Big Data

Principles and best practices of scalable realtime data systems

Nathan Marz
Samuel E. Ritchie

MEAP Edition
Manning Early Access Program
Big Data version 1

Copyright 2011 Manning Publications

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

Table of Contents

1. A new paradigm for Big Data

2. Data model for Big Data

3. Data storage on the batch layer

4. MapReduce and batch processing

5. Batch processing with Cascading

6. Basics of the serving layer

7. Storm and the speed layer

8. Incremental batch processing

9. Layered architecture in-depth

10. Piping the system together

11. Future of NoSQL and Big Data processing

Appendix A Hadoop

Appendix B Thrift

Appendix C Storm

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

A New Paradigm for Big Data

In the past decade the amount of data being created has skyrocketed. More than 30000 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 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, who created an innovative distributed key-value store called Dynamo. The open source community responded in the years following with projects like Hadoop, HBase, MongoDB, Cassandra, RabbitMQ, and countless other projects.

We will learn how to use this new breed of technologies to build robust and

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

scalable Big Data systems. We will be exploring a new set of techniques for handling Big Data. Managing the complexity of these systems is as important as scaling. As our tools become more complex and we must worry about concepts like fault-tolerance, consistency, and availability in our application code, it is imperative that we find ways to eliminate complexity throughout the rest of our systems. Some of the most basic ways people handle data in traditional systems is too complex for building robust Big Data systems. The simpler, alternative approach is the new paradigm for Big Data that we will be exploring.

In this first chapter, we will explore the "Big Data problem" and why we need a new paradigm for Big Data. We'll look at the perils of some of the traditional techniques for scaling and disocver 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. We'll take a look at how recent trends in technology encourage the use of new kinds of systems, and finally we'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 Scaling with a traditional database

Let's start by taking a look at some of the problems you'll run into when trying to scale a traditional database to Big Data. The most pervasive traditional database I will be referring to are relational databases, such as MySQL, Oracle, or Postgres. Relational databases exhibit the following characteristics:

1. *Centralized*: Relational databases are designed to be run on one machine. You can shard and/or replicate a relational database, but those lead to their own set of headaches as we will discuss. The best way to scale a relational database is to get a more powerful machine.
2. *One-size-fits-all*: Relational databases try to satisfy all query workloads. This includes both "known queries", queries that are done repeatedly by the application, and "one-off queries", queries that are done infrequently to answer arbitrary questions. Relational databases try to accomplish being one-size-fits-all databases through their query language, SQL, and features such as indexing and views.

With that, let's take a closer look at some of the perils you'll run into when working with traditional databases. We'll do this by observing as a simple application scales and evolves.

The example application we'll look at is a simple web analytics application. The application tracks the number of pageviews to any url a customer wishes to track.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

A web page pings the application's web server with its url everytime a pageview is received.

Additionally, the application occasionally runs a query to extract the top 100 most viewed urls from the database.

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

Figure 1.1 Relational schema for simple analytics application

The initial implementation for this application is simple. The data is all kept on a relational database with a schema as shown in Figure 1.1. When a webpage pings the web server with a page view, the web server increments the pageview count for that url in a transaction.

At some point, the write throughput will become too high for the relational database to handle. In the next few sections I'll discuss two very common techniques used to scale applications: inserting a queue to do asynchronous, batched updates and sharding the database across multiple machines to increase write throughput. You'll see that these techniques scale at the cost of significant complexity and loss of fault tolerance, and that to achieve both robustness and scale we'll need to rethink the entire application.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

1.1.1 Adding asynchronous, batched updates

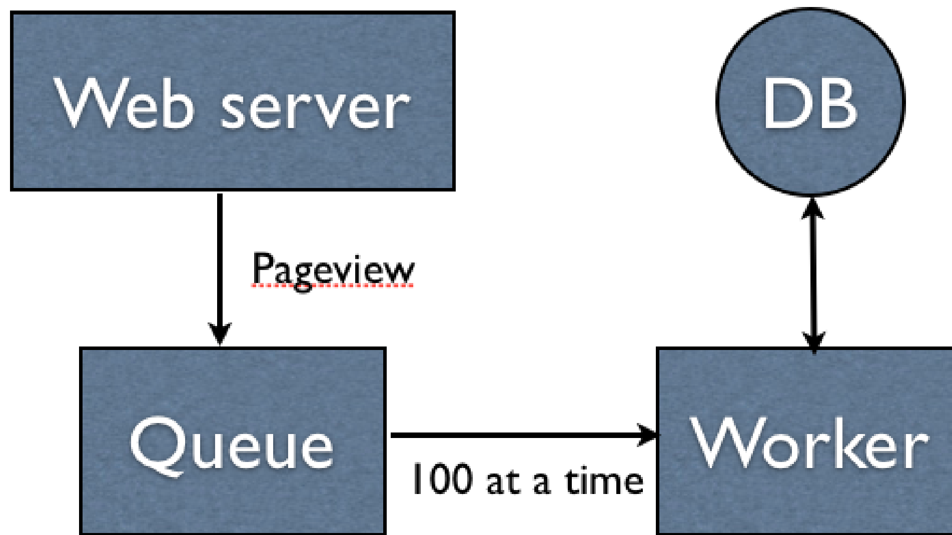

Figure 1.2 Batching updates with queue and worker

When the application grows big enough, the single database won't be able to handle the load of all the incoming requests. A common technique used to increase performance is to batch write requests to the database. Rather than increment only one url in a given request, it is more efficient to increment multiple urls in a single request. This is accomplished by inserting a queue into the architecture as shown in Figure 1.2. A worker continuously pulls items off the queue in batches and sends one request to the database for each batch.

