures. This new pattern of processing has proven dramatically better than the old file-shipping style of log analysis.

Figure 5-4 shows the first step in stream-based global log processing. Here, we show web servers as the software generating logs, but it could as well be any kind of data that produces log files. As log messages are appended to any log file on any machine in the data center, that message is written to a message stream that is on a small cluster. The log messages could be written directly to the log stream instead of to files, but writing the log messages to a local file first minimizes the chance of the write to the stream hanging up the processing.

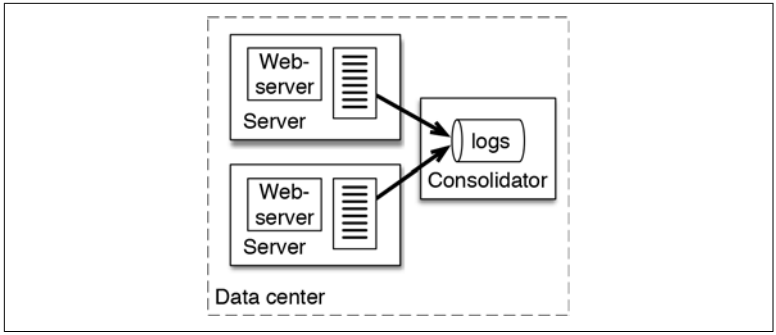


Figure 5-4. Multiple processes in the same data center can send logged events to a log stream on a consolidator machine. These processes are shown as web servers, but they could be any kind of machine or process that produces measurements or event logs. Similarly, we show these processes and the consolidator as if they are in an on premises data center, but they could as easily be in a single group of machines in a cloud such as Amazon Web Services (AWS) or Google Cloud Platform.

The topics used to write these messages to the log stream are commonly composed using a combination of the data center name, the name of the node running the application generating the log message, and possibly a message type. Message types are commonly used to distinguish different kinds of log lines or to differentiate between different kinds of measurements that might be contained in the log lines.

So far, there is little advantage in writing log messages to a stream. There are clear differences, however, because message streams can be easily replicated to a central data center with minimal configuration or setup, as shown in Figure 5-5.

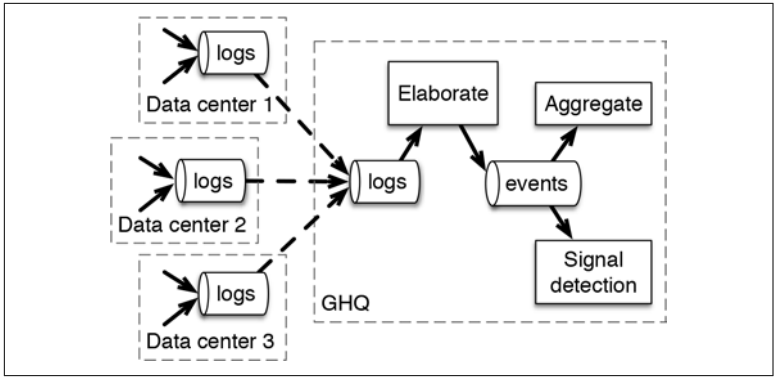


Figure 5-5. You can replicate the log event streams from Figure 5-4 back to a central stream for analysis. With some data platforms, such replication is built in. For other streaming technologies, such as Apache Kafka, the replication has to be implemented. In any case, having non-overlapping topics at the source allows all of the data to funnel into a single stream.

Exactly how the replication of the stream to the central cluster is done varies depending on what kind of software you use to implement the stream. If you use Apache Kafka, you can configure an application called Mirror Maker to copy messages from one broker cluster to another. If you use MapR as a data platform for the streams, you can instead use the native stream replication that functions at the data-platform level itself.

In any case, the central data center (labeled GHQ in Figure 5-5) will have a stream whose contents are the union of all of the messages from any of many processes running in any of many data centers. You can process these messages to aggregate or elaborate them as shown in Figure 5-5. You also can monitor them to detect anomalies or patterns that might indicate some sort of failure.

The important point in this pattern is not just that log messages from all of the processes in all of the data centers can be processed centrally. Instead, it is that you can accomplish the transport of messages to the central point with a very small amount of application code because it is done at using core capabilities of the data platform itself. The result is substantially simpler and more reliable than using systems like Flume because message streams as a data construct provides a more opaque data abstraction.

Edge Computing

Gathering data from multiple locations and then processing it centrally, as with the previous design, *Stream-Based Log Processing*, is well and good, but there are times when you actually need to *do* something out at the edge with the data you are collecting. That locality of computation might be preferable for a number of reasons, possibly because you need very low latency, because your system has to keep working even if the network fails, or just because the total amount of raw information is larger than you really want to transmit back to headquarters. The type of computation can vary in complexity from simple processing or aggregation to running full-scale machine learning models.

This sort of system has proven useful in connecting drilling equipment, large medical instruments, and telecommunications equipment. In each of the cases we have seen, machine learning models, control software, or diagnostics software runs at each of many field locations on very small clusters of one to five small computers. Data moves back to one or more central systems using file mirroring or stream replication where the data is analyzed to produce operational reports as well as new models for the edge clusters. These new models are then transported back down to the edge clusters so that these new and improved models can be used in the field.

Even without machine learning, you can use edge computing to decrease the amount of data retained. This is commonly done by using a relatively simple anomaly detector on the data so that data that is predictable and, well, boring can be discarded while data that is unpredictable or unexpected is retained and transmitted to the central systems for analysis. This is useful in telecommunications systems for which the total amount of diagnostic data acquired is simply too large to return. It is also commonly done as a form of data compression for process control systems, as well.

As an example, in autonomous car development efforts, the raw data stream can be 2 GB/s or more. The advanced machine learning models that are used to understand the environment and control the car can be used to select data segments that are interesting or novel from the standpoint of the control system thus decreasing the total amount of data by roughly a factor of 1,000 without impairing the quality of the model training process.

Figure 5-6 shows an outline of the common structure of all of these systems.

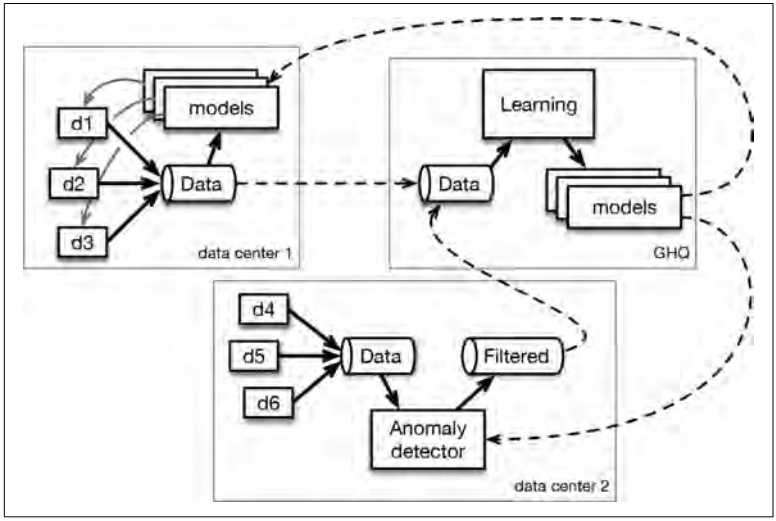


Figure 5-6. Examples of edge computing. In data center 1, models are used to control data sources d1, d2, and d3. The data produced by these data sources is processed locally by control models but is also send to a galactic headquarters (GHQ) cluster to be used to train new models. In data center 2, on the other hand, data from sources d4, d5, and d6 are processed by an anomaly detector and only an interesting subset of the raw data is transported back to the GHQ.

Here, both strategies of edge computing, model execution, and anomaly detection are shown in a single figure. Most systems would not use a mixture of strategies; we show both here simply for illustration purposes.

