<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A new paradigm for Big Data</td>
<td>1</td>
</tr>
<tr>
<td><strong>PART 1</strong></td>
<td><strong>BATCH LAYER</strong></td>
<td><strong>25</strong></td>
</tr>
<tr>
<td>2</td>
<td>Data model for Big Data</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>Data model for Big Data: Illustration</td>
<td>47</td>
</tr>
<tr>
<td>4</td>
<td>Data storage on the batch layer</td>
<td>54</td>
</tr>
<tr>
<td>5</td>
<td>Data storage on the batch layer: Illustration</td>
<td>65</td>
</tr>
<tr>
<td>6</td>
<td>Batch layer</td>
<td>83</td>
</tr>
<tr>
<td>7</td>
<td>Batch layer: Illustration</td>
<td>111</td>
</tr>
<tr>
<td>8</td>
<td>An example batch layer: Architecture and algorithms</td>
<td>139</td>
</tr>
<tr>
<td>9</td>
<td>An example batch layer: Implementation</td>
<td>156</td>
</tr>
<tr>
<td><strong>PART 2</strong></td>
<td><strong>SERVING LAYER</strong></td>
<td><strong>177</strong></td>
</tr>
<tr>
<td>10</td>
<td>Serving layer</td>
<td>179</td>
</tr>
<tr>
<td>11</td>
<td>Serving layer: Illustration</td>
<td>196</td>
</tr>
</tbody>
</table>
## BRIEF CONTENTS

**PART 3  SPEED LAYER .............................................................205**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Realtime views</td>
<td>207</td>
</tr>
<tr>
<td>13</td>
<td>Realtime views: Illustration</td>
<td>220</td>
</tr>
<tr>
<td>14</td>
<td>Queuing and stream processing</td>
<td>225</td>
</tr>
<tr>
<td>15</td>
<td>Queuing and stream processing: Illustration</td>
<td>242</td>
</tr>
<tr>
<td>16</td>
<td>Micro-batch stream processing</td>
<td>254</td>
</tr>
<tr>
<td>17</td>
<td>Micro-batch stream processing: Illustration</td>
<td>269</td>
</tr>
<tr>
<td>18</td>
<td>Lambda Architecture in depth</td>
<td>284</td>
</tr>
</tbody>
</table>
Part 1 focuses on the batch layer of the Lambda Architecture. Chapters alternate between theory and illustration.

Chapter 2 discusses how you model and schematize the data in your master dataset. Chapter 3 illustrates these concepts using the tool Apache Thrift.

Chapter 4 discusses the requirements for storage of your master dataset. You’ll see that many features typically provided by database solutions are not needed for the master dataset, and in fact get in the way of optimizing master dataset storage. A simpler and less feature-full storage solution meets the requirements better. Chapter 5 illustrates practical storage of a master dataset using the Hadoop Distributed Filesystem.

Chapter 6 discusses computing arbitrary functions on your master dataset using the MapReduce paradigm. MapReduce is general enough to compute any scalable function. Although MapReduce is powerful, you’ll see that higher-level abstractions make it far easier to use. Chapter 7 shows a powerful high-level abstraction to MapReduce called JCascalog.

To connect all the concepts together, chapters 8 and 9 implement the complete batch layer for the running example SuperWebAnalytics.com. Chapter 8 shows the overall architecture and algorithms, while chapter 9 shows the working code in all its details.
In the last chapter you saw what can go wrong when using traditional tools for building data systems, and we went back to first principles to derive a better design. You saw that every data system can be formulated as computing functions on data, and you learned the basics of the Lambda Architecture, which provides a practical way to implement an arbitrary function on arbitrary data in real time.

At the core of the Lambda Architecture is the master dataset, which is highlighted in figure 2.1. The master dataset is the source of truth in the Lambda Architecture. Even if you were to lose all your serving layer datasets and speed layer datasets, you could reconstruct your application from the master dataset. This is because the batch views served by the serving layer are produced via functions on the master dataset, and since the speed layer is based only on recent data, it can construct itself within a few hours.

The master dataset is the only part of the Lambda Architecture that absolutely must be safeguarded from corruption. Overloaded machines, failing disks, and
The data in the speed layer realtime views has a high turnover rate, so any errors are quickly expelled.

The master dataset is the source of truth in your system and cannot withstand corruption.

Any errors introduced into the serving layer batch views are overwritten because they are continually rebuilt from the master dataset.