Unfortunately, this approach is only a band-aid to the scaling problem. Batches can only be made so big and a new scaling technique will be needed once the write throughput gets high enough.

There is a conceptual benefit to this approach though. It decouples the web tier from the database. If the database can't keep up with writes, the queue will grow bigger rather than timeout to the client.

1.1.2 Sharding a relational database

The standard technique to scale a write-heavy application is to shard the database. Rather than keep all the data for a table on one machine, the table will instead be spread across multiple machines. A simple function is used that maps a key to the machine it belongs on.

Sharding will scale this pageview application, as the load of writes is now spread across multiple machines. However, sharding introduces new problems into

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

the back-end.

First, sharding is a manual, multi-step process. The difficulty of adding new shards increases the more shards already exist: it's a lot harder to go from 16 shards to 32 shards than it is to go from 2 shards to 4 shards. The process involves writing scripts to create the new shards and move every row to the correct shard. For this to happen in a timely manner, the resharding process must be parallelized. The process typically involves backup and verification steps to reduce the risk of catastrophic error. Resharding without taking any downtime is even more complex.

Second, sharding a database makes the code using that database more complex. Rather than simply run queries on a key, the code must first find what shard to talk to. This becomes even worse for our analytics query to find the 100 urls with the most pageviews. Rather than simply ask for the top 100 urls, the query needs to get the top 100 urls from each shard and then merge the results in the code.

Doing sharding manually is a lot of work given that you just want to track pageviews for URLs. With the databases we will be using that are designed for Big Data, the distributed nature of the data will be hidden from our application code. Scaling will be as easy as adding new machines to the cluster. Since the databases are aware of their distributed nature, they will handle any necessary resharding internally. [1]

Footnote 1 For the most part - may need to run a rebalancing task but this is much easier than doing it manually.

1.1.3 Complexity of adding new features

Adding new features to this application is complex. Let's observe what happens when trying to add a new feature that keeps track of the last time a url was viewed by anyone.

First, we need to add a new column to the schema for the "last_viewed timestamp". This schema migration needs to be run on every shard, and updating a schema can be slow. The migration will need to be monitored for success.

The original pageview information was discarded after the aggregate view count was updated in the database. Since that information was not kept, it's impossible to know what the last_viewed timestamp should be for each url. The common solution to this is to default the value to "null" indicating that the value is unknown. Unfortunately, this affects the application code that queries the database and serves the data to the user: the application needs to handle null as a special case and display to the user that the last viewed timestamp is unknown for that url.

This problem of having different versions of data gets compounded the more

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

the application evolves over time. The "unknown case" manifests itself over and over and the application code gets riddled with code to handle the different versions of data.

The complexity of adding new features is caused by two problems with the pageview application. The first problem is that only an aggregate of the pageview information is kept rather than the raw pageview information. The last_viewed timestamps could be deduced from the raw pageviews if they were available, but they are not. The second problem is that even if the raw pageviews were kept, the system isn't setup to run a massive computation to recompute the databases so that every url has a last_viewed timestamp.

Both these problems won't exist in the approach we take in this book to building Big Data systems. Because the systems will be able to handle arbitrary amounts of data, we will collect the raw data rather than just an aggregate form of it. And since batch, massive scale computation will be at the core of the systems we build, doing schema migrations that involve recomputing the entire database will be easy.

1.1.4 Incremental approaches are vulnerable to human error

One of the key problems in the pageview application is that it relies on incremental algorithms at its core. When a new pageview is received, the count is incremented by one, rather than recomputed by fetching and counting all the known pageviews.

Incremental algorithms are vulnerable to human error. And if there's one guarantee in software development, it's that humans aren't perfect and bugs inevitably reach production. If a bug was deployed to production that incremented pageviews by two instead of by one, there would be no way to fix the corrupt values in the database. Information was lost through the incremental process of updating pageview counts. This is a fundamental weakness of incremental algorithms. Human errors happen and your systems need to be designed to be resilient to human error. There should always be a recovery process.

In the Big Data world, we will keep the raw data and use recomputation algorithms when possible to make our algorithms resilient to human error. When a mistake is made, we can just remove bad data and use recomputation to fix our databases.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

1.1.5 Fault tolerance of sharded database is bad

A sharded relational database is not very fault tolerant. The more it's sharded, the less fault tolerant it is. There's always some probability of a machine going down, whether because of a disk going bad, a disk filling up, or the machine losing power. The more machines you have, the more likely it is that some machine will be unavailable.

The way to achieve fault tolerance is to replicate each shard across multiple machines. That way, if a machine goes down, its data can be served off of another machine. Unfortunately, there are an enormous amount of complications to doing this, and implementing this scheme is equivalent to writing an entire NoSQL database.

The Big Data databases we will be using make use of the sharding plus replication strategy internally, but they hide the complexities of it from you.

1.2 The CAP Theorem

Distributing data and computations across many machines has proven to be the best way to scale data systems. Unfortunately, there is a strong negative result called the CAP theorem that places limits on how fault-tolerant distributed systems can be to machine failure. The CAP theorem explores the relationship between three desired properties of data systems -- consistency, availability, and partition-tolerance -- and states that a distributed database can have at most two of these properties. Let's look at the definition of these properties and then see how the CAP theorem impacts the construction of actual data systems.

1.2.1 Definition

Consistency means that after you do a successful write, future reads will always take that write into account. Consider the example of writing to the database that Sally's location is "Atlanta", where previously her location was "Chicago". In a consistent database, all future reads of the value are guaranteed to get "Atlanta" back. In an inconsistent database, future readers may or may not get "Atlanta" back. Some readers may get "Atlanta" and others may get "Chicago".

