Big Data

Principles and best practices of scalable real-time data systems

Nathan Marz
James Warren

6$0 3/(&+$37(5

MANNING

Big Data

by Nathan Marz
and James Warren

Chapter 1

brief contents

1 A new paradigm for Big Data

This chapter covers

- Typical problems encountered when scaling a traditional database
- Why NoSQL is not a panacea
- Thinking about Big Data systems from first principles
- Landscape of Big Data tools
- Introducing SuperWebAnalytics.com

In the past decade the amount of data being created has skyrocketed. More than 30,000 gigabytes of data are generated *every second*, and the rate of data creation is only accelerating.

The data we deal with is diverse. Users create content like blog posts, tweets, social network interactions, and photos. Servers continuously log messages about what they're doing. Scientists create detailed measurements of the world around us. The internet, the ultimate source of data, is almost incomprehensibly large.

This astonishing growth in data has profoundly affected businesses. Traditional database systems, such as relational databases, have been pushed to the limit. In an

increasing number of cases these systems are breaking under the pressures of "Big Data." Traditional systems, and the data management techniques associated with them, have failed to scale to Big Data.

To tackle the challenges of Big Data, a new breed of technologies has emerged. Many of these new technologies have been grouped under the term *NoSQL*. In some ways, these new technologies are more complex than traditional databases, and in other ways they're simpler. These systems can scale to vastly larger sets of data, but using these technologies effectively requires a fundamentally new set of techniques. They aren't one-size-fits-all solutions.

Many of these Big Data systems were pioneered by Google, including distributed filesystems, the MapReduce computation framework, and distributed locking services. Another notable pioneer in the space was Amazon, which created an innovative distributed key/value store called Dynamo. The open source community responded in the years following with Hadoop, HBase, MongoDB, Cassandra, RabbitMQ, and countless other projects.

This book is about complexity as much as it is about scalability. In order to meet the challenges of Big Data, we'll rethink data systems from the ground up. You'll discover that some of the most basic ways people manage data in traditional systems like relational database management systems (RDBMSs) are too complex for Big Data systems. The simpler, alternative approach is the new paradigm for Big Data that you'll explore. We have dubbed this approach the *Lambda Architecture*.

In this first chapter, you'll explore the "Big Data problem" and why a new paradigm for Big Data is needed. You'll see the perils of some of the traditional techniques for scaling and discover some deep flaws in the traditional way of building data systems. By starting from first principles of data systems, we'll formulate a different way to build data systems that avoids the complexity of traditional techniques. You'll take a look at how recent trends in technology encourage the use of new kinds of systems, and finally you'll take a look at an example Big Data system that we'll build throughout this book to illustrate the key concepts.

1.1 How this book is structured

You should think of this book as primarily a theory book, focusing on how to approach building a solution to any Big Data problem. The principles you'll learn hold true regardless of the tooling in the current landscape, and you can use these principles to rigorously choose what tools are appropriate for your application.

This book is not a survey of database, computation, and other related technologies. Although you'll learn how to use many of these tools throughout this book, such as Hadoop, Cassandra, Storm, and Thrift, the goal of this book is not to learn those tools as an end in themselves. Rather, the tools are a means of learning the underlying principles of architecting robust and scalable data systems. Doing an involved compare-and-contrast between the tools would not do you justice, as that just distracts from learning the underlying principles. Put another way, you're going to learn how to fish, not just how to use a particular fishing rod.

In that vein, we have structured the book into *theory* and *illustration* chapters. You can read just the theory chapters and gain a full understanding of how to build Big Data systems—but we think the process of mapping that theory onto specific tools in the illustration chapters will give you a richer, more nuanced understanding of the material.

Don't be fooled by the names though—the theory chapters are very much example-driven. The overarching example in the book—SuperWebAnalytics.com—is used in both the theory and illustration chapters. In the theory chapters you'll see the algorithms, index designs, and architecture for SuperWebAnalytics.com. The illustration chapters will take those designs and map them onto functioning code with specific tools.

1.2 Scaling with a traditional database

Let's begin our exploration of Big Data by starting from where many developers start: hitting the limits of traditional database technologies.

Suppose your boss asks you to build a simple web analytics application. The application should track the number of pageviews for any URL a customer wishes to track. The customer's web page pings the application's web server with its URL every time a pageview is received. Additionally, the application should be able to tell you at any point what the top 100 URLs are by number of pageviews.

You start with a traditional relational schema for the pageviews that looks something like figure 1.1. Your back end consists of an RDBMS with a table of that schema and a web server. Whenever someone loads a web page being tracked by your application, the web page pings your web server with the pageview, and your web server increments the corresponding row in the database.

Column name	Type
id	integer
user_id	integer
url	varchar(255)
pageviews	bigint

Figure 1.1 Relational schema for simple analytics application

Let's see what problems emerge as you evolve the application. As you're about to see, you'll run into problems with both scalability and complexity.

1.2.1 Scaling with a queue

The web analytics product is a huge success, and traffic to your application is growing like wildfire. Your company throws a big party, but in the middle of the celebration you start getting lots of emails from your monitoring system. They all say the same thing: "Timeout error on inserting to the database."

You look at the logs and the problem is obvious. The database can't keep up with the load, so write requests to increment pageviews are timing out.

You need to do something to fix the problem, and you need to do something quickly. You realize that it's wasteful to only perform a single increment at a time to the database. It can be more efficient if you batch many increments in a single request. So you re-architect your back end to make this possible.

Instead of having the web server hit the database directly, you insert a queue between the web server and the database. Whenever you receive a new pageview, that event is added to the queue. You then create a worker process that reads 100 events at a time off the queue, and batches them into a single database update. This is illustrated in figure 1.2.

This scheme works well, and it resolves the timeout issues you were getting. It even has the added bonus that if the database ever gets overloaded again, the queue will just get bigger instead of timing out to the web server and potentially losing data.

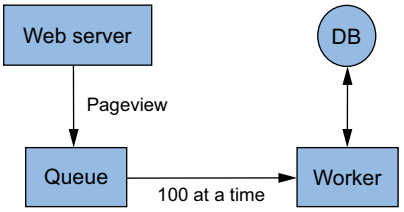

Figure 1.2 Batching updates with queue and worker

1.2.2 Scaling by sharding the database

Unfortunately, adding a queue and doing batch updates was only a band-aid for the scaling problem. Your application continues to get more and more popular, and again the database gets overloaded. Your worker can't keep up with the writes, so you try adding more workers to parallelize the updates. Unfortunately that doesn't help; the database is clearly the bottleneck.

You do some Google searches for how to scale a write-heavy relational database. You find that the best approach is to use multiple database servers and spread the table across all the servers. Each server will have a subset of the data for the table. This is known as *horizontal partitioning* or *sharding*. This technique spreads the write load across multiple machines.