Figure 2.1  The master dataset in the Lambda Architecture serves as the source of truth for your Big Data system. Errors at the serving and speed layers can be corrected, but corruption of the master dataset is irreparable.

power outages all could cause errors, and human error with dynamic data systems is an intrinsic risk and inevitable eventuality. You must carefully engineer the master dataset to prevent corruption in all these cases, as fault tolerance is essential to the health of a long-running data system.

There are two components to the master dataset: the data model you use and how you physically store the master dataset. This chapter is about designing a data model for the master dataset and the properties such a data model should have. You’ll learn about physically storing a master dataset in the next chapter.

In this chapter you’ll do the following:

- Learn the key properties of data
- See how these properties are maintained in the fact-based model
- Examine the advantages of the fact-based model for the master dataset
- Express a fact-based model using graph schemas

Let’s begin with a discussion of the rather general term data.
2.1 **The properties of data**

In keeping with the applied focus of the book, we’ll center our discussion around an example application. Suppose you’re designing the next big social network—FaceSpace. When a new user—let’s call him Tom—joins your site, he starts to invite his friends and family. What information should you store regarding Tom’s connections? You have a number of choices, including the following:

- The sequence of Tom’s friend and unfriend events
- Tom’s current list of friends
- Tom’s current number of friends

Figure 2.2 exhibits these options and their relationships.

This example illustrates information dependency. Note that each layer of information can be derived from the previous one (the one to its left), but it’s a one-way process. From the sequence of friend and unfriend events, you can determine the other quantities. But if you only have the number of friends, it’s impossible to determine exactly who they are. Similarly, from the list of current friends, it’s impossible to determine if Tom was previously a friend with Jerry, or whether Tom’s network has been growing as of late.

The notion of dependency shapes the definitions of the terms we’ll use:

- **Information** is the general collection of knowledge relevant to your Big Data system. It’s synonymous with the colloquial usage of the word *data*.
- **Data** refers to the information that can’t be derived from anything else. Data serves as the axioms from which everything else derives.
- **Queries** are questions you ask of your data. For example, you query your financial transaction history to determine your current bank account balance.
- **Views** are information that has been derived from your base data. They are built to assist with answering specific types of queries.

![Figure 2.2](image-url)  
Figure 2.2 Three possible options for storing friendship information for FaceSpace. Each option can be derived from the one to its left, but it’s a one-way process.
Figure 2.3 The relationships between data, views, and queries

Figure 2.3 illustrates the FaceSpace information dependency in terms of data, views, and queries.

It’s important to observe that one person’s data can be another’s view. Suppose FaceSpace becomes a monstrous hit, and an advertising firm creates a crawler that scrapes demographic information from user profiles. FaceSpace has complete access to all the information Tom provided—for example, his complete birthdate of March 13, 1984. But Tom is sensitive about his age, and he only makes his birthday (March 13) available on his public profile. His birthday is a view from FaceSpace’s perspective because it’s derived from his birthdate, but it’s data to the advertiser because they have limited information about Tom. This relationship is shown in figure 2.4.

Having established a shared vocabulary, we can now introduce the key properties of data: rawness, immutability, and perpetuity (or the “eternal trueness of data”).

Figure 2.4 Classifying information as data or a view depends on your perspective. To FaceSpace, Tom’s birthday is a view because it’s derived from the user’s birthdate. But the birthday is considered data to a third-party advertiser.
Foundational to your understanding of Big Data systems is your understanding of these three key concepts.

If you’re coming from a relational background, this could be confusing. Typically you constantly update and summarize your information to reflect the current state of the world; you’re not concerned with immutability or perpetuity. But that approach limits the questions you can answer with your data, and it fails to robustly discourage errors and corruption. By enforcing these properties in the world of Big Data, you achieve a more robust system and gain powerful capabilities.

We’ll delve further into this topic as we discuss the rawness of data.

### 2.1.1 Data is raw

A data system answers questions about information you’ve acquired in the past. When designing your Big Data system, you want to be able to answer as many questions as possible. In the FaceSpace example, your FaceSpace data is more valuable than the advertiser’s because you can deduce more information about Tom. We’ll colloquially call this property *rawness*. If you can, you want to store the rawest data you can get your hands on. The rawer your data, the more questions you can ask of it.

The FaceSpace example helps illustrate the value of rawness, but let’s consider another example to help drive the point home. Stock market trading is a fountain of information, with millions of shares and billions of dollars changing hands on a daily basis. With so many trades taking place, stock prices are historically recorded daily as an opening price, high price, low price, and closing price. But those bits of data often don’t provide the big picture and can potentially skew your perception of what happened. For instance, look at figure 2.5. It records the price data for Google, Apple, and Amazon stocks on a day when Google announced new products targeted at their competitors.