Availability means you can always read from and write to the system. An unavailable database will return errors back to the client when the client requests an unavailable portion of the data. Dealing with unavailability can be awkward. Oftentime your best option is to bubble up the error back to the user, which can seriously detract from the user experience of your product.

A partition in a distributed systems means that messages fail to get delivered

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

between any two nodes in the system. A partition-tolerant system maintains its properties even in the event of one or more partitions. Partitions can occur due to problems in the network (like losing a network switch) or due to machines going down.

It is tempting to think that you don't need partition-tolerance and pick consistency and availability. Unfortunately, you can't really do this. Partitions can and do happen, whether because of network failure or high load on your system causes multiple nodes to go down. Since partitions are a fact of nature, it doesn't make sense to build a partition-untolerant system. Otherwise, when a partition happens, your database can't give you any guarantees. So you have to choose between availability and consistency.

1.2.2 Eventual Consistency

A database that chooses consistency and partition-tolerance (commonly abbreviated CP) is easy to understand: reads and writes are always consistent but sometimes you'll get an error when you do a read or write. What about a highly available and partition-tolerant database (AP)? Is it completely inconsistent or does it provide a different kind of consistency guarantee?

Highly available databases commonly provide an*eventual consistency* guarantee. During a partition, these databases may be inconsistent. So you may do a write and read a stale value, or different readers may read different values for the same key at the same time. When the partition goes away, the nodes of the databases will talk to each other and reconcile the inconsistency. Eventual consistency is the best guarantee you can get when you require your database to be highly available. It's important to remember that this is a limitation of nature and not of our tools.

Eventual consistency is not free. It requires work on the part of the programmer to tell the database how to reconcile values to make them consistent again. This is best explained through example.

Suppose you have a simple application that updates counters in an eventually consistent database. Suppose the same key K is being incremented by two writers "W1" and "W2". The database achieves high availability by keeping a replica of each key on another node. Let's call the two replicas of a key "R1" and "R2". This setup looks something like Figure 1.4.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

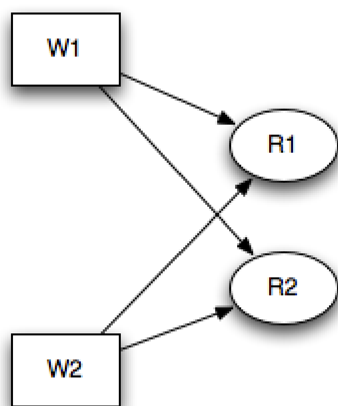

Figure 1.3 Highly available database without a partition

Now suppose a partition forms in the network between the two replicas. Now W1 can only increment R1, and W2 can only increment R2. The values of K at R1 and R2 will quickly get out of sync, and each replica will represent a different set of updates since the two replicas were last consistent. A partition is illustrated in Figure 1.4.

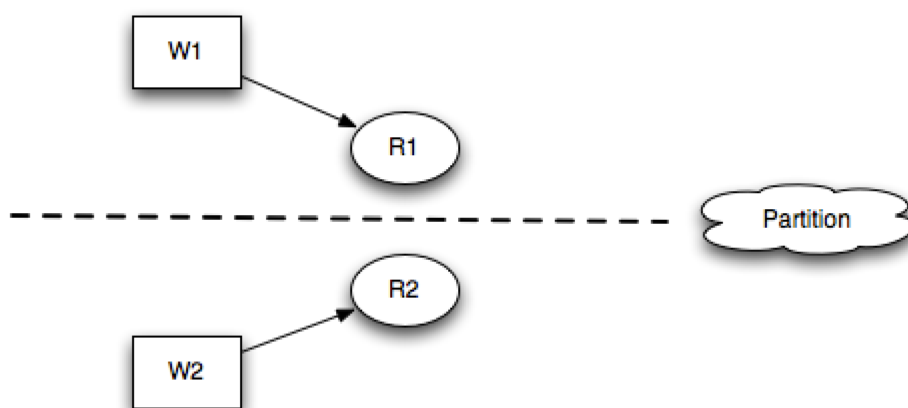

Figure 1.4 Highly available database with a partition

When the partition goes away, the database can now detect that R1 and R2 are out of sync. Unfotunately, it has no idea how to repair the value. Should it pick one of the replicas as the new value, or should it somehow merge the replicas together? It is up to you as a developer to tell the database how to make the value consistent again. The next time you read K, the database will give you all replicas for the key along with a history of how the replicas diverged since they were last consistent. In our example, the history might look something like Figure 1.5.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

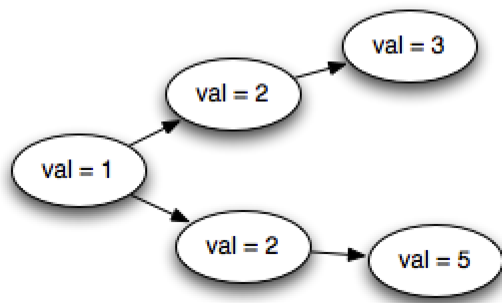

Figure 1.5 History of a value that diverged due to a partition

This history is represented in a data structure called a vector clock, and using this information you need to repair the value. This process is called "read repair".

As you can see, read-repair is complex. Read-repair code is extremely susceptible to developer error; if and when you make a mistake, faulty read-repairs will introduce irreversible corruption into the database.

It is clear that there is something fundamentally wrong with this way of building data systems. Not only do you have to figure out how to store, read, and write your data, you need to figure out how to repair your data as a core part of your data workflow. It is all too likely that you will make a mistake and cause corruption in your database.