Customer 360

The goal of a *Customer 360* pattern is to establish a high-performance, consolidated store of complete histories for every customer. When this is done and the entire history for a single customer is viewed as a single consistent list of events, many kinds of processing become enormously simpler. The basic idea is that the nonrelational, highly flexible nature of state-of-the-art big data allows dramatically simpler interpretation of the data without hav-

ing to join hundreds of tables from incompatible snowflake schemas together.

Having a coherent *Customer 360* database makes many analytical tasks much easier. For example, extracting predictive features for event prediction is relatively simple to frame as a computation on the complete history of a single customer. You just need to iterate through a customer's history twice: once to find the events that you want to predict, and once to find out what transactions preceded the event and thus might be predictive. You can also do a variety of ad hoc behavioral queries very easily if you can access all of a customer's history at once. For instance, a mobile operator might know that several cell towers were misbehaving during a certain period of time. With a complete history of each handset, it is possible to find all customers who had dropped calls near the problematic towers at the right time. Reaching out to these customers, possibly with some sort of compensation, could help with potential attrition by showing that you care about the service that they are getting. Both of these examples would likely be too difficult to do relative to the expected value if you have to access a number of databases.

The idealized view of one data store to rule them all often gives way a bit to a structure more like the one shown in Figure 5-7. Here, as in the idealized view, many data sources are concentrated into a central store. These streams are accumulated in a reference database that is keyed by a common customer identifier. The records in these streams are nearly or fully denormalized so that they can cross from one machine to another, maintaining their internal consistency and integrity.

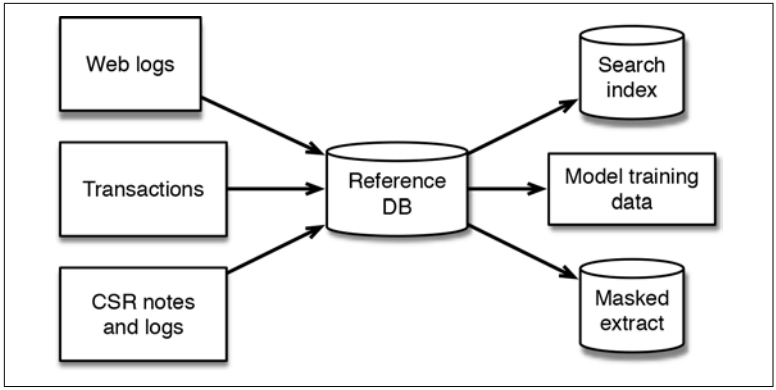


Figure 5-7. In a Customer 360 system, all kinds of information for a single customer are collected into a reference database and kept in a way so that customer histories can be accessed very quickly and with comprehensive retrieval of all desired data. In practice, internal customers of the data have specialized enough needs that it pays to extract views of the reference database into smaller, special-purpose subdatabases.

This reference database is stored in a NoSQL database such as HBase or MapR-DB. The key advantage that these databases offer for an application like this is that good key design will allow all of the records for any single customer to be stored nearly contiguously on disk. This means that a single customer's data can be read very quickly—so fast, indeed, that the inherent expansion in the data caused by denormalization can often be more than compensated by the speed advantage of contiguous reads. In addition, a modern document database can store data whose schema is not known ahead of time. When you are merging data from a number of sources, it helps if you can take data verbatim rather than changing the schema of your reference database every time one of your upstream sources changes.

When building a *Customer 360* database like this, it is likely that you will quickly find that your internal customers of this data will need specialized access to the data. For instance, one common requirement is to be able to search for patterns in the customer histories using a search engine. Search engines like ElasticSearch fill the requirement for search, but they are not generally suitable for use as a primary data store. The easy middle ground is to replicate a filtered extract of the updates to the main database to the search

engine in near real time. You can easily implement this near real-time replication using a *Change Data Capture* (CDC) pattern, as described later in this chapter.

Another important consumer of *Customer 360* data might be a team of machine learning experts building a predictive model. These teams typically prefer no database at all; instead, they prefer to get data in flat files. A common way to deal with this requirement is to run periodic extracts from the main database to get the record set that the team needs into a flat file and then, at least on a MapR system, to use filesystem mirroring to deploy the file or files to the cluster that the machine learning team is using. This method isolates the unpredictable machine load of the machine learning software from the production environment for the reference database. Alternately, programs like `rsync` can incrementally copy data from the master machine to the machine learning environment thus moving much less data than a full copy.

Transactionally correct mirroring is not available on HDFS, however, so a workaround is required on Hadoop systems to allow this type of data delivery. The typical approach used on non-MapR systems is to invoke a MapReduce program called `distcp` to copy the files to the development cluster. Careful management is required to avoid changing the files and directories being copied during the copy, but this alternative approach can make the *Customer 360* use case work well on Hadoop systems.

Another common reason for custom extracts is to comply with security standards. The reference database typically contains sensitive information, possibly in encrypted or masked form. Permission schemes on columns in the reference database are used to enforce role-based limitations on who can access data in the database. Different versions of sensitive information are likely stored in different columns to give flexibility in terms of what data people can see. To secure the sensitive information in the reference database even more stringently, it is common to produce special versions of the reference database with all sensitive data masked or even omitted. Such an extract can be manipulated much more freely than the original and can be hosted on machines with lower security profiles, making management and access easier. Security-cleared extracts like this may be more useful even than the original data for many applications.

Recommendation Engine

The motivation for building a *recommendation engine* generally is to improve customer experience by better understanding what will appeal to particular customers. This is done by an analysis of the customers' preferences communicated through their actions. The improved experience can result in increased sales, longer retention for services, stickier websites, or higher efficiency for marketing spend. In short, happier customers generally result in improved business and customers who find things they like tend to be happier.

Big data systems provide an excellent platform for building and deploying a recommendation system, particularly because good recommendation requires very large datasets to train a model. You can build and deploy a simple but very powerful recommender easily by exploiting search technology running on a data platform. In fact, such a recommender can be much simpler than you might think. Let's take a look at how that works.

The goal of a recommendation engine is to present customers with opportunities that they might not otherwise find by normal browsing and searching. This is done by using historical user behavior for the entire population of users to find patterns that are then cross-referenced to the recent behavior of a specific user. Recommendations can be presented to users explicitly in the form of a list of recommended items or offers but can also be used more subtly to make a user's overall experience more relevant to what they want to do. As an example, a "What's New" page could literally just show new items in reverse chronological order of introduction, or it could show all items introduced recently ordered by a recommendation engine. The latter approach tends to engage users more strongly.

There are two major kinds of recommendation systems that are commonly used in production. One is based on machine learning, typically using an algorithm called *alternating least squares*. This system tends to be more complex and deploying the recommendations requires a specialized recommender engine. The other major kind is based solely on counting how many times different items co-occur in users' activity histories. This result of this co-occurrence analysis is then inserted into a conventional search engine, often a preexisting one for deployment of recommendations in production. The second approach is much simpler than the first and the difference in performance (if any) is typically not very significant.

Recommendation systems of this type work by reading large amounts of historical data and doing a large analysis. This analysis is typically run as a batch or offline process because it can take tens of minutes to hours to run. The output of the analysis consists of so-called recommendation indicators and is transferred to a system that can match these indicators to recent behavior of a specific user to make recommendations in real time as soon as new behavior is observed. You can implement the system that makes these real-time recommendations using pretty much any conventional search engine. This implementation choice is very convenient because search engines are often already being used. Another advantage of this design for recommendation is that the more computationally expensive and time-consuming part of the project—building and training the recommendation model—is done offline, ahead of time, allowing recommendations to be made for users in real time, online, as outlined in Figure 5-8.