This data suggests that Amazon may not have been affected by Google’s announcement, as its stock price moved only slightly. It also suggests that the announcement had either no effect on Apple, or a positive effect.

But if you have access to data stored at a finer time granularity, you can get a clearer picture of the events on that day and probe further into potential cause and effect.

<table>
<thead>
<tr>
<th>Company</th>
<th>Symbol</th>
<th>Previous</th>
<th>Open</th>
<th>High</th>
<th>Low</th>
<th>Close</th>
<th>Net</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google</td>
<td>GOOG</td>
<td>564.68</td>
<td>567.70</td>
<td>573.99</td>
<td>566.02</td>
<td>569.30</td>
<td>+4.62</td>
</tr>
<tr>
<td>Apple</td>
<td>AAPL</td>
<td>572.02</td>
<td>575.00</td>
<td>576.74</td>
<td>571.92</td>
<td>574.50</td>
<td>+2.48</td>
</tr>
<tr>
<td>Amazon</td>
<td>AMZN</td>
<td>225.61</td>
<td>225.01</td>
<td>227.50</td>
<td>223.30</td>
<td>225.62</td>
<td>+0.01</td>
</tr>
</tbody>
</table>

Financial reporting promotes daily net change in closing prices. What conclusions would you draw about the impact of Google’s announcements?

*Figure 2.5* A summary of one day of trading for Google, Apple, and Amazon stocks: previous close, opening, high, low, close, and net change.
Apple held steady throughout the day. Google’s stock price had a slight boost on the day of the announcement.

Amazon’s stock dipped in late-day trading.

**Figure 2.6** Relative stock price changes of Google, Apple, and Amazon on June 27, 2012, compared to closing prices on June 26 (www.google.com/finance). Short-term analysis isn’t supported by daily records but can be performed by storing data at finer time resolutions.

effect relationships. Figure 2.6 depicts the minute-by-minute relative changes in the stock prices of all three companies, which suggests that both Amazon and Apple were indeed affected by the announcement, Amazon more so than Apple.

Also note that the additional data can suggest new ideas you may not have considered when examining the original daily stock price summary. For instance, the more granular data makes you wonder if Amazon was more greatly affected because the new Google products compete with Amazon in both the tablet and cloud-computing markets.

Storing raw data is hugely valuable because you rarely know in advance all the questions you want answered. By keeping the rawest data possible, you maximize your ability to obtain new insights, whereas summarizing, overwriting, or deleting information limits what your data can tell you. The trade-off is that rawer data typically entails more of it—sometimes much more. But Big Data technologies are designed to manage petabytes and exabytes of data. Specifically, they manage the storage of your data in a distributed, scalable manner while supporting the ability to directly query the data.

Although the concept is straightforward, it’s not always clear what information you should store as your raw data. We’ll provide a couple of examples to help guide you in making this decision.

**UNSTRUCTURED DATA IS RAWER THAN NORMALIZED DATA**

When deciding what raw data to store, a common hazy area is the line between *parsing* and *semantic normalization*. Semantic normalization is the process of reshaping free-form information into a structured form of data.
For example, FaceSpace may request Tom’s location. He may input anything for that field, such as *San Francisco, CA, SF, North Beach*, and so forth. A semantic normalization algorithm would try to match the input with a known place—see figure 2.7.

If you come across a form of data such as an unstructured location string, should you store the unstructured string or the semantically normalized form? We argue that it’s better to store the unstructured string, because your semantic normalization algorithm may improve over time. If you store the unstructured string, you can renormalize that data at a later time when you have improved your algorithms. In the preceding example, you may later adapt the algorithm to recognize North Beach as a neighborhood in San Francisco, or you may want to use the neighborhood information for another purpose.

**STORE UNSTRUCTURED DATA WHEN...**  
As a rule of thumb, if your algorithm for extracting the data is simple and accurate, like extracting an age from an HTML page, you should store the results of that algorithm. If the algorithm is subject to change, due to improvements or broadening the requirements, store the unstructured form of the data.

**MORE INFORMATION DOESN’T NECESSARILY MEAN RAWER DATA**

It’s easy to presume that more data equates to rawer data, but that’s not always the case. Let’s say that Tom is a blogger, and he wants to add his posts to his FaceSpace profile. What exactly should you store once Tom provides the URL of his blog?