What's missing from these systems is the notion of "human fault-tolerance." If there's any certainty in software development, it's that bugs make it to production and the correctness of our applications cannot be reliant on developers being perfect all the time. Mistakes shouldn't cause irreversible corruption -- there should always be a way to recover. The CAP theorem is a result about the degree to which data systems can be fault-tolerant to machine failure, but we need to go beyond that and build systems that are human fault-tolerant as well.

To achieve this goal, we will look at what we're doing wrong that's causing all this complexity. By starting from first principles, you can avoid the complexity and build human fault-tolerant systems that are easy to reason about. Let's talk more about the properties we desire from Big Data systems and then look at the first principles that are going to allow us to satisfy those properties.

1.3 Desired Properties of a Big Data System

The properties we 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

1.3.1 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 we can easily reason about the system.

As discussed before, we also want our systems to "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. By building in "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 will be described in depth in Chapters 2 and 3.

1.3.2 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, we will need to be able to achieve low latency updates *when we need them* in our Big Data systems. More importantly, we need to be able to achieve low latency reads and updates without compromising the robustness of the system. We will learn how to achieve low latency updates in the discussion of the "speed layer" in Chapter 5.

1.3.3 Scalable

Scalability is the ability to maintain performance in the face of increasing data and/or load by adding resources to the system. The systems we build will be horizontally scalable across all layers of the system stack: scaling is accomplished by adding more machines.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

1.3.4 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 systems we will build will generalize to applications as diverse as financial management systems, social media analytics, scientific applications, and social networking.

1.3.5 Extensible

We don't want to have to reinvent the wheel each time we want to add a related feature or make a change to how our 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 we will learn.

1.3.6 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.

Ad hoc queries don't need to be low latency because they're only needed for offline analysis. They just need to be doable in a reasonable amount of time. We'll learn how to do ad hoc queries in Chapter 3 when we discuss batch processing.

1.3.7 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 small an *implementation complexity* as possible. We 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 the system.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

We will combat implementation complexity by preferring to rely on simple algorithms and simple components. A trick that we will employ is to push complexity out of our core components and into pieces of our system whose outputs are discardable after a few hours. The most complex components we use, like read/write distributed databases, will be in this layer where outputs are eventually discardable. We will discuss this technique in depth when we discuss the "speed layer" in Chapter 5.

1.3.8 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.

We will accomplish "debuggability" through the functional nature of the "batch layer" and by preferring to use "recomputation" algorithms when possible. We will go in depth into how to debug these systems in Chapter 12.

Achieving all these properties together in one system seems like a daunting challenge. But by starting from first principles, these properties will naturally emerge from the resulting system design. Let's begin that journey by exploring those first principles.

1.4 First principles

Before we can even begin to approach the design of a system that satisfies our desired properties, first we have to define the problem we're trying to solve. What is the purpose of a data system? What is data?

Data applications range from storing and retrieving objects, joins, aggregations, stream processing, continuous computation, machine learning, and so on and so on. The definitions we choose must generalize to all data systems if they are going to be a foundation for desiging a Big Data system.

It turns out data systems can be summarized by one simple equation. Let's look at the equation and then rigorously define what each piece of it means.

Query = Function(All data)

That's it. This equation summaries the entire field of databases and data systems. Everything in the field -- the past 50 years of RDBMS's, indexing, OLAP, OLTP, MapReduce, ETL, distributed filesystems, stream processors, NoSQL, etc. -- is summarized by that equation in one way or another.

A data system answers questions about a dataset. Those questions are called "queries". And this equation states that a query is just a function of all the data you

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

have.

This equation may seem too general to be useful. It doesn't seem to capture any of the intricacies of data system design. But what matters is that every data system falls into that equation. The equation is a starting point from which we can explore data systems, and the equation will eventually lead to a method for satisfying all the properties we enumerated.

There are two concepts in this equation: "data" and "queries". These are distinct concepts that are often conflated in the database field, so let's be rigorous about what these concepts mean.

1.4.1 Data

Let's start with "data". A piece of data is an indivisible unit that you hold to be true for no other reason that it exists. It is like an axiom in mathematics.

There are two crucial properties to note about data. First, data is inherently time based. A piece of data is a fact that you know to be true at some moment of time. For example, suppose Sally enters into her social network profile that she lives in Chicago. The data you take from that input is that she lived in Chicago as of the particular moment in time that she entered that information into her profile. Suppose that on a later date Sally updates her profile location to Atlanta. Then you know that she lived in Atlanta as of that particular time. The fact that she lives in Atlanta now doesn't change the fact that she used to live in Chicago. Both pieces of data are true.

The second property of data follows immediately from the first: data is inherently immutable. Because of its connection to a point in time, the truthfulness of a piece of data never changes. One cannot go back in time to change the truthfulness of a piece of data. This means that there are only two main operations you can do with data: read existing data and add more data. CRUD (Create-read-update-delete), the standard operations of relational databases, has become CR.

I've left out the "Update" operation. This is because updates don't make sense with immutable data. For example, "updating" Sally's location really means that you're adding a new piece of data saying she lives in a new location as of a more recent time.

I've also left out the "Delete" operation. Again, most cases of deletes are better represented as creating new data. For example, if Bob stops following Mary on

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

Twitter, that doesn't change the fact that he used to follow her. So instead of deleting the data that says he follows her, you'd add a new data record that says he un-followed her at some moment in time.

There are a few cases where you do want to permanently delete data, such as regulations requiring you to purge data after a certain amount of time. These cases are easily supported by the data system design I'm going to show as we'll discuss later on in the book.

This definition of data is almost certainly different than what you're used to, especially if you come from the relational database world where updates are the norm. There are two reasons for this. First, this definition of data is extremely generic: it's hard to think of a kind of data that doesn't fit under this definition. Second, the immutability of data is the key property we're going to exploit in designing a human fault-tolerant data system that's easy to reason about.

