Big Data

Principles and best practices of scalable realtime data systems

Nathan Marz
James Warren

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Big Data
Version 17

Copyright 2014 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

brief contents

1. *A new paradigm for big data*

2. *Data model for big data*

3. *Data storage on the batch layer*

4. *Batch layer: scalability*

5. *Batch layer: abstraction and composition*

6. *Batch layer: tying it all together*

7. *Serving layer*

8. *Speed layer: Realtime views*

9. *Speed layer: queuing and stream processing*

10. *Speed layer: micro-batch stream processing*

11. *Lambda Architecture in-depth*

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

1 A New Paradigm for Big Data

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 are simpler. These systems can scale to vastly larger sets of data, but using these technologies effectively requires a fundamentally new set of techniques. They are not 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, you must rethink data systems from the ground up. You will discover that some of the most basic ways people manage data in traditional systems like the relational database management system (RDBMS) is

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

too complex for Big Data systems. The simpler, alternative approach is the new paradigm for Big Data that you will be exploring. We, the authors, have dubbed this approach the "Lambda Architecture".

In this chapter, you will 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. Then, starting from first principles of data systems, you'll learn a different way to build data systems that avoids the complexity of traditional techniques. Finally you'll take a look at an example Big Data system that we'll be building throughout this book to illustrate the key concepts.

1.1 What this book is and is not about

This book is not a survey of database, computation, and other related technologies. While you will 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 upon themselves. Rather, the tools are a means to learning the underlying principles of architecting robust and scalable data systems.

Put another way, you are going to learn how to fish, not just how to use a particular fishing rod. Different situations require different tools. If you understand the underlying principles of building these systems, then you will be able to effectively map the requirements to the right set of tools.

At many points in this book, there will be a choice of technologies to use. Doing an involved compare-and-contrast between the tools would not be doing you, the reader, justice, as that just distracts from learning the principles of building data systems. Instead, the approach we take is to make clear the requirements for a particular situation, and explain why a particular tool meets those requirements. Then, we will use that tool to illustrate the application of the concepts. For example, we will be using Thrift as the tool for specifying data schemas and Cassandra for storing realtime state. Both of these tools have alternatives, but that doesn't matter for the purposes of this book since these tools are sufficient for illustrating the underlying concepts.

By the end of this book, you will have a thorough understanding of the principles of data systems. You will be able to use that understanding to choose the right tools for your specific application.

Let's begin our exploration of data systems by seeing what can go wrong when using traditional tools to solve Big Data problems.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

1.2 Scaling with a traditional database

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

You have a lot of experience using relational databases to build web applications, so you start with a traditional relational schema for the pageviews that looks something like Figure 1.1. Whenever someone loads a webpage being tracked by your application, the webpage pings your web server with the pageview and your web server increments the corresponding row in the RDBMS.

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

Figure 1.1 Relational schema for simple analytics application

Your plan so far makes sense -- at least in the world before Big Data. But as you'll soon find out, you're going to run into problems with both scale and complexity as you evolve the application.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

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 do 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 backend 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 1000 events at a time off the queue and batches them into a single database update. This is illustrated in Figure 1.2.

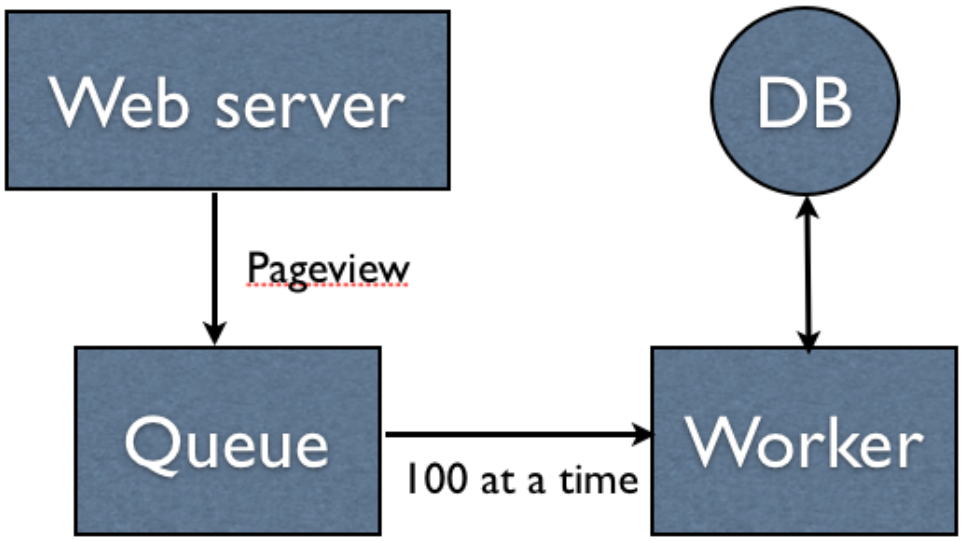

Figure 1.2 Batching updates with queue and worker

This scheme works great and 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.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

1.2.2 Scaling by sharding the database

Unfortunately, adding a queue and doing batch updates was only a band-aid to 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 work; 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". It is also known as sharding. This technique spreads the write load across multiple machines.