Storing the pure text of the blog entries is certainly a possibility. But any phrases in italics, boldface, or large font were deliberately emphasized by Tom and could prove useful in text analysis. For example, you could use this additional information for an index to make FaceSpace searchable. We’d thus argue that the annotated text entries are a rawer form of data than ASCII text strings.

At the other end of the spectrum, you could also store the full HTML of Tom’s blog as your data. While it’s considerably more information in terms of total bytes, the color scheme, stylesheets, and JavaScript code of the site can’t be used to derive any additional information about Tom. They serve only as the container for the contents of the site and shouldn’t be part of your raw data.
2.1.2 Data is immutable

Immutable data may seem like a strange concept if you’re well versed in relational databases. After all, in the relational database world—and most other databases as well—update is one of the fundamental operations. But for immutability you don’t update or delete data, you only add more. By using an immutable schema for Big Data systems, you gain two vital advantages:

- **Human-fault tolerance**—This is the most important advantage of the immutable model. As we discussed in chapter 1, human-fault tolerance is an essential property of data systems. People will make mistakes, and you must limit the impact of such mistakes and have mechanisms for recovering from them. With a mutable data model, a mistake can cause data to be lost, because values are actually overridden in the database. With an immutable data model, no data can be lost. If bad data is written, earlier (good) data units still exist. Fixing the data system is just a matter of deleting the bad data units and recomputing the views built from the master dataset.

- **Simplicity**—Mutable data models imply that the data must be indexed in some way so that specific data objects can be retrieved and updated. In contrast, with an immutable data model you only need the ability to append new data units to the master dataset. This doesn’t require an index for your data, which is a huge simplification. As you’ll see in the next chapter, storing a master dataset is as simple as using flat files.

The advantages of keeping your data immutable become evident when comparing with a mutable schema. Consider the basic mutable schema shown in figure 2.8, which you could use for FaceSpace.

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>age</th>
<th>gender</th>
<th>employer</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>25</td>
<td>female</td>
<td>Apple</td>
<td>Atlanta, GA</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>36</td>
<td>male</td>
<td>SAS</td>
<td>Chicago, IL</td>
</tr>
<tr>
<td>3</td>
<td>Tom</td>
<td>28</td>
<td>male</td>
<td>Google</td>
<td>San Francisco, CA</td>
</tr>
<tr>
<td>4</td>
<td>Charlie</td>
<td>25</td>
<td>male</td>
<td>Microsoft</td>
<td>Washington, DC</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

*Figure 2.8 A mutable schema for FaceSpace user information. When details change—say, Tom moves to Los Angeles—previous values are overwritten and lost.*

---

1 There are a few scenarios in which you can delete data, but these are special cases and not part of the day-to-day workflow of your system. We’ll discuss these scenarios in section 2.1.3.
Should Tom move to Los Angeles, you’d update the highlighted entry to reflect his current location—but in the process, you’d also lose all knowledge that Tom ever lived in San Francisco.

With an immutable schema, things look different. Rather than storing a current snapshot of the world, as done by the mutable schema, you create a separate record every time a user’s information evolves. Accomplishing this requires two changes. First, you track each field of user information in a separate table. Second, you tie each unit of data to a moment in time when the information is known to be true. Figure 2.9 shows a corresponding immutable schema for storing FaceSpace information.

Tom first joined FaceSpace on April 4, 2012, and provided his profile information. The time you first learn this data is reflected in the record’s timestamp. When he subsequently moves to Los Angeles on June 17, 2012, you add a new record to the location table, timestamped by when he changed his profile—see figure 2.10.

You now have two location records for Tom (user ID #3), and because the data units are tied to particular times, they can both be true. Tom’s current location involves a simple query on the data: look at all the locations, and pick the one with the most recent timestamp. By keeping each field in a separate table, you only record the information that changed. This requires less space for storage and guarantees that each record is new information and is not simply carried over from the last record.

One of the trade-offs of the immutable approach is that it uses more storage than a mutable schema. First, the user ID is specified for every property, rather than just once per row, as with a mutable approach. Additionally, the entire history of events is stored rather than just the current view of the world. But Big Data isn’t called “Big Data” for
nothing. You should take advantage of the ability to store large amounts of data using Big Data technologies to get the benefits of immutability. The importance of having a simple and strongly human-fault tolerant master dataset can’t be overstated.