1.4.2 Queries

The second concept in the equation is the "query". A query is a derivation from a set of data. In this sense, a query is like a theorem in mathematics. For example, "What is Sally's current location?" is a query. You would compute this query by returning the most recent data record about Sally's location. Queries are functions of the complete dataset, so they can do anything: aggregations, join together different types of data, and so on. So you might query for the number of female users of your service, or you might query a dataset of tweets for what topics have been trending in the past few hours.

I've defined a query as a function on the complete dataset. Of course, many queries don't need the complete dataset to run -- they only need a subset of the dataset. But what matters is that my definition encapsulates all possible queries.

Now that data and queries have been defined and shown to encapsulate all data systems, let's begin our exploration of how to build a system that allows you to run any query on any dataset within your performance constraints.

1.5 Thought experiment on the ideal Big Data architecture

The simplest possible data system computes queries by literally running a function on the complete dataset to compute queries. If you could do this within your latency constraints, then you'd be done. There would be nothing else to build.

Of course, it's infeasible to expect a function on a complete dataset to finish quickly. Many queries, such as those that serve a website, require millisecond response times. It's impossible to run a function on even a small dataset that fast,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

much less a petabyte scale dataset. However, let's do a thought experiment. Let's pretend that you can compute any query function instantly, and let's look at the properties of such a system. Doing this thought experiment will inform as to what a real Big Data architecture should look like.

1.5.1 Interaction with the CAP theorem

Let's start by looking at how this system interacts with the CAP theorem. Remember, the CAP theorem states a limitation of nature; you must choose between availability and consistency in any data system you build. As you are about to see, the CAP theorem does not affect your ability to reason about the system. It adds no complexity and requires no additional work on your part to maintain consistency.

If you choose consistency over availability, then queries will always incorporate all data that has been written in the past. During partitions though, the system will sometimes fail to write data or compute queries.

Things get much more interesting when you choose availability over consistency. In this case, the system is eventually consistent without any of the complexities of eventual consistency. Since the system is highly available, you can always write new data and compute queries. In failure scenarios, queries will return results that don't incorporate previously written data. Eventually that data will be consistent and queries will incorporate that data into their computations.

The key here is that data is immutable. Immutable data means there's no such thing as an update, so it's impossible for different replicas of a piece of data to become inconsistent. From the perspective of queries, a piece of data either exists or doesn't exist. There are no divergent values, vector clocks, or read-repair. By having immutable data and computing queries from scratch each time, you completely avoid the complexity normally caused by the CAP theorem. There is just data and functions on that data. Eventual consistency does not get in the way of reasoning about the system, and the developer does not have to do anything to enforce eventual consistency.

What caused complexity in my discussion of eventual consistency from before was the interaction between incremental updates and the CAP theorem. Incremental updates and the CAP theorem really don't play well together; mutable values require read-repair in an eventually consistent system. By rejecting incremental updates, embracing immutable data, and computing queries from scratch each time, you avoid that complexity.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

1.5.2 Human fault-tolerance

Such a system is also phenomenally human fault-tolerant. There's only two mistakes a human can make in such a system: write bad data or deploy a buggy query function.

Writing bad data does not cause corruption in this system. All data is immutable and each piece of data is independent, so writing data never replaces older data. Recovering from writing bad data is as simple as deleting the offending pieces of data, and then queries will return correct results. Note that this is in stark contrast to traditional databases, such as RDBMS's, where updates are the norm. In a traditional database an update destroys an older piece of data. It can be hard or impossible to recover from such corruption.

It's even easier to fix a system that has a buggy query function deployed. All you have to do is fix the bug. Since the query recomputes its results from scratch each time, queries will be correct once the bug is fixed.

This recomputation-based system has some very nice properties. It is easy to reason about: you just think in terms of data and queries and everything works. Of course, what we just went through was a thought experiment; a system like this is infeasible. However, let's extract from this thought experiment the key aspects of the system that made it human fault-tolerant and easy to reason about.

There are three distinguishing features of this recomputation-based system:

- The system makes it easy to store and scale an immutable, constantly-growing dataset.
- The primary write operation is adding new immutable facts of data.
- The system avoids the complexity of the CAP theorem by recomputing queries from raw data.

Traditional databases like the RDBMS do not have any of these properties. The lack of immutability of data in an RDBMS makes it lack human fault-tolerance; if you write bad data you will irreversibly destroy good data.

One of the reasons RDBMS's don't treat data as immutable is for practicality: you don't have enough space on a single server to store an immutable, constantly-growing dataset. In the Big Data world, you have much looser space constraints due to the nature of systems being horizontally scalable. Storing an immutable dataset becomes feasible in the Big Data context.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

Let's now build on these principles and see what a Big Data architecture that satisifes all our desired properties looks like.

1.6 Realtime Big Data architecture

Computing arbitrary functions on an arbitrary dataset in realtime is a daunting problem. It is not possible with a single tool or approach. Instead, we will have to use a variety of tools and techniques to build a complete Big Data system.

The main idea is to build Big Data systems as a series of layers. Each layer will satisfy a subset of the properties and build upon the functionality provided by the layers beneath it. We will be spending the whole book discussing how to design, implement, and deploy each layer, but the high level ideas of how the whole system fits together is fairly easy to understand.