The technique you use to shard the database 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 evenly 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 awhile 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 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 as 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 worker scripts active at once. One time 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 has been misplaced.

Does this sound familiar? Has a situation like this ever happened to you? The good news is that Big Data systems will be able to help you tackle problems like these. However, back in our example, you haven't yet learned about Big Data systems, and your problems are still compounding...

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

1.2.3 Fault-tolerance issues begin

Eventually you have so many shards that it's not uncommon for the disk on one of the database machines to go bad. So that portion of the data is unavailable while that machine is down. You do a few things to address this:

- You update your queue/worker system to put increments for unavailable shards on a separate "pending" queue that is attempted to be flushed once every 5 minutes.
- You use the database's replication capabilities to add a slave to each shard to 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 for every URL instead of by one. You don't notice until 24 hours later but by then the damage is done: many of the values in your database are inaccurate. 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 to 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 Analysis of problems with traditional architecture

In developing the web analytics application, you started with one web server and one database and ended with a web of queues, workers, shards, replicas, and web servers. Scaling your application forced your backend to become much more complex. Unfortunately, operating the backend became much more complex as well! Consider some of the serious challenges that emerged with your new architecture:

- *Fault-tolerance is hard:* As the number of machines in the backend grew, it became increasingly more likely that a machine would go down. All the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

complexity of keeping the application working even under failures has to be managed manually, such as setting up replicas and managing a failure queue. Nor was your architecture fully fault-tolerant: if the master node for a shard is down, you're unable to execute writes to that shard. Making writes highly-available is a much more complex problem that your architecture doesn't begin to address.

- *Complexity pushed to application layer:* The distributed nature of your data is not abstracted away from you. Your application needs to know which shard to look at for each key. Queries such as the "Top 100 URLs" query had to be modified to query every shard and then merge the results together.
- *Lack of human fault-tolerance:* As the system gets more and more complex, it becomes more and more likely that a mistake will be made. Nothing prevents you from reading/writing data from the wrong shard, and logical bugs can irreversibly corrupt the database.

  Mistakes in software are inevitable, so 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 is essential especially when Big Data adds so many more complexities to building applications.
- *Maintenance is an enormous amount of work:* Scaling your sharded database is time-consuming and error-prone. The problem is that you have to manage all the constraints of what is allowed where yourself. What you really want is for the database to be self-aware of its distributed nature and manage the sharding process for you.

The Big Data techniques you are going to learn will address these scalability and complexity issues in dramatic fashion. First of all, the databases and computation systems you use for Big Data are self-aware of their distributed nature. So things like sharding and replication are handled for you. You will 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 machines and the data will automatically rebalance onto that new machine.

Another core technique you will learn about is making your data immutable. Instead of storing the pageview counts as your core dataset, which you

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

continuously mutate as new pageview come in, you store the raw pageview information. That raw pageview information is never modified. So when you make mistake, you might write bad data, but at least you didn'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 would be wary of using immutable data because of how fast such a dataset would grow. But since Big Data techniques can scale to so much data, you have the ability to design systems in different ways.

1.3 NoSQL as a paradigm shift

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. This set of tools has been categorized under the term "NoSQL." These systems can handle very large scales of data but with serious tradeoffs.

Hadoop, for example, can run 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 since the databases are mutable, they're not human fault-tolerant.

These tools on their own are not a panacea. However, 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 will be learning throughout the book.

1.4 First principles

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

Let's start with an intuitive definition of what a data system does: "A data system answers questions based on information that was acquired in the past". 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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

bits and pieces together to produce their answers. A bank account balance, for example, is based on combining together 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 the friend list, and the friend list is derived from all the times the user added and removed friends from her profile.

When you keep tracing back where information is derived from, you eventually end up at the most raw form of information -- information that was not derived from anywhere else. This is the information you hold to be true simply because it exists. Let's call this information "data".

Consider the example of the "friend count" on a social network profile. The "friend count" is ultimately derived from events triggered by users: adding and removing friends. So the data underlying the "friend count" are the "add friend" and "remove friend" events. You could, of course, choose to only store the existing friend relationships, but the rawest form of data you could store are the individual add and remove events.