The sharding technique you use is to choose the shard for each key by taking the hash of the key modded by the number of shards. Mapping keys to shards using a hash function causes the keys to be uniformly distributed across the shards. You write a script to map over all the rows in your single database instance, and split the data into four shards. It takes a while to run, so you turn off the worker that increments pageviews to let it finish. Otherwise you'd lose increments during the transition.

Finally, all of your application code needs to know how to find the shard for each key. So you wrap a library around your database-handling code that reads the number of shards from a configuration file, and you redeploy all of your application code. You have to modify your top-100-URLs query to get the top 100 URLs from each shard and merge those together for the global top 100 URLs.

As the application gets more and more popular, you keep having to reshard the database into more shards to keep up with the write load. Each time gets more and more painful because there's so much more work to coordinate. And you can't just run one script to do the resharding, as that would be too slow. You have to do all the resharding in parallel and manage many active worker scripts at once. You forget to update the application code with the new number of shards, and it causes many of the increments to be written to the wrong shards. So you have to write a one-off script to manually go through the data and move whatever was misplaced.

1.2.3 Fault-tolerance issues begin

Eventually you have so many shards that it becomes a not-infrequent occurrence for the disk on one of the database machines to go bad. That portion of the data is unavailable while that machine is down. You do a couple of things to address this:

- You update your queue/worker system to put increments for unavailable shards on a separate "pending" queue that you attempt to flush once every five minutes.
- You use the database's replication capabilities to add a slave to each shard so you have a backup in case the master goes down. You don't write to the slave, but at least customers can still view the stats in the application.

You think to yourself, "In the early days I spent my time building new features for customers. Now it seems I'm spending all my time just dealing with problems reading and writing the data."

1.2.4 Corruption issues

While working on the queue/worker code, you accidentally deploy a bug to production that increments the number of pageviews by two, instead of by one, for every URL. You don't notice until 24 hours later, but by then the damage is done. Your weekly backups don't help because there's no way of knowing which data got corrupted. After all this work trying to make your system scalable and tolerant of machine failures, your system has no resilience to a human making a mistake. And if there's one guarantee in software, it's that bugs inevitably make it to production, no matter how hard you try to prevent it.

1.2.5 What went wrong?

As the simple web analytics application evolved, the system continued to get more and more complex: queues, shards, replicas, resharding scripts, and so on. Developing applications on the data requires a lot more than just knowing the database schema. Your code needs to know how to talk to the right shards, and if you make a mistake, there's nothing preventing you from reading from or writing to the wrong shard.

One problem is that your database is not self-aware of its distributed nature, so it can't help you deal with shards, replication, and distributed queries. All that complexity got pushed to you both in operating the database and developing the application code.

But the worst problem is that the system is not engineered for human mistakes. Quite the opposite, actually: the system keeps getting more and more complex, making it more and more likely that a mistake will be made. Mistakes in software are inevitable, and if you're not engineering for it, you might as well be writing scripts that randomly corrupt data. Backups are not enough; the system must be carefully thought out to limit the damage a human mistake can cause. Human-fault tolerance is not optional. It's essential, especially when Big Data adds so many more complexities to building applications.

1.2.6 How will Big Data techniques help?

The Big Data techniques you're going to learn will address these scalability and complexity issues in a dramatic fashion. First of all, the databases and computation systems you use for Big Data are aware of their distributed nature. So things like sharding and replication are handled for you. You'll never get into a situation where you accidentally query the wrong shard, because that logic is internalized in the database. When it comes to scaling, you'll just add nodes, and the systems will automatically rebalance onto the new nodes.

Another core technique you'll learn about is making your data immutable. Instead of storing the pageview counts as your core dataset, which you continuously mutate as new pageviews come in, you store the raw pageview information. That raw pageview information is never modified. So when you make a mistake, you might write bad data, but at least you won't destroy good data. This is a much stronger human-fault tolerance guarantee than in a traditional system based on mutation. With traditional databases, you'd be wary of using immutable data because of how fast such a dataset would grow. But because Big Data techniques can scale to so much data, you have the ability to design systems in different ways.

1.3 NoSQL is not a panacea

The past decade has seen a huge amount of innovation in scalable data systems. These include large-scale computation systems like Hadoop and databases such as Cassandra and Riak. These systems can handle very large amounts of data, but with serious trade-offs.

Hadoop, for example, can parallelize large-scale batch computations on very large amounts of data, but the computations have high latency. You don't use Hadoop for anything where you need low-latency results.

NoSQL databases like Cassandra achieve their scalability by offering you a much more limited data model than you're used to with something like SQL. Squeezing your application into these limited data models can be very complex. And because the databases are mutable, they're not human-fault tolerant.

These tools on their own are not a panacea. But when intelligently used in conjunction with one another, you can produce scalable systems for arbitrary data problems with human-fault tolerance and a minimum of complexity. This is the Lambda Architecture you'll learn throughout the book.

1.4 First principles

To figure out how to properly build data systems, you must go back to first principles. At the most fundamental level, what does a data system do?

Let's start with an intuitive definition: *A data system answers questions based on information that was acquired in the past up to the present.* So a social network profile answers questions like "What is this person's name?" and "How many friends does this person have?" A bank account web page answers questions like "What is my current balance?" and "What transactions have occurred on my account recently?"

Data systems don't just memorize and regurgitate information. They combine bits and pieces together to produce their answers. A bank account balance, for example, is based on combining the information about all the transactions on the account.

Another crucial observation is that not all bits of information are equal. Some information is derived from other pieces of information. A bank account balance is derived from a transaction history. A friend count is derived from a friend list, and a friend list is derived from all the times a user added and removed friends from their profile.

When you keep tracing back where information is derived from, you eventually end up at information that's not derived from anything. This is the rawest information you have: information you hold to be true simply because it exists. Let's call this information *data*.

You may have a different conception of what the word *data* means. Data is often used interchangeably with the word *information*. But for the remainder of this book, when we use the word *data*, we're referring to that special information from which everything else is derived.

If a data system answers questions by looking at past data, then the most general-purpose data system answers questions by looking at the *entire* dataset. So the most general-purpose definition we can give for a data system is the following:

```
query = function(all data)
```

Anything you could ever imagine doing with data can be expressed as a function that takes in all the data you have as input. Remember this equation, because it's the crux of everything you'll learn. We'll refer to this equation over and over.

The Lambda Architecture provides a general-purpose approach to implementing an arbitrary function on an arbitrary dataset and having the function return its results with low latency. That doesn't mean you'll always use the exact same technologies every time you implement a data system. The specific technologies you use might change depending on your requirements. But the Lambda Architecture defines a consistent approach to choosing those technologies and to wiring them together to meet your requirements.

