Big Data

Principles and best practices of scalable realtime data systems

Nathan Marz
James Warren

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Big Data
Version 11

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

brief contents

1 A New Paradigm for Big Data

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

This astonishing growth in data has profoundly affected businesses. Traditional database systems, such as relational databases, have been pushed to the limit. In an increasing number of cases these systems are breaking under the pressures of "Big Data." Traditional systems, and the data management techniques associated with them, have failed to scale to Big Data.

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

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

This book is about complexity as much as it is about scalability. In order to meet the challenges of Big Data, you must rethink data systems from the ground up. You will discover that some of the most basic ways people manage data in traditional systems like the relational database management system (RDBMS) is

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

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

1.1 What this book is and is not about

This book is not a survey of database, computation, and other related technologies. While you will learn how to use many of these tools throughout this book, such as Hadoop, Cassandra, Storm, and Thrift, the goal of this book is not to learn those tools as an end upon themselves. Rather, the tools are a means to learning the underlying principles of architecting robust and scalable data systems.

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

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

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

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

1.2 Scaling with a traditional database

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

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

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

Figure 1.1 Relational schema for simple analytics application

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

1.2.1 Scaling with a queue

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

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

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

Instead of having the web server hit the database directly, you insert a queue between the web server and the database. Whenever you receive a new pageview, that event is added to the queue. You then create a worker process that reads 1000 events at a time off the queue and batches them into a single database update. This is illustrated in Figure 1.2.

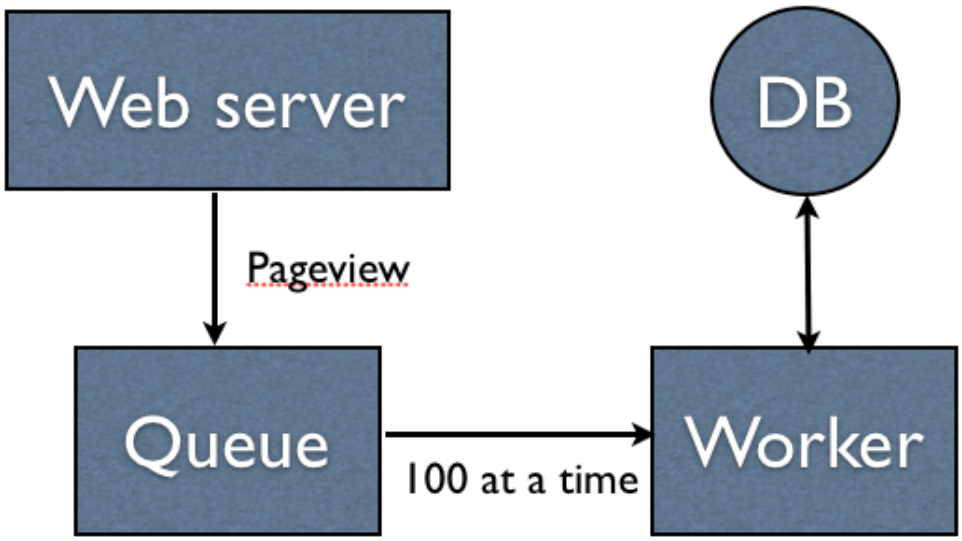


Figure 1.2 Batching updates with queue and worker

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

1.2.2 Scaling by sharding the database

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

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

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

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

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

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

1.2.3 Fault-tolerance issues begin

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

- You update your queue/worker system to put increments for unavailable shards on a separate "pending" queue that is attempted to be flushed once every 5 minutes.
- You use the database's replication capabilities to add a slave to each shard to have a backup in case the master goes down. You don't write to the slave, but at least customers can still view the stats in the application.

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

1.2.4 Corruption issues

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

1.2.5 Analysis of problems with traditional architecture

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

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

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

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

  Mistakes in software are inevitable, so if you're not engineering for it you might as well be writing scripts that randomly corrupt data. Backups are not enough, the system must be carefully thought out to limit the damage a human mistake can cause. Human fault-tolerance is not optional. It is essential especially when Big Data adds so many more complexities to building applications.
- *Maintenance is an enormous amount of work:* Scaling your sharded database is time-consuming and error-prone. The problem is that you have to manage all the constraints of what is allowed where yourself. What you really want is for the database to be self-aware of its distributed nature and manage the sharding process for you.

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

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

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

1.3 NoSQL as a paradigm shift

The past decade has seen a huge amount of innovation in scalable data systems. These include large scale computation systems like Hadoop and databases such as Cassandra and Riak. This set of tools has been categorized under the term "NoSQL." These systems can handle very large scales of data but with serious tradeoffs.

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

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

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

1.4 First principles

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

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

Data systems don't just memorize and regurgitate information. They combine

bits and pieces together to produce their answers. A bank account balance, for example, is based on combining together the information about all the transactions on the account.

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

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

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

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

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

Figure 1.3 Basis of all possible data systems

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

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

The Lambda Architecture, which we will be introducing later in this chapter, provides a general purpose approach to implementing an arbitrary function on an arbitrary dataset and having the function return its results with low latency. That doesn't mean you'll always use the exact same technologies everytime you implement a data system. The specific technologies you use might change depending on your requirements. But the Lambda Architecture defines a consistent approach to choosing those technologies and how to wire them together to meet your requirements.

Before we dive into the Lambda Architecture, let's discuss the properties a data system must exhibit.

1.5 Desired Properties of a Big Data System

The properties you should strive for in Big Data systems are as much about complexity as they are about scalability. Not only must a Big Data system perform well and be resource-efficient, it must be easy to reason about as well. Let's go over each property one by one. You don't need to memorize these properties, as we will revisit them as we use first principles to show how to achieve these properties.

1.5.1 Robust and fault-tolerant

Building systems that "do the right thing" is difficult in the face of the challenges of distributed systems. Systems need to behave correctly in the face of machines going down randomly, the complex semantics of consistency in distributed databases, duplicated data, concurrency, and more. These challenges make it difficult just to reason about what a system is doing. Part of making a Big Data system robust is avoiding these complexities so that you can easily reason about the system.

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

1.5.2 Low latency reads and updates

The vast majority of applications require reads to be satisfied with very low latency, typically between a few milliseconds to a few hundred milliseconds. On the other hand, the update latency requirements vary a great deal between applications. Some applications require updates to propogate immediately, while in other applications a latency of a few hours is fine. Regardless, you will need to be able to achieve low latency updates *when you need them* in your Big Data systems. More importantly, you need to be able to achieve low latency reads and updates without compromising the robustness of the system. You will learn how to achieve low latency updates in the discussion of the "speed layer" in Chapter 7.

1.5.3 Scalable

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

1.5.4 General

A general system can support a wide range of applications. Indeed, this book wouldn't be very useful if it didn't generalize to a wide range of applications! The Lambda Architecture generalizes to applications as diverse as financial management systems, social media analytics, scientific applications, and social networking.

1.5.5 Extensible

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

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

1.5.6 Allows ad hoc queries

Being able to do ad hoc queries on your data is extremely important. Nearly every large dataset has unanticipated value within it. Being able to mine a dataset arbitrarily gives opportunities for business optimization and new applications. Ultimately, you can't discover interesting things to do with your data unless you can ask arbitrary questions of it. You will learn how to do ad hoc queries in Chapters 4 and 5 when we discuss batch processing.

1.5.7 Minimal maintenance

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

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

You combat implementation complexity by relying on simple algorithms and simple components. A trick employed in the Lambda Architecture is to push complexity out of the core components and into pieces of the system whose outputs are discardable after a few hours. The most complex components used, like read/write distributed databases, are in this layer where outputs are eventually discardable. We will discuss this technique in depth when we discuss the "speed layer" in Chapter 7.

1.5.8 Debuggable

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

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

1.6 Lambda Architecture

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

The Lambda Architecture solves the problem of computing arbitrary functions on arbitrary data in realtime by decomposing the problem into three layers: the batch layer, the serving layer, and the speed layer. You will be spending the whole book learning how to design, implement, and deploy each layer, but the high level ideas of how the whole system fits together are fairly easy to understand.

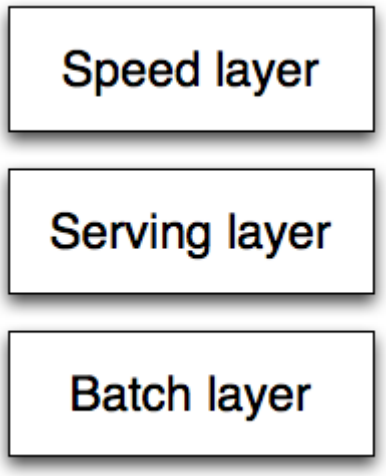


Figure 1.4 Lambda Architecture

Everything starts from the "query = function(all data)" equation. Ideally, you could literally run your query functions on the fly on the complete dataset to get the results. Unfortunately, even if this were possible it would take a huge amount of resources to do and would be unreasonably expensive. Imagine having to read a petabyte dataset everytime you want to answer the query of someone's current location.

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

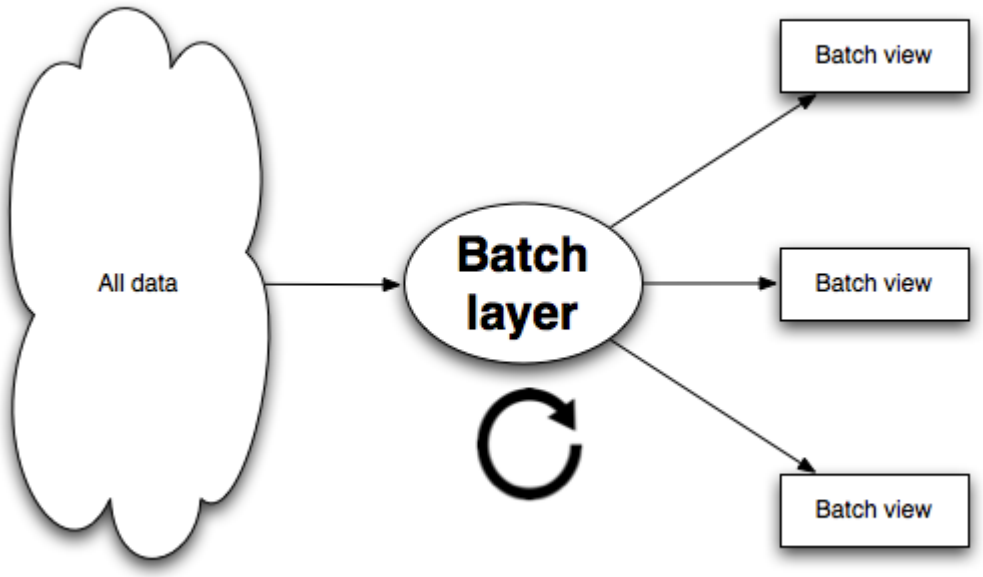


Figure 1.5 Batch layer

In this system, you run a function on all the data to get the batch view. Then when you want to know the value for a query function, you use the precomputed results to complete the query rather than scan through all the data. The batch view makes it possible to get the values you need from it very quickly since it's indexed.

Since this discussion is somewhat abstract, let's ground it with an example. Suppose you're building a web analytics application (again), and you want to query the number of pageviews for a URL on any range of days. If you were computing the query as a function of all the data, you would scan the dataset for pageviews for that URL within that time range and return the count of those results. This of course would be enormously expensive, as you would have to look at all the pageview data for every query you do.

The batch view approach instead runs a function on all the pageviews to precompute an index from a key of [url, day] to the count of the number of pageviews for that URL for that day. Then, to resolve the query, you retrieve all values from that view for all days within that time range and sum up the counts to get the result. The precomputed view indexes the data by url, so you can quickly retreive all the data points you need to complete the query.

You might be thinking that there's something missing from this approach as described so far. Creating the batch view is clearly going to be a high latency operation, as it's running a function on all the data you have. By the time it

finishes, a lot of new data will have collected that's not represented in the batch views, and the queries are going to be out of date by many hours. You're right, but let's ignore this issue for the moment because we'll be able to fix it. Let's pretend that it's okay for queries to be out of date by a few hours and continue exploring this idea of precomputing a batch view by running a function on the complete dataset.

1.6.1 Batch Layer

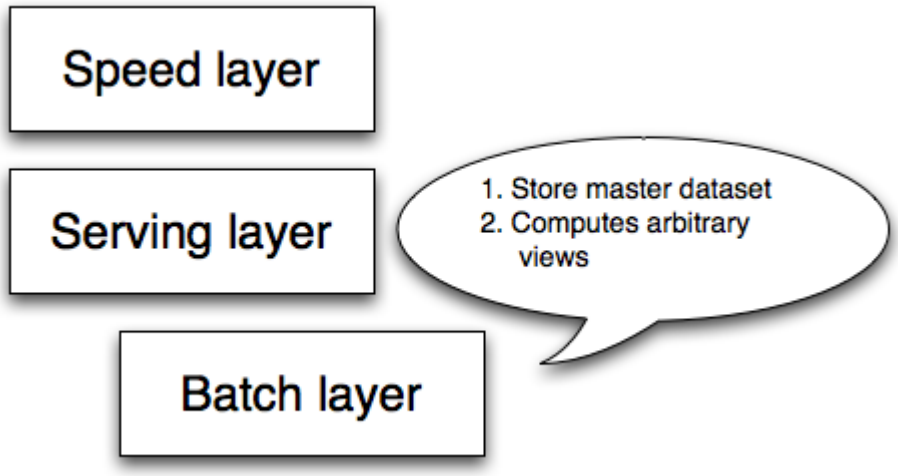


Figure 1.6 Batch layer

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

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

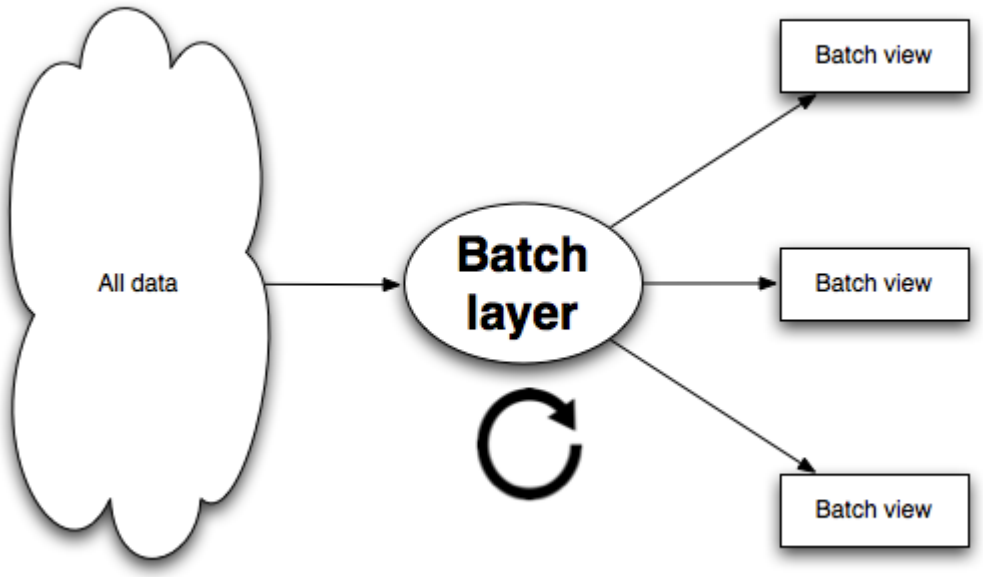


Figure 1.7 Batch layer

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

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

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

The nice thing about the batch layer is that it's so simple to use. Batch computations are written like single-threaded programs, yet automatically parallelize across a cluster of machines. This implicit parallelization makes batch layer computations scale to datasets of any size. It's easy to write robust, highly scalable computations on the batch layer.

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

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

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

This code computes the number of pageviews for every URL given an input dataset of raw pageviews. What's interesting about this code is that all the concurrency challenges of scheduling work, merging results, and dealing with runtime failures (such as machines going down) is done for you. Because the algorithm is written in this way, it can be automatically distributed on a MapReduce cluster, scaling to however many nodes you have available. So if you have 10 nodes in your MapReduce cluster, the computation will finish about 10x faster than if you only had one node! At the end of the computation, the output directory will contain some number of files with the results. You will learn how to write programs like this in Chapter 5.

1.6.2 Serving Layer

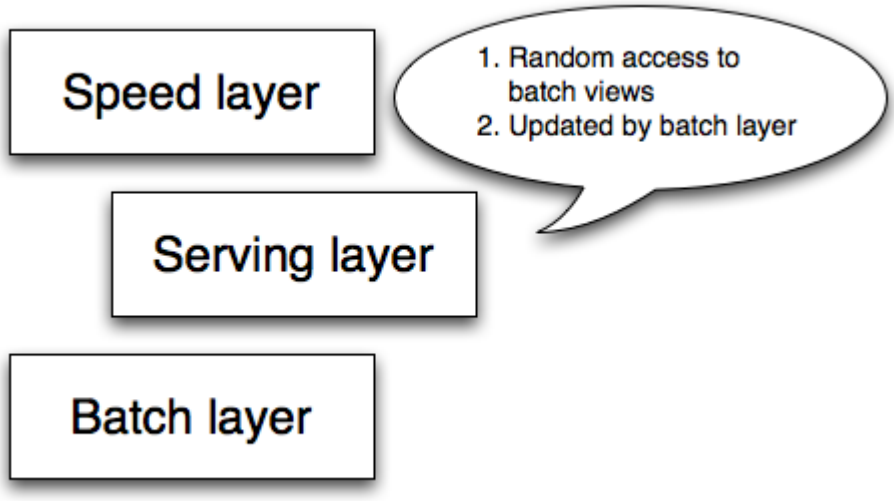


Figure 1.8 Serving layer

The batch layer emits batch views as the result of its functions. The next step is to load the views somewhere so that they can be queried. This is where the serving layer comes in. For example, your batch layer may precompute a batch view containing the pageview count for every [url, hour] pair. That batch view is

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

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

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

1.6.3 Batch and serving layers satisfy almost all properties

So far you've seen how the batch and serving layers can support arbitrary queries on an arbitrary dataset with the tradeoff that queries will be out of date by a few hours. The long update latency is due to new pieces of data taking a few hours to propogate through the batch layer into the serving layer where it can be queried. The important thing to notice is that other than low latency updates, the batch and serving layers satisfy every property desired in a Big Data system as outlined in Section 1.3. Let's go through them one by one:

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

- *Extensible:* Adding a new view is as easy as adding a new function of the master dataset. Since the master dataset can contain arbitrary data, new types of data can be easily added. If you want to tweak a view, you don't have to worry about supporting multiple versions of the view in the application. You can simply recompute the entire view from scratch.
- *Allows ad hoc queries:* The batch layer supports ad-hoc queries innately. All the data is conveniently available in one location and you're able to run any function you want on that data.
- *Minimal maintenance:*The batch and serving layers are comprised of very few pieces, yet they generalize arbitrarily. So you only have to maintain a few pieces for a huge number of applications. As explained before, the serving layer databases are simple because they don't do random writes. Since a serving layer database has so few moving parts, there's lots less that can go wrong. As a consequence, it's much less likely that anything will go wrong with a serving layer database so they are easier to maintain.
- *Debuggable:*You will always have the inputs and outputs of computations run on the batch layer. In a traditional database, an output can replace the original input -- for example, when incrementing a value. In the batch and serving layers, the input is the master dataset and the output is the views. Likewise you have the inputs and outputs for all the intermediate steps. Having the inputs and outputs gives you all the information you need to debug when something goes wrong.