### 2.1.3 Data is eternally true

The key consequence of immutability is that each piece of data is true in perpetuity. That is, a piece of data, once true, must always be true. Immutability wouldn’t make sense without this property, and you saw how tagging each piece of data with a timestamp is a practical way to make data eternally true.

This mentality is the same as when you learned history in school. The fact *The United States consisted of thirteen states on July 4, 1776,* is always true due to the specific date; the fact that the number of states has increased since then is captured in additional (also perpetual) data.

In general, your master dataset consistently grows by adding new immutable and eternally true pieces of data. There are some special cases, though, in which you do delete data, and these cases are not incompatible with data being eternally true. Let’s consider the cases:

- **Garbage collection**—When you perform garbage collection, you delete all data units that have low value. You can use garbage collection to implement data-retention policies that control the growth of the master dataset. For example, you may decide to implement a policy that keeps only one location per person per year instead of the full history of each time a user changes locations.

- **Regulations**—Government regulations may require you to purge data from your databases under certain conditions.

In both of these cases, deleting the data is not a statement about the truthfulness of the data. Instead, it’s a statement about the value of the data. Although the data is eternally true, you may prefer to “forget” the information either because you must or because it doesn’t provide enough value for the storage cost.

We’ll proceed by introducing a data model that uses these key properties of data.
Deleting immutable data?
You may be wondering how it is possible to delete immutable data. On the face of it, this seems like a contradiction. It is important to distinguish that the deleting we are referring to is a special and rare case. In normal usage, data is immutable, and you enforce that property by taking actions such as setting the appropriate permissions. Since deleting data is rare, the utmost care can be taken to ensure that it is done safely. We believe deleting data is most safely accomplished by producing a second copy of the master dataset with the offending data filtered out, running analytic jobs to verify that the correct data was filtered, and then and only then replacing the old version of the master dataset.

2.2 The fact-based model for representing data
Data is the set of information that can’t be derived from anything else, but there are many ways you could choose to represent it within the master dataset. Besides traditional relational tables, structured XML and semistructured JSON documents are other possibilities for storing data. We, however, recommend the fact-based model for this purpose. In the fact-based model, you deconstruct the data into fundamental units called (unsurprisingly) facts. In the discussion of immutability, you got a glimpse of the fact-based model, in that the master dataset continually grows with the addition of immutable, timestamped data. We’ll now expand on what we already discussed to explain the fact-based model in full. We’ll first introduce the model in the context of the FaceSpace example and discuss its basic properties. We’ll then continue with discussing how and why you should make your facts identifiable. To wrap up, we’ll explain the benefits of using the fact-based model and why it’s an excellent choice for your master dataset.

2.2.1 Example facts and their properties
Figure 2.11 depicts examples of facts about Tom from the FaceSpace data, as well as two core properties of facts—they are atomic and timestamped.

Facts are atomic because they can’t be subdivided further into meaningful components. Collective data, such as Tom’s friend list in the figure, are represented as multiple, independent facts. As a consequence of being atomic, there’s no redundancy of information across distinct facts. Facts having timestamps should come as no surprise, given our earlier discussion about data—the timestamps make each fact immutable and eternally true.

These properties make the fact-based model a simple and expressive model for your dataset, yet there is an additional property we recommend imposing on your facts: identifiability.

Making facts identifiable
Besides being atomic and timestamped, facts should be associated with a uniquely identifiable piece of data. This is most easily explained by example.
Suppose you want to store data about pageviews on FaceSpace. Your first approach might look something like this (in pseudo-code):

```c++
struct PageView {
    DateTime timestamp
    String url
    String ip_address
}
```

Facts using this structure don’t uniquely identify a particular pageview event. If multiple pageviews come in at the same time for the same URL from the same IP address, each pageview will have the exact same data record. Consequently, if you encounter two identical pageview records, there’s no way to tell whether they refer to two distinct events or if a duplicate entry was accidentally introduced into your dataset.

To distinguish different pageviews, you can add a nonce to your schema—a 64-bit number randomly generated for each pageview:

```c++
struct PageView {
    Datetime timestamp
    String url
    String ip_address
    Long nonce
}
```

The addition of the nonce makes it possible to distinguish pageview events from each other, and if two pageview data units are identical (all fields, including the nonce), you know they refer to the exact same event.

Making facts identifiable means that you can write the same fact to the master dataset multiple times without changing the semantics of the master dataset. Your queries can filter out the duplicate facts when doing their computations. As it turns out, and as you’ll see later, having distinguishable facts makes implementing the rest of the Lambda Architecture much easier.
Duplicates aren’t as rare as you might think
At a first look, it may not be obvious why we care so much about identity and duplicates. After all, to avoid duplicates, the first inclination would be to ensure that an event is recorded just once. Unfortunately life isn’t always so simple when dealing with Big Data.