Let's now discuss the properties a data system must exhibit.

1.5 Desired properties of a Big Data system

The properties you should strive for in Big Data systems are as much about complexity as they are about scalability. Not only must a Big Data system perform well and be resource-efficient, it must be easy to reason about as well. Let's go over each property one by one.

1.5.1 Robustness and fault tolerance

Building systems that "do the right thing" is difficult in the face of the challenges of distributed systems. Systems need to behave correctly despite machines going down randomly, the complex semantics of consistency in distributed databases, duplicated data, concurrency, and more. These challenges make it difficult even to reason about

what a system is doing. Part of making a Big Data system robust is avoiding these complexities so that you can easily reason about the system.

As discussed before, it's imperative for systems to be *human-fault tolerant*. This is an oft-overlooked property of systems that we're not going to ignore. In a production system, it's inevitable that someone will make a mistake sometime, such as by deploying incorrect code that corrupts values in a database. If you build immutability and recomputation into the core of a Big Data system, the system will be innately resilient to human error by providing a clear and simple mechanism for recovery. This is described in depth in chapters 2 through 7.

1.5.2 Low latency reads and updates

The vast majority of applications require reads to be satisfied with very low latency, typically between a few milliseconds to a few hundred milliseconds. On the other hand, the update latency requirements vary a great deal between applications. Some applications require updates to propagate immediately, but in other applications a latency of a few hours is fine. Regardless, you need to be able to achieve low latency updates *when you need them* in your Big Data systems. More importantly, you need to be able to achieve low latency reads and updates without compromising the robustness of the system. You'll learn how to achieve low latency updates in the discussion of the speed layer, starting in chapter 12.

1.5.3 Scalability

Scalability is the ability to maintain performance in the face of increasing data or load by adding resources to the system. The Lambda Architecture is horizontally scalable across all layers of the system stack: scaling is accomplished by adding more machines.

1.5.4 Generalization

A general system can support a wide range of applications. Indeed, this book wouldn't be very useful if it didn't generalize to a wide range of applications! Because the Lambda Architecture is based on functions of all data, it generalizes to all applications, whether financial management systems, social media analytics, scientific applications, social networking, or anything else.

1.5.5 Extensibility

You don't want to have to reinvent the wheel each time you add a related feature or make a change to how your system works. Extensible systems allow functionality to be added with a minimal development cost.

Oftentimes a new feature or a change to an existing feature requires a migration of old data into a new format. Part of making a system extensible is making it easy to do large-scale migrations. Being able to do big migrations quickly and easily is core to the approach you'll learn.

1.5.6 Ad hoc queries

Being able to do ad hoc queries on your data is extremely important. Nearly every large dataset has unanticipated value within it. Being able to mine a dataset arbitrarily

gives opportunities for business optimization and new applications. Ultimately, you can't discover interesting things to do with your data unless you can ask arbitrary questions of it. You'll learn how to do ad hoc queries in chapters 6 and 7 when we discuss batch processing.

1.5.7 Minimal maintenance

Maintenance is a tax on developers. Maintenance is the work required to keep a system running smoothly. This includes anticipating when to add machines to scale, keeping processes up and running, and debugging anything that goes wrong in production.

An important part of minimizing maintenance is choosing components that have as little *implementation complexity* as possible. You want to rely on components that have simple mechanisms underlying them. In particular, distributed databases tend to have very complicated internals. The more complex a system, the more likely something will go wrong, and the more you need to understand about the system to debug and tune it.

You combat implementation complexity by relying on simple algorithms and simple components. A trick employed in the Lambda Architecture is to push complexity out of the core components and into pieces of the system whose outputs are discardable after a few hours. The most complex components used, like read/write distributed databases, are in this layer where outputs are eventually discardable. We'll discuss this technique in depth when we discuss the speed layer in chapter 12.

1.5.8 Debuggability

A Big Data system must provide the information necessary to debug the system when things go wrong. The key is to be able to trace, for each value in the system, exactly what caused it to have that value.

"Debuggability" is accomplished in the Lambda Architecture through the functional nature of the batch layer and by preferring to use recomputation algorithms when possible.

Achieving all these properties together in one system may seem like a daunting challenge. But by starting from first principles, as the Lambda Architecture does, these properties emerge naturally from the resulting system design.

Before diving into the Lambda Architecture, let's take a look at more traditional architectures—characterized by a reliance on incremental computation—and at why they're unable to satisfy many of these properties.

1.6 The problems with fully incremental architectures

At the highest level, traditional architectures look like figure 1.3. What characterizes these architectures is the use of read/write databases and maintaining the state in those databases incrementally as new data is seen. For example, an incremental approach to counting pageviews would be to process a new pageview by adding one to the counter for its URL. This characterization of architectures is a

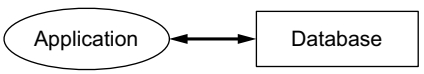

Figure 1.3 Fully incremental architecture

lot more fundamental than just relational versus non-relational—in fact, the vast majority of both relational and non-relational database deployments are done as fully incremental architectures. This has been true for many decades.

It's worth emphasizing that fully incremental architectures are so widespread that many people don't realize it's possible to avoid their problems with a different architecture. These are great examples of *familiar complexity*—complexity that's so ingrained, you don't even think to find a way to avoid it.

The problems with fully incremental architectures are significant. We'll begin our exploration of this topic by looking at the general complexities brought on by any fully incremental architecture. Then we'll look at two contrasting solutions for the same problem: one using the best possible fully incremental solution, and one using a Lambda Architecture. You'll see that the fully incremental version is significantly worse in every respect.

1.6.1 Operational complexity

There are many complexities inherent in fully incremental architectures that create difficulties in operating production infrastructure. Here we'll focus on one: the need for read/write databases to perform online compaction, and what you have to do operationally to keep things running smoothly.

In a read/write database, as a disk index is incrementally added to and modified, parts of the index become unused. These unused parts take up space and eventually need to be reclaimed to prevent the disk from filling up. Reclaiming space as soon as it becomes unused is too expensive, so the space is occasionally reclaimed in bulk in a process called *compaction*.

Compaction is an intensive operation. The server places substantially higher demand on the CPU and disks during compaction, which dramatically lowers the performance of that machine during that time period. Databases such as HBase and Cassandra are well-known for requiring careful configuration and management to avoid problems or server lockups during compaction. The performance loss during compaction is a complexity that can even cause cascading failure—if too many machines compact at the same time, the load they were supporting will have to be handled by other machines in the cluster. This can potentially overload the rest of your cluster, causing total failure. We have seen this failure mode happen many times.