You may have a different conception for what the word "data" means. Data is often used interchangably with the word "information". However, for the remainder of the book when we use the word "data", we are referring to that special information from which everything else is derived.

You answer questions on your data by running functions that take data as input. Your function that answers the "friend count" question can derive the friend count by looking at all the add and remove friend events. Different functions may look at different portions of the dataset and aggregate information in different ways. The most general purpose data system can answer questions by running functions that take in the *entire dataset* as input. In fact, any query can be answered by running a function on the complete dataset. So the most general purpose definition of a query is this:

Figure 1.3 Basis of all possible data systems

Remember this equation, because it is the crux of everything you will learn. We

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

will be referring to this equation over and over. The goal of a data system is to compute arbitrary functions on arbitrary data.

The Lambda Architecture, which we will be introducing later in this chapter, 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 everytime 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 how to wire them together to meet your requirements.

Before we dive into the Lambda Architecture, let's 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. You don't need to memorize these properties, as we will revisit them as we use first principles to show how to achieve these properties.

1.5.1 Robust and fault-tolerant

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

Additionally, it is imperative for systems to be "human fault-tolerant." This is an oft-overlooked property of systems that we are not going to ignore. In a production system, it's inevitable that someone is going to make a mistake sometime, like by deploying incorrect code that corrupts values in a database. You will learn how to bake immutability and recomputation into the core of your systems to make your systems innately resilient to human error. Immutability and recomputation will be described in depth in Chapters 2 through 5.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

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 propogate immediately, while in other applications a latency of a few hours is fine. Regardless, you will 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 will learn how to achieve low latency updates in the discussion of the "speed layer" in Chapter 7.

1.5.3 Scalable

Scalability is the ability to maintain performance in the face of increasing data and/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 General

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! The Lambda Architecture generalizes to applications as diverse as financial management systems, social media analytics, scientific applications, and social networking.

1.5.5 Extensible

You don't want to have to reinvent the wheel each time you want to 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 change to an existing feature requires a migration of old data into a new format. Part of a system being 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 will learn.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

1.5.6 Allows 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 will learn how to do ad hoc queries in Chapters 4 and 5 when we discuss batch processing.

1.5.7 Minimal maintenance

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 small an *implementation complexity* as possible. That is, 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 will discuss this technique in depth when we discuss the "speed layer" in Chapter 7.

1.5.8 Debuggable

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.

Achieving all these properties together in one system seems like a daunting challenge. But by starting from first principles, these properties naturally emerge from the resulting system design. Let's now take a look at the Lambda Architecture which derives from first principles and satisifes all of these properties.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

1.6 Lambda Architecture

Computing arbitrary functions on an arbitrary dataset in realtime is a daunting problem. There is 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 Lambda Architecture solves the problem of computing arbitrary functions on arbitrary data in realtime by decomposing the problem into three layers: the batch layer, the serving layer, and the speed layer. You will be spending 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.

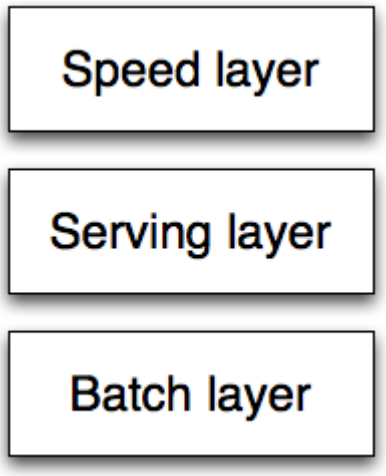

Figure 1.4 Lambda Architecture

Everything starts from the "query = function(all data)" equation. Ideally, you could literally run your query functions on the fly on the complete dataset 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 everytime you want to answer the query of someone's current location.

The 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 quickly with random reads. This system looks like this:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