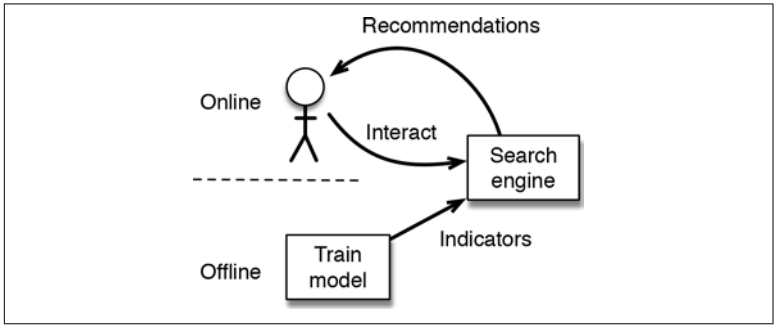


Figure 5-8. The beauty of this two-part design for a recommendation engine is that by dividing the computation of recommendations into two parts, most of the computation can be done offline. That offline computation prepares information called indicators that a standard search engine can use to deliver customized recommendations in real time. The indicators all recommendations for users to be created in real time.

The offline part of the computation is shown in Figure 5-9. User behavioral history is analyzed both for co-occurrence of behavior and for cross-occurrence. In co-occurrence, behaviors are compared like-to-like. An example might be that if you want to recommend songs to a listener, you would analyze previous song-listening behavior. To recommend books for purchase, you would analyze previous book purchases. With cross-occurrence, in contrast, you

would analyze past behavior of one type to make recommendations of a different type. An example would be using past behavior consisting of reading reviews for a product to recommend purchase of that item or others. Using multiple cross-occurrences together with co-occurrence is a valuable way to improve recommender performance.

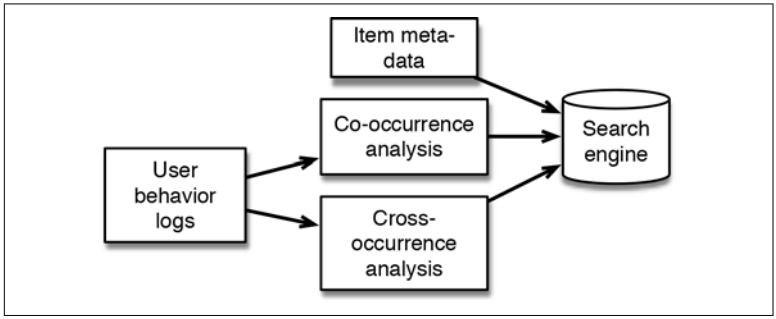


Figure 5-9. A rough structure for the offline portion of a recommendation analysis system. Historical behavior is recorded in user behavior logs. These logs are examined to generate recommendation indicators by doing co-occurrence and cross-occurrence analysis. These indicators are inserted into a search engine together with conventional item metadata that would normally have been in the search engine.

Note that recent research by Schelter and Celebi is outlining ways that the offline part of this computation can be done incrementally, possibly even in strict real time. This would allow a percolation pattern to be used to update indicators in a recommendation system within seconds, even as new patterns of behavior emerge or as new content is introduced.

You can find more information on how recommendation engines are built in our previous book, *Practical Machine Learning: Innovations in Recommendation* (O'Reilly, 2014). That book provides a very short introduction into how to build a recommendation engine and describes the theory and basic practice.

Marketing Optimization

The goal of *Marketing Optimization* is to understand what causes customers to ultimately buy products across both marketing and sales cycles. In very few businesses, the marketing that get customers to engage with a company and the sales process that ensues are rela-

tively simple. An example might be a web-only company that has only a few online marketing programs. In contrast, many businesses are at the other extreme and have a large number of marketing contacts with customers, and the sales process consists of many interactions, as well. For businesses with anything but the simplest sales cycles, determining which actions actually help sell things to customers and which things either don't help or even impede sales is both difficult and very important. In some cases, a company has enough products that just deciding which products to talk about at which times can make a significant difference to the business.

The best practice for this problem is to first establish as complete a history of interactions with customers as possible. Typically, this takes the form of some kind of *Customer 360* database. The simplest marketing optimization system and usually the first one implemented is a recommendation system of some kind. The goal here is to recognize which customers are likely to be in a position where offering a particular product to them is likely to result in a sale or other desired response.

Recommendation systems are very common in online business, but it is unusual to integrate online and offline experiences as inputs to a recommender, and it is unusual to drive recommendations uniformly to both online and offline customer interactions.

The next step in complexity beyond an indicator-based recommendation system is to build per-product sales models. These models can use behavioral features, including recommendation indicators and detailed timing of past transactions and marketing efforts, to attempt to guide the direct sales process by determining which products have the highest propensity to sell if pitched. These models are more complex than the models implicit in a normal recommender, and building them is likely to take a considerable amount of computation, but for complex sales cycles, the results can be very significant. The level of effort to build these models, however, is substantial and should only be undertaken if the product line and sales cycle justify the additional complexity. Simpler search engine–based recommenders are much more appropriate for many companies, including most business-to-consumer companies, both because the sales cycle tends to be simpler, but also because spending time on complex machine learning development is probably worth it only if there is sufficient leverage to reward the development. Extreme sales volume is one way to provide this leverage; high per-unit net profit

is another way. For companies that don't have these factors, it is often much more valuable to spend time adding more logging to user interactions and tuning the user experience to better incorporate recommendations from simpler recommendation systems, instead.

Object Store

The goal of a large *Object Store* is to store a large number of data objects that need to be accessed individually, often by name, but that are not necessarily of interest for wholesale analysis. This has wide utility in a number of businesses. One common use case is in financial companies where all emails, text messages, phone calls, and even recordings of meetings may be recorded this way and retained for potential use in proving regulatory compliance. In such a case, it is important for all of the saved objects to be accessible via standard file APIs so that standard software for, say, voice-to-text conversion can be used. Another use case is for media streaming for which a large number of video or audio files would be stored in different versions in different encodings. Typically, there would be a large number of thumbnail images or other extracted information associated with each media item. Commonly, these related files would be arranged in directories, and it is very useful to have automated ways to replicate data to other clusters in other data centers and to allow web servers to read files directly.

In terms of how the objects in an object store are stored, the simplest possible implementation is to simply store all objects as individual files, one per object, rather than a database. This is particularly true because large objects, often over a megabyte on average, are relatively common. Often the underlying purpose for a large object store is to provide access to media such as videos or audio recordings; sometimes the objects have to do with messaging systems or systems data. Typically, the number of objects is in the tens of millions to tens of billions, and the sizes of the objects involved range from tens of kilobytes to hundreds of megabytes. One common requirement is to make objects accessible on the web. Downtime is typically unacceptable in these systems, and 99.9th percentile response, exclusive of transmission time, must typically be in the tens of milliseconds.

Objects in systems like this often come from a very large number of internet-connected devices. These devices are commonly the primary consumer of these objects, but large scans of objects are common requirements, as well. For instance, if you are building a video serving site on a large object store, it will occasionally be necessary to transcode files into new formats or to extract thumbnail images or run image classifiers. In media systems, the total number of files is typically much larger than the number of individual videos being served because of the requirement to have multiple encoding formats at multiple bit rates along with additional media assets like thumbnail images and preview clips. A good rule of thumb is to expect roughly 100 times more files than you have conceptual objects such as a video. Decoding audio to text is another common use.

Traditionally, large object stores have been built on top of special storage hardware at very high cost, or purpose built using a combination of databases (to store object locations) and conventional file systems at very high cost in operational complexity.

You can use Hadoop systems to create an object store, but they aren't very good at it because they cannot store very many objects. With an HDFS-based system, a completely file-oriented implementation will work only at moderate to small scale due to the file count limit that comes from the basic architecture of HDFS. Depending on the file size distribution, you might be able to use a combination of HBase to store smaller files and HDFS to store larger files. Any HDFS-based solutions will require special software to be written to translate requests into the HDFS API.

With a MapR-based system, in contrast, you can simply use the system as a very large file system because the very large number of files and their potentially large size are not a problem for MapR-FS. Using NFS or a direct POSIX interface to allow direct access by conventional web services also works well with such a solution. Such files can also be accessed using the Amazon Web Services' Simple Storage Service (Amazon S3) API. Any of these options works as a stable production system.