To manage compaction correctly, you have to schedule compactions on each node so that not too many nodes are affected at once. You have to be aware of how long a compaction takes—as well as the variance—to avoid having more nodes undergoing compaction than you intended. You have to make sure you have enough disk capacity on your nodes to last them between compactions. In addition, you have to make sure you have enough capacity on your cluster so that it doesn't become overloaded when resources are lost during compactions.

All of this can be managed by a competent operational staff, but it's our contention that the best way to deal with any sort of complexity is to get rid of that complexity

altogether. The fewer failure modes you have in your system, the less likely it is that you'll suffer unexpected downtime. Dealing with online compaction is a complexity inherent to fully incremental architectures, but in a Lambda Architecture the primary databases don't require any online compaction.

1.6.2 Extreme complexity of achieving eventual consistency

Another complexity of incremental architectures results when trying to make systems highly available. A highly available system allows for queries and updates even in the presence of machine or partial network failure.

It turns out that achieving high availability competes directly with another important property called *consistency*. A consistent system returns results that take into account all previous writes. A theorem called the CAP theorem has shown that it's impossible to achieve both high availability and consistency in the same system in the presence of network partitions. So a highly available system sometimes returns stale results during a network partition.

The CAP theorem is discussed in depth in chapter 12—here we wish to focus on how the inability to achieve full consistency and high availability at all times affects your ability to construct systems. It turns out that if your business requirements demand high availability over full consistency, there is a huge amount of complexity you have to deal with.

In order for a highly available system to return to consistency once a network partition ends (known as *eventual consistency*), a lot of help is required from your application. Take, for example, the basic use case of maintaining a count in a database. The obvious way to go about this is to store a number in the database and increment that number whenever an event is received that requires the count to go up. You may be surprised that if you were to take this approach, you'd suffer massive data loss during network partitions.

The reason for this is due to the way distributed databases achieve high availability by keeping multiple replicas of all information stored. When you keep many copies of the same information, that information is still available even if a machine goes down or the network gets partitioned, as shown in figure 1.4. During a network partition, a system that chooses to be highly available has clients update whatever replicas are reachable to them. This causes replicas to diverge and receive different sets of updates. Only when the partition goes away can the replicas be merged together into a common value.

Suppose you have two replicas with a count of 10 when a network partition begins. Suppose the first replica gets two increments and the second gets one increment. When it comes time to merge these replicas together, with values of 12 and 11, what should the merged value be? Although the correct answer is 13, there's no way to know just by looking at the numbers 12 and 11. They could have diverged at 11 (in which case the answer would be 12), or they could have diverged at 0 (in which case the answer would be 23).

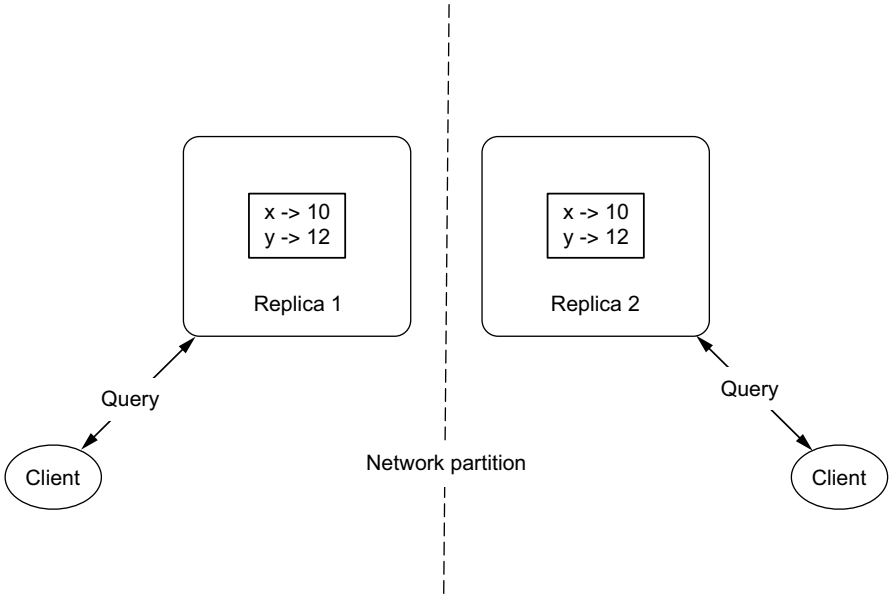

Figure 1.4 **Using replication to increase availability**

To do highly available counting correctly, it's not enough to just store a count. You need a data structure that's amenable to merging when values diverge, and you need to implement the code that will repair values once partitions end. This is an amazing amount of complexity you have to deal with just to maintain a simple count.

In general, handling eventual consistency in incremental, highly available systems is unintuitive and prone to error. This complexity is innate to highly available, fully incremental systems. You'll see later how the Lambda Architecture structures itself in a different way that greatly lessens the burdens of achieving highly available, eventually consistent systems.

1.6.3 Lack of human-fault tolerance

The last problem with fully incremental architectures we wish to point out is their inherent lack of human-fault tolerance. An incremental system is constantly modifying the state it keeps in the database, which means a mistake can also modify the state in the database. Because mistakes are inevitable, the database in a fully incremental architecture is guaranteed to be corrupted.

It's important to note that this is one of the few complexities of fully incremental architectures that can be resolved without a complete rethinking of the architecture. Consider the two architectures shown in figure 1.5: a synchronous architecture, where the application makes updates directly to the database, and an asynchronous architecture, where events go to a queue before updating the database in the background. In both cases, every event is permanently logged to an events datastore. By keeping every event, if a human mistake causes database corruption, you can go back

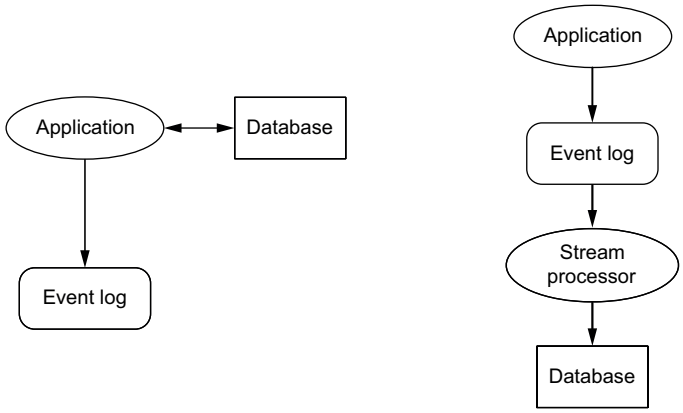

Figure 1.5 Adding logging to fully incremental architectures

to the events store and reconstruct the proper state for the database. Because the events store is immutable and constantly growing, redundant checks, like permissions, can be put in to make it highly unlikely for a mistake to trample over the events store. This technique is also core to the Lambda Architecture and is discussed in depth in chapters 2 and 3.