Once FaceSpace becomes a hit, it will require hundreds, then thousands, of web servers. Building the master dataset will require aggregating the data from each of these servers to a central system—no trivial task. There are data collection tools suitable for this situation—Facebook’s Scribe, Apache Flume, syslog-ng, and many others—but any solution must be fault tolerant.

One common “fault” these systems must anticipate is a network partition where the destination datastore becomes unavailable. For these situations, fault-tolerant systems commonly handle failed operations by retrying until they succeed. Because the sender will not know which data was last received, a standard approach is to resend all data yet to be acknowledged by the recipient. But if part of the original attempt did make it to the metastore, you’d end up with duplicates in your dataset.

There are ways to make these kinds of operations transactional, but it can be fairly tricky and entail performance costs. An important part of ensuring correctness in your systems is avoiding tricky solutions. By embracing distinguishable facts, you remove the need for transactional appends to the master dataset and make it easier to reason about the correctness of the full system. After all, why place difficult burdens on yourself when a small tweak to your data model can avoid those challenges altogether?

To quickly recap, the fact-based model
- Stores your raw data as atomic facts
- Keeps the facts immutable and eternally true by using timestamps
- Ensures each fact is identifiable so that query processing can identify duplicates

Next we’ll discuss the benefits of choosing the fact-based model for your master dataset.

2.2.2 Benefits of the fact-based model
With a fact-based model, the master dataset will be an ever-growing list of immutable, atomic facts. This isn’t a pattern that relational databases were built to support—if you come from a relational background, your head may be spinning. The good news is that by changing your data model paradigm, you gain numerous advantages. Specifically, your data
- Is queryable at any time in its history
- Tolerates human errors
- Handles partial information
- Has the advantages of both normalized and denormalized forms

Let’s look at each of these advantages in turn.
THE DATASET IS QUERYABLE AT ANY TIME IN ITS HISTORY
Instead of storing only the current state of the world, as you would using a mutable, relational schema, you have the ability to query your data for any time covered by your dataset. This is a direct consequence of facts being timestamped and immutable. “Updates” and “deletes” are performed by adding new facts with more recent timestamps, but because no data is actually removed, you can reconstruct the state of the world at the time specified by your query.

THE DATA IS HUMAN-FAULT TOLERANT
Human-fault tolerance is achieved by simply deleting any erroneous facts. Suppose you mistakenly stored that Tom moved from San Francisco to Los Angeles—see figure 2.12.

By removing the Los Angeles fact, Tom’s location is automatically “reset” because the San Francisco fact becomes the most recent information.

THE DATASET EASILY HANDLES PARTIAL INFORMATION
Storing one fact per record makes it easy to handle partial information about an entity without introducing NULL values into your dataset. Suppose Tom provided his age and gender but not his location or profession. Your dataset would only have facts for the known information—any “absent” fact would be logically equivalent to NULL. Additional information that Tom provides at a later time would naturally be introduced via new facts.

THE DATA STORAGE AND QUERY PROCESSING LAYERS ARE SEPARATE
There is another key advantage of the fact-based model that is in part due to the structure of the Lambda Architecture itself. By storing the information at both the batch and serving layers, you have the benefit of keeping your data in both normalized and denormalized forms and reaping the benefits of both.

NORMALIZATION IS AN OVERLOADED TERM
Data normalization is completely unrelated to the semantic normalization term that we used earlier. In this case, data normalization refers to storing data in a structured manner to minimize redundancy and promote consistency.

<table>
<thead>
<tr>
<th>user id</th>
<th>location</th>
<th>timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Atlanta, GA</td>
<td>2012/03/29 08:12:24</td>
</tr>
<tr>
<td>2</td>
<td>Chicago, IL</td>
<td>2012/04/12 14:47:51</td>
</tr>
<tr>
<td>3</td>
<td>San Francisco, CA</td>
<td>2012/04/04 18:31:24</td>
</tr>
<tr>
<td>4</td>
<td>Washington, DC</td>
<td>2012/04/09 11:52:30</td>
</tr>
<tr>
<td>3</td>
<td>Los Angeles, CA</td>
<td>2012/06/17 20:09:48</td>
</tr>
</tbody>
</table>

Figure 2.12 To correct for human errors, simply remove the incorrect facts. This process automatically resets to an earlier state by “uncovering” any relevant previous facts.
The fact-based model for representing data

Data in this table is denormalized because the same information is stored redundantly—in this case, the company name can be repeated.