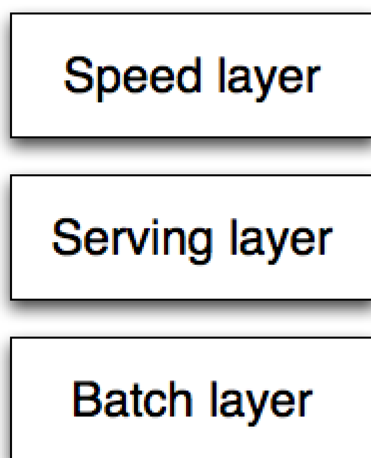

Figure 1.6 Layered architecture

Since a query is a function of all the data, the easiest way to make queries run fast is to precompute them. Whenever there's new data, you just recompute everything from the master dataset. Recomputing things from scratch can be slow, so by the time the precomputation finishes the queries may be out of date by a few hours. However, building such a system is fairly straightforward and is a strong first step to solving the more general problem of computing queries on the complete dataset in realtime. All that will be left is to compensate for that last few hours of data.

I call the layer that does the precomputation the "batch layer" and the layer that provides random access to the results of the precomputation the "serving layer". Finally, the layer that compensates for the last few hours of data is the "speed layer". The layers and how they interact are described in the next few sections.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

1.6.1 Batch layer

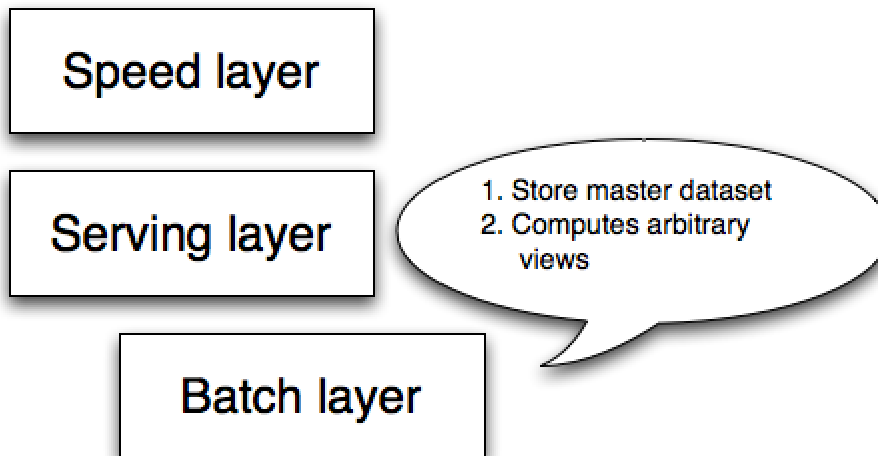

Figure 1.7 Batch layer

At the bottom of our architecture is the batch layer. The batch layer stores the master copy of the dataset and precomputes queries on that master dataset. The master dataset is append-only and can be thought of us a very large list of records. The batch layer is good at storing lots of data and doing high latency, high throughput computations. Computations done in the batch layer typically scan the full dataset.[2]We will be using Hadoop as the basis for the batch layer.

Footnote 2 This is not strictly true. We will be doing some partitioning of the dataset, but you rely on "scan and filter" computations instead of indexing individual records.

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 parallism "for free". It's easy to write robust, highly scalable computations on the batch layer. The batch layer scales by adding new machines.

On the batch layer we will be computing views of the master dataset. A view is simply a function of the complete dataset that precomputes some query. For example, if your master dataset contains all the tweets that ever happened on Twitter, one view may be the number of times any given URL has been tweeted. Another view may assign a score for how influential each person is on Twitter. Another view may classify each Twitter account as being a human, bot, or company account. What makes the batch layer powerful is its arbitrariness: you can compute any view on the batch layer, and you can do so with simple code.

In the batch layer, we will start off by literally writing functions of the complete dataset to compute the views. To update a view, you would throw away the old

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

view and compute a new one from scratch.

You may be wondering about the feasability of always recomputing views from scratch. After all, it seems wasteful to recompute the entire view when only a small part of it may have changed since the last recompute. The reason we start with the full recompute approach is because full recomputes are sometimes necessary, whether for the first run of the application, to do a schema migration, or to fix a mistake that has been made. Additionally, full recomputes are easy to build.

For integrating smaller amounts of new data into our views, we will not be doing full recomputes as it's too resource-intensive in practice. We want hours of turnaround on our batch workflows rather than days or weeks. What we will do as a compromise to make the batch layer more resource-efficient is introduce some incrementalization into the batch layer. The idea is that for most views, most of the view doesn't change given a few hours of new data. We'll learn techniques for how to partially update a view to make the batch layer vastlly more resource efficient. Incrementalization adds complexity, so we will introduce the minimum amount of incrementalization necessary to maintain the simplicity of the full recompute approach.

A view computed by the batch layer is not servable, as the batch layer just produces flat files of records. This is where the serving layer comes in.

1.6.2 Serving layer

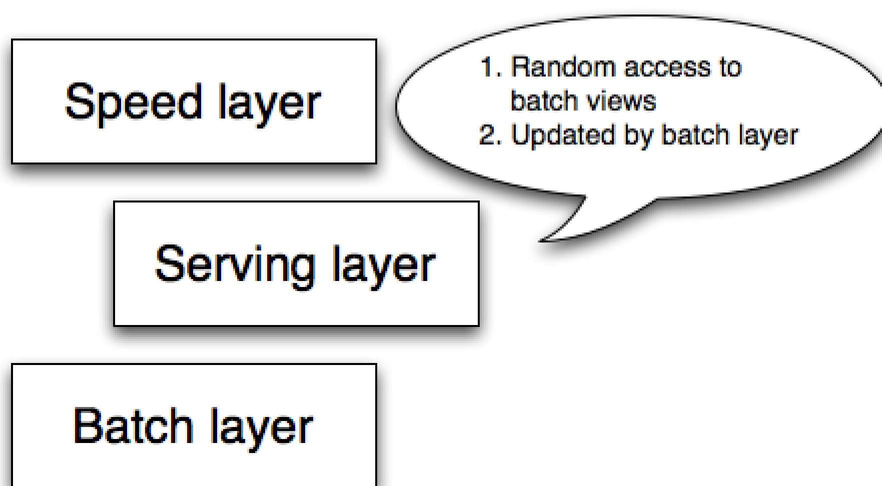

Figure 1.8 Serving layer

The serving layer makes views produced by the batch layer queryable. The serving layer is optimized for batch updates and low-latency reads. Like the batch layer,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

the serving layer easily scales by adding more machines. There are a few choices of databases to use for the serving layer, like Voldemort, HBase and ElephantDB, with different data models and query capabilities. We'll be using ElephantDB because it's the simplest to use and has facilities for doing incremental batch processing and recovering from mistakes. However, there are tradeoffs to consider when choosing a serving layer database that we will discuss in depth in Chapter 4.