Although fully incremental architectures with logging can overcome the human-fault tolerance deficiencies of fully incremental architectures without logging, the logging does nothing to handle the other complexities that have been discussed. And as you'll see in the next section, any architecture based purely on fully incremental computation, including those with logging, will struggle to solve many problems.

1.6.4 Fully incremental solution vs. Lambda Architecture solution

One of the example queries that is implemented throughout the book serves as a great contrast between fully incremental and Lambda architectures. There's nothing contrived about this query—in fact, it's based on real-world problems we have faced in our careers multiple times. The query has to do with pageview analytics and is done on two kinds of data coming in:

- *Pageviews*, which contain a user ID, URL, and timestamp.
- *Equivs*, which contain two user IDs. An equiv indicates the two user IDs refer to the same person. For example, you might have an equiv between the email *sally@gmail.com* and the username *sally*. If sally@gmail.com also registers for the username *sally2*, then you would have an equiv between sally@gmail.com and sally2. By transitivity, you know that the usernames *sally* and *sally2* refer to the same person.

The goal of the query is to compute the number of unique visitors to a URL over a range of time. Queries should be up to date with all data and respond with minimal latency (less than 100 milliseconds). Here's the interface for the query:

```
long uniquesOverTime(String url, int startHour, int endHour)
```

What makes implementing this query tricky are those equivs. If a person visits the same URL in a time range with two user IDs connected via equivs (even transitively), that should only count as one visit. A new equiv coming in can change the results for any query over any time range for any URL.

We'll refrain from showing the details of the solutions at this point, as too many concepts must be covered to understand them: indexing, distributed databases, batch processing, HyperLogLog, and many more. Overwhelming you with all these concepts at this point would be counterproductive. Instead, we'll focus on the characteristics of the solutions and the striking differences between them. The best possible fully incremental solution is shown in detail in chapter 10, and the Lambda Architecture solution is built up in chapters 8, 9, 14, and 15.

The two solutions can be compared on three axes: accuracy, latency, and throughput. The Lambda Architecture solution is significantly better in all respects. Both must make approximations, but the fully incremental version is forced to use an inferior approximation technique with a 3–5x worse error rate. Performing queries is significantly more expensive in the fully incremental version, affecting both latency and throughput. But the most striking difference between the two approaches is the fully incremental version's need to use special hardware to achieve anywhere close to reasonable throughput. Because the fully incremental version must do many random access lookups to resolve queries, it's practically required to use solid state drives to avoid becoming bottlenecked on disk seeks.

That a Lambda Architecture can produce solutions with higher performance in every respect, while also avoiding the complexity that plagues fully incremental architectures, shows that something very fundamental is going on. The key is escaping the shackles of fully incremental computation and embracing different techniques. Let's now see how to do that.

1.7 Lambda Architecture

Computing arbitrary functions on an arbitrary dataset in real time is a daunting problem. There's no single tool that provides a complete solution. Instead, you have to use a variety of tools and techniques to build a complete Big Data system.

The main idea of the Lambda Architecture is to build Big Data systems as a series of layers, as shown in figure 1.6. Each layer satisfies a subset of the properties and builds upon the functionality provided by the layers beneath it. You'll spend the whole book learning how to design, implement, and deploy each layer, but the high-level ideas of how the whole system fits together are fairly easy to understand.

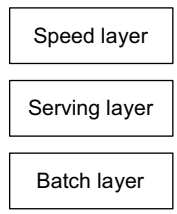

Figure 1.6 Lambda Architecture

Everything starts from the *query = function(all data)* equation. Ideally, you could run the functions on the fly to get the results. Unfortunately, even if this were possible, it would take a huge amount of resources to do and would be unreasonably expensive.

Imagine having to read a petabyte dataset every time you wanted to answer the query of someone's current location.

The most obvious alternative approach is to precompute the query function. Let's call the precomputed query function the *batch view*. Instead of computing the query on the fly, you read the results from the precomputed view. The precomputed view is indexed so that it can be accessed with random reads. This system looks like this:

```
batch view = function(all data)
query = function(batch view)
```

In this system, you run a function on all the data to get the batch view. Then, when you want to know the value for a query, you run a function on that batch view. The batch view makes it possible to get the values you need from it very quickly, without having to scan everything in it.

Because this discussion is somewhat abstract, let's ground it with an example. Suppose you're building a web analytics application (again), and you want to query the number of pageviews for a URL on any range of days. If you were computing the query as a function of all the data, you'd scan the dataset for pageviews for that URL within that time range, and return the count of those results.

The batch view approach instead runs a function on all the pageviews to precompute an index from a key of [url, day] to the count of the number of pageviews for that URL for that day. Then, to resolve the query, you retrieve all values from that view for all days within that time range, and sum up the counts to get the result. This approach is shown in figure 1.7.

It should be clear that there's something missing from this approach, as described so far. Creating the batch view is clearly going to be a high-latency operation, because it's running a function on all the data you have. By the time it finishes, a lot of new data will have collected that's not represented in the batch views, and the queries will be out of date by many hours. But let's ignore this issue for the moment, because we'll

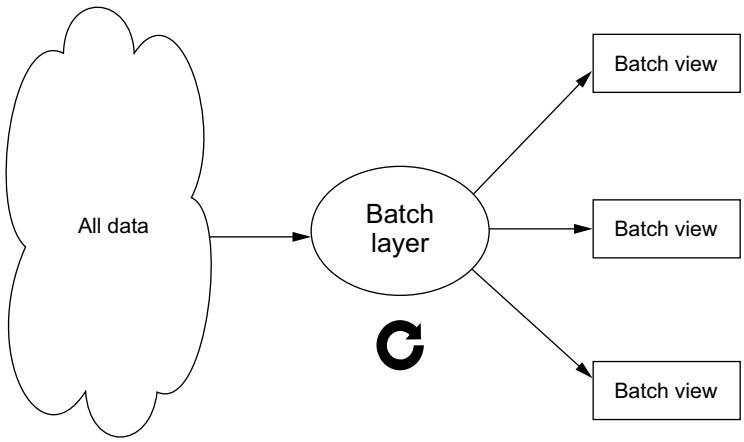

Figure 1.7 Architecture of the batch layer

be able to fix it. Let's pretend that it's okay for queries to be out of date by a few hours and continue exploring this idea of precomputing a batch view by running a function on the complete dataset.

1.7.1 Batch layer

The portion of the Lambda Architecture that implements the *batch view = function(all data)* equation is called the *batch layer*. The batch layer stores the master copy of the dataset and precomputes batch views on that master dataset (see figure 1.8). The master dataset can be thought of as a very large list of records.