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

1.6.4 Speed layer

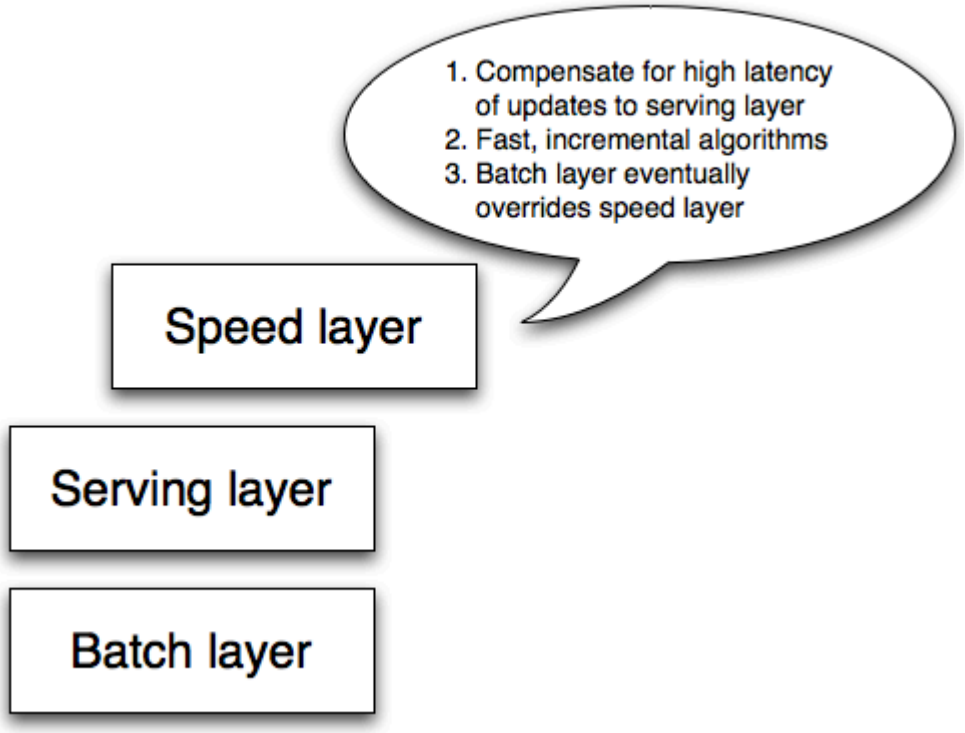


Figure 1.9 Speed layer

The serving layer updates whenever the batch layer finishes precomputing a batch view. This means that the only data not represented in the batch views is the data that came in while the precomputation was running. All that's left to do to have a fully realtime data system – that is, arbitrary functions computed on arbitrary data in realtime – is to compensate for those last few hours of data. This is the purpose of the speed layer.

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

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

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

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

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

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

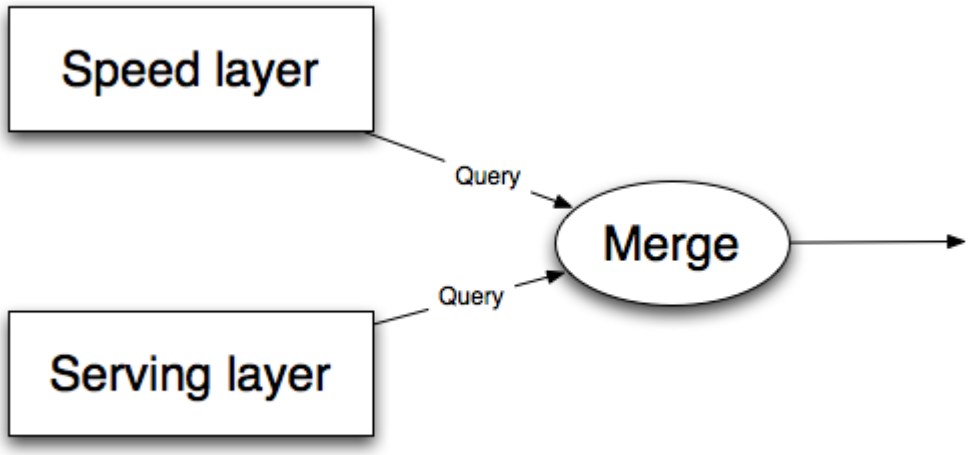


Figure 1.10 Satisfying application queries

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

1.7 Summary of the Lambda Architecture

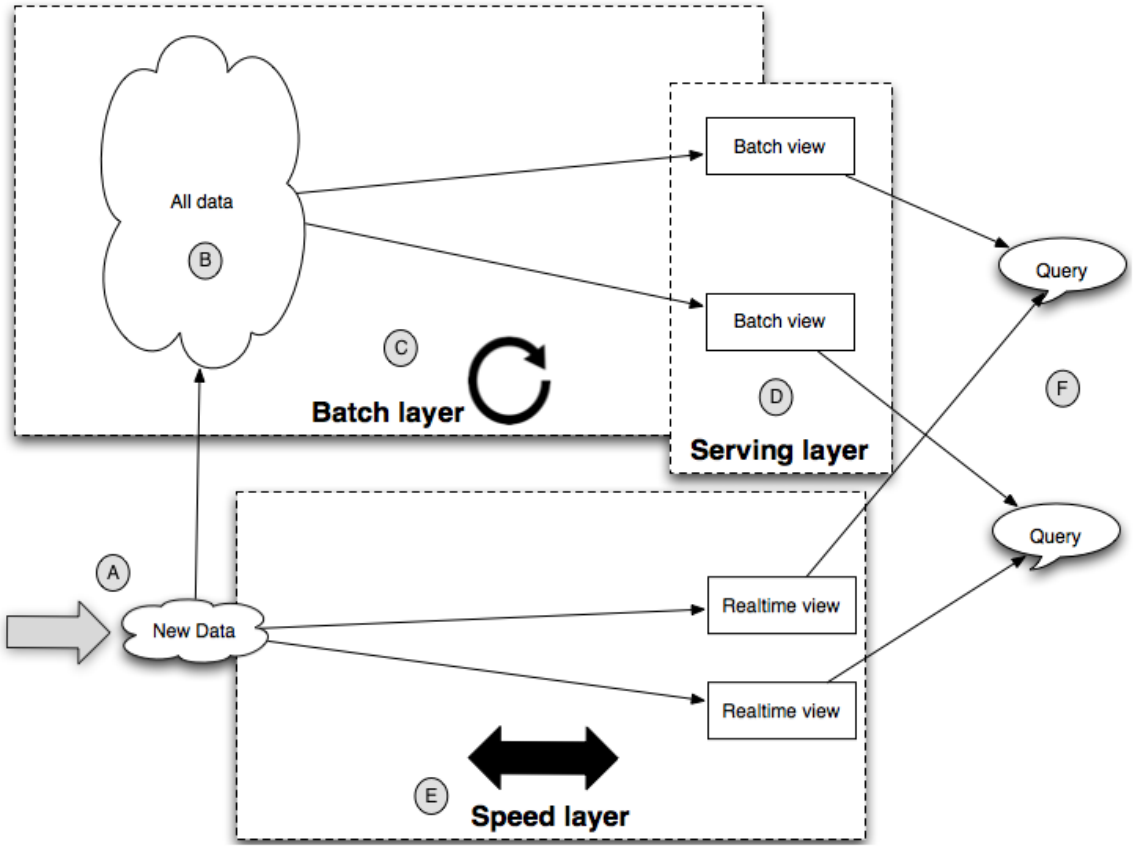


Figure 1.11 Lambda Architecture diagram

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

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

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

1.8 Example application: SuperWebAnalytics.com

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

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

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

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

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

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

1.9 Summary

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

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

Another benefit is how much more robust your applications will be. There are many reasons why your applications will be more robust. As one example, you'll have the ability to run computations on your whole dataset to do migrations or fix things that go wrong. You'll never have to deal with situations where there are

multiple versions of a schema active at the same time. When you change your schema, you will have the capability to update all data to the new schema. Likewise, if an incorrect algorithm is accidentally deployed to production and corrupts data you're serving, you can easily fix things by recomputing the corrupted values. As you'll explore, there are many other reasons why your Big Data applications will be more robust.

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

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

2 Data model for Big Data

This chapter covers:

- Properties of data
- The fact-based data model
- Benefits of a fact-based model for Big Data
- Graph schemas and serialization frameworks
- A complete model implementation using Apache Thrift

In the last chapter you saw what can go wrong when using traditional tools for building data systems and 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 we highlight 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 as the speed layer is based only on recent data it can construct itself within a few hours.

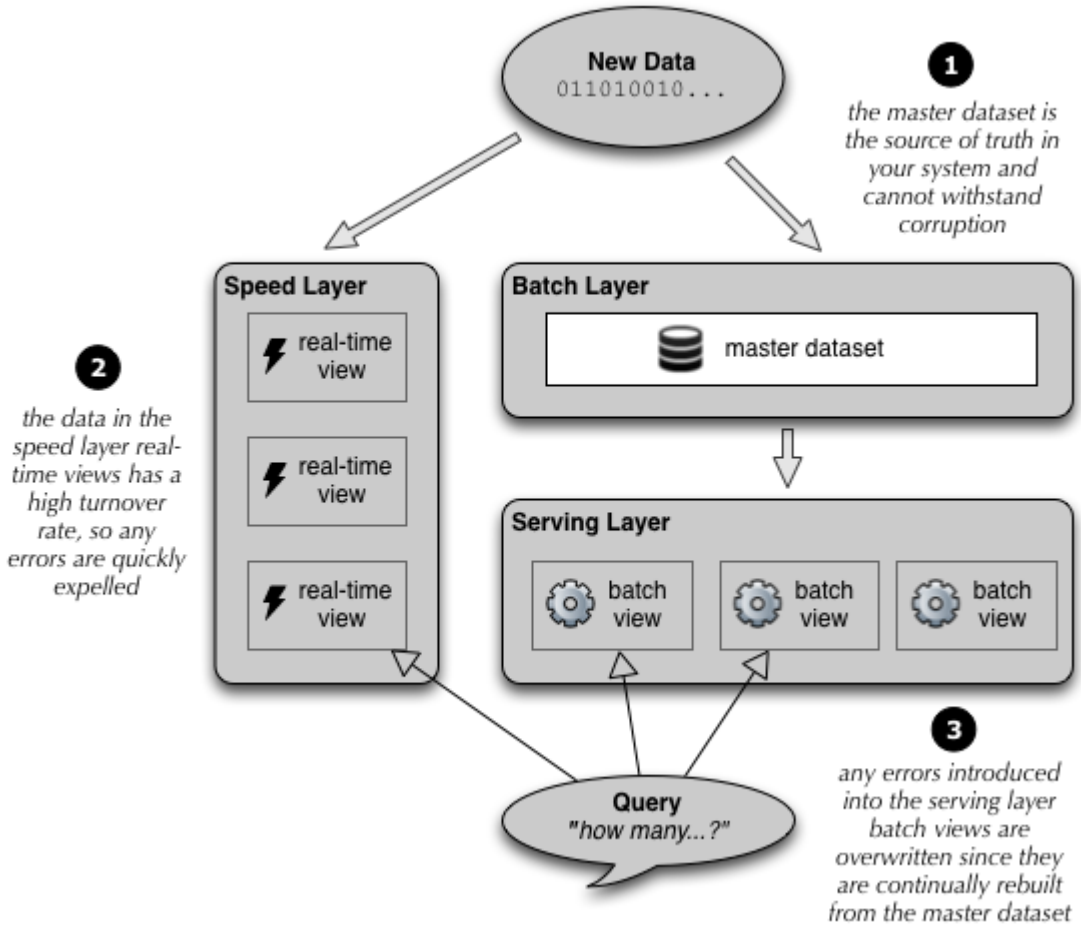


Figure 2.1 The master dataset in the Lambda Architecture serves as the source of truth of your Big Data system. Errors at the serving and speed layers can be corrected, but corruption at the master dataset is irreparable.

The master dataset is the only part of the Lambda Architecture that absolutely must be safeguarded from corruption. Overloaded machines, failing disks, and 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 to use, and how to physically store it. This chapter is about designing a data model for the master dataset and the properties such a data model should have. You will learn about physically storing a master dataset in the next chapter.

To provide a roadmap for your undertaking, you will

- 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
- implement a graph schema using Apache Thrift

Let's begin with a discussion of the rather general term *data*.

2.1 The properties of data

Keeping with the applied focus of the book, we will center our discussion around an example application. Suppose you are 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. So what information should you store regarding Tom's connections? You have a number of choices, ranging from potentially storing

- 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.

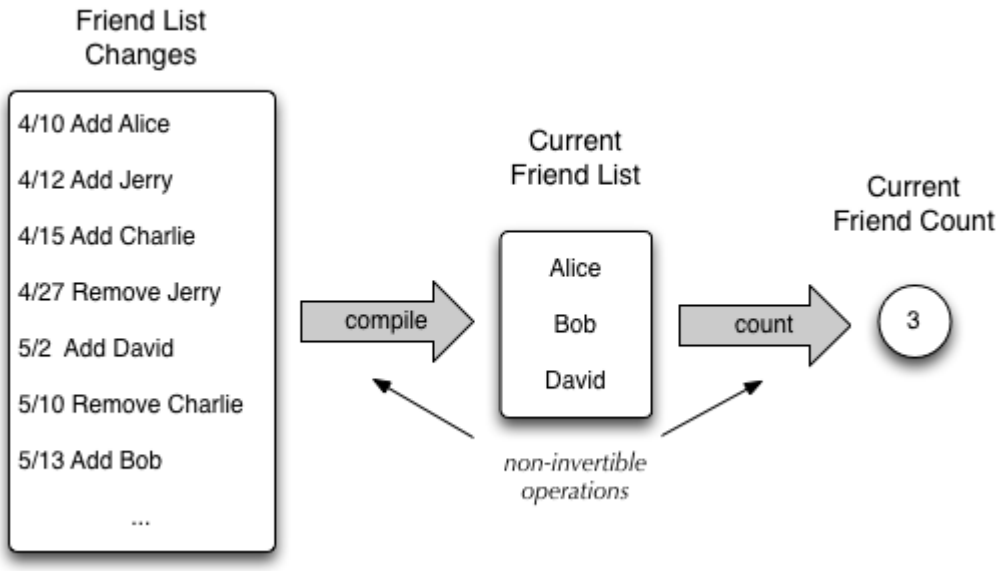


Figure 2.2 Three possible options for storing friendship information for FaceSpace. Each option can be derived from the one to its left, but it's a one way process.

This example illustrates information dependency. Note that each layer of information can be derived from the previous one, but it's a one way process. From the sequence of friend and unfriend events, we can determine the other quantities. However, 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 will use:

- *Information* is the general collection of knowledge relevant to your Big Data System. It is synonymous with the colloquial usage of the word "data".
- *Data* will refer 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.

In Figure 2.3, we re-illustrate the FaceSpace information dependency in terms of data, views and queries.

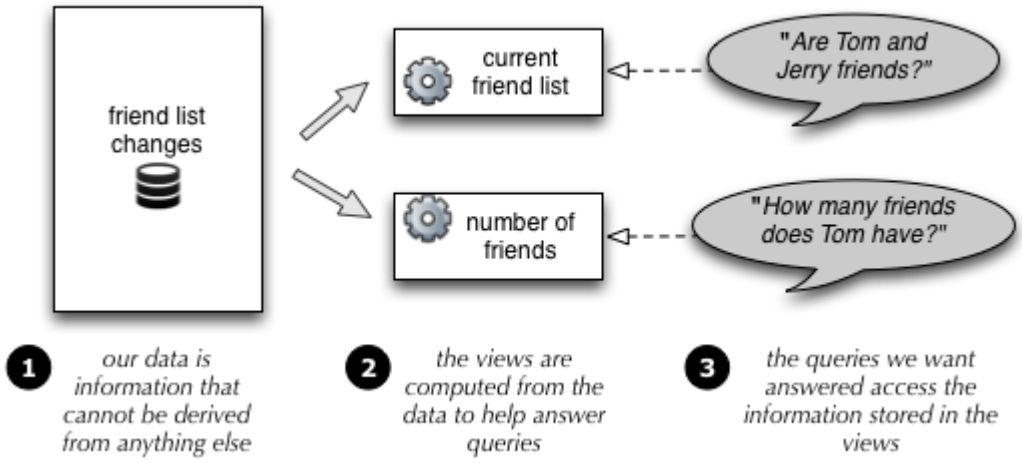


Figure 2.3 The relationships between data, views and queries.