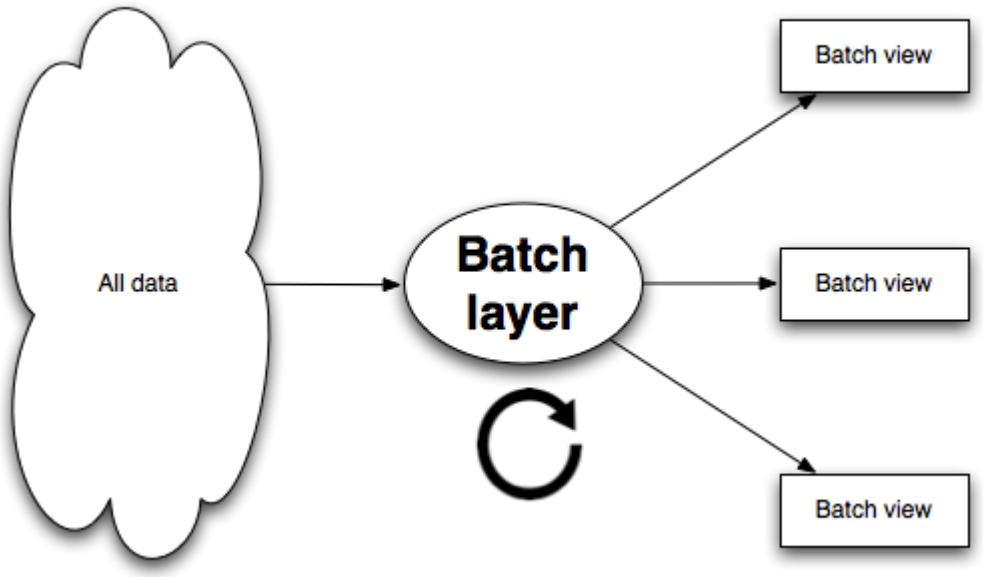

Figure 1.5 Batch layer

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 function, you use the precomputed results to complete the query rather than scan through all the data. The batch view makes it possible to get the values you need from it very quickly since it's indexed.

Since 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 would scan the dataset for pageviews for that URL within that time range and return the count of those results. This of course would be enormously expensive, as you would have to look at all the pageview data for every query you do.

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. The precomputed view indexes the data by url, so you can quickly retreive all the data points you need to complete the query.

You might be thinking 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, as it's running a function on all the data you have. By the time it

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

finishes, a lot of new data will have collected that's not represented in the batch views, and the queries are going to be out of date by many hours. You're right, but let's ignore this issue for the moment because we'll 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.

1.6.1 Batch Layer

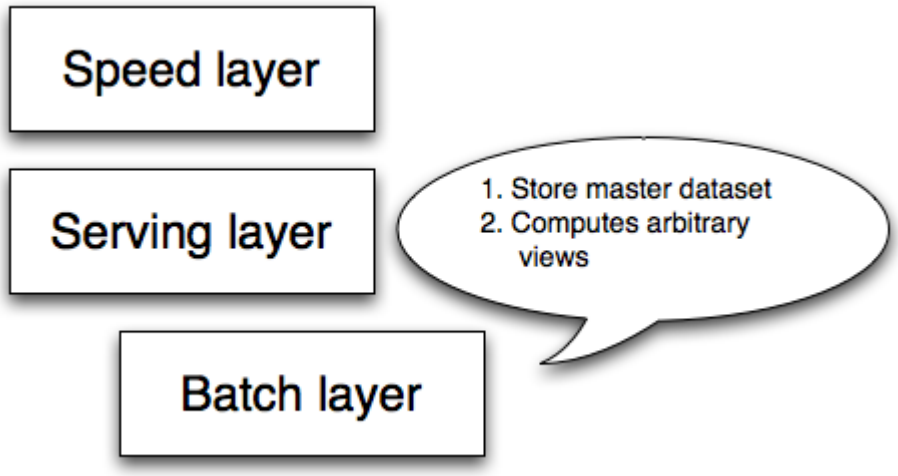

Figure 1.6 Batch layer

The portion of the Lambda Architecture that precomputes the batch views is called the "batch layer". The batch layer stores the master copy of the dataset and precomputes batch views on that master dataset. The master dataset can be thought of us a very large list of records.

The batch layer needs to be able to do two things to do its job: store an immutable, constantly growing master dataset, and compute arbitrary functions on that dataset. The key word here is "arbitrary." If you're going to precompute views on a dataset, you need to be able to do so for *any view* and *any dataset*. There's a class of systems called "batch processing systems" that are built to do exactly what the batch layer requires. They are very good at storing immutable, constantly growing datasets, and they expose computational primitives to allow you to compute arbitrary functions on those datasets. Hadoop is the canonical example of a batch processing system, and we will use Hadoop in this book to demonstrate the concepts of the batch layer.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