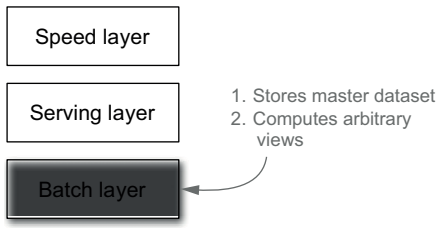

Figure 1.8 Batch layer

The batch layer needs to be able to do two things: store an immutable, constantly growing master dataset, and compute arbitrary functions on that dataset. This type of processing is best done using batch-processing systems. Hadoop is the canonical example of a batch-processing system, and Hadoop is what we'll use in this book to demonstrate the concepts of the batch layer.

The simplest form of the batch layer can be represented in pseudo-code like this:

```
function runBatchLayer():
  while(true):
    recomputeBatchViews()
```

The batch layer runs in a `while(true)` loop and continuously recomputes the batch views from scratch. In reality, the batch layer is a little more involved, but we'll come to that later in the book. This is the best way to think about the batch layer at the moment.

The nice thing about the batch layer is that it's so simple to use. Batch computations are written like single-threaded programs, and you get parallelism for free. It's easy to write robust, highly scalable computations on the batch layer. The batch layer scales by adding new machines.

Here's an example of a batch layer computation. Don't worry about understanding this code—the point is to show what an inherently parallel program looks like:

```
Api.execute(Api.hfsSeqfile("/tmp/pageview-counts"),
     new Subquery("?url", "?count")
         .predicate(Api.hfsSeqfile("/data/pageviews"),
             "?url", "?user", "?timestamp")
         .predicate(new Count(), "?count");
```

This code computes the number of pageviews for every URL given an input dataset of raw pageviews. What's interesting about this code is that all the concurrency challenges of scheduling work and merging results is done for you. Because the algorithm is written in this way, it can be arbitrarily distributed on a MapReduce cluster, scaling to however many nodes you have available. At the end of the computation, the output

directory will contain some number of files with the results. You'll learn how to write programs like this in chapter 7.

1.7.2 Serving layer

The batch layer emits batch views as the result of its functions. The next step is to load the views somewhere so that they can be queried. This is where the serving layer comes in. The serving layer is a specialized distributed database that loads in a batch view and makes it possible to do random reads on it (see figure 1.9). When new batch views are available, the serving layer automatically swaps those in so that more up-to-date results are available.

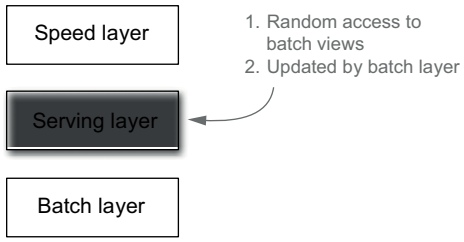

Figure 1.9 Serving layer

A serving layer database supports batch updates and random reads. Most notably, it doesn't need to support random writes. This is a very important point, as random writes cause most of the complexity in databases. By not supporting random writes, these databases are extremely simple. That simplicity makes them robust, predictable, easy to configure, and easy to operate. ElephantDB, the serving layer database you'll learn to use in this book, is only a few thousand lines of code.

1.7.3 Batch and serving layers satisfy almost all properties

The batch and serving layers support arbitrary queries on an arbitrary dataset with the trade-off that queries will be out of date by a few hours. It takes a new piece of data a few hours to propagate through the batch layer into the serving layer where it can be queried. The important thing to notice is that other than low latency updates, the batch and serving layers satisfy every property desired in a Big Data system, as outlined in section 1.5. Let's go through them one by one:

- *Robustness and fault tolerance*—Hadoop handles failover when machines go down. The serving layer uses replication under the hood to ensure availability when servers go down. The batch and serving layers are also human-fault tolerant, because when a mistake is made, you can fix your algorithm or remove the bad data and recompute the views from scratch.
- *Scalability*—Both the batch and serving layers are easily scalable. They're both fully distributed systems, and scaling them is as easy as adding new machines.
- *Generalization*—The architecture described is as general as it gets. You can compute and update arbitrary views of an arbitrary dataset.
- *Extensibility*—Adding a new view is as easy as adding a new function of the master dataset. Because the master dataset can contain arbitrary data, new types of data can be easily added. If you want to tweak a view, you don't have to worry

about supporting multiple versions of the view in the application. You can simply recompute the entire view from scratch.

- *Ad hoc queries*—The batch layer supports ad hoc queries innately. All the data is conveniently available in one location.
- *Minimal maintenance*—The main component to maintain in this system is Hadoop. Hadoop requires some administration knowledge, but it's fairly straightforward to operate. As explained before, the serving layer databases are simple because they don't do random writes. Because a serving layer database has so few moving parts, there's lots less that can go wrong. As a consequence, it's much less likely that anything *will* go wrong with a serving layer database, so they're easier to maintain.
- *Debuggability*—You'll always have the inputs and outputs of computations run on the batch layer. In a traditional database, an output can replace the original input—such as when incrementing a value. In the batch and serving layers, the input is the master dataset and the output is the views. Likewise, you have the inputs and outputs for all the intermediate steps. Having the inputs and outputs gives you all the information you need to debug when something goes wrong.

The beauty of the batch and serving layers is that they satisfy almost all the properties you want with a simple and easy-to-understand approach. There are no concurrency issues to deal with, and it scales trivially. The only property missing is low latency updates. The final layer, the speed layer, fixes this problem.

1.7.4 Speed layer

The serving layer updates whenever the batch layer finishes precomputing a batch view. This means that the only data not represented in the batch view is the data that came in while the precomputation was running. All that's left to do to have a fully realtime data system—that is, to have arbitrary functions computed on arbitrary data in real time—is to compensate for those last few hours of data. This is the purpose of the speed layer. As its name suggests, its goal is to ensure new data is represented in query functions as quickly as needed for the application requirements (see figure 1.10).

You can think of the speed layer as being similar to the batch layer in that it produces views based on data it receives. One big difference is that the speed layer only looks at recent data, whereas the batch layer looks at all the data at once. Another big difference is that in order to achieve the smallest latencies possible, the speed layer doesn't look at all the new data at once. Instead, it updates the realtime views as it receives new data instead of recomputing the views from scratch like the batch layer does. The speed layer does incremental computation instead of the recomputation done in the batch layer.

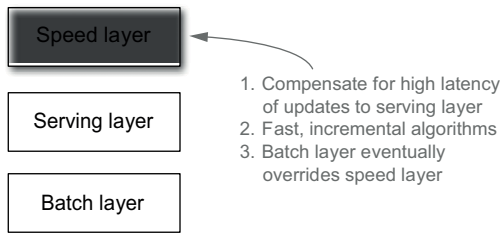

Figure 1.10 Speed layer

We can formalize the data flow on the speed layer with the following equation:

```
realtime view = function(realtime view, new data)
```

A realtime view is updated based on new data and the existing realtime view.