The serving layer is tied to the batch layer and updates when the batch layer finishes updating the views. Typically this means the serving layer will update every few hours.

1.6.3 Batch and serving layers satisfy almost all properties

The batch and serving layers support arbitrary queries on an arbitrary dataset with the tradeoff that queries will be out of date by a few hours. It takes a new piece of data a few hours for the data 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 we desire in a Big Data system as outlined in Section 1.2. Let's go through them one by one:

- *Robust and fault tolerant:* 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, since when a mistake is made you can fix our algorithm or remove the bad data and recompute the views from scratch.
- *Scalable:* Both the batch layer and serving layers are easily scalable. Just add new machines and they will be taken advantage of automatically.
- *General:* The architecture described is as general as it gets. You can compute and update arbitrary views of an arbtirary dataset.
- *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.
- *Minimal maintenance:*The main component to maintain in this system is Hadoop. Hadoop requires some administration knowledge, but is fairly

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

straightforward to operate. The serving layer databases are simple because they don't do random writes. Random writes causes most of the complexity in a database. 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 you get parallelism for free. The only property we're missing is low latency updates. The final layer, the speed layer, fixes this problem.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

1.6.4 Speed layer

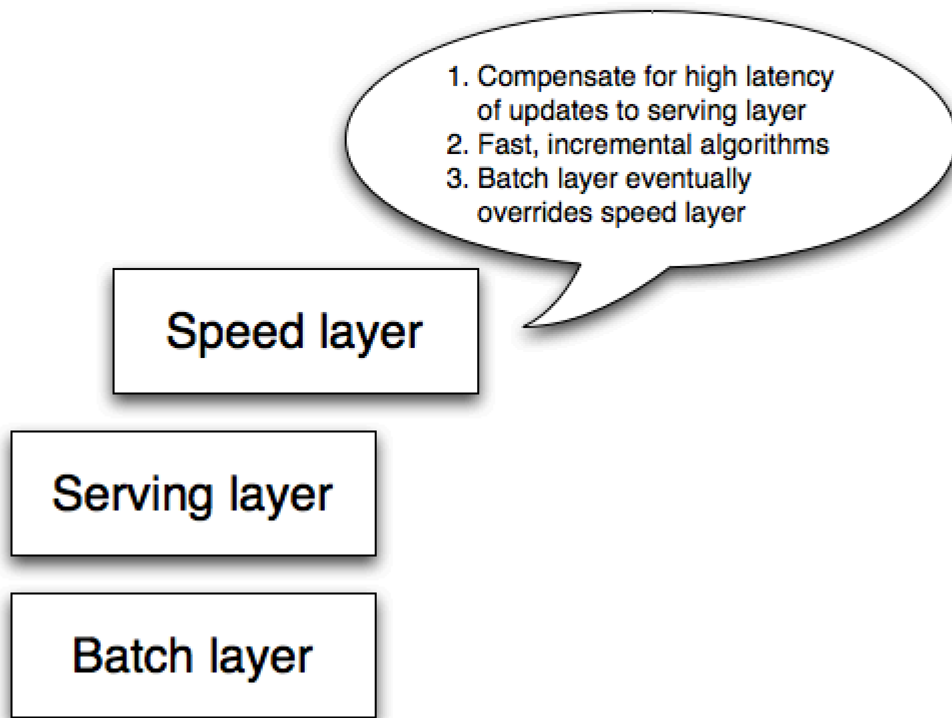

Figure 1.9 Speed layer

The speed layer exists to compensate for the high latency of updates to the serving layer. The trick, and one of the cruxes of this book, is that the speed layer only needs to compensate for data that hasn't made it into the serving layer yet. This means that the speed layer only needs to handle hours of data rather than years of data! Additionally, you will be able to throw data away from the speed layer once it's available in the serving layer.

The speed layer tends to use complex, highly incremental algorithms to achieve low latency updates. The beauty is that this complexity is all transient since the serving layer eventually replaces computations done in the speed layer.

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 constantly 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

challenging if the sets of uniques get large. It's easy to do the unique count on the batch layer since you look at all the data at once, but on the speed layer you might use a bloom filter[3]as an approximation.

Footnote 3 Bloom filters are a compact set representation that sometimes has false positives. Unique counts computed using a bloom filter will therefore be approximate.

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" since 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 does not affect the robustness of our results. The transient nature of the speed layer gives you the flexibility to be very aggressive when it comes to making tradeoffs for performance. Of course, for computations that can be done exactly in an incremental fashion the system is fully accurate.

Incremental algorithms cause a lot of complexity: they interact poorly with the CAP theorem, they cause corruption when a mistake is made, and when things go wrong they can be hard to debug. Additionally, the complexity is transient since the batch layer constantly overrides the speed layer. This "complexity isolation" is one of the greatest strengths of the layered architecture.