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. As FaceSpace, we have complete access to all the information Tom provided - for example, his complete birthdate of March 13, 1984. However, 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 our perspective since it's derived from his birthdate, yet it is data to the advertiser since they have limited information about Tom. This relationship is shown in Figure 2.4

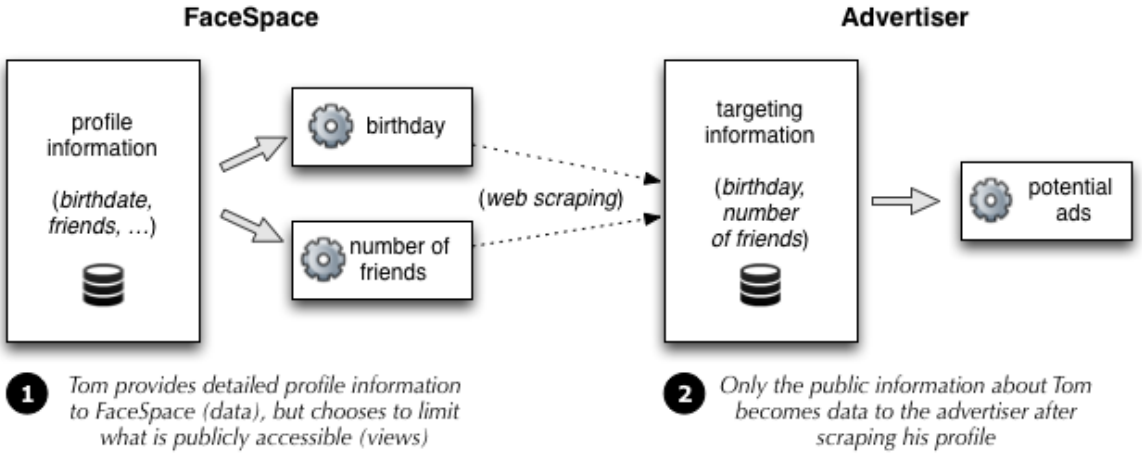


Figure 2.4 Classifying information as data or a view depends upon your perspective. As the owner of FaceSpace, Tom's birthday is a view since it is derived from the user's birthdate. However, this information is considered data to a third party advertiser.

Having established a shared vocabulary, we can introduce the key properties of data: *rawness*, *immutability*, and *perpetuity* - or the "eternal trueness of data". 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 are not concerned with immutability or perpetuity. However, that approach limits the questions you can answer with your data, as well as fails to be robust to errors and corruption. It doesn't have to be so in the world of Big Data by enforcing these properties.

We will delve further into this topic as we discuss 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 data is more valuable than the advertiser's since you can deduce more information about Tom. We 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.

While the FaceSpace example helps illustrate the value of rawness, we offer another 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 stock on a day when Google announced new products targeted at their competitors.

Company	Symbol	Previous	Open	High	Low	Close	Net
Google	GOOG	564.68	567.70	573.99	566.02	569.30	+4.62
Apple	AAPL	572.02	575.00	576.74	571.92	574.50	+2.48
Amazon	AMZN	225.61	225.01	227.50	223.30	225.62	+0.01

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 stock: previous close, opening, high, low, close and net change.

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

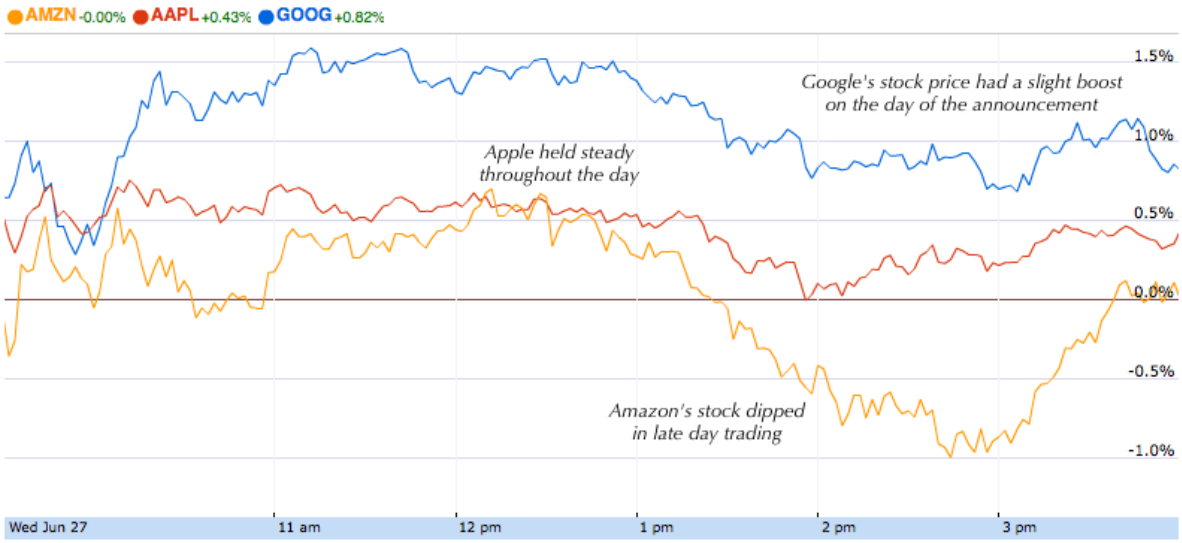


Figure 2.6 Relative stock price changes of Google, Apple and Amazon on June 27, 2012 compared to closing prices on June 26. Short term analysis isn't supported by daily records but can be performed by storing data at a finer time resolution.

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 us wonder if Amazon was more greatly affected because the new Google products compete with Amazon in both the table 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 tradeoff is that rawer data typically entails more of it - sometimes much more. However, 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.

While the concept is straightforward, it is not always clear what information you should store as your raw data. We offer a couple of examples to help guide you when you are faced with making this decision.

UNSTRUCTURED DATA IS RAWER THAN NORMALIZED DATA

When deciding what to store for your raw data, a common hazy area is the line between parsing and *semantic normalization*. Semantic normalization is the process of reshaping free-form information to convert it 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.

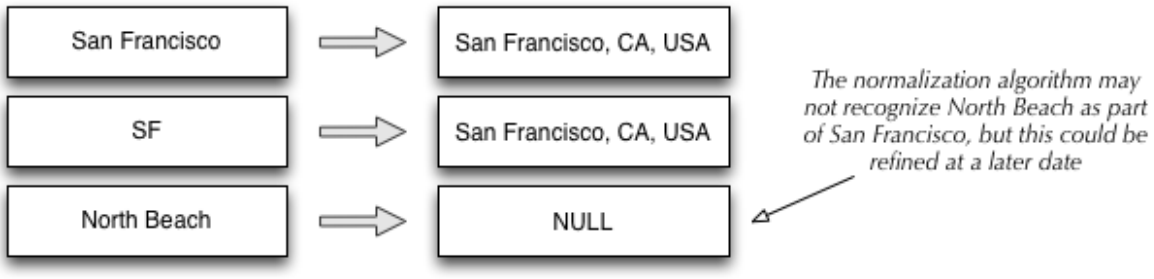


Figure 2.7 Semantic normalization of unstructured location responses to city, state and country. A simple algorithm would normalize "North Beach" to NULL if it does not recognize it as a San Francisco neighborhood.

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 example above, 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.

TIP	**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 it'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 just the pure text of the blog entries is certainly a possibility. However, 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, we could also store the full HTML of Tom's blog as your data. While it is considerably more information in terms of total bytes, the color scheme, stylesheets and JavaScript code of the site cannot be used to derive any additional information about Tom. They serve only as the container for the contents of the site and should not 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 with relational databases. After all, in the relational database world - and most other databases as well - *update* is one of the fundamental operations. However, for immutability, you don't update or delete data, you only add more. [1] By using an immutable schema for Big Data systems, you gain two vital advantages:

Footnote 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 will discuss these scenarios in Section 2.1.4.

1. *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 off the master dataset.

2. *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 does not require an index for your data, which is a huge simplification. As you will 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 that you could use for FaceSpace:

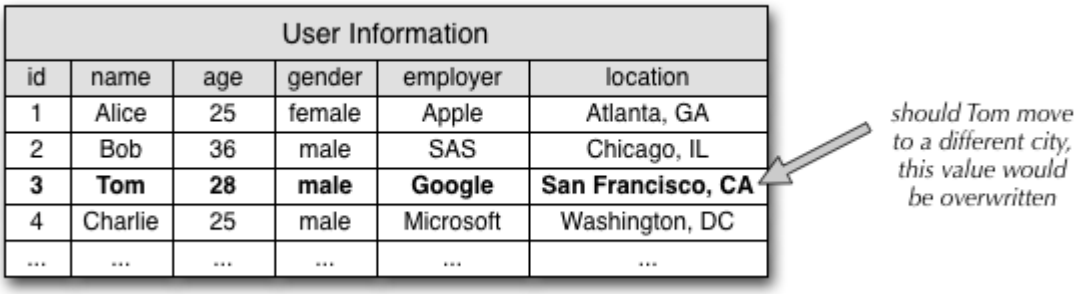


Figure 2.8 A mutable schema for FaceSpace user information. When details change - say Tom moves to Los Angeles - previous values are overwritten and lost.

Should Tom move to Los Angeles, you would update the highlighted entry to reflect his current location - but in the process you would also lose all knowledge that Tom ever lived in San Francisco.

With an immutable schema, things look different. Rather than store 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. You also 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.

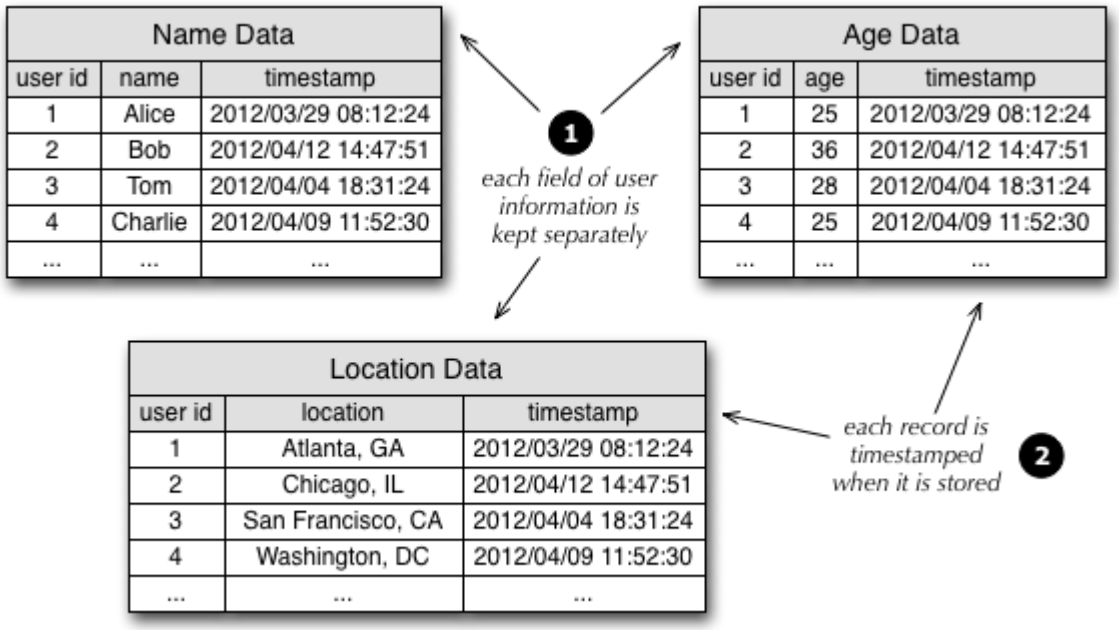


Figure 2.9 An equivalent immutable schema for FaceSpace user information. Each field is tracked in a separate table, and each row has a timestamp for when it is known to be true. (Gender and employer data are omitted for space but are stored similarly.)

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.

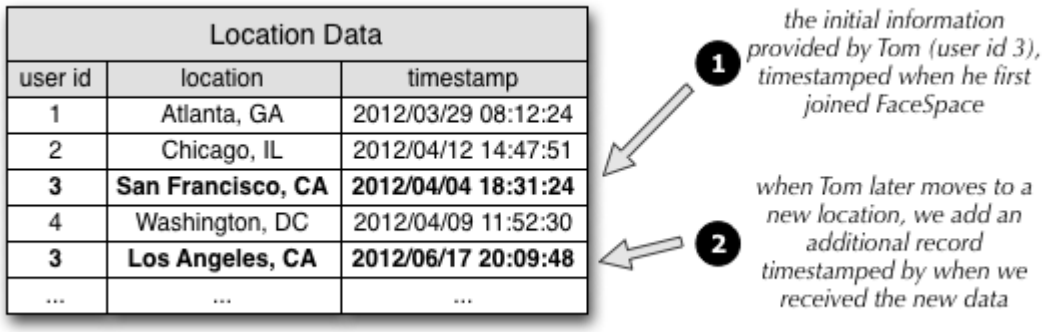


Figure 2.10 Instead of updating prexisting records, an immutable schema uses new records to represent changed information. An immutable schema thus can store multiple records for the same user.(Other tables omitted since they remain unchanged.)

You now have two location records for Tom (user id #3), and since the data units are tied to particular times, they can both be true. Tom's *current location* is 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 as well as guarantees each record is new information and not simply carried over from the last record.

One of the tradeoffs 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 strong human fault-tolerant master dataset cannot be overstated.

2.1.3 Data is eternally true

The key consequence from 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 would be captured in additional (and also perpetual) data.

In general, your master dataset is consistently growing 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 first consider the cases:

1. *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 you 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.
2. *Regulations:* Government regulations may require you to purge data from your databases in certain conditions.

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

We proceed by introducing a data model that uses these key properties of data.

2.2 The fact-based model for representating data

While data is the set of information that can't be derived from anything else, there are many ways we could choose to represent it within the master dataset. Besides traditional relational tables, structured XML and semi-structured JSON documents are other possibilities for storing data. We, however, recommend the fact-based model for this purpose. In the fact-based model, we deconstruct the data into fundamental units that we (unsurprisingly) call facts.

In the discussion of immutability you saw 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 fact-based model in the context of our 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 An example of the fact-based model

Figure 2.11 depicts some examples of facts from the FaceSpace data regarding Tom.

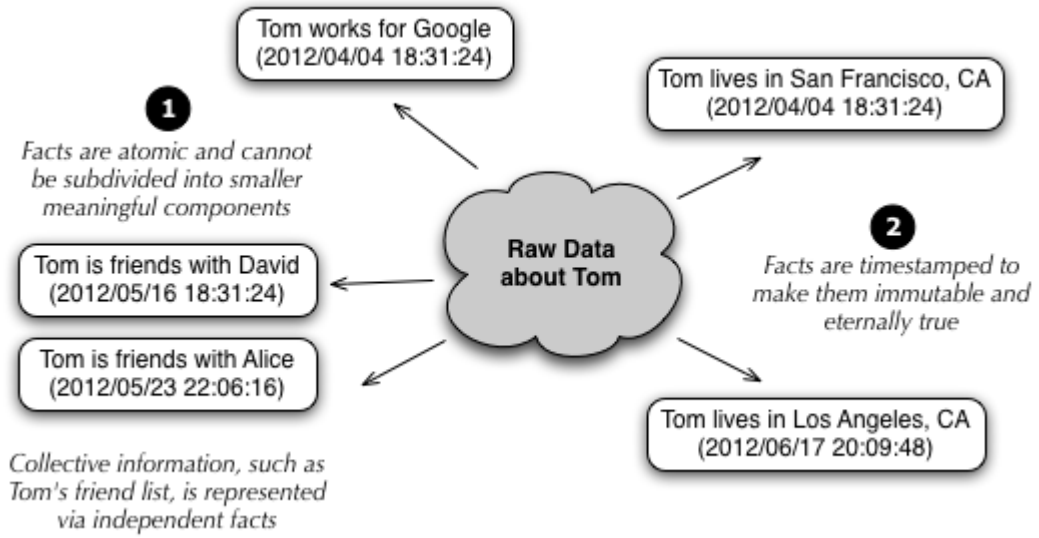


Figure 2.11 All of the raw data concerning Tom are deconstructed into time-stamped, atomic units we call facts.

This example demonstrates the core properties of facts. First, facts are timestamped. This should come as no surprise given our earlier discussion about

data - the timestamps make each fact immutable and eternally true.

Second, facts are atomic since they cannot 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 is no redundancy of information across distinct facts.

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*. We next discuss in depth how and why you make facts identifiable.

2.2.2 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 (pseudocode):

```
struct PageView:
  DateTime timestamp
  String url
  String ip_address
```

Facts using this structure do not 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 be 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.

Here's an alternative way to model pageviews in which you can distinguish between different pageviews:

```
struct PageView:
  Datetime timestamp
  String url
  String ip_address
  Long nonce
```

1 the nonce, combined with the other fields, uniquely identifies a particular pageview

When a pageview fact is created, a random 64 bit number is chosen as a nonce to distinguish this pageview from other pageviews that occur for the same URL at the same time and from the same ip address. 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 will see later, having distinguishable facts makes implementing the rest of the Lambda Architecture much easier.

SIDEBAR **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 an event it 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 available. For these situations, fault-tolerant systems commonly handle failed operations by retrying until success. Since the sender would not know which data was last received, a standard approach would be to resend all data yet to be acknowledged by the recipient. However, if part of the original attempt did make it to the metastore, you'll end up with duplicates in your dataset.

Now 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, and
- 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.

2.2.3 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.

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 since no data is actually removed, you can reconstruct the state of the world at the specified time of your query.

THE DATA IS HUMAN FAULT-TOLERANT

Human fault tolerance is achieved by simply deleting any erroneous facts. Suppose you had mistakenly stored that Tom moved from San Francisco to Los Angeles - see Figure 2.12.

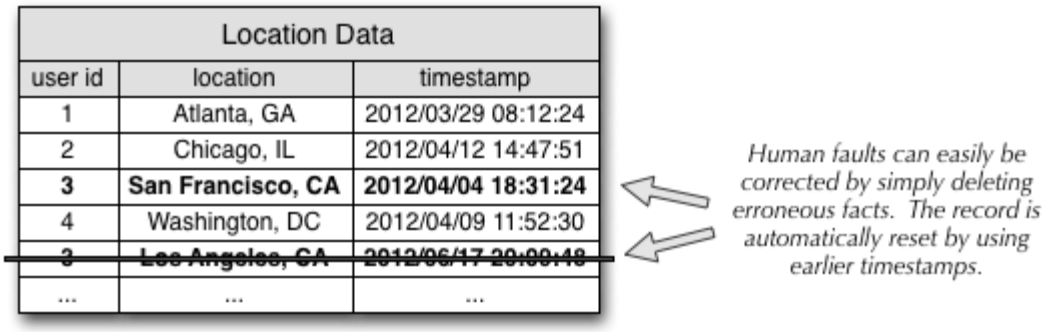


Figure 2.12 To correct for human errors, simply remove the incorrect facts. This process automatically resets to an earlier state by "uncovering" any relevant predated facts.

By removing the Los Angeles fact, Tom's location is automatically "reset" since the San Francisco fact becomes the most recent information.