The Lambda Architecture in full is summarized by these three equations:

```
batch view = function(all data)
realtime view = function(realtime view, new data)
query = function(batch view. realtime view)
```

A pictorial representation of these ideas is shown in figure 1.11. Instead of resolving queries by just doing a function of the batch view, you resolve queries by looking at both the batch and realtime views and merging the results together.

The speed layer uses databases that support random reads and random writes. Because these databases support random writes, they're orders of magnitude more complex than the databases you use in the serving layer, both in terms of implementation and operation.

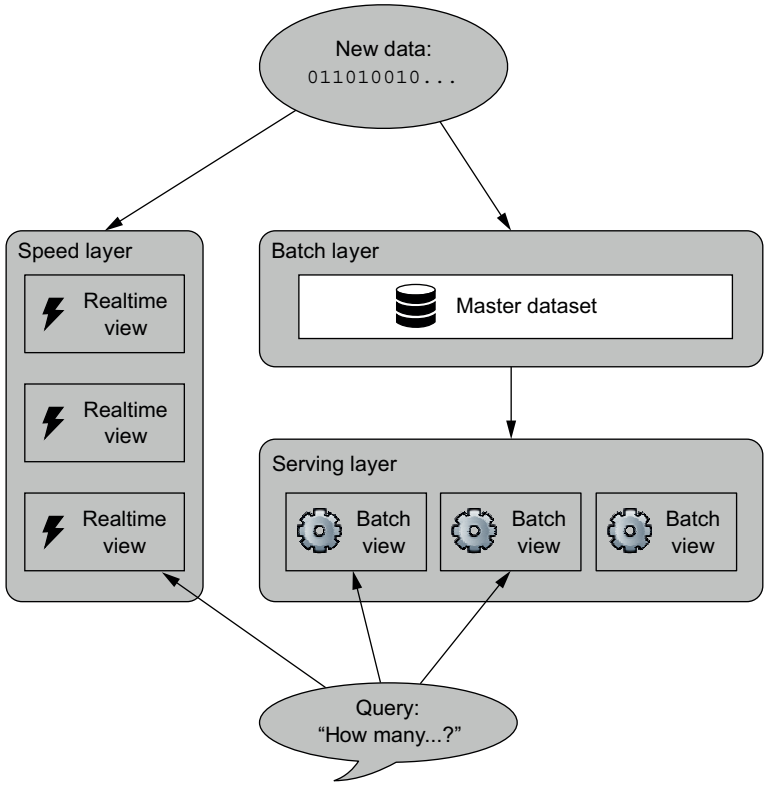

Figure 1.11 Lambda Architecture diagram

The beauty of the Lambda Architecture is that once data makes it through the batch layer into the serving layer, the corresponding results in the realtime views *are no longer needed*. This means you can discard pieces of the realtime view as they're no longer needed. This is a wonderful result, because the speed layer is far more complex than the batch and serving layers. This property of the Lambda Architecture is called *complexity isolation*, meaning that complexity is pushed into a layer whose results are only temporary. If anything ever goes wrong, you can discard the state for the entire speed layer, and everything will be back to normal within a few hours.

Let's continue the example of building a web analytics application that supports queries about the number of pageviews over a range of days. Recall that the batch layer produces batch views from [url, day] to the number of pageviews.

The speed layer keeps its own separate view of [url, day] to number of pageviews. Whereas the batch layer recomputes its views by literally counting the pageviews, the speed layer updates its views by incrementing the count in the view whenever it receives new data. To resolve a query, you query both the batch and realtime views as necessary to satisfy the range of dates specified, and you sum up the results to get the final count. There's a little work that needs to be done to properly synchronize the results, but we'll cover that in a future chapter.

Some algorithms are difficult to compute incrementally. The batch/speed layer split gives you the flexibility to use the exact algorithm on the batch layer and an approximate algorithm on the speed layer. The batch layer repeatedly overrides the speed layer, so the approximation gets corrected and your system exhibits the property of *eventual accuracy*. Computing unique counts, for example, can be challenging if the sets of uniques get large. It's easy to do the unique count on the batch layer, because you look at all the data at once, but on the speed layer you might use a HyperLogLog set as an approximation.

What you end up with is the best of both worlds of performance and robustness. A system that does the exact computation in the batch layer and an approximate computation in the speed layer exhibits eventual accuracy, because the batch layer corrects what's computed in the speed layer. You still get low latency updates, but because the speed layer is transient, the complexity of achieving this doesn't affect the robustness of your results. The transient nature of the speed layer gives you the flexibility to be very aggressive when it comes to making trade-offs for performance. Of course, for computations that can be done exactly in an incremental fashion, the system is fully accurate.

1.8 Recent trends in technology

It's helpful to understand the background behind the tools we'll use throughout this book. There have been a number of trends in technology that deeply influence the ways in which you can build Big Data systems.

1.8.1 CPUs aren't getting faster

We've started to hit the physical limits of how fast a single CPU can go. That means that if you want to scale to more data, you must be able to parallelize your computation.

This has led to the rise of shared-nothing parallel algorithms and their corresponding systems, such as MapReduce. Instead of just trying to scale by buying a better machine, known as *vertical scaling*, systems scale by adding more machines, known as *horizontal scaling*.

1.8.2 Elastic clouds

Another trend in technology has been the rise of elastic clouds, also known as *Infrastructure as a Service*. Amazon Web Services (AWS) is the most notable elastic cloud. Elastic clouds allow you to rent hardware on demand rather than own your own hardware in your own location. Elastic clouds let you increase or decrease the size of your cluster nearly instantaneously, so if you have a big job you want to run, you can allocate the hardware temporarily.

Elastic clouds dramatically simplify system administration. They also provide additional storage and hardware allocation options that can significantly drive down the price of your infrastructure. For example, AWS has a feature called *spot instances* in which you bid on instances rather than pay a fixed price. If someone bids a higher price than you, you'll lose the instance. Because spot instances can disappear at any moment, they tend to be significantly cheaper than normal instances. For distributed computation systems like MapReduce, they're a great option because fault tolerance is handled at the software layer.

1.8.3 Vibrant open source ecosystem for Big Data

The open source community has created a plethora of Big Data technologies over the past few years. All the technologies taught in this book are open source and free to use.