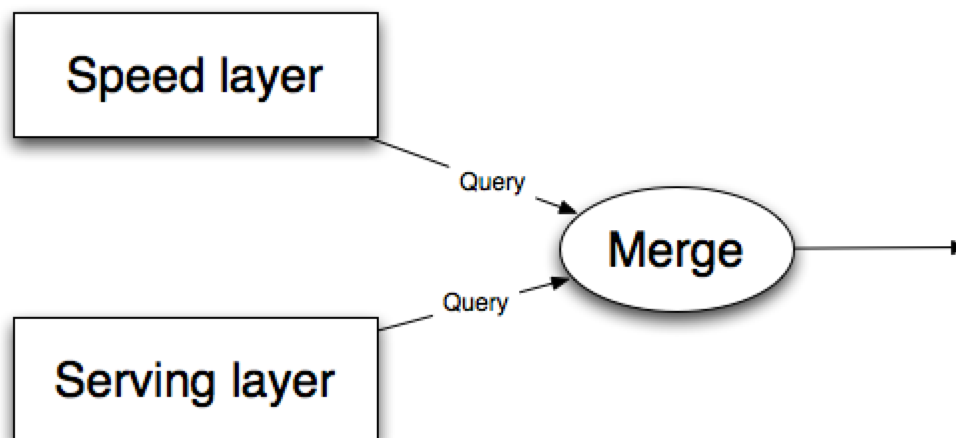

Figure 1.10 Satisfying application queries

The model we will use in the speed layer is that of "stream processing." Whereas in the batch layer we do computations on large, fixed-size datasets, in

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

stream processing you process an infinite stream of messages one message at a time. The complexity of the speed layer comes from using highly incremental algorithms and needing to engineer for databases that aren't fully consistent.

To satisfy a query from the application, you query the results from the serving layer and speed layer and merge them to get the final result. This is shown in Figure 1.10

1.7 A glimpse into the Big Data way

We have a lot of ground to cover to learn the tools and techniques for building Big Data applications. Every aspect of how you manage your data -- storage, schemafication, processing, and serving -- will most likely be different from how you built applications in the past.

In the meantime, let's get a taste for what these Big Data applications look like. The code snippet in the following listing computes the number of pageviews for each url given an arbitrarily sized collection of raw pageviews. This code reads from records that are formatted in text format: every line of a text file contains a single url. This is *not* how we will store data in practice, but this is the simplest way to show the processing code at this point.

Listing 1.1 Counting pageviews per url in parallel

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

What's interesting about this code is that you get parallelism "for free." 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 in text format.

The core of our Big Data systems will be batch computations written like this. Since you get the parallelism for free, and there are no concurrency issues to worry

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

about, these computations are easy to write. The drawback to batch computation is that it has high latency -- a batch workflow that updates the databases that serve the application will take hours to complete. However, we will be able to achieve low latency on updates without sacrificing the simplicity of the batch approach. The speed layer will utilize more complex algorithms to compensate for the high latency of batch computation.

1.8 Benefits of Big Data systems

The architecture described in this book does for more than just scale existing systems. Your systems will acquire some new capabilities as compared to traditional relational databases and will have simpler operational properties.

Benefit #1: You will be able collect more data and get more value out of your data. These Big Data systems are easily scalable. The powerful capabilities to store and process data they provide will lead you to collect even more data. Increasing the amount and types of data your store will lead to more opportunities to mine your data, produce analytics, and build new applications.

Benefit #2: Applications will be more robust. 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 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 we'll explore, there are many other reasons why your Big Data applications will be more robust.

Benefit #3: Performance will be more predictable. Although these Big Data systems as a whole are 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.

1.9 Recent trends in technology

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

1.9.1 Commoditization of Hardware

In terms of price/performance tradeoff, you get the most bang for your buck using off-the-shelf components. This means that it's better to scale by adding more machines rather than scaling by buying better hardware. The first appoach is called "horizontal scaling", and the second approach is called "vertical scaling." Almost all the systems we will be using are horizontally scalable.

1.9.2 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 near instantaneously, so if you have a big job you want to run you can allocate the hardware temporarily.

Elastic clouds drastically 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 will 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 are a great option because fault-tolerance is handled at the software layer.

1.9.3 Open Source

The open source community has created a plethora of Big Data technologies over the past few years. All the technologies we will be using are open-source and free to use.

There are five categories of open source projects we will be using:

1. 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 will be using is Hadoop. The Hadoop project has two subprojects: the Hadoop Distributed Filesystem (HDFS) and Hadoop MapReduce. HDFS is a distributed fault-tolerant storage system and can scale to petabytes of data. MapReduce is a horizontally scalable computation framework that integrates with HDFS.

2. Serialization frameworks: Serialization frameworks provide tools and

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

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 notable serialization frameworks are Thrift, Protocol Buffers, and Avro. We will be using Thrift, but any of them could be used.

3. Random-access NoSQL databases: There have been a plethora of "NoSQL" databases created 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 are *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 choose appropriately. We will be using Cassandra as part of the example application we'll be building.

4. 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 will be using Kestrel in this book, a message queue developed at Twitter, because it's the simplest of the queues to operate and use that still has all the features you require for realtime computation.

5. 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 will be using Storm in this book (written by the author). 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 of 10gen, and RabbitMQ is a product of SpringSource, a division of VMWare.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787

1.10 Our example application: SuperWebAnalytics.com

We will be building an example Big Data application throughout this book to illustrate the concepts. We will be building the data management layer for a Google Analytics like service. The service will be able to track billions of page views 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.

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.

1.11 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 we build things the right way, both robustness and scalability are achievable in the same system.

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 we will start learning how to build a layered architecture. We will start at the very core of the stack with how you model and schemify the master copy of your dataset.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=787