THE DATASET EASILY HANDLES PARTIAL INFORMATION

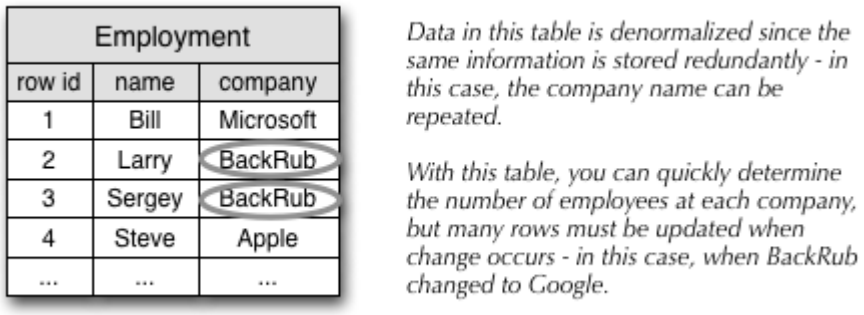
Storing one fact per record makes it easy to handle partial information about an entity without introducing NULL values into your dataset. Suppose Tom provided his age and gender but not his location or profession. Your dataset would only have facts for the known information - any "absent" fact would be logically equivalent to NULL. Additional information 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 benefits of keeping your data in both normalized and denormalized forms and reaping the benefits of both.

TIP	**Normalization is an overloaded term** Data normalization is completely unrelated to the term semantic normalization that we used earlier. In this case, data normalization refers to storing data in a structured manner to minimize redundancy and promote consistency.

Let's set the stage with an example involving relational tables - the context where data normalization is most frequently encountered. Suppose you wanted to store the employment information for various people of interest. Figure 2.13 offers a simple schema to suit this purpose.



Employment

row id	name	company
1	Bill	Microsoft
2	Larry	BackRub
3	Sergey	BackRub
4	Steve	Apple
...	...	...

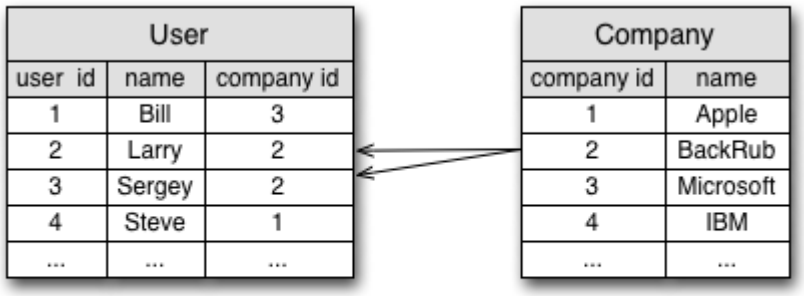
Data in this table is denormalized since the same information is stored redundantly - in this case, the company name can be repeated.

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.

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, yet 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.



User

user id	name	company id
1	Bill	3
2	Larry	2
3	Sergey	2
4	Steve	1
...	...	...

Company

company id	name
1	Apple
2	BackRub
3	Microsoft
4	IBM
...	...

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.

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 removes the risk of inconsistency, but you must join the tables to answer queries - a potentially expensive computation.

With relational databases, query processing is performed directly on the data at the storage level. You therefore must weigh the importance of query efficiency versus data consistency and choose between the two schema types. However, these objectives are cleanly separated in the Lambda Architecture. Take another look at the batch and server layers in Figure 2.15.

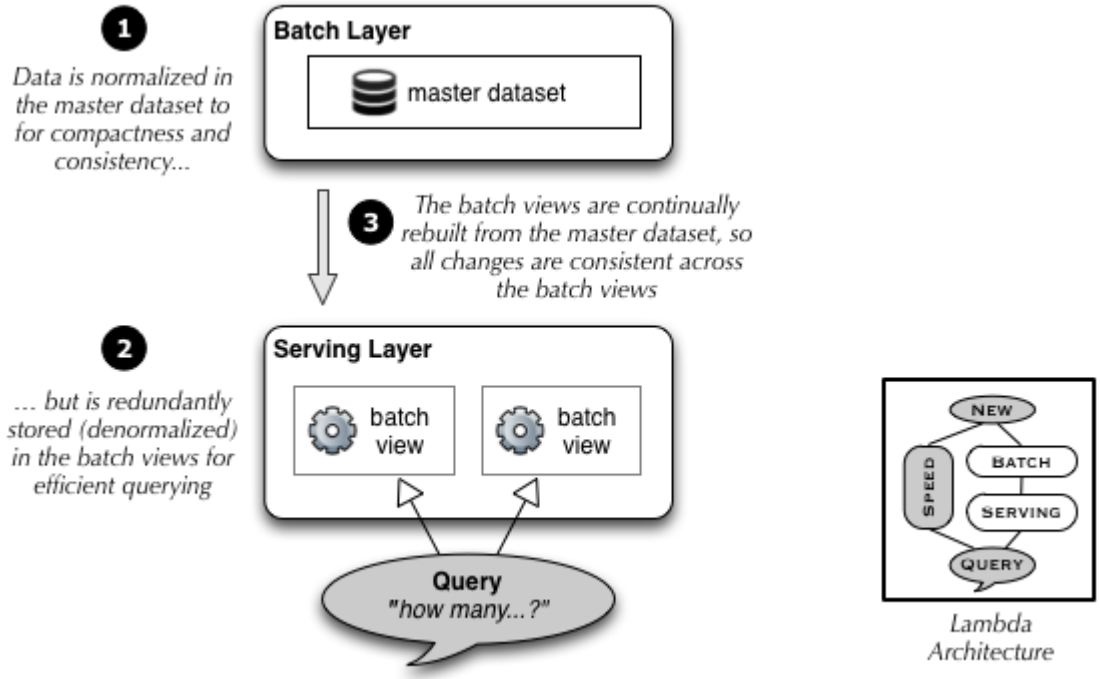


Figure 2.15 The Lambda Architecture has the benefits of both normalization and denormalization by separating objectives at different layers.

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 since 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 since it will be continually rebuilt from the master dataset. This has the additional benefit in that the batch views and the 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.

2.3 Graph schemas and serialization frameworks

Each fact within a fact-based model captures a single piece of information. However, the facts alone do not convey the structure behind the data. That is, there is no description of the type of facts contained in the dataset, nor any explanation of the relationships between them. In this section we introduce *graph schemas* - graphs that capture the structure of a dataset stored using the fact-based model. We will 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 contains a representation of the relationships between the FaceSpace facts. It provides a useful visualization of your users, their individual information, and the friendships between them.

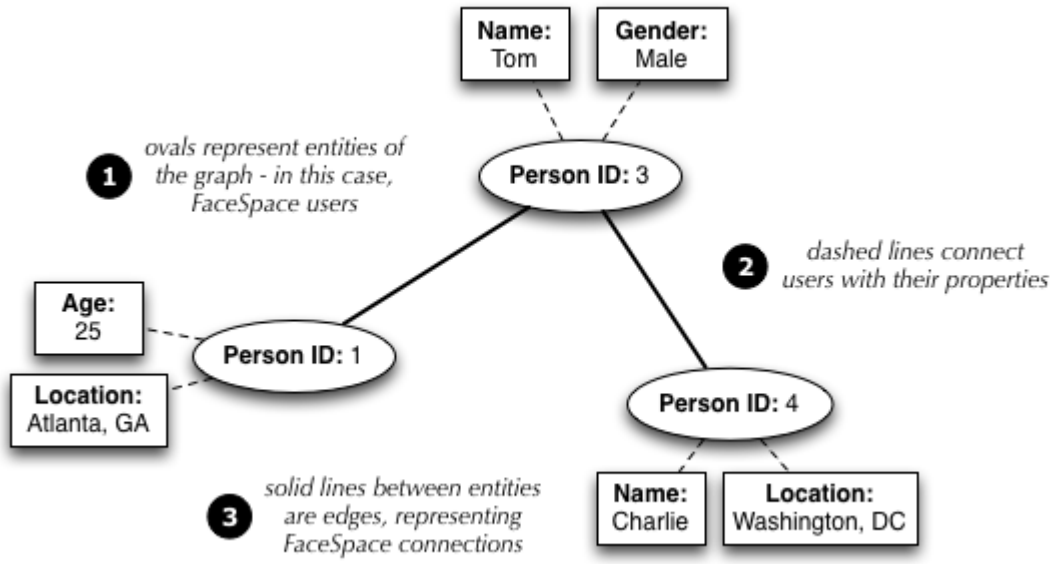


Figure 2.16 Visualizing the relationship between FaceSpace facts

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

1. Nodes are the entities in the system. In this example, the nodes are our 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.

2. Edges are relationships between nodes. The connotation in FaceSpace is straightforward - an edge between users represents a FaceSpace friendship. We could later add additional edge types between users to identify co-workers, family members, or classmates.
3. Properties are information about entities. In this example, age, gender, location, and all other individual information are properties.

WARNING **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 is the full listing of all types of nodes, edges and properties, and it provides a complete description of the data contained within a dataset. We next 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 type of information contained in the dataset. You're all set, right? Well, not quite. You still need to decide in what format you will store your facts. A first idea might be to use a semi-structured text format like JSON. This would provide simplicity and flexibility, allowing essentially anything to be written to the master dataset. However, in this case it's too flexible for our needs.

To illustrate this problem, suppose we 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 is no way to ensure that all subsequent facts will follow the same format. From human error, the dataset could also possibly include facts like

```
{"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 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 cannot 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 of what data they can expect - that each fact will have a timestamp, 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 the time of creating the data rather than when trying to use the data later on in a different system. The closer the error appears to the bug, the easier it is to catch and fix.

TIP **Enforceable schema catch only syntactic errors.**

Enforceable schemas only check the syntax of a fact - that is, that all the required fields are all present and of the expected type. It does not check the semantic truthfulness of the data. If you enter the incorrect value for Tom's age but use the proper format, no error will be found.

This is analogous to inserting data into a relational database. When you add a row to a table, the database server verifies that all required fields are present and that each field matches the expected type. The validity of the data is still the responsibility of the user.

Enforceable schemas are implemented 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 and retrieved from your master dataset.

The framework also provides a controlled means for your schema to evolve - for example, if you later wanted to add FaceSpace user's e-mail addresses to the dataset. It provides the flexibility to add new types of facts while guaranteeing that all facts meet the desired conditions.

We are aware that in this section we have only discussed the concepts of

enforceable schemas and serialization frameworks, and that you may be hungry for details. Not to worry, for we believe the best way to learn is by doing. In the next section we will implement the fact-based model for SuperWebAnalytics.com in its entirety.

2.4 A complete data model for SuperWebAnalytics.com

We've covered a lot of material in this chapter, and in this section we aim to tie it all together using the SuperWebAnalytics.com example. We begin with Figure 2.17, which contains a graph schema suitable for our purpose.

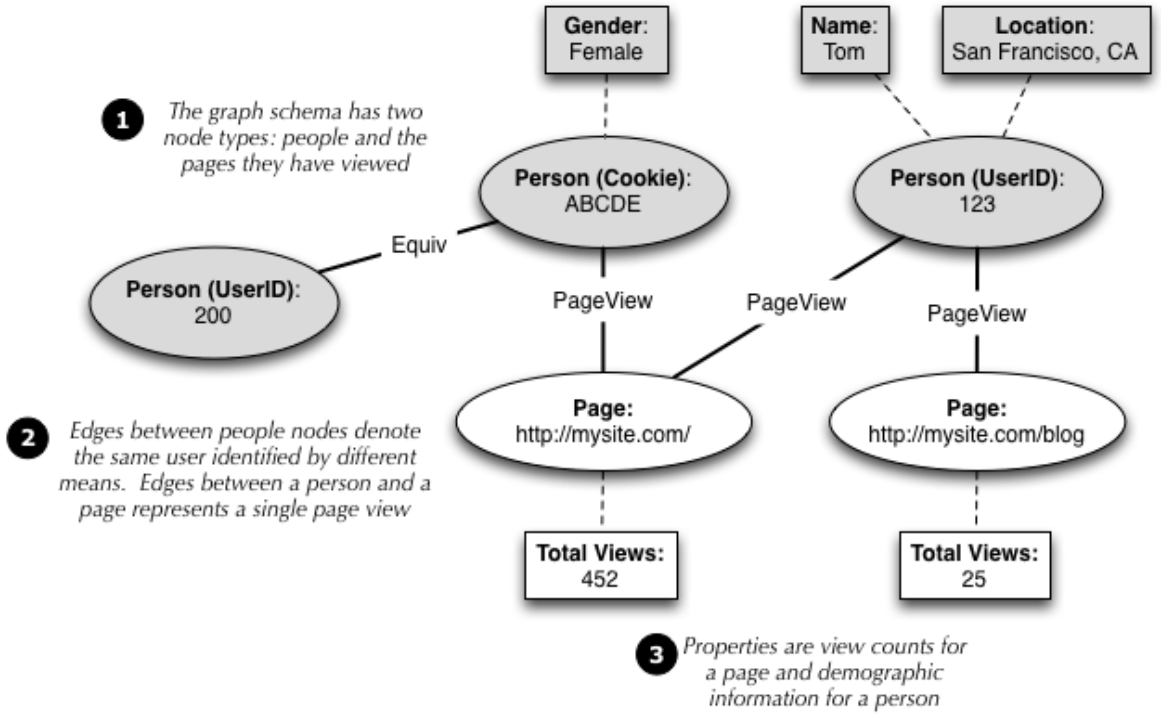


Figure 2.17 The graph schema for SuperWebAnalytics.com. There are two node types: people and edges. People nodes and their properties are slightly shaded to distinguish the two.

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 identify from people we can only identify using a web browser cookie.

Edges in the schema are rather simple. A *pageview* edge occurs between a person and a page for each distinct view, while 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 page view counts, and people have basic demographic information: name, gender and location.

We next introduce Apache Thrift as a serialization framework to make this schema enforceable.

2.4.1 Using Thrift as a serialization framework.

Apache Thrift [2] is a tool that can be used to define statically typed, enforceable schemas. It provides an interface definition language to describe the schema in terms of generic data types, and this description can be later used to automatically generate the actual implementation in multiple programming languages.

Footnote 2 http://thrift.apache.org/. Thrift was initially developed at Facebook for building cross-language services. It can be used for many purposes, but we'll limit our discussion to its usage as a serialization framework.

SIDEBAR **Other serialization frameworks**

There are other tools that can be used for this purpose, such as Protocol Buffers or Avro. Remember, the purpose of this book is not to provide a survey of all possible tools for every situation, but to use an appropriate tool to illustrate the fundamental concepts. As a serialization framework, Thrift is practical, thoroughly tested and widely used.

The workhorses of Thrift are the *struct* and *union* type definitions. They are composed of other fields, such as

- Primitive data types: strings, integers, longs and doubles
- Collections of other types: lists, maps and sets
- Other structs and unions

In general, unions are useful for representing nodes, structs are natural representations of edges, and properties use a combination of both. This will become more clear in the type definitions needed to represent the SuperWebAnalytics.com schema components.

NODES

In computer science, a union is a single value that may have any of several representations. This is exactly the case for the person nodes - an individual is identified either by a user id or a browser cookie, but not both. In Thrift, unions are defined by listing all possible representations:

```
union PersonID {
  1: string cookie;
  2: i64 user_id;
```

```
}

union PageID {
  1: string url;
}
```

Note that unions can also be used for nodes with a single representation. Unions allow the schema to evolve as the data evolves - we will discuss this further later in the section.

EDGES

Each edge can be represented as a struct containing two nodes. The name of an edge struct indicates the relationship it represents, and the fields in the edge struct contain the entities involved in the relationship. The schema definition is very simple.

```
struct EquivEdge {
  1: required PersonID id1;
  2: required PersonID id2;
}

struct PageViewEdge {
  1: required PersonID person;
  2: required PageID page;
  3: required i64 nonce;
}
```

The fields of a Thrift struct can be denoted as *required* or *optional*. If a field is defined as required, then a value for that field must be provided else Thrift will give an error upon serialization or deserialization. Since each edge in a graph schema must have two nodes, they are required fields in this example.

PROPERTIES

Last, let's define the properties. A property contains a node and a value for the property. The value can be one of many types, so that is best represented using a union structure. Let's start by defining the schema for page properties. There is only one property for pages so it's really simple.

```
union PagePropertyValue {
  1: i32 page_views;
```

```
}

struct PageProperty {
  1: required PageID id;
  2: required PagePropertyValue property;
}
```

Next let's define the properties for people. As you can see, the location property is more complex and requires another struct to be defined.