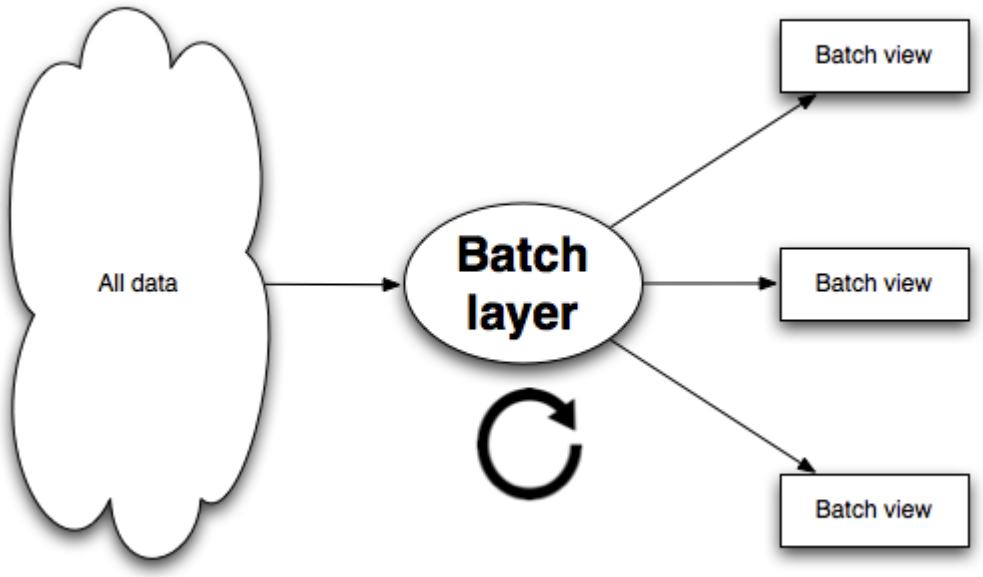

Figure 1.7 Batch layer

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 will be a little more involved, but we'll come to that in a later chapter. 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, yet automatically parallelize across a cluster of machines. This implicit parallelization makes batch layer computations scale to datasets of any size. It's easy to write robust, highly scalable computations on the batch layer.

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.

```
Pipe pipe = new Pipe("counter");
pipe = new GroupBy(pipe, new Fields("url"));
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