Stream of Events as a System of Record

The use of a message streaming system as a log of business events, and even as the primary system of record, is showing up in more

and more businesses lately. The basic idea is that as the business progresses, you keep a history of high-level business events in a message stream. This approach has the virtue that the stream contains what amounts to a complete and relatively easily understood history of the business at all times in the past. This makes it enormously easier to replay the past, but it may not be obvious how to implement such a system well. The final result can, however, be fairly simple, however, and getting a good streaming backbone in place can really help react to business microcycles better than traditional systems.

The motive for doing this was described in Chapter 1 where we talked about measuring the performance of outgoing emails. In that example, it was clear that we need to do analyses that are not known *a priori* and which require detailed knowledge of events such as sending and receiving emails. In Chapter 3, we had another example with fraud control in which we needed to determine what we knew as of a moment that fraud could still be prevented. Event orientation was important then because we typically don't know ahead of time what patterns of fraud we want to control (the thieves don't copy us on the memos when they invent new schemes).

A publicly available example of an event-based system of record is NASDAQ, which retains all transactions that affect the order book for equities and can perform a complete replay of these events down to the exact millisecond. They even allow public access to this event stream (for a price, of course). Externally, this is advantageous because it allows regulators, brokers, and traders to see exactly how trades were executed, but the fact that a precise history of events is kept has internal benefits, as well. Most notably, the production of a complete historical log of business events makes it much easier to validate that internal processes and software perform exactly as intended. You also can run new versions against historical events to demonstrate that they also behave exactly as specified.

The precise technology used by NASDAQ is, not surprisingly, not described publicly, and implementing a fair exchange system that creates such a log of operations reliably while responding correctly to all transactions within hard real-time limits is difficult. The fact remains, however, that if you create a stream of business events, it is very easy to reuse that stream for all kinds of purposes, and those purposes can have high value to the business. It is also clear that having such a log in terms of actual business events rather than in a

form that exposes the details of how internal systems are actually implemented has substantial advantages, not least because internal system details that are well hidden can be changed without external repercussions.

One of the key aspects of a stream of business events is that you can reconstruct the state of the business at any point in time. This is illustrated in Figure 5-10 in which an incoming stream of business events is used to populate a database, `db1`, which contains a current state of the business. A replica of the event stream is used to populate a second database, `db2`, which is paused at some past time presumably so that the state of the business back then can be interrogated.

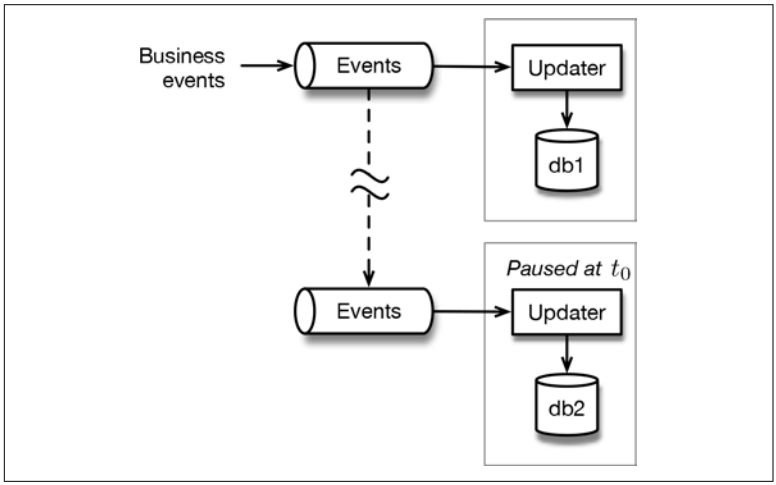


Figure 5-10. You can use business events to populate databases with current estimates of the business state. Pausing the update of the database can give snapshots of historical states. Ordinary databases normally cannot give you this historical view.

But the capabilities of this approach go further. The two databases don't even need to record the same sort of state. One might be simply counting transactions; the other might be keeping account balances.

Figure 5-11 presents more complete picture of the common design idiom for such a streaming system of record that could be used for the email, fraud, or NASDAQ use cases. The idea here is that multiple critical business systems take a responsibility for writing any

business events to a stream (labeled "Events" in the figure) before they are confirmed back to the origin of the event. Moreover, all such systems undertake to only update any databases local to the system from the event stream itself. The event stream becomes the system of record, superior in authority to any of the databases. If one of the critical systems fails, it is assumed that subsequent user requests will be re-routed to any surviving systems.

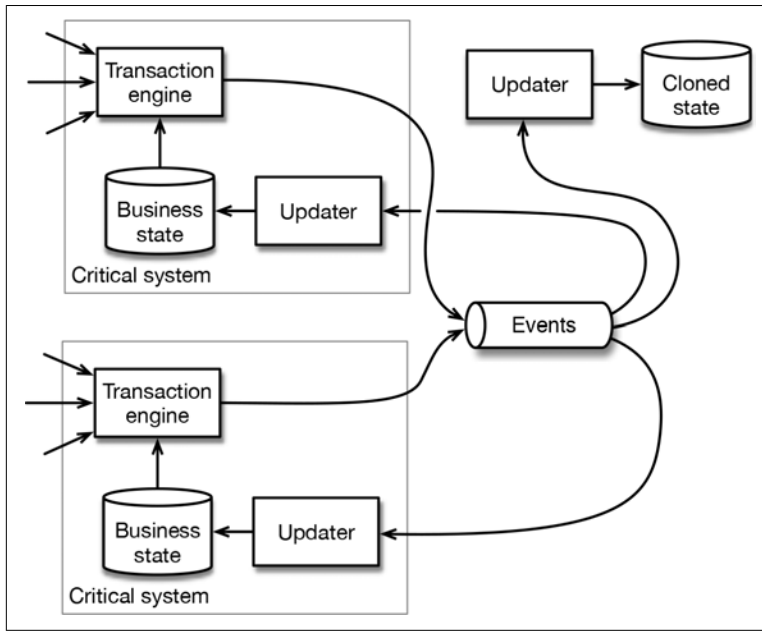


Figure 5-11. Critical subsystems are systems that both use and update state, emitting business events in the process. Each such system sends all such events to a business-wide stream of record that can provide a scalable and inspectable implementation of critical business processes. You need to consider possible race conditions, but you can avoid this potential problem with good design.

For many applications, this kind of design is perfectly adequate just as it stands, say, for handling updates to a user's profile on some web service. The reason is simply that updates for a single user occur at relatively long intervals (typically seconds) and will almost always be routed to a single critical system. Indeed, the total update rate might be so slow that only one such system need be active at a time. Even if some updates were lost due to failure of a communications or a critical system while making an update, that would not be the end of the

world (or the business). Typically, the user would simply repeat the update after being notified of a timeout.

To handle critical applications that cannot have duplicate transactions, such as online banking, it is common to add a unique ID to each transaction that is processed by the system. You should add this ID as far back in the system as possible, preferably right next to the origin of the transaction. In the case that everything goes well, the transaction will be confirmed back to the origin. It is relatively simple to design the system so that transactions are confirmed only when they have succeeded and also been written successfully to the event stream.

For cases in which communications or a critical system fails before, during, or after the processing of a transaction, however, the user might be left in doubt about whether the transaction has succeeded. In such a case, after communications to any critical system is restored, the user can be prompted to inspect the recent history of transactions to verify whether the transaction in question was applied. Of course, not all business systems can be designed to allow this user-driven inspection and corrections process, but if it is possible, this kind of design can be very robust while still very simple.

It should be noted, however, that if more than one critical system accepts transactions at a time, there is a potential for self-contradictory events to be pushed into the events stream. Avoiding this in the general case can be quite difficult, but in many practical situations, you can minimize the impact of this risk. For example, if all events involve just one account, or user profile or package location, you can build a very reliable system this way. In systems for which extreme availability is important (such as with package locations coming from mobile bar code readers), you can build a very reliable system by making sure that all updates are intrinsically ordered, possibly by including an accurate time in each event. That allows events to be buffered at multiple stages, possibly redundantly, but still allows correct integration of delayed messages and events. For cases in which events are generated by devices with integrated GPS receivers, accurate times are easily available which makes this much easier.