```


struct Location {
  1: optional string city;
  2: optional string state;
  3: optional string country;
}

enum GenderType {
  MALE = 1,
  FEMALE = 2
}

union PersonPropertyValue {
  1: string full_name;
  2: GenderType gender;
  3: Location location;
}

struct PersonProperty {
  1: required PersonID id;
  2: required PersonPropertyValue property;
}
```

The location struct is interesting because the city, state, and country fields could have been stored as separate pieces of data. However, in this case they are so closely related it makes sense to put them all into one struct as optional fields. When consuming location information, you will almost always want all of those fields.

2.4.2 Tying everything together into data objects

At this point, the edges and properties are defined as separate types. Ideally you would want to store all of the data together to provide a single interface to access your information. Furthermore, it also makes your data easier to manage if it's stored in a single dataset. This is accomplished by wrapping every property and edge type into a *DataUnit* union - see the following code listing.

Listing 2.1 Completing the SuperWebAnalytics.com schema

```
union DataUnit {
  1: PersonProperty person_property;
  2: PageProperty page_property;
  3: EquivEdge equiv;
  4: PageViewEdge page_view;
}

struct Pedigree {
  1: required i32 true_as_of_secs;
}

struct Data {
  1: required Pedigree pedigree;
  2: required DataUnit dataunit;
}
```

Each DataUnit is paired with its metadata that is kept in a *Pedigree* struct. The pedigree contains the timestamp for the information, but could also potentially contain debugging information or the source of the data. This final *Data* struct corresponds to a fact from the fact-based model.

2.4.3 Evolving your schema

The beauty of the fact-based model and graph schemas is that they can evolve as different types of data becomes available. A graph schema provides a consistent interface to arbitrarily diverse data, so it is 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.

Thrift is similarly designed so that schemas can be evolved over time. The key to evolving Thrift schemas is the numeric identifers associated with each field. Those ids are used to identify fields in their serialized form. When you want to change the schema but still be backwards compatible with existing data, you must obey the following rules.

- Fields may be renamed. This is because the serialized form of an object uses the field ids to identify fields, not the names.
- Fields may be removed, but you must never reuse that field id. When deserializing existing data, Thrift will ignore all fields with field ids not included in the schema. If you were to reuse a previously removed field id, Thrift will try to deserialize that old data into the new field which will lead to either invalid or incorrect data.

- Only optional fields can be added to existing structs. You can't add required fields because existing data won't have that field and thus wouldn't be deserializable. (Note this doesn't apply to unions since unions have no notion of required and optional fields.)

As an example, should you want change the SuperWebAnalytics.com schema to store a person's age and the links between webpages, you would make the following changes to your Thrift definition file:

Listing 2.2 Extending the SuperWebAnalytics.com schema

```
union PersonPropertyValue {
  1: string full_name;
  2: GenderType gender;
  3: Location location;
  4: i16 age;
}

struct LinkedEdge {
  1: required PageID source;
  2: required PageID target;
}

union DataUnit {
  1: PersonProperty person_property;
  2: PageProperty page_property;
  3: EquivEdge equiv;
  4: PageViewEdge page_view;
  5: LinkedEdge page_link;
}
```

Notice that adding a new age property is done by adding it to the corresponding union structure, and a new edge is incorporated by adding it into the DataUnit union.

2.5 Summary

How you model your master dataset sets the foundation of your Big Data system. The decisions made surrounding the master dataset determines the kind of analytics you can perform on your data and how you're going to consume that data. The structure of the master dataset must support evolution of the kinds of data stored, as 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.

In the next chapter, you'll learn how to physically store a master dataset in the batch layer so that it can be processed easily and efficiently.

3 Data storage on the batch layer

This chapter covers:

- Storage requirements for the master dataset
- The Hadoop Distributed File System (HDFS)
- Common tasks to maintain your dataset
- A record based abstraction to access your data

In the last chapter, you learned a data model for the master dataset and how to translate that data model into a graph schema. You saw the importance of making data immutable and eternal. The next step is to learn how to physically store that data in the batch layer.

Figure 3.1 provides a recap of where we are in the Lambda Architecture:

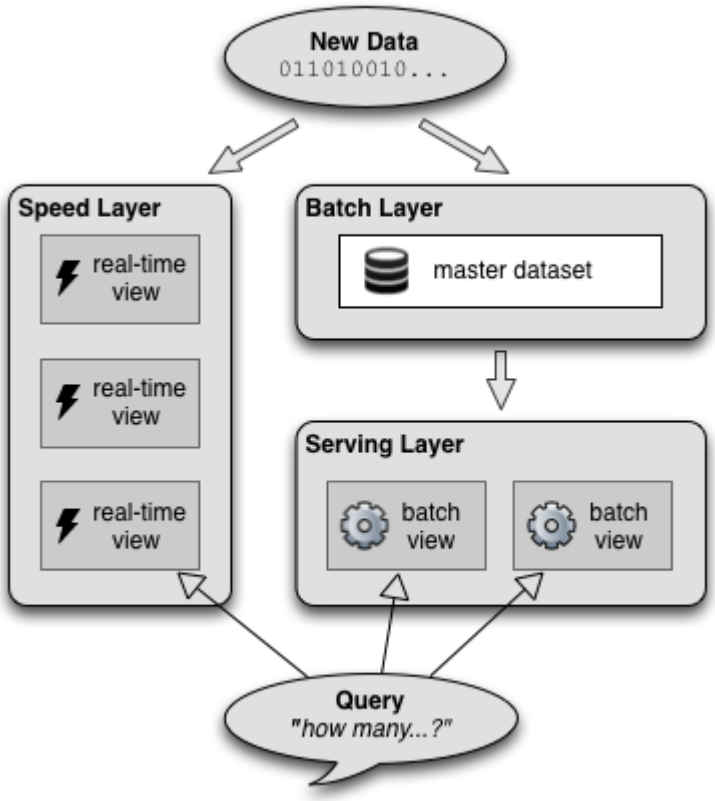


Figure 3.1 The batch layer must structure large, continually growing datasets in a manner that supports low maintenance as well as efficient creation of the batch views.

As with the last chapter, this chapter is dedicated to the master dataset. The master dataset is typically too large to exist on a single server, so you must choose how to distribute your data across multiple machines. The way you store your master dataset will impact how you consume it, so it is vital to devise your storage strategy with your usage patterns in mind.

In this chapter, you will

- determine the requirements for storing the master dataset
- examine a distributed file system that meets these requirements
- identify common tasks when using and maintaining your dataset
- use a library called Pail that abstracts away low level filesystem details when accessing your data
- implement the batch layer storage for our SuperWebAnalytics.com project

We begin by examining with how the role of batch layer within the Lambda Architecture affects how you should store your data.

3.1 Storage requirements for the master dataset

To determine the requirements for data storage, you must consider how your data is going to be written and how it will be read. The role of the batch layer within the Lambda Architecture affects both areas - we'll discuss each at a high level before providing a full list of requirements.

In the last chapter we emphasized two key properties of data: data is immutable and eternally true. Consequently each piece of your data will be written once and only once. There is no need to ever alter your data - the only write operation will be to add a new data unit to your dataset. The storage solution should therefore be optimized to handle a large, constantly growing set of data.

The batch layer is also responsible for computing functions on the dataset to produce the batch views. This means the batch layer storage system needs to be good at reading lots of data at once. In particular, random access to individual pieces of data is *not* required.

With this "write once, bulk read many times" paradigm in mind, we can create a checklist of requirements for the data storage - see Table 3.1.

Table 3.1 A checklist of storage requirements for the master dataset

Operation	Requisite	Discussion
writes	Efficient appends of new data	The basic write operation is to add new pieces of data, so it must be easy and efficient to append a new set of data objects to the master dataset.
	Scalable storage	The batch layer stores the complete dataset - potentially terabytes or petabytes of data. It must therefore be easy to scale the storage as your dataset grows.
reads	Support for parallel processing	Constructing the batch views requires computing functions on the entire master dataset. The batch storage must consequently support parallel processing to handle large amounts of data in a scalable manner.
	Ability to vertically partition data	Although the batch layer is built to run functions on the entire dataset, many computations don't require looking at all the data. For example, you may have a computation that only requires information collected during the past two weeks. The batch storage should allow you to partition your data so that a function only accesses data relevant to its computation. This process is called *vertical partitioning* and can greatly contribute to making the batch layer more efficient.
both	Tunable storage / processing costs	Storage costs money. You may choose to compress your data to help minimize your expenses. However, decompressing your data during computations can affect your performance. The batch layer should give you the flexibility to decide how to store and compress your data to suit your specific needs.

Let's now take a look at a specific batch layer storage solution that meets these requirements.

3.2 Implementing a storage solution for the batch layer

With our requirement checklist in hand, we can now consider options for the batch layer storage. There are many viable candidates, ranging from distributed file systems to key-value stores to document-oriented databases. This book emphasizes concepts, yet for this chapter we need to ground the discussion by focusing on a single platform. For this purpose, we have chosen the Hadoop Distributed File System (HDFS). Our reasons for doing so include that HDFS is

- an open-source project with an active developer community
- tightly coupled with Hadoop MapReduce, a distributed computing framework
- widely adopted and deployed in production systems by hundreds of companies

Regardless of the platform, the high level objectives for the batch layer storage remain the same: maintain a large, growing dataset in a robust, well-structured manner to efficiently generate the batch views. Many of the following details will only be applicable to HDFS, but high-level analogies will still apply to other technologies.

With that disclaimer, let's get started with HDFS. After a quick introduction, you'll learn how to store your master dataset using HDFS and how it meets the storage requirement checklist.

3.2.1 Introducing the Hadoop Distributed File System

HDFS and Hadoop MapReduce are the two prongs of the Hadoop project: a Java framework for distributed storage and distributed processing of large amounts of data. Hadoop is deployed across multiple servers, typically called a *cluster*, and HDFS is a distributed and scalable filesystem that manages how data is stored across the cluster. Hadoop is a project of significant size and depth, so we will only provide a high level description.

In a Hadoop cluster, there are two types of HDFS nodes: a single namenode and multiple datanodes. When you upload a file to HDFS, the file is first chunked into blocks of a fixed size, typically between 64MB and 256 MB. Each block is then replicated across multiple datanodes (typically three) that are chosen at random. The namenode keeps track of the file-to-block mapping and where each block is located. This design is shown in Figure 3.2.

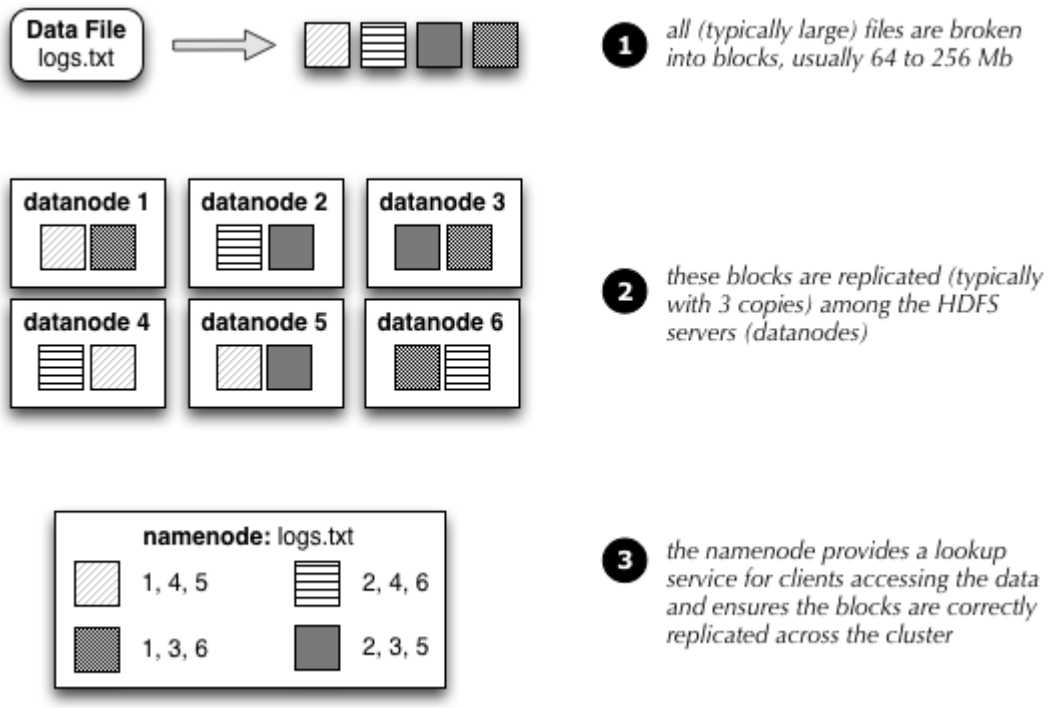


Figure 3.2 Files are chunked into blocks which are dispersed to datanodes in the cluster.

Distributing a file in this way across many nodes allows it to be easily processed in parallel. When a program needs to access a file stored in HDFS, it contacts the namenode to determine which datanodes host the file contents. This process is illustrated in Figure 3.3.

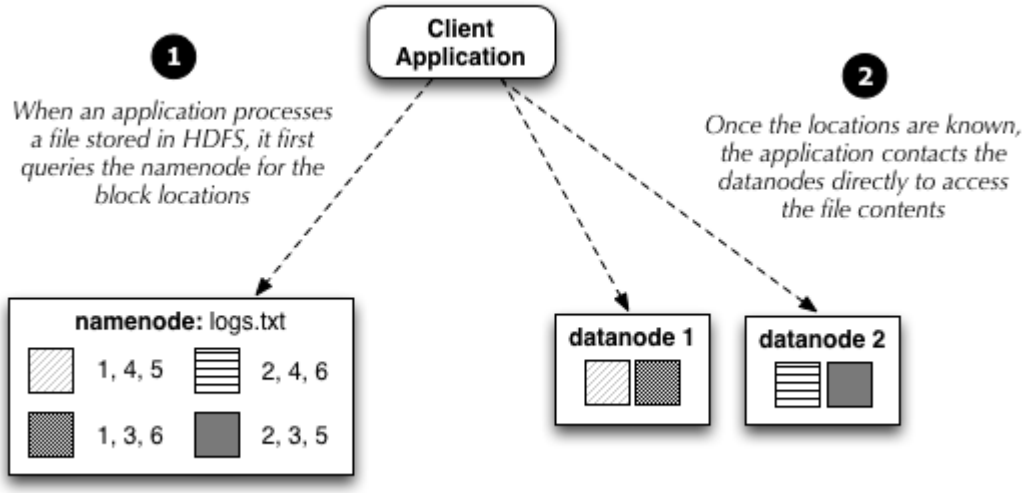


Figure 3.3 Files are chunked into blocks which are dispersed to datanodes in the cluster.

Additionally, by replicating each block across multiple nodes, your data

remains available even when individual nodes are offline.

SIDEBAR **Getting started with Hadoop**

Setting up Hadoop can be an arduous task. Hadoop has numerous configuration parameters that should be tuned for your hardware to perform optimally. To avoid getting bogged down in details, we recommend downloading a preconfigured virtual machine for your first encounter with Hadoop. A virtual machine will accelerate your learning of HDFS and MapReduce, and you will have an better understanding when setting up your own cluster.

At the time of this writing, Hadoop vendors Cloudera, Hortonworks and MapR all have made images publicly available. We recommend having access to Hadoop so you can follow along with the examples in this and later chapters.

Implementing a distributed file system is a difficult task, but this short primer covers the basics from a user perspective. Let's now explore how to store a master dataset using HDFS.

3.2.2 Storing a master dataset with HDFS

As a filesystem, HDFS offers support for files and directories. This makes storing a master dataset on HDFS straightforward. You store data units sequentially in files, with each file containing megabytes or gigabytes of data. All the files of a dataset are then stored together in a common folder in HDFS. To add new data to the dataset, you simply create and upload another file containing the new information. We will demonstrate this with a simple dataset.

Suppose you wanted to store all logins on a server. The following listing contains some example logins.

```
$ cat logins-2012-10-25.txt
alex     192.168.12.125   Thu Oct 25 22:33 - 22:46 (00:12)
bob      192.168.8.251    Thu Oct 25 21:04 - 21:28 (00:24)
charlie  192.168.12.82    Thu Oct 25 21:02 - 23:14 (02:12)
doug     192.168.8.13     Thu Oct 25 20:30 - 21:03 (00:33)
...
```

To store this data on HDFS, you create a directory for the dataset and upload the file.

```
$ hadoop fs -mkdir /logins   (1)
$ hadoop fs -put logins-2012-10-25.txt /logins   (2)
```

(1) The "hadoop fs" commands are Hadoop shell commands that interact directly with HDFS. A full list is available at http://hadoop.apache.org/.
(2) Uploading a file automatically chunks and distributes the blocks across the datanodes

You can list the directory contents:

```
$ hadoop fs -ls -R /logins   (1)
-rw-r--r--   3 hdfs hadoop  175802352 2012-10-26 01:38
    /logins/logins-2012-10-25.txt
```

(1) The ls command is based upon the UNIX command of the same name

And verify the contents of the file:

```
$ hadoop fs -cat /logins/logins-2012-10-25.txt
alex     192.168.12.125   Thu Oct 25 22:33 - 22:46 (00:12)
bob      192.168.8.251    Thu Oct 25 21:04 - 21:28 (00:24)
...
```

As we mentioned earlier, the file was automatically chunked into blocks and distributed among the datanodes when it was uploaded. You can identify the blocks and their locations through the following command:

```
$ hadoop fsck /logins/logins-2012-10-25.txt  -files -blocks -locations

/logins/logins-2012-10-25.txt 175802352 bytes, 2 block(s):   (1)
OK
0. blk_-1821909382043065392_1523 len=134217728   (2)
   repl=3 [10.100.0.249:50010, 10.100.1.4:50010, 10.100.0.252:50010]
1. blk_2733341693279525583_1524 len=41584624
   repl=3 [10.100.0.255:50010, 10.100.1.2:50010, 10.100.1.5:50010]
```

(1)

1. the file is stored in two blocks
2. the ip addresses and port numbers of the datanodes hosting each block

Nested folders provide an easy implementation of vertical partitioning. For our logins example, you may want to partition your data by login date. This could be accomplished by a layout shown in Figure 3.4. By storing each day's information in a separate subfolder, a function can pass over data not relevant to its computation.

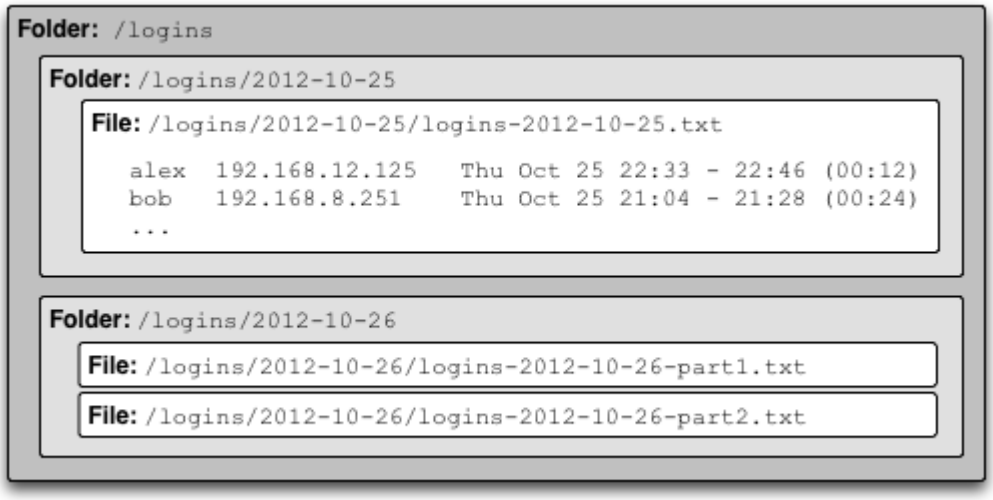


Figure 3.4 A vertical partitioning scheme for login data. By separating information for each date in a separate folder, a function can select only the folders containing data relevant to its computation.

We earlier asserted that HDFS meets the storage requirements of the batch layer, but we held off going through the checklist so we could provide more background. We now return to our list to verify our claim.

3.2.3 HDFS: Meeting the storage requirement checklist

Most of these points were discussed individually, but it is useful to compile the full list - see Table 3.2.

Table 3.2 How HDFS meets the storage requirement checklist

Operations	Criteria	Discussion
writes	Efficient appends of new data	Appending new data is as simple as adding a new file to the folder containing the master dataset.
	Scalable storage	HDFS evenly distributes the storage across a cluster of machines. You increase storage space and I/O throughput by adding more machines.
reads	Support for parallel processing	HDFS integrates with Hadoop MapReduce, a parallel computing framework that can compute nearly arbitrary functions on the data stored in HDFS.
	Ability to vertically partition data	Vertical partitioning is done grouping data into subfolders. A function can read only the select set of subfolders needed for its computation.
both	Tunable storage / processing costs	You have full control over how you store your data units within the HDFS files. You choose the file format for your data as well as the level of compression.

While HDFS is a powerful tool for storing your data, there are common tasks required to maintain your master dataset. We'll cover these tasks next, paving the way to introduce a library that abstracts away low-level interactions with HDFS so you can focus solely on your data.

3.3 Maintaining the batch storage layer in HDFS

Maintaining your dataset should be an easy chore given that the only write operation for the batch layer is to append new data. For HDFS, maintenance is effectively two operations:

- appending new files to the master dataset folder, and
- consolidating data to remove small files

These operations must be bulletproof to preserve the integrity and performance of the batch layer. In this section, we will cover these tasks as well as potential pitfalls. We introduce these issues both to further explain HDFS as well as to motivate a higher-level abstraction that can handle these difficulties for you. After all, your focus should be on using the data in your Big Data system, not worrying about maintaining it.

3.3.1 Appending to the master dataset

All Big Data systems must be capable of merging new data into the master dataset. In the context of HDFS, this means adding new files to the master dataset folder. Listing 3.1 provides a basic function that directly uses the HDFS API to merge the contents of two folders.

Listing 3.1 A rudimentary implementation to merge the contents of two HDFS folders

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class SimpleMerge {
  public static void mergeFolders(String destDir, String sourceDir)
    throws IOException
  {
    Path destPath = new Path(destDir);                                  (1)
    Path sourcePath = new Path(sourceDir);

    FileSystem fs = sourcePath.getFileSystem(new Configuration());      (2)
    for(FileStatus current: fs.listStatus(sourcePath)) {                (3)
      Path curPath = current.getPath();                                 (4)
      fs.rename(curPath, new Path(destPath, curPath.getName()));
    }
  }
}
```

(1) the HDFS API uses Path objects for both files and directories
(2) a FileSystem object is needed to manipulate paths
(3) a loop is needed to individually access each file in the source directory
(4) a new Path is constructed for each file and the moved to the destination folder

The most apparent aspect to this code is the low-level nature of the HDFS API:

four separate classes are needed to accomplish a rather basic task. Code written at this level introduces complexity that is not related to the task at hand.