With this table, you can quickly determine the number of employees at each company, but many rows must be updated when change occurs—in this case, when BackRub changed to Google.

Figure 2.13 A simple denormalized schema for storing employment information

Let’s set the stage with an example involving relational tables—the context where data normalization is most frequently encountered. Relational tables require you to choose between normalized and denormalized schemas based on what’s most important to you: query efficiency or data consistency. Suppose you wanted to store the employment information for various people of interest. Figure 2.13 offers a simple denormalized schema suitable for this purpose.

In this denormalized schema, the same company name could potentially be stored in multiple rows. This would allow you to quickly determine the number of employees for each company, but you would need to update many rows should a company change its name. Having information stored in multiple locations increases the risk of it becoming inconsistent.

In comparison, consider the normalized schema in figure 2.14.

Data in a normalized schema is stored in only one location. If BackRub should change its name to Google, there’s a single row in the Company table that needs to be altered. This removes the risk of inconsistency, but you must join the tables to answer queries—a potentially expensive computation.

For normalized data, each fact is stored in only one location and relationships between datasets are used to answer queries. This simplifies the consistency of data, but joining tables could be expensive.

Figure 2.14 Two normalized tables for storing the same employment information
The mutually exclusive choice between normalized and denormalized schemas is necessary because, for relational databases, queries are performed directly on the data at the storage level. You must therefore weigh the importance of query efficiency versus data consistency and choose between the two schema types.

In contrast, the objectives of query processing and data storage are cleanly separated in the Lambda Architecture. Take a look at the batch and server layers in figure 2.15.

In the Lambda Architecture, the master dataset is fully normalized. As you saw in the discussion of the fact-based model, no data is stored redundantly. Updates are easily handled because adding a new fact with a current timestamp “overrides” any previous related facts.

Similarly, the batch views are like denormalized tables in that one piece of data from the master dataset may get indexed into many batch views. The key difference is that the batch views are defined as functions on the master dataset. Accordingly, there is no need to update a batch view because it will be continually rebuilt from the master dataset. This has the additional benefit that the batch views and master dataset will never be out of sync. The Lambda Architecture gives you the conceptual benefits of full normalization with the performance benefits of indexing data in different ways to optimize queries.

In summary, all of these benefits make the fact-based model an excellent choice for your master dataset. But that’s enough discussion at the theoretical level—let’s dive into the details of practically implementing a fact-based data model.

Figure 2.15 The Lambda Architecture has the benefits of both normalization and denormalization by separating objectives at different layers.
2.3 **Graph schemas**

Each fact within a fact-based model captures a single piece of information. But the facts alone don’t convey the structure behind the data. That is, there’s no description of the types of facts contained in the dataset, nor any explanation of the relationships between them. In this section we’ll introduce **graph schemas**—graphs that capture the structure of a dataset stored using the fact-based model. We’ll discuss the elements of a graph schema and the need to make a schema enforceable.

Let’s begin by first structuring our FaceSpace facts as a graph.

### 2.3.1 Elements of a graph schema

In the last section we discussed FaceSpace facts in great detail. Each fact represents either a piece of information about a user or a relationship between two users. Figure 2.16 depicts a **graph schema** representing the relationships between the FaceSpace facts. It provides a useful visualization of your users, their individual information, and the friendships between them.

The figure highlights the three core components of a graph schema—**nodes**, **edges**, and **properties**:

- **Nodes are the entities in the system.** In this example, the nodes are the FaceSpace users, represented by a user ID. As another example, if FaceSpace allows users to identify themselves as part of a group, then the groups would also be represented by nodes.

- **Edges are relationships between nodes.** The connotation in FaceSpace is straightforward—an edge between users represents a FaceSpace friendship. You could...
later add additional edge types between users to identify coworkers, family members, or classmates.

- Properties are information about entities. In this example, age, gender, location, and all other pieces of individual information are properties.

**EDGES ARE STRICTLY BETWEEN NODES** Even though properties and nodes are visually connected in the figure, these lines are not edges. They are present only to help illustrate the association between users and their personal information. We denote the difference by using solid lines for edges and dashed lines for property connections.

The graph schema provides a complete description of all the data contained within a dataset. Next we’ll discuss the need to ensure that all facts within a dataset rigidly adhere to the schema.