Many businesses also have situations in which orders and stock levels need to be coordinated. This is commonly used as an example of a case for which traditional database semantics are required. As a

practical example, consider a situation with customers who want to reserve seats at concerts. Suppose you have business constraints such that you don't want to allow any customer to have more than one order pending, nor to buy more than five seats at any one concert, nor to buy seats at more than four upcoming concerts. You also don't want any seat at any concert to be sold to more than one customer. The problem here is that you have two kinds of read-modify-write operations (one against customers, one against seats) that seem to require an atomic transaction against two kinds of state, one for customers and one for seats. Each kind of state would traditionally be stored in different database tables. Using a database this way can be a classic scaling bottleneck.

In fact, the real-world business process is anything but atomic because it involves not just a database, but customers, as well (and humans don't support ACID transactions). Customers need to see available seats before they select them and need to select seats one by one before they commit to buying them. It takes a bit of time for them to actually complete the transaction. Abandoned purchases are also distressingly common, but these must not lock out other buyers forever. Customarily, the way that this is handled is that customers have an understanding that once they have selected some seats to purchase, they have a short period of time in which to buy them, typically a few minutes. It is also common that there is a rate limiter so that individual users are not allowed to perform actions faster than humanly possible. As such, the system can be reduced to updates to a user's state (commit to buy) and updates to seat states (temporarily commit a seat to a user, purchase a seat, release a seat). We can implement this business process by having one service for managing user state and another for managing seat state, as shown in Figure 5-12. Each of these systems follows the pattern previously shown in Figure 5-11, in which critical systems perform updates with limited extent and each system sends all business events to a shared stream.

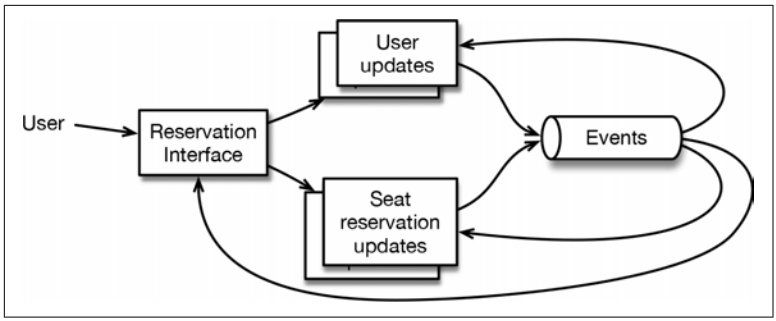


Figure 5-12. Cooperating services can create a good user experience without any multirow atomic updates to both user and seat reservations. This allows the user update service and the seat reservation update service to be sharded by user and seat, respectively, allowing scalability by limiting the scope of any updates. Sharding this way without global atomicity gives good scalability with relatively simple design. The user interface monitors the event stream so that appropriate updates can be made to user visible state via a mechanism such as web sockets.

In the process of selling tickets to users, the reservation interface makes synchronous remote procedure calls (RPCs) to the user update service or to the seat reservation update service. Both services write the events to the event stream just before returning a result to the interface. These events would also be used to update the state inside the service. The states for a user or a seat reservation evolve, as shown in Figure 5-13. The user interface would signal the intent to purchase tickets to the user state interface, which would record the intent internally or deny it if a different purchase session is in progress.

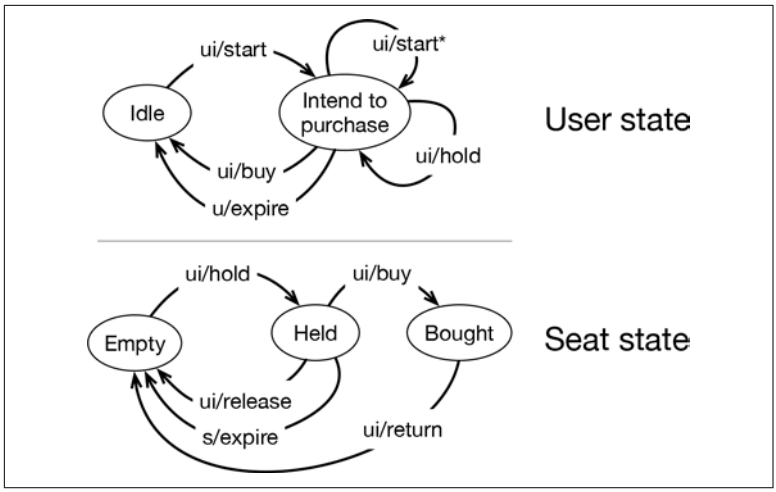


Figure 5-13. The user and seat reservation state machines are driven by synchronous calls from the reservation user interface to the user and seat reservation services. All call results are sent to the event stream so that the state can be updated cleanly by any other service. The asterisk on the ui/start call to the user state indicates that the request would fail if a conflicting session were already in progress and hadn't timed out.

Similar synchronous calls to the seat reservation system would cause a seat be held or purchased or returned. A seat could only be held by a user with a valid session and a seat could only be bought if it has a valid hold that hasn't timed out. The use of holds in this fashion has a dual purpose of matching the desired user experience and avoiding race conditions between users, seats, and the user interface.

Using time-limited reservations in this way is a classic method for factoring apart the updates to user and seat state and can be applied in a variety of situations beyond simple seat reservation systems. The practical effect of factoring transactions apart with reservations is that we can shard databases on a single entity so that no updates involve more than a single row. This allows the system to be scaled in terms of both number of entities (users and seats) and transaction rate.

How Fast Is Fast Enough?

Consider how fast your system really might need to be and don't overbuild to handle speeds that won't matter. Modern streaming and databases are really very fast compared to many real-world workloads. This means that relatively simple systems can handle surprisingly high loads. A single partition of a stream on a MapR system can insert records at a rate of more than 100,000 messages per second per topic. In contrast, NASDAQ has about 10 million trades per day. If these trades all occurred in about 20 minutes this would result in less than a million transactions per second even if you count all bids, offers, and trades. Partitioning by equity makes this transaction flow easy to handle. The moral is that simple streaming designs can have a lot of mileage.

Building a system like this that is reliable in a practical sense often is fairly different from designing a theoretical system. In a practical system, there are often practical bounds on how long an event can be in flight before being considered to have failed or how fast events for a single entity can happen. These bounds all have to do with real, measurable time and most theoretical descriptions of database correctness specifically disallow any consideration of timing. Using time effectively in our systems, however, can give us substantial practical advantages in terms of simplicity and scalability without compromising real-world correctness.

Deduplication of events can make similar use of such timing bounds to limit how many pending events to keep in a deduplication buffer. It should be kept in mind that the goal in a production system is to drive the probability of system failure below an acceptable limit. This typically means that the system is neither designed purely for maximum availability nor absolute consistency; rather, the system is designed to trade-off the consequences of different kinds of failure to find a business-optimal, user-friendly middle ground.

Table Transformation and Percolation

In some cases, it can be impractical to change a working system so that it uses a streaming system of record. In these situations, it can still be very useful to use a related pattern that uses a stream to directly track changes to the content of a table even though the code

making those updates remains unchanged. Another situation in which this is useful is when a table is already being used by a service, but we want to allow access to some version of that table without accidentally messing up the original or imposing any load on the original table. A third situation is when we want to be able to restore any version of a database from any time or replay the evolution of the database from past times. Figure 5-14 shows the basic idea.