Furthermore, the code is far from robust. Let's consider some adverse scenarios:

1. **Non-unique filenames:** If the same filename exists in both source and destination folders, you could overwrite - and hence lose - previously stored data.
2. **Non-standard file formats:** The new files are blindly merged into the destination folder. You could corrupt the master dataset if your new files have a different format.
3. **Inconsistent vertical partitioning:** Similarly, the directory structure of the new data may not match the desired vertical partitioning scheme. Each file may be valid, but queries against the merged dataset could return invalid results.

It is certainly possible to modify the function to handle these situations, but working with a low-level API requires more effort and increases the risk of introducing bugs into your code. A high-level abstraction for appending new data would be both a faster and safer solution.

The second HDFS maintenance task is needed if you regularly append small files to your master dataset, which we will briefly discuss now.

3.3.2 Consolidating the master dataset to eliminate small files

Hadoop HDFS and MapReduce are tightly integrated to form a framework for storing and processing large amounts of data. We will discuss MapReduce in detail in the next chapter, but a characteristic of Hadoop is that computing performance is significantly degraded when data is stored in many small files in HDFS. There can be an order of magnitude difference in performance between a MapReduce job that consumes 10GB stored in many small files versus a job processing that same data stored in a few large ones.

The reason is that a MapReduce job launches multiple tasks, one for each block in the input dataset. Each task requires some overhead to plan and coordinate its execution, and since each small file requires a separate task, the cost is repeatedly incurred. This property of MapReduce means you will want to consolidate your data should small files become abundant within your dataset. You can achieve this by either writing code that uses the HDFS API or a custom MapReduce job, but both will require considerable work and knowledge of Hadoop internals.

Instead of using the low-level HDFS API to append new data and consolidate small files, it would be preferable to have high-level functions for these tasks. We introduce one possible example next.

3.3.3 The need for a high level abstraction

In contrast to the code using the HDFS API, consider Listing 3.2 that uses the Pail library:

Listing 3.2 Abstractions of HDFS maintenance tasks

```
import java.io.IOException;
import backtype.hadoop.pail.Pail;

public class PailMove {

  public static void mergeData(String masterDir, String updateDir)
    throws IOException
  {
    Pail target = new Pail(masterDir);
    Pail source = new Pail(updateDir);
    target.absorb(source);
    target.consolidate();
  }
}
```

1. Pails are wrappers around HDFS folders
2. with the Pail library, apppends are one-line operations
3. small data files within the pail also can be consolidated with a single function call

With Pail, you can append folders in one line of code and consolidate small files in another. It throws an exception if either operation is invalid for any reason. Most importantly, a higher level abstraction like Pail allows you to work with your data directly rather than using containers like files and directories.

A QUICK RECAP

Before learning more about Pail, now is a good time to step back and regain the bigger perspective. Recall that the master dataset is the source of truth within the Lambda Architecture, and as such the batch layer must handle a large, growing dataset without fail. Furthermore, there must be an easy and effective means of transforming the data into batch views to answer actual queries.

There are many candidates to use for storing the master dataset, but HDFS is most commonly chosen for this purpose due to its integration with Hadoop MapReduce. However, the API for interacting with HDFS directly is low-level and requires in-depth knowledge to robustly perform tasks. A high-level abstraction

releases you from worrying about these details, letting you focus on using your data instead.

This chapter is more technical than the previous ones, but always keep in mind how everything integrates within the Lambda Architecture.

3.4 Data storage in the Batch Layer with Pail

Pail is a thin abstraction over files and folders from the dfs-datastores library (http://github.com/nathanmarz/dfs-datastores). This abstraction makes it significantly easier to manage a collection of records for batch processing. As the name suggests, Pail uses *pails*, folders that keep metadata about the dataset. By using this metadata, Pail allows you to safely act upon the batch layer without worrying about violating its integrity.

Under the hood, Pail is just a Java library that uses the standard Hadoop APIs. It handles the low-level filesystem interaction, providing an API that isolates you from the complexity of Hadoop's internals. The intent is to allow you to focus on the data itself instead of concerning yourself with how it is stored and maintained.

SIDEBAR **Why the focus on Pail?**

Pail, along with many other packages covered in this book, was written by Nathan while developing the Lambda Architecture. We introduce these technologies not to promote them but to discuss the context of their origins and the problems they solve. Feel free to use other libraries or develop your own - our emphasis is why these solutions are necessary and the benefits they provide.

You have already seen the characteristics of HDFS that make it a viable choice for storing the master dataset in the batch layer. As you explore Pail, keep in mind how it preserves the advantages of HDFS while streamlining operations on the data. After you have covered the basic operations of Pail, we will summarize the overall value provided by the library.

Now, let's dive right in and see how Pail works by creating and writing data to a pail.

3.4.1 Basic Pail operations

The best way to understand how Pail works is to follow along and run the presented code on your computer. To do this, you will need to download the source from GitHub and build the dfs-datastores library. If you don't have a Hadoop cluster or virtual machine available, your local filesystem will be treated as HDFS in the examples. You'll then be able to see the results of these commands by inspecting the relevant directories on your filesystem.

Let's start off by creating a new pail and storing some data.

```
public static void simpleIO() throws IOException {
  Pail pail = Pail.create("/tmp/mypail");                    (1)
  TypedRecordOutputStream os = pail.openWrite();             (2)
  os.writeObject(new byte[] {1, 2, 3});                      (3)
  os.writeObject(new byte[] {1, 2, 3, 4});
  os.writeObject(new byte[] {1, 2, 3, 4, 5});
  os.close();                                                (4)
}
```

(1) create a default pail in the specified directory
(2) provide an output stream to a new file in the Pail
(3) a pail without metadata is limited to storing byte arrays
(4) close the current file

When you check your filesystem, you'll see that a folder for "/tmp/mypail" was created and contains two files:

```
root:/ $ ls /tmp/mypail
f2fa3af0-5592-43e0-a29c-fb6b056af8a0.pailfile   (1)
pail.meta                                       (2)
```

(1) the records are stored within pailfiles
(2) the metadata describes the contents and structure of the pail

The pailfile contains the records you just stored. The file is created atomically, so all the records you created will appear at once - that is, an application that reads from the pail will not see the file until the writer closes it. Furthermore, pailfiles use globally unique names (so it will be named differently on your filesystem).

These unique names allow multiple sources to write concurrently to the same pail without conflict.

The other file in the directory contains the pail's metadata. This metadata describes the type of the data as well as how it is stored within the pail. The example did not specify any metadata when constructing the pail, so this file contains the default settings:

```
root:/ $ cat /tmp/mypail/pail.meta
---
format: SequenceFile   (1)
args: {}   (2)
```

(1) the format of files in the pail; a default pail stores data in key-value pairs within Hadoop SequenceFiles
(2) the arguments describe the contents of the pail; an empty map directs Pail to treat the data as uncompressed byte arrays

Later in the chapter you will see another pail.meta file containing more substantial metadata, but the overall structure will remain the same. We next cover how to store real objects in Pail, not just binary records.

3.4.2 Serializing objects into pails

To store objects within a pail, you must provide Pail with instructions for serializing and deserializing your objects to and from binary data. Let's return to the server logins example to demonstrate how this is done. Listing 3.3 has a simplified class to represent a login.

Listing 3.3 A no-frills class for logins

```
public class Login {
  public String userName;
  public long loginUnixTime;

  public Login(String _user, long _login) {
    userName = _user;
    loginUnixTime = _login;
  }
}
```

To store these Login objects in a pail, you need to create a class that

implements the PailStructure interface. Listing 3.4 defines a LoginPailStructure that describes how serialization should be performed.

Listing 3.4 Implementing the PailStructure interface

```
public class LoginPailStructure implements PailStructure<Login>{

  public Class getType() {                                    (1)
    return Login.class;
  }

  public byte[] serialize(Login login) {                      (2)
    ByteArrayOutputStream byteOut = new ByteArrayOutputStream();
    DataOutputStream dataOut = new DataOutputStream(byteOut);
    byte[] userBytes = login.userName.getBytes();
    try {
      dataOut.writeInt(userBytes.length);
      dataOut.write(userBytes);
      dataOut.writeLong(login.loginUnixTime);
      dataOut.close();
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    return byteOut.toByteArray();
  }

  public Login deserialize(byte[] serialized) {               (3)
    DataInputStream dataIn =
        new DataInputStream(new ByteArrayInputStream(serialized));
    try {
      byte[] userBytes = new byte[dataIn.readInt()];
      dataIn.read(userBytes);
      return new Login(new String(userBytes), dataIn.readLong());
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }

  public List<String> getTarget(Login object) {              (4)
    return Collections.EMPTY_LIST;
  }

  public boolean isValidTarget(String... dirs) {              (5)
    return true;
  }
}
```

(1) a pail with this structure will only store Login objects
(2) Login objects must be serialized when store in pailfiles
(3)

Logins are later reconstructed when read from pailfiles

4. the getTarget method defines the vertical partitioning scheme but is not used in this example
5. isValidTarget determines whether the given path matches the vertical partitioning scheme but is also not used in this example

By passing this LoginPailStructure to the Pail create function, the resulting pail will use these serialization instructions. You can then give it Login objects directly and Pail will handle the serialization automatically.

```
public static void writeLogins() throws IOException {
  Pail<Login> loginPail = Pail.create("/tmp/logins",
                                      new LoginPailStructure());  (1)
  TypedRecordOutputStream out = loginPail.openWrite();
  out.writeObject(new Login("alex", 1352679231));
  out.writeObject(new Login("bob", 1352674216));
  out.close();
}
```

1. create a pail with the new pail structure

Likewise, when you read the data, Pail will deserialize the records for you. Here's how you can iterate through all the objects you just wrote:

```
public static void readLogins() throws IOException {
  Pail<Login> loginPail = new Pail<Login>("/tmp/logins");
  for(Login l : loginPail) {  (1)
    System.out.println(l.userName + " " + l.loginUnixTime);
  }
}
```

1. a pail supports the Iterable interface for its object type

Once your data is stored within a pail, you can use Pail's built-in operations to safely act upon it.

3.4.3 Batch operations using Pail

Pail has built-in support for a number of common operations. These operations are where you will see the benefits of managing your records with Pail rather than doing it manually. The operations are all implemented using MapReduce so they scale regardless the amount of data in your pail, whether gigabytes or terabytes. We'll be talking about MapReduce a lot more in later chapters, but the key takeaway is that the operations are automatically parallelized and executed across a cluster of worker machines.

In the previous section we discussed the importance of append and consolidate operations. As you would expect, Pail has support for both. The append operation is particularly smart. It checks the pails to verify it is valid to append the pails together. For example, it won't allow you to append a pail containing strings to a pail containing integers. If the pails store the same type of records but in different file formats, it coerces the data to match the format of the target pail.

By default, the consolidate operation merges small files to create new files that are as close to 128MB as possible - a standard HDFS block size. This operation also uses a MapReduce job to accomplish its task.

For our logins example, suppose you had additional logins in a separate pail and wanted to merge the data into the original pail. The following code performs both the append and consolidate operations:

```
public static void appendData() throws IOException {
  Pail<Login> loginPail = new Pail<Login>("/tmp/logins");
  Pail<Login> updatePail = new Pail<Login>("/tmp/updates");
  loginPail.absorb(updatePail);
}
```

The major upstroke is that these built-in functions let you focus on what you want to do with your data rather than worry about how to manipulate files correctly.

3.4.4 Vertical partitioning with Pail

We earlier mentioned that you can vertically partition your data in HDFS by using multiple folders. Imagine trying to manage the vertical partitioning manually. It is all too easy to forget that two datasets are partitioned differently and mistakenly append them. Similarly, it wouldn't be hard to accidentally violate the partitioning structure when consolidating your data. Thankfully, Pail is smart about enforcing the structure of a pail and protects you from making these kinds of mistakes.

To create a partitioned directory structure for a pail, you must implement two additional methods of the PailStructure interface:

- **getTarget:** given a record, determines the directory structure where it should be stored and returns the path as a list of Strings.
- **isValidTarget:** given an array of Strings, builds a directory path and determines if it is consistent with the vertical partitioning scheme.

Pail uses these methods to enforce its structure and automatically map records to their correct subdirectory. The following code demonstrates how to partition Login objects so that records are grouped by the login date.

Listing 3.5 A vertical partitioning scheme for Login records

```
public class PartitionedLoginPailStructure extends LoginPailStructure {
  SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");

  public List<String> getTarget(Login object) {                    (1)
    ArrayList<String> directoryPath = new ArrayList<String>();
    Date date = new Date(object.loginUnixTime * 1000L);            (2)
    directoryPath.add(formatter.format(date));
    return directoryPath;
  }

  public boolean isValidTarget(String... strings) {                (3)
    if(strings.length != 1) return false;
    try {
      return (formatter.parse(strings[0]) != null);
    }
    catch(ParseException e) {
      return false;
    }
  }
}
```

(1) Logins are vertically partitioned in folders corresponding to the login date

2. the timestamp of the Login object is converted to an understandable form
3. isValidTarget verifies the directory structure has a depth of one and that the folder name is a date

With this new pail structure, Pail determines the correct subfolder whenever it writes a new Login object:

```
public static void partitionData() throws IOException {
  Pail<Login> pail = Pail.create("/tmp/partitioned_logins",
                                 new PartitionedLoginPailStructure());
  TypedRecordOutputStream os = pail.openWrite();
  os.writeObject(new Login("chris", 1352702020));   (1)
  os.writeObject(new Login("david", 1352788472));   (2)
  os.close();
}
```

1. 1352702020 is the timestamp for 2012-11-11, 22:33:40 PST
2. 1352788472 is the timestamp for 2012-11-12, 22:34:32 PST

Examining this new pail directory confirms the data was partitoned correctly.

```
root:/ $ ls -R /tmp/partitioned_logins
2012-11-11  2012-11-12  pail.meta

/tmp/partitioned_logins/2012-11-11:   (1)
d8c0822b-6caf-4516-9c74-24bf805d565c.pailfile

/tmp/partitioned_logins/2012-11-12:
d8c0822b-6caf-4516-9c74-24bf805d565c.pailfile
```

1. folders for the different login dates are created within the pail

3.4.5 Pail file formats and compression

Pail stores data in multiple files within its directory structure. You can control how Pail stores records in those files by specifying the file format Pail should be using. This lets you control the tradeoff between the amount of storage space Pail uses and the performance of reading records from fail. As discussed earlier in the chapter, this is a fundamental knob you need to have control of to match your application needs.

You can implement your own custom file format, but by default Pail uses Hadoop SequenceFiles. This format is very widely used, allows an individual file

to be processed in parallel via MapReduce, and has native support for compressing the records in the file.

To demonstrate these options, here's how to create a Pail that uses the SequenceFile format with gzip block compression:

```
public static void createCompressedPail() throws IOException {
  Map<String, Object> options = new HashMap<String, Object>();
  options.put(SequenceFileFormat.CODEC_ARG,
              SequenceFileFormat.CODEC_ARG_GZIP);                    (1)
  options.put(SequenceFileFormat.TYPE_ARG,
              SequenceFileFormat.TYPE_ARG_BLOCK);                    (2)
  LoginPailStructure struct = new LoginPailStructure();
  Pail compressed = Pail.create("/tmp/compressed",                   (3)
                        new PailSpec("SequenceFile", options, struct));
}
```

(1) contents of the pail will be gzip compressed
(2) blocks of records will be compressed together (as compared to compressing rows individually)
(3) create a new pail to store Login options with the desired format

You can then observe these properties in the pail's metadata.

```
root:/ $ cat /tmp/compressed/pail.meta
---
format: SequenceFile
structure: manning.LoginPailStructure                                (1)
args:
  compressionCodec: gzip                                             (2)
  compressionType: block
```

(1) the full class name of the LoginPailStructure
(2) the compression options for the pailfiles

Whenever records are added to this pail, they will be automatically compressed. This pail will use significantly less space at the cost of a higher CPU cost for reading and writing records.