```
pipe = new Every(
        pipe,
        new Count(new Fields("count")),
        new Fields("url", "count"));
Flow flow = new FlowConnector().connect(
                new Hfs(new TextLine(new Fields("url")), srcDir),
                new StdoutTap(),
                pipe);
flow.complete();
```

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, merging results, and dealing with runtime failures (such as machines going down) is done for you. Because the algorithm is written in this way, it can be automatically distributed on a MapReduce cluster, scaling to however many nodes you have available. So if you have 10 nodes in your MapReduce cluster, the computation will finish about 10x faster than if you only had one node! At the end of the computation, the output directory will contain some number of files with the results. You will learn how to write programs like this in Chapter 5.

1.6.2 Serving Layer

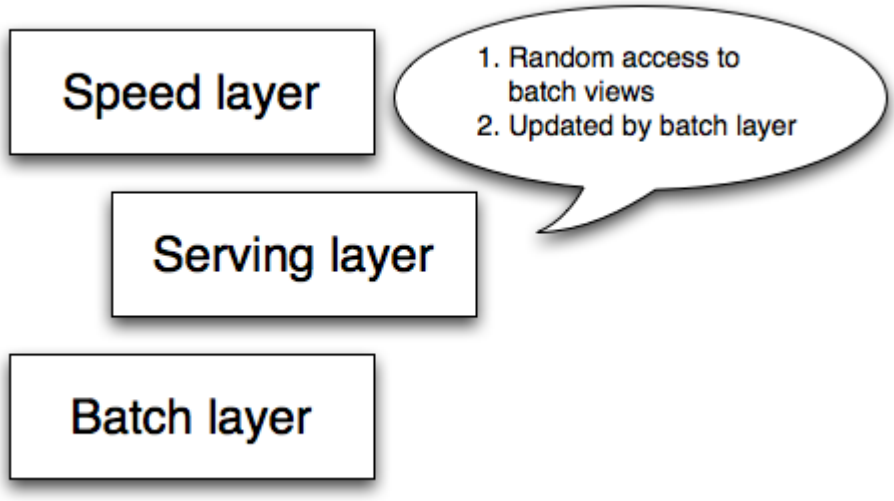

Figure 1.8 Serving layer

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. For example, your batch layer may precompute a batch view containing the pageview count for every [url, hour] pair. That batch view is

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

essentially just a set of flat files though: there's no way to quickly get the value for a particular URL out of that output.

The serving layer indexes the batch view and loads it up so it can be efficiently queried to get particular values out of the view. The serving layer is a specialized distributed database that loads in a batch views, makes them queryable, and continuously swaps in new versions of a batch view as they're computed by the batch layer. Since the batch layer usually takes at least a few hours to do an update, the serving layer is updated every few hours.

A serving layer database only requires batch updates and random reads. Most notably, it does not 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, serving layer databases can be very simple. That simplicity makes them robust, predictable, easy to configure, and easy to operate. ElephantDB, the serving layer database you will learn to use in this book, is only a few thousand lines of code.

1.6.3 Batch and serving layers satisfy almost all properties

So far you've seen how the batch and serving layers can support arbitrary queries on an arbitrary dataset with the tradeoff that queries will be out of date by a few hours. The long update latency is due to new pieces of data taking a few hours to propogate 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 1.3. Let's go through them one by one:

- *Robust and fault tolerant:* The batch layer handles failover when machines go down using replication and restarting computation tasks on other machines. 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, since when a mistake is made you can fix your algorithm or remove the bad data and recompute the views from scratch.
- *Scalable:* Both the batch layer and serving layers are easily scalable. They can both be implemented as fully distributed systems, whereupon scaling them is as easy as just adding new machines.
- *General:* The architecture described is as general as it gets. You can compute and update arbitrary views of an arbitrary dataset.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

- *Extensible:* Adding a new view is as easy as adding a new function of the master dataset. Since 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.
- *Allows ad hoc queries:* The batch layer supports ad-hoc queries innately. All the data is conveniently available in one location and you're able to run any function you want on that data.
- *Minimal maintenance:*The batch and serving layers are comprised of very few pieces, yet they generalize arbitrarily. So you only have to maintain a few pieces for a huge number of applications. As explained before, the serving layer databases are simple because they don't do random writes. Since 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 are easier to maintain.