### 2.3.2 The need for an enforceable schema

At this point, information is stored as facts, and a graph schema describes the types of facts contained in the dataset. You’re all set, right? Well, not quite. You still need to decide in what format you’ll store your facts.

A first idea might be to use a semistructured text format like JSON. This would provide simplicity and flexibility, allowing essentially anything to be written to the master dataset. But in this case it’s too flexible for our needs.

To illustrate this problem, suppose you chose to represent Tom’s age using JSON:

```
{"id": 3, "field":"age", "value":28, "timestamp": 1333589484}
```

There are no issues with the representation of this single fact, but there’s no way to ensure that all subsequent facts will follow the same format. As a result of human error, the dataset could also possibly include facts like these:

```
{"name":"Alice", "field":"age", "value":25,  
 "timestamp":"2012/03/29 08:12:24"}
{"id":2, "field":"age", "value":36}
```

Both of these examples are valid JSON, but they have inconsistent formats or missing data. In particular, in the last section we stressed the importance of having a timestamp for each fact, but a text format can’t enforce this requirement. To effectively use your data, you must provide guarantees about the contents of your dataset.

The alternative is to use an enforceable schema that rigorously defines the structure of your facts. Enforceable schemas require a bit more work up front, but they guarantee all required fields are present and ensure all values are of the expected type. With these assurances, a developer will be confident about what data they can expect—that each fact will have a timestamp, that a user’s name will always be a string, and so forth. The key is that when a mistake is made creating a piece of data, an enforceable schema will give errors at that time, rather than when someone is trying
to use the data later in a different system. The closer the error appears to the bug, the easier it is to catch and fix.

In the next chapter you’ll see how to implement an enforceable schema using a serialization framework. A serialization framework provides a language-neutral way to define the nodes, edges, and properties of your schema. It then generates code (potentially in many different languages) that serializes and deserializes the objects in your schema so they can be stored in and retrieved from your master dataset.

We’re aware that at this point you may be hungry for details. Not to worry—we believe the best way to learn is by doing. In the next section we’ll design the fact-based model for SuperWebAnalytics.com in its entirety, and in the following chapter we’ll implement it using a serialization framework.

### 2.4 A complete data model for SuperWebAnalytics.com

In this section we aim to tie together all the material from the chapter using the SuperWebAnalytics.com example. We’ll begin with figure 2.17, which contains a graph schema suitable for our purpose.

In this schema there are two types of nodes: people and pages. As you can see, there are two distinct categories of people nodes to distinguish people with a known identity from people you can only identify using a web browser cookie.

![Graph Schema](image-url)

The graph schema has two node types: people and the pages they have viewed.

Edges between people nodes denote the same user identified by different means. Edges between a person and a page represent a single pageview.

Properties are view counts for a page and demographic information for a person.

**Figure 2.17** The graph schema for SuperWebAnalytics.com. There are two node types: people and pages. People nodes and their properties are slightly shaded to distinguish the two.
CHAPTER 2  Data model for Big Data

Edges in the schema are rather simple. A *pageview* edge occurs between a person and a page for each distinct view, whereas an *equiv* edge occurs between two person nodes when they represent the same individual. The latter would occur when a person initially identified by only a cookie is fully identified at a later time.

Properties are also self-explanatory. Pages have total pageview counts, and people have basic demographic information: name, gender, and location.

One of the beauties of the fact-based model and graph schemas is that they can evolve as different types of data become available. A graph schema provides a consistent interface to arbitrarily diverse data, so it’s easy to incorporate new types of information. Schema additions are done by defining new node, edge, and property types. Due to the atomicity of facts, these additions do not affect previously existing fact types.

2.5 Summary

How you model your master dataset creates the foundation for your Big Data system. The decisions made about the master dataset determine the kind of analytics you can perform on your data and how you’ll consume that data. The structure of the master dataset must support evolution of the kinds of data stored, because your company’s data types may change considerably over the years.

The fact-based model provides a simple yet expressive representation of your data by naturally keeping a full history of each entity over time. Its append-only nature makes it easy to implement in a distributed system, and it can easily evolve as your data and your needs change. You’re not just implementing a relational system in a more scalable way—you’re adding whole new capabilities to your system as well.
Web-scale applications like social networks, real-time analytics, or e-commerce sites deal with a lot of data, whose volume and velocity exceed the limits of traditional database systems. These applications require architectures built around clusters of machines to store and process data of any size, or speed. Fortunately, scale and simplicity are not mutually exclusive.

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

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

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

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

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