3.4.6 Summarizing the benefits of Pail

Having invested the time probing the inner workings of Pail, it's important to understand the benefits it provides over raw HDFS. Table 3.3 summarizes the impact of Pail in regard to our earlier checklist of batch layer storage requirements.

Table 3.3 The advantages of Pail for storing the master dataset.

Operations	Criteria	Discussion
writes	Efficient appends of new data	Pail has a first class interface for appending data and prevents you from performing invalid operations - something the raw HDFS API won't do for you.
	Scalable storage	The namenode holds the entire HDFS namespace in memory and can be taxed if the filesystem contains a vast number of small files. Pail's consolidate operator decreases the total number of HDFS blocks and eases the demand upon the namenode.
reads	Support for parallel processing	The number of tasks in a MapReduce job is determined by the number of blocks in the dataset. Consolidating the contents of a pail lowers the number of required tasks and increases the efficiency of processing the data.
	Ability to vertically partition data	Output written into a pail is automatically partitioned with each fact stored in its appropriate directory. This directory structure is strictly enforced for all Pail operations.
both	Tunable storage / processing costs	Pail has built-in support to coerce data into the format specified by the pail structure. This coercion occurs automatically while performing operations on the pail.

That concludes our whirlwind tour through Pail. It is a useful and powerful abstraction for interacting with your data in the batch layer while isolating you from the details of the underlying filesystem.

3.5 Putting it all together for SuperWebAnalytics.com

This chapter has been quite a journey. You began with the storage requirements of the batch layer, ventured into the properties and benefits of using HDFS to store the master dataset, and finally arrived at a high level library to simplify maintaining and interacting with your data. You'll conclude this chapter by wrapping it all together for our SuperWebAnalytics.com example.

When you last left this project, you had created a graph schema to express the entities, edges and properties of the dataset. The structure of these facts was defined using Thrift - the snippet below serves as a quick reminder.

```
struct Data {                                    ❶
  1: required Pedigree pedigree;
  2: required DataUnit dataunit;
}

union DataUnit {                                 ❷
  1: PersonProperty person_property;
  2: PageProperty page_property;
  3: EquivEdge equiv;
  4: PageViewEdge page_view;
}

union PersonPropertyValue {                      ❸
  1: string full_name;
  2: GenderType gender;
  3: Location location;
}
```

❶ all facts in the dataset are represented as a timestamp and a base unit of data
❷ the fundamental data unit describes the edges and properties of the dataset
❸ property value can be of multiple types

A key observation is that the unions of a graph schema provide a natural vertical partitioning of the data. This is illustrated in Figure 3.5

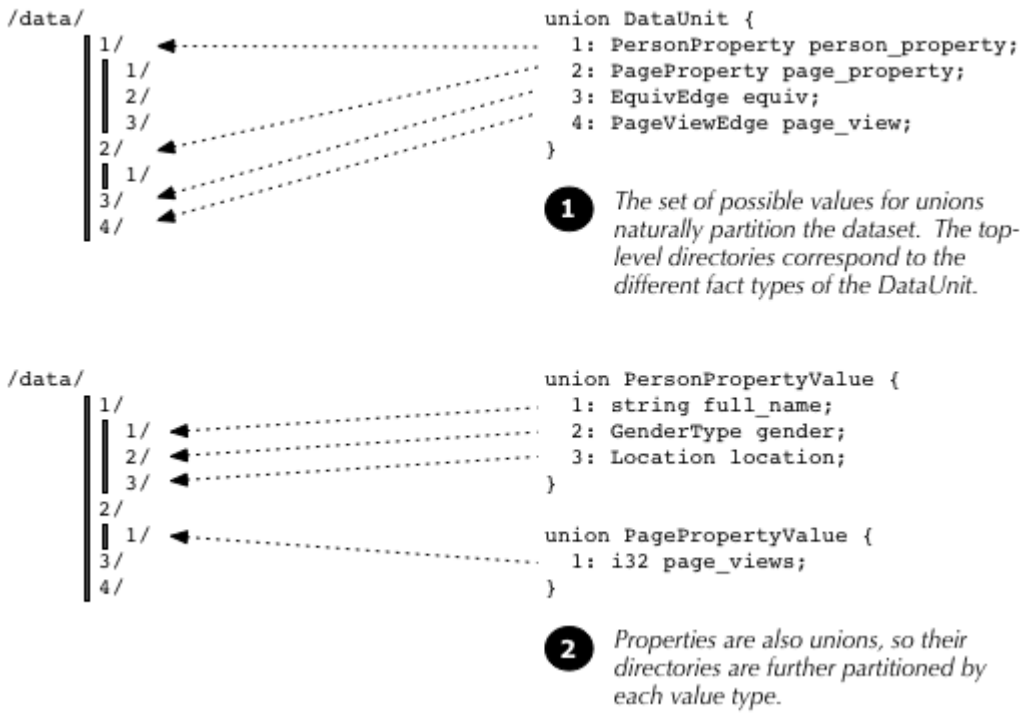


Figure 3.5 The unions within a graph schema provide a natural vertical partitioning scheme for a dataset.

To use HDFS and Pail for SuperWebAnalytics.com, we must define a structured pail to store Data objects that also enforces this vertical partitioning scheme. This code is a bit involved, so we will present it in steps:

- First, you will create an abstract pail structure for storing Thrift objects. Thrift serialization is independent of the type of data being stored, and the code is cleaner by separating this logic.
- Next, you will derive a pail structure from the abstract class for storing SuperWebAnalytics.com Data objects.
- Finally you will define a further subclass that will implement the desired vertical partitioning schem.

Throughout this section, don't worry about the details of the code. What matters is that this code works for any graph schema, and it continues to work even as the schema evolves over time.

3.5.1 A Structured Pail for Thrift Objects

Creating a pail structure for Thrift objects is surprisingly easy since Thrift does the heavy lifting for you. Listing 3.6 demonstrates how to use Thrift utilities to serialize and deserialize your data.

Listing 3.6 A generic abstract pail structure for serializing Thrift objects

```
public abstract class ThriftPailStructure<T extends Comparable>
  implements PailStructure<T>                                    (1)
{
  private transient TSerializer ser;                             (2)
  private transient TDeserializer des;

  private TSerializer getSerializer() {                          (3)
    if(ser==null) ser = new TSerializer();
    return ser;
  }

  private TDeserializer getDeserializer() {
    if(des==null) des = new TDeserializer();
    return des;
  }

  public byte[] serialize(T obj) {
    try {
      return getSerializer().serialize((TBase)obj);              (4)
    } catch (TException e) {
      throw new RuntimeException(e);
    }
  }

  public T deserialize(byte[] record) {
    T ret = createThriftObject();                                (5)
    try {
      getDeserializer().deserialize((TBase)ret, record);
    } catch (TException e) {
      throw new RuntimeException(e);
    }
    return ret;
  }

  protected abstract T createThriftObject();                     (6)
}
```

(1) Java Generics allow the pail structure to be used for any Thrift object
(2) TSerializer and TDeserializer are Thrift utilities for serializing objects to and from binary arrays
(3) the Thrift utilities are lazily built, constructed only when required
(4) the data object is cast to a basic Thrift object for serialization
(5) a new data object is constructed prior to deserialization
(6) the constructor of the data object must be implemented in the child class

3.5.2 A Basic Pail for SuperWebAnalytics.com

Next you can define a basic class for storing SuperWebAnalytics.com Data objects by creating a concrete subclass of ThriftPailStructure - see Listing 3.7.

Listing 3.7 A concrete implementation for Data objects

```
public class DataPailStructure extends ThriftPailStructure<Data> {
  public Class getType() {                                          (1)
    return Data.class;
  }

  protected Data createThriftObject() {                             (2)
    return new Data();
  }

  public List<String> getTarget(T object) {                         (3)
    return Collections.EMPTY_LIST;
  }

  public boolean isValidTarget(String... dirs) {
    return true;
  }
}
```

(1) specify the Pail stores Data objects
(2) needed by ThriftPailStructure to create an object for deserialization
(3) this pail structure does not use vertical partitioning

3.5.3 A Split Pail to Vertically Partition the DataSet.

The last step is to create a pail structure that implements the vertical partitioning strategy for a graph schema. It is also the most complex step. All of the following snippets are extracted from the SplitDataPailStructure class that accomplishes this task.

At a high-level, the SplitDataPailStructure code inspects the DataUnit class to create a map between thrift ids and classes to process the corresponding type. Figure 3.6 demonstrates this map for SuperAnalytics.com.

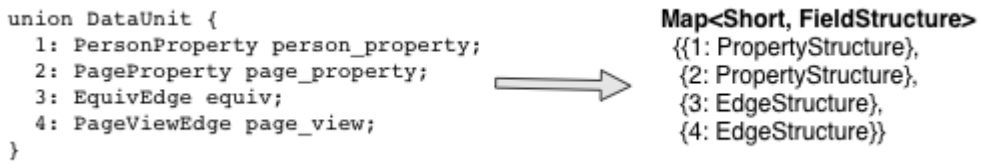


Figure 3.6 The SplitDataPailStructure field map for the DataUnit class of SuperWebAnalytics.com

Listing 3.8 contains the code to generate the field map. It works for any graph schema, not just this example.

Listing 3.8 Code to generate the field map for a graph schema

```
public class SplitDataPailStructure extends DataPailStructure {

  public static HashMap<Short, FieldStructure> validFieldMap =
    new HashMap<Short, FieldStructure>();                          (1)

  static {
    for(DataUnit._Fields k: DataUnit.metaDataMap.keySet()) {       (4)
      FieldValueMetaData md = DataUnit.metaDataMap.get(k).valueMetaData;
      FieldStructure fieldStruct;
      if(md instanceof StructMetaData &&
        ((StructMetaData) md).structClass
          .getName().endsWith("Property"))                         (3)
      {
        fieldStruct = new PropertyStructure(
          ((StructMetaData) md).structClass);
      } else {
        fieldStruct = new EdgeStructure();                         (4)
      }
      validFieldMap.put(k.getThriftFieldId(), fieldStruct);
    }
  }

  // remainder of class elided
}
```

(1) FieldStructure is an interface for both edges and properties
(2) Thrift code to inspect and iterate over the DataUnit object
(3) properties are identified by the class name of the inspected object
(4) if class name doesn't end with "Property", it must be an edge

As mentioned in the code annotation, FieldStructure is an interface shared by both PropertyStructure and EdgeStructure. The definition of the interface is as follows:

```
protected static interface FieldStructure {
  public boolean isValidTarget(String[] dirs);
  public void fillTarget(List<String> ret, Object val);
}
```

We will later provide the details for the EdgeStructure and PropertyStructure classes, but we first show how this interface is used to accomplish the vertical partitioning of the table.

```
// methods are from SplitDataPailStructure

public List<String> getTarget(Data object) {
  List<String> ret = new ArrayList<String>();
  DataUnit du = object.get_dataunit();
  short id = du.getSetField().getThriftFieldId();
  ret.add("" + id);                                          (1)
  validFieldMap.get(id).fillTarget(ret, du.getFieldValue()); (2)
  return ret;
}

public boolean isValidTarget(String[] dirs) {
  if(dirs.length==0) return false;
  try {
    short id = Short.parseShort(dirs[0]);
    FieldStructure s = validFieldMap.get(id);
    if(s==null)                                              (3)
      return false;
    else
      return s.isValidTarget(dirs);                          (4)
  } catch(NumberFormatException e) {
    return false;
  }
}
```

(1) the top-level directory is determined by inspecting the DataUnit
(2) any further partitioning is passed to the FieldStructure
(3) the validity check first verifies the DataUnit field id is in the field map
(4) any additional checks are passed to the FieldStructure

The SplitDataPailStructure is responsible for the top-level directory of the vertical partitioning, and it passes the responsibility of any additional subdirectories to the FieldStructure classes. Therefore, once we define the EdgeStructure and PropertyStructure classes, our work will be done.

Edges are structs and hence cannot be further partitioned. This makes the

EdgeStructure class trivial.

```
protected static class EdgeStructure implements FieldStructure {
  public boolean isValidTarget(String[] dirs) { return true; }
  public void fillTarget(List<String> ret, Object val) { }
}
```

However, properties are unions like the DataUnit class. The code similarly uses inspection to create a set of valid Thrift field ids for the given property class. For completeness we provide the full listing of the class in Listing 3.9, but the keynotes are the construction of the set and the use of this set in fulfilling the FieldStructure contract.

Listing 3.9

```
protected static class PropertyStructure implements FieldStructure {
  private short valueId;                                          (1)
  private HashSet<Short> validIds;                                (2)

  public PropertyStructure(Class prop) {
    try {
      Map<TFieldIdEnum, FieldMetaData> propMeta = getMetadataMap(prop);
      Class valClass = Class.forName(prop.getName() + "Value");
      valueId = getIdForClass(propMeta, valClass);                (3)

      validIds = new HashSet<Short>();
      Map<TFieldIdEnum, FieldMetaData> valMeta
        = getMetadataMap(valClass);
      for(TFieldIdEnum valId: valMeta.keySet()) {
        validIds.add(valId.getThriftFieldId());                   (4)
      }
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }

  public boolean isValidTarget(String[] dirs) {
    if(dirs.length < 2) return false;                             (5)
    try {
      short s = Short.parseShort(dirs[1]);
      return validIds.contains(s);
    } catch(NumberFormatException e) {
      return false;
    }
  }

  public void fillTarget(List<String> ret, Object val) {
    ret.add("" + ((TUnion) ((TBase)val)
      .getFieldValue(valueId))
      .getSetField()
      .getThriftFieldId());                                       (6)
    }
  }

  private static Map<TFieldIdEnum, FieldMetaData>
    getMetadataMap(Class c)                                       (7)
{
    try {
      Object o = c.newInstance();
      return (Map) c.getField("metaDataMap").get(o);
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
```

```
  }

  private static short getIdForClass(
    Map<TFieldIdEnum, FieldMetaData> meta, Class toFind)
  {
    for(TFieldIdEnum k: meta.keySet()) {
      FieldValueMetaData md = meta.get(k).valueMetaData;
      if(md instanceof StructMetaData) {
        if(toFind.equals(((StructMetaData) md).structClass)) {
          return k.getThriftFieldId();
        }
      }
    }

    throw new RuntimeException("Could not find " + toFind.toString() +
      " in " + meta.toString());
  }
```

1. a Property is a Thrift struct containing a property value field; this is the property value Thrift field id
2. the set of Thrift ids of the property value types
3. parse the Thrift metadata to get the field id of the property value
4. parse the metadata to get all valid field ids of the property value
5. the vertical partitioning of a property value has a depth of at least two
6. use the thrift ids to create the directory path for the current fact
7. getMetadataMap and getIdForClass are helper functions for inpecting Thrift objects

After that last bit of code, take a break - you've earned it. The good news is that this was a one-time cost. Once you have defined a pail structure for your master dataset, future interaction with the batch layer will be straightforward. Moreover, this code can be applied to any project where you have created a Thrift graph schema.

3.6 Conclusion

The high level requirements for storing data in the Lambda Architecture batch layer are straightforward. You observed that these requirements could be mapped to a required checklist for a storage solution, and you saw that HDFS can be used for this purpose.

You learned that maintaining a dataset within HDFS involves the common tasks of appending new data to the master dataset and consolidating small files. You witnessed that accomplishing these tasks using the HDFS API directly requires in-depth knowledge of Hadoop internals and is prone to human error.

You then were introduced to the Pail abstraction. Pail isolates you from the file formats and directory structure of HDFS, making it easy to do robust, enforced

vertical partitioning and perform common operations on your dataset. Without the Pail abstraction, performing appends and consolidating files manually is tedious and difficult.

The Pail abstraction plays an important role in making robust batch workflows. However, it ultimately takes very few lines of code in the code. Vertical partitioning happens automatically, and tasks like appends and consolidation are simple one-liners. This means you can focus on how you want to process your records rather than the details of how to store those records.

In the next chapter, you'll learn how to leverage the storage of the records to do efficient batch processing.

4 Batch layer: scalability

This chapter covers:

- Computing functions on the batch layer
- Splitting a query into precomputed and on the fly components
- Recomputation versus incremental algorithms
- The meaning of scalability
- The MapReduce paradigm
- Using Hadoop MapReduce

The goal of a data system is to answer arbitrary questions about your data. Any question you could ask of your dataset can be implemented as a function that takes all of your data as input. Ideally, you could run these functions on the fly whenever you query your dataset. Unfortunately, a function that uses your entire dataset as input will take a very long time to run. You need a different strategy if you want your queries answered quickly.

In the Lambda Architecture, the batch layer precomputes the master dataset into batch views so that queries can be resolved with low latency. This requires striking a balance between what will be precomputed and what will be computed at execution time to complete the query. By doing a little bit of computation on the fly to complete queries, you save yourself from needing to precompute ridiculously large batch views. The key is to precompute just enough information so that the query can be completed quickly.

In the last two chapters, you learned how to form a data model for your dataset and how to store your data in the batch layer in a scalable way. With this chapter, you'll take the next step of learning how to compute arbitrary functions on that data. We will start by introducing some motivating examples that we will use to

illustrate the concepts of computation on the batch layer. Then you'll learn in detail how to compute indexes of the master dataset that the application layer will use to complete queries. You'll examine the tradeoffs between recomputation algorithms, the style of algorithm emphasized in the batch layer, and incremental algorithms, the kind of algorithms typically used with relational databases. You'll see what it means for the batch layer to be scalable, and then you'll learn about Hadoop MapReduce, a tool that can be used to practically implement computation on the batch layer.

4.1 Motivating examples

Let's consider some example queries to motivate the theoretical discussions in this chapter. These queries will be used to illustrate the concepts of batch computation. Each example shows how you would compute the query as a function that takes in the entire master dataset as input. Later you will modify these implementations to use precomputation rather than being executed completely on the fly.

4.1.1 Number of pageviews over time

The first example query operates over a dataset of pageviews, where each pageview record contains a URL and timestamp. The goal of the query is to determine the total number of pageviews of a URL for a range given in hours. This query can be written in pseudocode like so:

```
function pageviewsOverTime(masterDataset, url, startHour, endHour) {
    pageviews = 0
    for(record in masterDataset) {
        if(record.url == url &&
           record.time >= startHour &&
           record.time <= endHour) {
           pageviews += 1
           }
    }
    return pageviews
}
```

To compute this query using a function of the entire dataset, you simply iterate through every record and keep a counter of all the pageviews for that URL that fall within the specified range. After exhausting all the records, you then return the final value of the counter.