- *Debuggable:*You will always have the inputs and outputs of computations run on the batch layer. In a traditional database, an output can replace the original input -- for example, 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 trivially scales. The only property missing is low latency updates. The final layer, the speed layer, fixes this problem.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

1.6.4 Speed layer

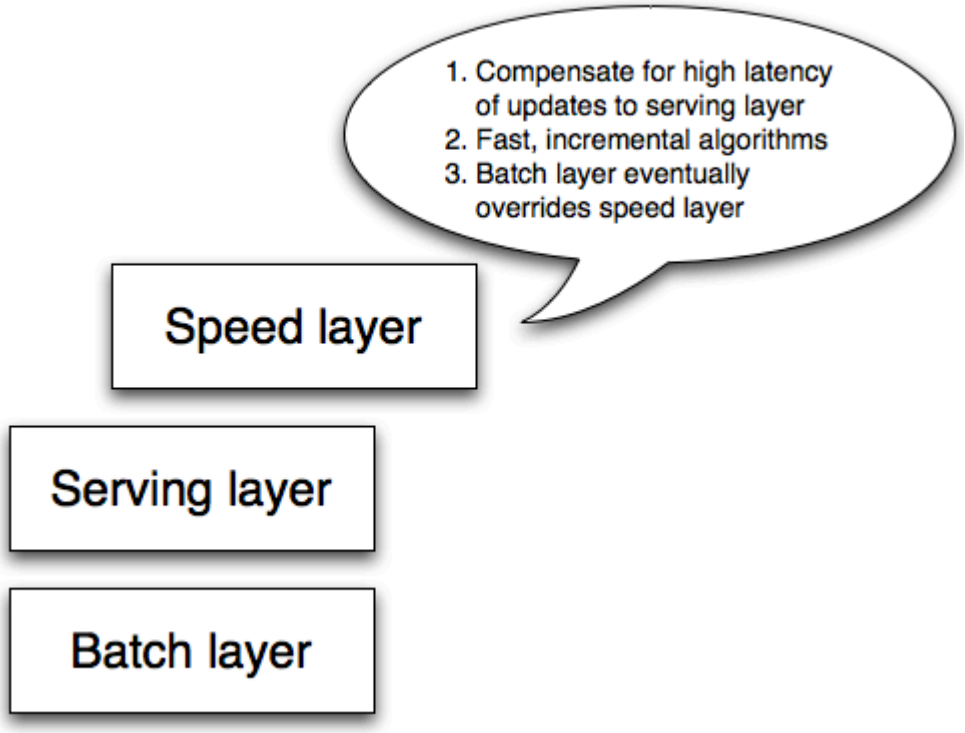

Figure 1.9 Speed layer

The serving layer updates whenever the batch layer finishes precomputing a batch view. This means that the only data not represented in the batch views 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, arbitrary functions computed on arbitrary data in realtime – is to compensate for those last few hours of data. This is the purpose of the speed layer.

You can think of the speed layer as similar to the batch layer in that it produces views based on data it receives. There are some key differences though. One big difference is that in order to achieve the fastest latencies possible, the speed layer doesn't look at all the new data at once. Instead, it updates the realtime view as it receives new data instead of recomputing them like the batch layer does. This is called "incremental updates" as opposed to "recomputation updates". Another big difference is that the speed layer only produces views on recent data, whereas the batch layer produces views on the entire dataset.

Let's continue the example of computing the number of pageviews for a url over a range of time. The speed layer needs to compensate for pageviews that

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

haven't been incorporated in the batch views, which will be a few hours of pageviews. Like the batch layer, the speed layer maintains a view from a key [url, hour] to a pageview count. Unlike the batch layer, which recomputes that mapping from scratch each time, the speed layer modifies its view as it receives new data. When it receives a new pageview, it increments the count for the corresponding [url, hour] in the database.

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

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, since the speed layer is way 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 entire speed layer and everything will be back to normal within a few hours. This property greatly limits the potential negative impact of the complexity of the speed layer.

The last piece of the Lambda Architecture is merging the resutlts from the batch and realtime views to quickly compute query functions. For the pageview example, you get the count values for as many of the hours in the range from the batch view as possible. Then, you query the realtime view to get the count values for the remaining hours. You then sum up all the individual counts to get the total number of pageviews over that range. There's a little work that needs to be done to get the synchronization right between the batch and realtime views, but we'll cover that in a future chapter. The pattern of merging results from the batch and realtime views is shown in figure 1.10.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

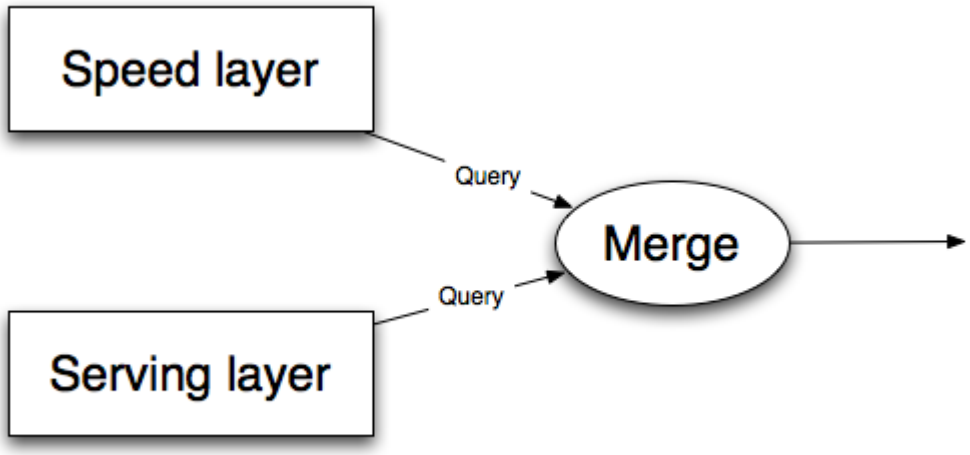

Figure 1.10 Satisfying application queries

We've covered a lot of material in the past few sections. Let's do a quick summary of the Lambda Architecture to nail down how it works.

1.7 Summary of the Lambda Architecture

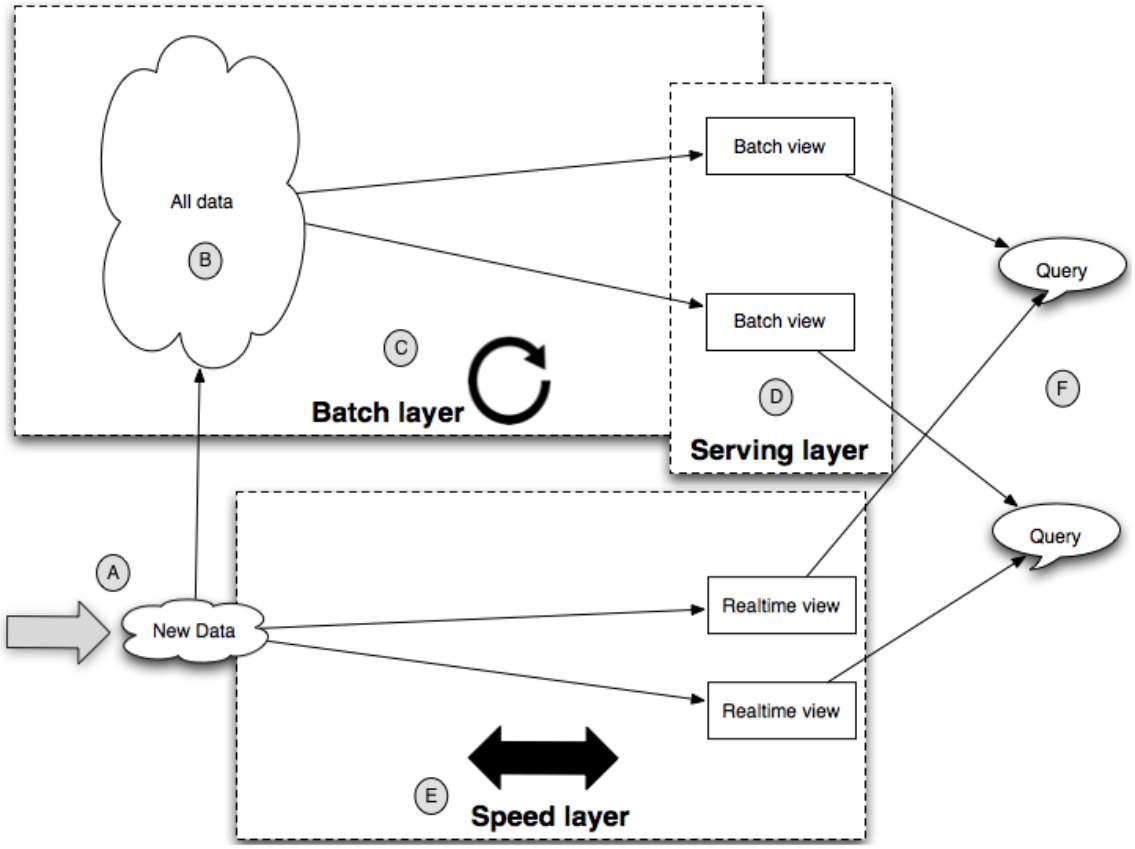

Figure 1.11 Lambda Architecture diagram

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

The complete Lambda Architecture is represented pictorially in Figure 1.11. We will be referring to this diagram over and over in the rest of the chapters. Let's go through the diagram piece by piece.

- (A): All new data is sent to both the batch layer and the speed layer. In the batch layer, new data is appended to the master dataset. In the speed layer, the new data is consumed to do incremental updates of the realtime views.
- (B): The master dataset is an immutable, append-only set of data. The master dataset only contains the rawest information that is not derived from any other information you have. We will have a thorough discussion on the importance of immutability in the upcoming chapter.