There are five categories of open source projects you'll learn about. Remember, this is not a survey book—the intent is not to just teach a bunch of technologies. The goal is to learn the fundamental principles so that you'll be able to evaluate and choose the right tools for your needs:

- *Batch computation systems*—Batch computation systems are high throughput, high latency systems. Batch computation systems can do nearly arbitrary computations, but they may take hours or days to do so. The only batch computation system we'll use is Hadoop. The Hadoop project has two subprojects: the Hadoop Distributed File System (HDFS) and Hadoop MapReduce. HDFS is a distributed, fault-tolerant storage system that can scale to petabytes of data. MapReduce is a horizontally scalable computation framework that integrates with HDFS.
- *Serialization frameworks*—Serialization frameworks provide tools and libraries for using objects between languages. They can serialize an object into a byte array from any language, and then deserialize that byte array into an object in any language. Serialization frameworks provide a Schema Definition Language for defining objects and their fields, and they provide mechanisms to safely version objects so that a schema can be evolved without invalidating existing objects. The three notable serialization frameworks are Thrift, Protocol Buffers, and Avro.

- *Random-access NoSQL databases*—There has been a plethora of NoSQL databases created in the past few years. Between Cassandra, HBase, MongoDB, Voldemort, Riak, CouchDB, and others, it's hard to keep track of them all. These databases all share one thing in common: they sacrifice the full expressiveness of SQL and instead specialize in certain kinds of operations. They all have different semantics and are meant to be used for specific purposes. They're *not* meant to be used for arbitrary data warehousing. In many ways, choosing a NoSQL database to use is like choosing between a hash map, sorted map, linked list, or vector when choosing a data structure to use in a program. You know beforehand exactly what you're going to do, and you choose appropriately. Cassandra will be used as part of the example application we'll build.
- *Messaging/queuing systems*—A messaging/queuing system provides a way to send and consume messages between processes in a fault-tolerant and asynchronous manner. A message queue is a key component for doing realtime processing. We'll use Apache Kafka in this book.
- *Realtime computation system*—Realtime computation systems are high throughput, low latency, stream-processing systems. They can't do the range of computations a batch-processing system can, but they process messages extremely quickly. We'll use Storm in this book. Storm topologies are easy to write and scale.

As these open source projects have matured, companies have formed around some of them to provide enterprise support. For example, Cloudera provides Hadoop support, and DataStax provides Cassandra support. Other projects are company products. For example, Riak is a product of Basho technologies, MongoDB is a product of 10gen, and RabbitMQ is a product of SpringSource, a division of VMWare.

1.9 Example application: SuperWebAnalytics.com

We'll build an example Big Data application throughout this book to illustrate the concepts. We'll build the data management layer for a Google Analytics–like service. The service will be able to track billions of pageviews per day.

The service will support a variety of different metrics. Each metric will be supported in real time. The metrics range from simple counting metrics to complex analyses of how visitors are navigating a website.

These are the metrics we'll support:

- *Pageview counts by URL sliced by time*—Example queries are "What are the pageviews for each day over the past year?" and "How many pageviews have there been in the past 12 hours?"
- *Unique visitors by URL sliced by time*—Example queries are "How many unique people visited this domain in 2010?" and "How many unique people visited this domain each hour for the past three days?"
- *Bounce-rate analysis*—"What percentage of people visit the page without visiting any other pages on this website?"

We'll build out the layers that store, process, and serve queries to the application.

1.10 Summary

You saw what can go wrong when scaling a relational system with traditional techniques like sharding. The problems faced go beyond scaling as the system becomes more complex to manage, extend, and even understand. As you learn how to build Big Data systems in the upcoming chapters, we'll focus as much on robustness as we do on scalability. As you'll see, when you build things the right way, both robustness and scalability are achievable in the same system.

The benefits of data systems built using the Lambda Architecture go beyond just scaling. Because your system will be able to handle much larger amounts of data, you'll be able to collect even more data and get more value out of it. Increasing the amount and types of data you store will lead to more opportunities to mine your data, produce analytics, and build new applications.

Another benefit of using the Lambda Architecture is how robust your applications will be. There are many reasons for this; for example, you'll have the ability to run computations on your whole dataset to do migrations or fix things that go wrong. You'll never have to deal with situations where there are multiple versions of a schema active at the same time. When you change your schema, you'll have the capability to update all data to the new schema. Likewise, if an incorrect algorithm is accidentally deployed to production and corrupts the data you're serving, you can easily fix things by recomputing the corrupted values. As you'll see, there are many other reasons why your Big Data applications will be more robust.

Finally, performance will be more predictable. Although the Lambda Architecture as a whole is generic and flexible, the individual components comprising the system are specialized. There is very little "magic" happening behind the scenes, as compared to something like a SQL query planner. This leads to more predictable performance.

Don't worry if a lot of this material still seems uncertain. We have a lot of ground yet to cover and we'll revisit every topic introduced in this chapter in depth throughout the course of the book. In the next chapter you'll start learning how to build the Lambda Architecture. You'll start at the very core of the stack with how you model and schematize the master copy of your dataset.

DATABASES

Big Data

Marz • Warren

Free eBook
SEE INSERT

Web-scale applications like social networks, real-time analytics, or e-commerce sites deal with a lot of data, whose volume and velocity exceed the limits of traditional database systems. These applications require architectures built around clusters of machines to store and process data of any size, or speed. Fortunately, scale and simplicity are not mutually exclusive.

Big Data teaches you to build big data systems using an architecture designed specifically to capture and analyze web-scale data. This book presents the Lambda Architecture, a scalable, easy-to-understand approach that can be built and run by a small team. You'll explore the theory of big data systems and how to implement them in practice. In addition to discovering a general framework for processing big data, you'll learn specific technologies like Hadoop, Storm, and NoSQL databases.

What's Inside

- Introduction to big data systems
- Real-time processing of web-scale data
- Tools like Hadoop, Cassandra, and Storm
- Extensions to traditional database skills

This book requires no previous exposure to large-scale data analysis or NoSQL tools. Familiarity with traditional databases is helpful.

Nathan Marz is the creator of Apache Storm and the originator of the Lambda Architecture for big data systems. **James Warren** is an analytics architect with a background in machine learning and scientific computing.

"Transcends individual tools or platforms. Required reading for anyone working with big data systems."
—Jonathan Esterhazy, Groupon

"A comprehensive, example-driven tour of the Lambda Architecture with its originator as your guide."
—Mark Fisher, Pivotal

"Contains wisdom that can only be gathered after tackling many big data projects. A must-read."
—Pere Ferrera Bertran, Datasalt

"The de facto guide to streamlining your data pipeline in batch and near-real time."
—Alex Holmes
Author of *Hadoop in Practice*

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/BigData

ISBN 13: 978-1-617290-34-3
ISBN 10: 1-617290-34-3
54999
9 781617 290343

MANNING $49.99 / Can $57.99 [INCLUDING eBOOK]