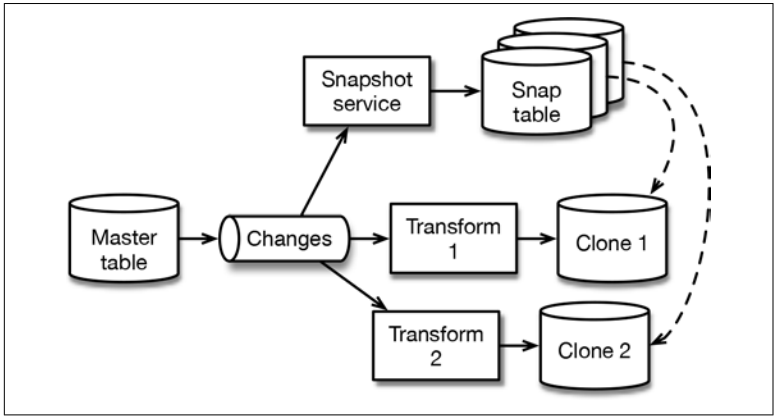


Figure 5-14. With changed data streaming, changes made to a master table are put into a stream (labeled Changes here). You can use this change stream to create snapshots of the table with each snapshot copy corresponding to an offset in the change stream. Clones of the master table can also be created via a transformation. You can even implement these clones by using completely different technology than the master table, but they can still be initialized using snapshots from the snapshot service.

In this pattern, one table is considered the master copy of the data. Changes to this table are inserted into a stream using whatever CDC system is available for the master table. These changes are best stored in a stream so that you can back up and replay them at any time. The first consumer in the figure of these changes is a so-called snapshot service. The point of the snapshot service is to make and save copies of the master table at different points in time. Each of these snapshots includes the stream offset of the last change in the snapshot. This allows you to quickly construct a current copy of the database by copying the snapshot and replaying all changes starting from the offset specified in the snapshot table. This initialization is illustrated in Figure 5-14 with a dashed line.

One of the key advantages of this pattern is not so much to make identical copies of the table; rather, it's to create functional clones of the master table that include transformed or selected values from the master or are stored using a different technology than the master table. For instance, the clone might include only a few columns from the master. Alternatively, the clone might be stored in a very fast technology that is not robust to failures. Such a failure would not be a huge problem because you could easily reconstitute the in-memory form of the table from snapshots. Another option would be to have the clone be heavily indexed to support, say, full-text search.

This kind of design is not as useful as keeping business-level events in a stream. The reason is that table updates tend to be on much lower level of abstraction and, as such, harder to interpret in the context of the business. In many cases, a single business event can cause multiple table updates and the updates for different business events may be intertwined. As such, it is relatively easy to translate from business events to table updates, but it is often quite difficult to reverse the process. On the other hand, the table cloning pattern is architecturally much less invasive, and thus easier to get implemented. Cloning is particularly handy when you have lots of consumers who cannot handle the full detail in the master table and only want an extract containing some rows or some columns or who want a fast materialized aggregate value. In Figure 5-14, for example, Clone 1 and Clone 2 need not have the same data at all.

The idea of selective or transformed clones of a table is particularly useful when you have a very detailed customer 360 system. In such systems the customer tables with all previous transactions are too large and detailed for most internal consumers. To simplify using such a system it can help to provide drastically simplified clones that are automatically kept synchronized with the master table.

It is also possible to adapt the table cloning pattern to implement so-called percolation computations. The basic idea is that the data inserted into the master table is only skeletal in nature. Each time such a skeleton is inserted, however, a consumer of the change stream elaborates the skeleton data and inserts additional information into the original entry, as illustrated in Figure 5-15.

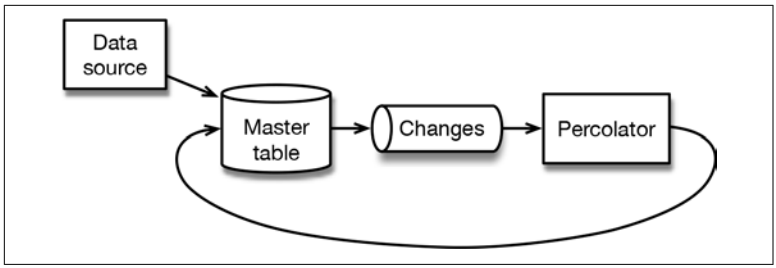


Figure 5-15. With percolation, a change capture stream is used not for replication, but to trigger further changes in a table. This allows the data source to insert (or change) only the barest minimum of data in the master table. Subsequent enrichment or aggregation can be done by the percolator.

Percolation can be very handy when you have a system that inserts data into a table, but you don't want to have that system spend the time to compute the fully elaborated form of that data. With percolation, the basic data source can be kept very simple, and all the work of elaborating the data beyond the original form can be deferred out of the critical path. Percolation can also become a dangerous data pattern if the changes are not idempotent or if the percolated changes cause more percolation. If you find yourself building elaborate patterns of percolation, you probably will be happier with streaming system of record pattern.

CHAPTER 6

Tips and Tricks

So, what does it take to put artificial intelligence (AI), machine learning, or large-scale analytics into production? In part, it depends on decisions you make as you design and implement your workflows and how you set up your cluster(s) to begin with. The technologies available for building these systems are powerful and have huge potential, but we are still discovering ways that we can use them. Whether you are experienced or a newcomer to these technologies, there are key decisions and strategies that can help ensure your success. This chapter offers suggestions that can help you make choices about how to proceed.

The following list is not a comprehensive "how-to" guide, nor is it detailed documentation about large-scale analytical tools. Instead, it's an eclectic mix. We provide technical and strategic tips—some major and some relatively minor or specialized—that are based on what has helped other users we have known to succeed. Some of these tips will be helpful before you begin, whereas others are intended for more seasoned users, to guide choices as you work in development and production settings.

Tip #1: Pick One Thing to Do First

If you work with large volumes of data and need scalability and flexibility, you can use machine learning and advanced analytics in a wide variety of ways to reduce costs, increase revenues, advance your research, and keep you competitive. But adopting these technologies is a big change from conventional computing, and if you

want to be successful quickly, it helps to focus initially on one specific use for this new technology.

Don't expect to know at the start all the different ways that you might eventually want to use machine learning or advanced analytics. Instead, examine your needs (immediate or long-term goals), pick one need that offers a near-term advantage, and begin planning your initial project. As your team becomes familiar with what is feasible and with the ecosystem tools required for your specific goal, you'll be well positioned to try other things as you see new ways in which advanced analytical systems may be useful to you.

There's no single starting point that's best for everyone. In Chapter 5, we describe some common design patterns for machine learning and advanced analytics. Many of those would make reasonable first projects. As you consider where to begin, whether it comes from our list or not, make sure that there is a good match between what you need done and what such a system does well. For your first project, don't think about picking the right tool for the job; be a bit opportunistic and pick the right job for the tool.

By focusing on one specific goal to start with, the learning curve that you face can be a little less steep. For example, for your first project, you might want to pick one with a fairly short development horizon. You can more quickly see whether your planning is correct, determine if your architectural flow is effective, and begin to gain familiarity with what you can actually achieve. This approach can also get you up and running quickly and let you develop the expertise needed to handle the later, larger, and likely more critical projects.

Many if not most of the successful and large-scale data systems today started with a single highly focused project. That first project led in a natural way to the next project and the next one after that. There is a lot of truth in the idea that big data didn't cause these systems to be built, but that instead, building them created big data. As soon as there is a cluster available, you begin to see the possibilities of working with much larger (and new) datasets. As soon as you can build and deploy machine learning models, you begin to see places to use such models everywhere. It is amazing to find out how many people had analytical projects in their hip pocket and how much value can be gained from bringing them to life.

Tip #2: Shift Your Thinking

Think in a different way so that you change the way that you design systems. This idea of changing how you think may be one of the most important bits of advice we can offer to someone moving from a traditional computing environment. This mental transition may sound trivial, but it actually matters a lot if you are to take full advantage of the potential that advanced analytical systems and machine learning offer. Here's why.