- (C): The batch layer precomputes query functions from scratch. The results of the batch layer are called "batch views." The batch layer runs in a while(true) loop and continuously recomputes the batch views from scratch. The strength of the batch layer is its ability to compute arbitrary functions on arbitrary data. This gives it the power to support any application.
- (D): The serving layer indexes the batch views produced by the batch layer and makes it possible to get particular values out of a batch view very quickly. The serving layer is a scalable database that swaps in new batch views as they're made available. Because of the latency of the batch layer, the results available from the serving layer are always out of date by a few hours.
- (E): The speed layer compensates for the high latency of updates to the serving layer. It uses fast incremental algorithms and read/write databases to produce realtime views that are always up to date. The speed layer only deals with recent data, because any data older than that has been absorbed into the batch layer and accounted for in the serving layer. The speed layer is significantly more complex than the batch and serving layers, but that complexity is compensated by the fact that the realtime views can be continuously discarded as data makes its way through the batch and serving layers. So the potential negative impact of that complexity is greatly limited.
- (F): Queries are resolved by getting results from both the batch and realtime views and merging them together.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

We will be building an example Big Data application throughout this book to illustrate a complete implementation of the Lambda Architecture. Let's now introduce that sample application.

1.8 Example application: SuperWebAnalytics.com

The example application we will be building throughout the book is the data management layer for a Google Analytics like service. The service will be able to track billions of page views per day.

SuperWebAnalytics.com will support a variety of different metrics. Each metric will be supported in real-time. The metrics we will support are:

1. Page view counts by URL sliced by time. Example queries are "What are the pageviews for each day over the past year?". "How many pageviews have there been in the past 12 hours?"

2. Unique visitors by URL sliced by time. Example queries are "How many unique people visited this domain in 2010?" "How many unique people visited this domain each hour for the past three days?"

3. Bounce rate analysis. "What percentage of people visit the page without visiting any other pages on this website?"

We will be building out the layers that store, process, and serve queries to the application.

1.9 Summary

You saw what can go wrong when scaling a relational system with traditional techniques like sharding. The problems faced went beyond scaling as the system became complex to manage, extend, and even understand. As you learn how to build Big Data systems in the upcoming chapters, we will 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 will 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 is how much more robust your applications will be. There are many reasons why your applications will be more robust. As one 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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787

multiple versions of a schema active at the same time. When you change your schema, you will have the capability to update all data to the new schema. Likewise, if an incorrect algorithm is accidentally deployed to production and corrupts data you're serving, you can easily fix things by recomputing the corrupted values. As you'll explore, 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 will be revisiting every topic introduced in this chapter in depth throughout the course of the book. In the next chapter you will start learning how to build the Lambda Architecture. You will start at the very core of the stack with how you model and schemify the master copy of your dataset.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=787