The methods and patterns that work best for large-scale computing are very different from the methods and patterns that work in more traditional environments, especially those that involve relational databases and data warehouses. A significant shift in thinking is required for the operations, analytics, and applications development teams. This change is what will let you build systems that make good use of what new data technologies do. It is undeniably very hard to change the assumptions that are deeply ingrained by years of experience working with traditional systems. The flexibility and capabilities of these new systems are a great advantage, but to be fully realized, you must pair them with your own willingness to think in new ways.

The following subsections look at some specific examples of how to do this.

Learn to Delay Decisions

This advice likely feels counterintuitive. We're not advocating procrastination in general—we don't want to encourage bad habits—but it is important to shift your thinking away from the standard idea that you need to completely design and structure how you will format, transform, and analyze data from the start, before you ingest, store, or analyze any of it.

This change in thinking is particularly hard to do if you're used to using relational databases, where the application life cycle of planning, specifying, designing, and implementing can be fairly important and strict. In traditional systems, just how you prepare data—that is, do Extract, Transform, and Load (ETL)—is critically important; you need to choose well before you load, because changing your mind late in the process with a traditional system can be disastrous. This means that with traditional systems such as relational

databases, your early decisions really need to be fully and carefully thought through and locked down.

With modern tools that support more flexible data models, you don't need to be locked into your first decisions. It's not only unnecessary to narrow your options too much from the start, it's also not advised. To do so limits too greatly the valuable insights you can unlock through various means of data exploration.

It's not that you should store data without any regard at all for how you plan to use it. Instead, the new idea here is that the massively lower cost of large-scale data storage and the ability to use a wider variety of data formats means that you can load and use data in relatively raw form, including unstructured or semistructured formats. This is useful because it leaves you open to use it for a known project but also to decide later how else you may want to use the same data. This flexibility is particularly useful because you can use the data for a variety of different projects, some of which you've not yet conceived at the time of data ingestion. The big news is that you're not stuck with your first decisions.

Save More Data

If you come from a traditional data storage background, you're probably used to automatically thinking in terms of extracting, transforming, summarizing, and then discarding the raw data. Even where you run analytics on all incoming data for a particular project, you likely do not save more than a few weeks or months of data because the costs of doing so would quickly become prohibitive.

With modern systems, that changes dramatically. You can benefit by saving much longer time spans of your data because data storage can be orders of magnitude less expensive than before. These longer histories can prove valuable to give you a finer-grained view of operations or for retrospective studies such as forensics. Predictive analytics on larger data samples tends to give you a more accurate result. You don't always know what will be of importance in data at the time it is ingested, and the insights that you can gain from a later perspective will not be possible if the pertinent data has already been discarded.

"Save more data" means saving data for longer time spans, from larger-scale systems, and also from new data sources. Saving data

from more sources also opens the way to data exploration—experimental analysis of data alongside your mainstream needs that may unlock surprising new insights. This data exploration is also a reason for delaying decisions about how to process or downsample data when it is first collected.

Saving data longer can even simplify the basic architecture of system components such as message-queuing systems. Traditional queuing systems worry about deleting messages as soon as the last consumer has acknowledged receipt, but new systems keep messages for a much longer time period and expire them based on size or age. If messages that should be processed in seconds will actually persist for a week, most of the need for fancy acknowledgement mechanisms vanishes. It also becomes easier to replay data. Your architecture may have similar assumptions and similar opportunities.

Rethink How Your Deployment Systems Work

Particularly when you have container-based deployments combined with streaming architecture, you can often have much more flexible deployment systems. A particular point of interest is the ability to deploy more than one version of a model at a time for comparison purposes and to facilitate very simple model roll forward and roll back. If you don't have a system capable of doing this, you might want to consider making that change.

Tip #3: Start Conservatively but Plan to Expand

A good guideline for your initial purchase of a cluster is to start conservatively and expand at a later date. Don't try to commit to finalizing your cluster size from the start; you'll know more six months down the line about how you want to use your system and therefore how large a cluster makes sense than you will at first. If you are using a good data platform, this should be fairly easy and not overly disruptive, even if you expand by very large factors. Some rough planning can be helpful to budget the overall costs of seeing your big data project through to production, but you can make these estimates much better after a bit of experience.

That said, make sure to provide yourself with a reasonably sized cluster for your initial development projects. You need to have suffi-

cient computing power and storage capacity to make your first project a success, so give it adequate resources. Remember also that extra uses for your cluster will pop out of the woodwork almost as soon you get it running. When ideas for new uses arise, be sure to consider whether your initial cluster can handle them or whether it's time to expand. Capacity planning is a key to success.

A common initial cluster configuration as of the writing of this book is 10 to 30 machines, each with 12 to 24 disks and 128 to 256 GB of RAM. It is not uncommon to consider NVMe-based flash storage, especially for clusters doing machine learning. If you need to slim this down initially, go for fewer nodes with good specifications rather than having more nodes that give very poor performance. If you can, go for at least multiple 10 Gb/s network ports, but consider faster networking if you are using flash storage. It is also becoming more and more common to include one or more nodes equipped with Graphics Processing Units (GPUs) for machine learning. It isn't unusual to have a relatively heterogeneous cluster with some GPU +flash machines for compute-intensive jobs combined with machines with large amounts of spinning drives for cold storage. Just remember, you can always add more hardware at any time.

Tip #4 Dealing with Data in Production

We mentioned in Chapter 1 that the view of being in production should extend to data, and that you need to treat data to as a production asset much earlier than you have to treat code that way. This is particularly true because your current (supposedly nonproduction) data may later be incorporated retrospectively by some future project which effectively makes your current data a production asset. That can even happen without you even realizing it. So how can you prevent unwitting dependencies on unreliable data? How do you deal with that possibility for current and future (as yet unknown) projects?

There are no hard-and-fast answers to this problem; it's actually quite hard. But you can do some things to avoid getting too far off the path.

One of the first things that you can do is to make sure that you distinguish between data that is "production" from data that is "preproduction." Then, only allow production processes to depend on production data and allow production data to be written only by

production services. Nonproduction should not have permission to write to any "production" grade data, although it might be allowed to read from that data. To make this easier, it helps to have a data platform that allows administrative control over permissions for entire categories of data. The idea here is that once data has been tainted by being the product of a nonproduction process, it cannot be restored to production grace. This provides the social pressure necessary to make sure that production data is only ever produced by production code, which has sufficient controls to allow the data to be re-created later if a problem is found.

It won't always be possible to maintain this level of production quality purity, but having a permission scheme in place will force an organizational alert when production processes try to depend on preproduction data. That will give you a chance to examine that data and bring the processes that produce that data under necessary control to get stability.

Machine learning is a bit of an exception. The problem is that training data for a model is often not produced by a fully repeatable process and thus isn't production grade. The next best bet is to freeze and preserve the training data itself as it was during training for all production models. This is in addition to version controlling the model building process itself.

Unfortunately, it is more and more common that the training data for a model is more than 1 GB in size or larger. Training sets in the terabyte range and above are becoming more and more common. Version control systems already have problems with objects as large as 100 MB, so conventional version control is not plausible for training data. Data platform snapshots, however, should not be a problem even for petabyte-scale objects. Engineering provenance over models can thus be established by version controlling the code and snapshotting the data.

Tip #5: Monitor for Changes in the World and Your Data

The world changes, and your data will, too. Advanced analytics and machine learning systems almost never deal with an entirely closed and controlled world. Instead, when you deploy such a system there is almost always something that is outside your control. This is par-

ticularly true of large-scale systems, systems that are in long-term production, systems that get information from the internet, and systems that have adversaries such as fraudsters. The data you see in your system is very unlikely to be static. It will likely change due to updated formats, new data, loss of data sources, or enemy (or competitor or fraudster) actions.

Recognizing that this is inevitable, it is important that you be prepared and that you watch for what comes your way. There are many techniques to look for shifts of this sort, but here are two particular techniques that are relatively easy but still provide substantial value. The first method is to look at the shape of your incoming data. You can do this by clustering your input data to find patterns of highly related input features. Later, for each new data point, you find which cluster the new data point is closest to and how far it is from the center of that cluster. This method reduces multidimensional input data into a one-dimensional signal and reduces the problem of monitoring your input data to that of monitoring one-dimensional signals (the distances) and arrival times (when points are added to each cluster). You also can use more advanced kinds of autoencoders, but the key is reducing complex input into a (mathematically) simpler discrepancy score for monitoring. You can find more information about monitoring for change this way in our book *Practical Machine Learning: A New Look at Anomaly Detection* (O'Reilly, 2014).

A second technique involves looking at the distribution of scores that come out of your machine learning models. Because these models learn regularities in your input data, the score distributions are important indicators of what is happening in general in your data and changes in those distributions can be important leading indicators of changes that might cause problems for your system. You can learn more about monitoring machine learning models in our book *Machine Learning Logistics: Model Management in the Real World* (O'Reilly, 2017).

Tip #6: Be Realistic About Hardware and Network Quality

AI, machine learning, and advanced analytics offer huge potential cost savings as well as top-line opportunities, especially as you scale up. But it isn't magic. If you set a bad foundation, these systems can-

not make up for inadequate hardware and setup. If you try to run on a couple of poor-quality machines with a few disks and shaky network connections, you won't see very impressive results.

Be honest with yourself about the quality of your hardware and your network connections. Do you have sufficient disk space? Enough disk bandwidth? Do you have a reasonable balance of cores to disk to network bandwidth? Do you have a reasonable balance of CPU and disk capacity for the scale of data storage and analysis you plan to do? And, perhaps most important of all, how good are your network connections?

A smoothly running cluster will put serious pressure on the disks and network—it's supposed to do so. Make sure each machine can communicate with every other machine at the full bandwidth for your network. Get good-quality switches and be certain that the system is connected properly.

To do this, plan time to test your hardware and network before you install anything, even if you think your systems are working fine. This helps avoid problems and makes it easier to isolate the source of problems if they do arise. If you do not take these preparatory steps and a problem occurs, you won't know whether it is hardware or a software issue that's at fault. Lots of people waste lots of time on this. Trying to build a high-performance cluster with misconfigured network, disk controllers, or memory is so common that we require a hardware audit before installing clusters.

The good news is that we have some pointers to good resources for how to test machines for performance. For more details, see the Appendix at the end of this book.

Tip #7: Explore New Data Formats

Decisions to use new data formats, such as semi-structured or unstructured data, have resulted in some of the most successful advanced data projects that we have seen.These formats may be unfamiliar to you if you've worked mainly with traditional databases, but they can provide substantial benefits by allowing you to "future-proof" your data. Some useful new formats such as Parquet or well-known workhorses like JSON allow nested data with very flexible structure. Parquet is a binary data form that allows efficient

columnar access, and JSON allows the convenience of a human readable form of data, as displayed in Example 6-1.

Example 6-1. Nested data showing a VIN number expanded to show all of the information it contains in more readable form

```
{
   "VIN":"3FAFW33407M000098",
   "manufacturer":"Ford",
   "model": {
     "base": "Ford F-Series, F-350",
     "options": ["Crew Cab", "4WD", "Dual Rear Wheels"]
   },
   "engine":{
     "class": "V6,Essex",
     "displacement": "3.8 L",
     "misc": ["EFI","Gasoline","190hp"]
   },
   "year":2007
}
```

Nested data formats such as JSON (shown in Example 6-1) are very expressive and help future-proof your applications by making data format migration safer and easier. Social media sources and web-oriented APIs such as Twitter streams often use JSON for just this reason.

Nested data provides you with some interesting new options. Think of this analogy: Nested data is like a book. A book is a single thing, but it contains subsets of content at different levels, such as chapters, figure legends, and individual sentences. Nested data can be treated as a unit, but the data at each internal layer can also be used in detail if desired.

Nested data formats such as JSON or Parquet combine flexibility with performance. A key benefit of this flexibility is future-proofing your applications. Old applications will silently ignore new data fields, and new applications can still read old data. Combined with a little bit of discipline, these methods lead to very flexible and robust interfaces. This style of data structure migration was pioneered by Google and has proved very successful in a wide range of companies.

Besides future-proofing, nested data formats let you encapsulate structures. Just as with programming languages, encapsulation allows data to be more understandable and allows you to hide irrele-

vant details in your code. As an example, if you copy the engine data in the VIN structure in Example 6-1, you can be confident that even if the details contained in an engine data structure change, your copy will still work precisely because the details aren't mentioned.

These new formats seem very strange at first if you come from a relational data background, but they quickly become second nature if you give them a try. One of your challenges for success is to encourage your teams to begin to consider unstructured and semi-structured data among their options. From a business perspective, access to semi-structured, unstructured, and nested data formats gives you a chance to reap the benefits of analyzing social data, of combining insights from diverse sources, and reducing development time through more efficient workflows for many projects.

Tip #8: Read Our Other Books (Really!)

We've written several short books published by O'Reilly Media that provide pointers to handy ways to build Hadoop applications for practical machine learning, such as how to do more effective anomaly detection (*Practical Machine Learning: A New Look at Anomaly Detection*), how to build a simple but very powerful recommendation engine (*Practical Machine Learning: Innovations in Recommendation*), how to build high-performance streaming systems (*Streaming Architecture: New Designs Using Apache Kafka and MapR Streams*), and how to manage the logistics involved in the deployment of machine learning systems (*Machine Learning Logistics: Model Management in the Real World*). Each of these short books takes on a single use case and elaborates on the most important aspects of that use case in an approachable way. In our current book, we are doing the opposite, treating many use cases at a considerably lower level of detail. Both high- and low-detail approaches are useful.

So, check those other books out; you may find lots of good tips that fit your project.

Appendix

Additional Resources

These resources will help you to build production artificial intelligence and analytics systems:

- "Big News for Big Data in Kubernetes." Video of talk by Ted Dunning at Berlin Buzzwords, 12 June, 2018; *http://bit.ly/2LWAsEF*
- "What Matters for Data-Intensive Applications in Production." Video of talk by Ellen Friedman at Frankfurt Convergence June 2018; *https://youtu.be/Cr39GFNMFm8*
- "How to Manage a DataOps Team." Article by Ellen Friedman in *RTInsights*, June 2018; *http://bit.ly/2vHhlDK*
- "Getting Started with MapR Streams." Blog by Tugdual Grall; *http://bit.ly/2OLLVVw*
- Cluster validation scripts *http://bit.ly/2KCBznw*

Selected O'Reilly Publications by Ted Dunning and Ellen Friedman

- *Machine Learning Logistics: Model Management in the Real World* (September 2017)
- *Data Where You Want It: Geo-Distribution of Big Data and Analytics* (March 2017)

- *Streaming Architecture: New Designs Using Apache Kafka and MapR Streams* (March 2016)
- *Sharing Big Data Safely: Managing Data Security* (September 2015)
- *Real-World Hadoop* (January 2015)
- *Time Series Databases: New Ways to Store and Access Data* (October 2014)
- *Practical Machine Learning: A New Look at Anomaly Detection* (June 2014)
- *Practical Machine Learning: Innovations in Recommendation* (January 2014)

O'Reilly Publication by Ellen Friedman and Kostas Tzoumas

- *Introduction to Apache Flink: Stream Processing for Real Time and Beyond* (September 2016)

About the Authors

Ted Dunning is chief application architect at MapR Technologies. He's also a board member for the Apache Software Foundation, a PMC member and committer of the Apache Mahout, Apache Zookeeper, and Apache Drill projects, and a mentor for various incubator projects. Ted has years of experience with machine learning and other big data solutions across a range of sectors.

Ellen Friedman is principal technologist for MapR Technologies. Ellen is a committer on the Apache Drill and Apache Mahout projects and coauthor of a number of books on computer science, including *Machine Learning Logistics*, *Streaming Architecture*, the *Practical Machine Learning* series, and *Introduction to Apache Flink*.