4.1.2 Gender inference

The next example query operates over a dataset of name records and predicts the likely gender for a person. The algorithm first performs semantic normalization on the names for the person, doing conversions like "Bob" to "Robert" and "Bill" to "William". The algorithm then makes use of a model that provides the probability of a gender for each name. The resulting inference algorithm looks like this:

```
function genderInference(masterDataset, personId) {
    names = new Set()
    for(record in masterDataset) {                    (1)
        if(record.personId == personId) {
            names.add(normalizeName(record.name))
        }
    }
    maleProbSum = 0.0
    for(name in names) {                              (2)
        maleProbSum += maleProbabilityOfName(name)
    }
    maleProb = maleProbSum / names.size()
    if(maleProb > 0.5) {                              (3)
        return "male"
    } else {
        return "female"
    }
}
```

(1) normalize all names associated with the person
(2) average each name's probability of being male
(3) return the gender with the highest likelihood

An interesting aspect of this query is that the results can change as the name normalization algorithm and name-to-gender model improve over time, not just when new data is received.

4.1.3 Influence score

The final example operates over a Twitter-inspired dataset containing "reaction" records. Each reaction record contains a sourceId and responderId field, indicating that responderId retweeted or replied to sourceId's post. The query determines an influencer score for each person in the social network. The score is computed in two steps. First, the top influencer for each person is selected based on the amount of reactions the influencer caused in that person. Then, someone's influence score is set to be the number of people for which he or she was the top influencer. The algorithm to determine a user's influence score is as follows:

```
function influence_score(masterDataset, personId) {
    influence = new Map()
    for(record in masterDataset) {                          (1)
        curr = influence.get(record.sourceId) || new Map(default=0)
        curr[record.responderId] += 1
        influence.set(record.sourceId, curr)
    }

    score = 0
    for(entry in influence) {                               (2)
        if(topKey(entry.value) == personId) {
            score += 1
        }
    }
    return score
}
```

(1) compute amount of influence between all pairs of people
(2) count how many people for which personId is the top influencer

In this code, the "topKey" function is mocked since it's straightforward to implement. Otherwise, the algorithm simply counts the number of reactions between each pair of people and then counts the number of people for which the queried user is the top influencer.

4.2 Computing on the batch layer

Let's take a step back and review how the Lambda Architecture works at a high level. When processing queries, each layer in the Lambda Architecture has a key, complementary role, as shown in Figure 4.1.

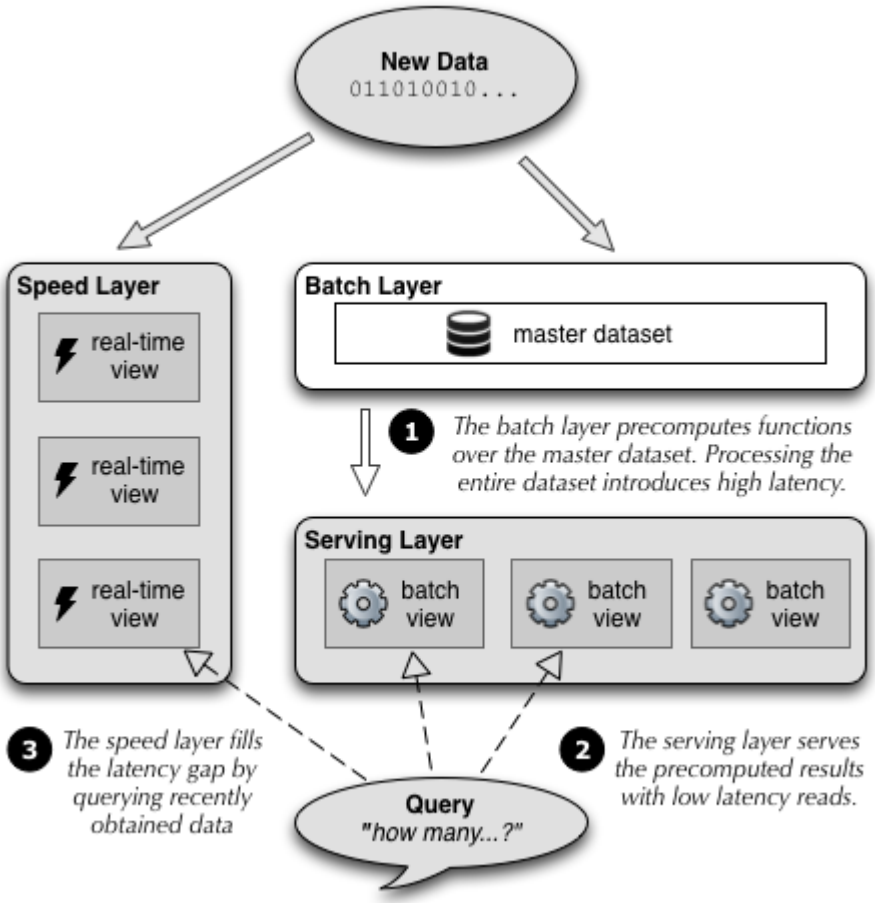


Figure 4.1 The roles of the Lambda Architecture layers in servicing queries on the dataset.

The batch layer runs functions over the master dataset to precompute intermediate data for your queries, which are stored in batch views in the serving layer. The speed layer compensates for the high latency of the batch layer by providing low latency updates using data that has yet to be precomputed into a batch view. Queries are then satisfied by processing data from the serving layer views and the speed layer views and merging the results.

A linchpin of the architecture is that for *any* query, it is possible to precompute the data in the batch layer to expedite its processing by the serving layer. These precomputations over the master dataset take time, but you should view the high latency of the batch layer as an opportunity to do deep analyses of the data and connect diverse pieces of data together. Remember, low latency query serving is achieved through other parts of the Lambda Architecture.

A naive strategy for computing on the batch layer would be to precompute all possible queries and cache the results in the serving layer. Such an approach is illustrated in Figure 4.2.

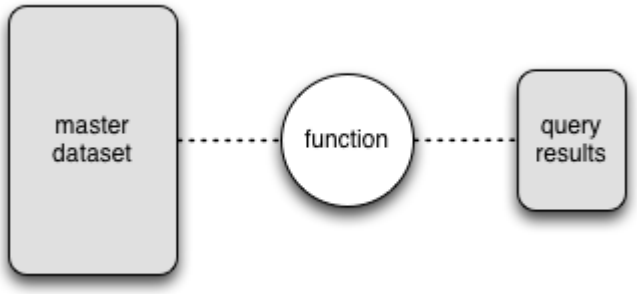


Figure 4.2 Precomputing a query by running a function on the master dataset directly

Unfortunately you can't always precompute *everything*. Consider the pageviews over time query as an example. If you wanted to precompute every potential query, you would need to determine the answer for every possible range of hours for every URL. However, the number of ranges of hours within a given timeframe can be huge. In a one year period, there are approximately 380 million distinct hour ranges. Therefore to precompute the query, you would need to precompute and index 380 million values *for every URL*. This is obviously infeasible and an unworkable solution.

Instead, you can precompute an intermediate result and then use these results to complete queries on the fly, as shown in Figure 4.3.

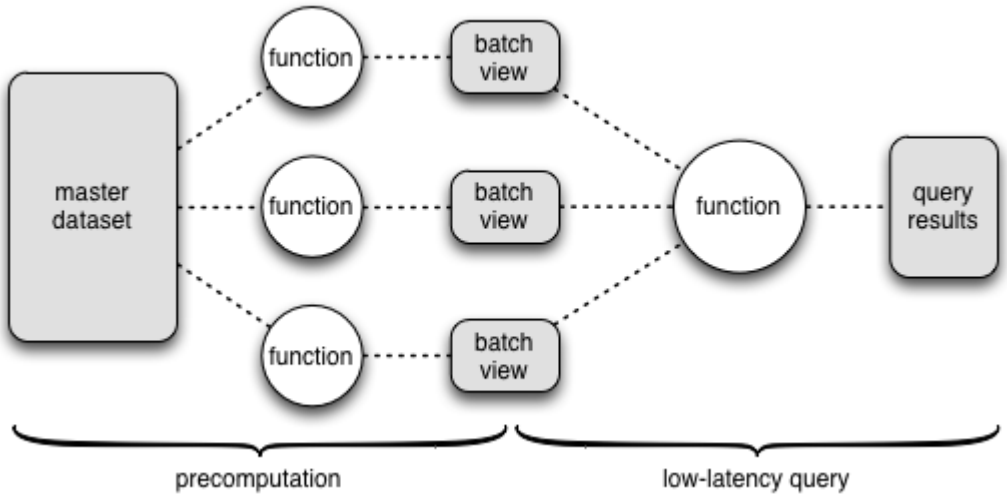


Figure 4.3 Splitting a query into precomputation and on the fly components

For the pageviews over time query, you can precompute the number of pageviews for every hour for each URL. This is illustrated in Figure 4.4.

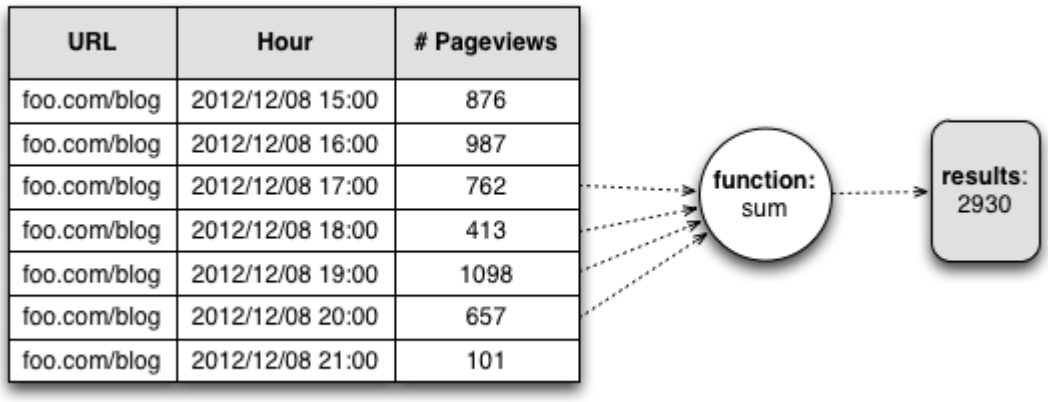


Figure 4.4 Computing the number of pageviews by querying an indexed batch view

To complete a query, you retrieve from the index the number of pageviews for every hour in the range and sum the results. For a single year, you only need to precompute and 8760 values per URL (365 days, 24 hours per day). This is certainly a much more manageable number.

4.3 Recomputation algorithms vs. incremental algorithms

Since your master dataset is continually growing, you must have a strategy for updating your batch views when new data becomes available. You could choose a *recomputation* algorithm, throwing away the old batch views and recomputing functions over the entire master dataset. Alternatively, an *incremental* algorithm would update the views directly when new data arrives.

As a basic example, consider a batch view containing the total number of records in your master dataset. A recomputation algorithm would update the count by first appending the new data into the master dataset and then counting all the records from scratch. This strategy is shown in Figure 4.5.

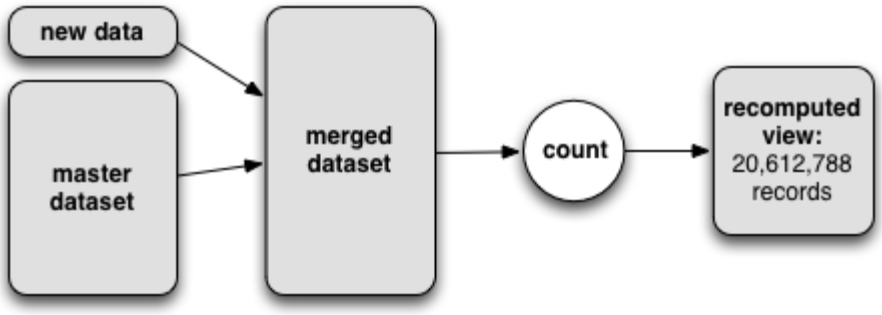


Figure 4.5 A recomputing algorithm to update the number of records in the master dataset. New data is appended to the master dataset, then all records are counted.

An incremental algorithm, on the other hand, would count the number of new data records and add it to the existing count, as demonstrated in Figure 4.6.

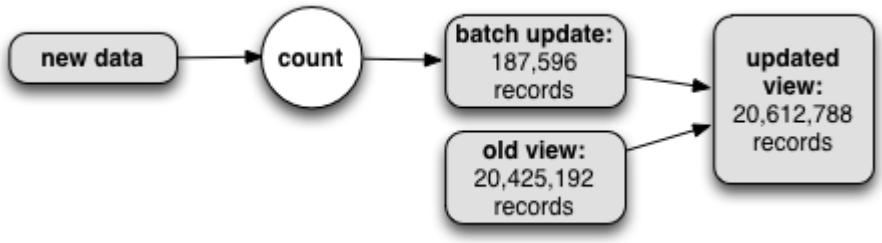


Figure 4.6 An incremental algorithm to update the number of records in the master dataset. Only the new data set is counted, with the total used to update the batch view directly.

You might be wondering why would you ever use a recomputation algorithm when you can use a vastly more efficient incremental algorithm instead. However, efficiency is not the only factor to be considered. The key tradeoffs to weigh between the two approaches are performance, human fault-tolerance, and the generality of the algorithm. We will discuss both types of algorithms in regard to each of these contexts. In doing so, you will discover that while incremental approaches can provide additional efficiency, you *must* have recomputation versions of your algorithms.

4.3.1 Performance

There are two aspects to the performance of a batch layer algorithm: the amount of resources required to update a batch view with new data, and the size of the batch views produced.

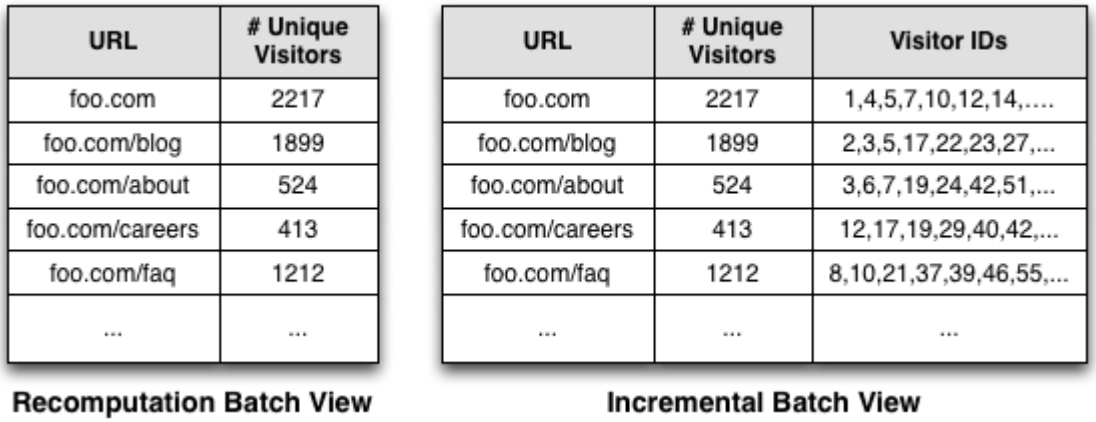
An incremental algorithm almost always uses significantly less resources to update a view since it uses new data and the current state of the batch view to perform an update. For a task such as computing pageviews over time, the view will be significantly smaller than the master dataset because of the aggregation. A recomputation algorithm looks at the entire master dataset, so the amount of resources needed for an update can be multiple orders of magnitude higher than an incremental algorithm.

However, the size of the batch view for an incremental algorithm can be significantly larger than the corresponding batch view for a recomputation algorithm. This is because the view needs to be formulated in a way such that it can be incrementally updated. We will demostrate through two separate examples.

First, suppose you needed to compute the average number of pageviews for

each URL within a particular domain. The batch view generated by a recomputation algorithm would contain a map from each URL to its corresponding average. However, this is not suitable for an incremental algorithm, since updating the average incrementally requires that you also know the number of records used for computing the previous average. An incremental view would therefore store both the average and the total count for each URL, increasing the size of the incremental view over the recomputation-based view by a constant factor.

In other scenarios, the increase in the batch view size for an incremental algorithm is much more severe. Next consider a query that computes the number of unique visitors for each URL. Figure 4.7 demonstrates the differences between batch views using recomputation and incremental algorithms.



URL	# Unique Visitors
foo.com	2217
foo.com/blog	1899
foo.com/about	524
foo.com/careers	413
foo.com/faq	1212
...	...

Recomputation Batch View

URL	# Unique Visitors	Visitor IDs
foo.com	2217	1,4,5,7,10,12,14,....
foo.com/blog	1899	2,3,5,17,22,23,27,...
foo.com/about	524	3,6,7,19,24,42,51,...
foo.com/careers	413	12,17,19,29,40,42,...
foo.com/faq	1212	8,10,21,37,39,46,55,...
...	...	...

Incremental Batch View

Figure 4.7 A comparison between a recomputation view and an incremental view for determining the number of unique visitors per URL.

A recomputation view only requires a map from the URL to the unique count. In contrast, an incremental algorithm only examines the new pageviews, so its view must contain the full set of visitors for each URL so it can determine which records in the new data correspond to return visits. As such, the incremental view could potentially be as large as the master dataset! The batch view generated by an incremental algorithm isn't always this large, but it can be much, much larger than the corresponding recomputation-based view.

4.3.2 Human fault-tolerance

The lifetime of a data system is extremely long: bugs can and will be deployed to production during that time period. You therefore must consider how your batch update algorithm will tolerate such mistakes. In this regard, recomputation algorithms are inherently human fault-tolerant, whereas with an incremental algorithm human mistakes can cause serious problems.

Consider as an example a batch layer algorithm that computes a global count of the number of records in the master dataset. Now suppose you make a mistake and deploy an algorithm that increments the global count for each record by two instead of by one. If your algorithm is recomputation-based, all that is required is to fix the algorithm and redeploy the code - your batch view will be correct the next time the batch layer runs. This is because the recomputation-based algorithm recomputes the batch view from scratch.

However, if your algorithm is incremental, then correcting your view isn't so simple. The only option is to identify the records that were overcounted, determine how many times each one was overcounted, and then correct the count for each affected record. Accomplishing this with a high decree of confidence is not always possible! You may have detailed logging that helps you with these tasks, but they may not always have the required information since you cannot anticipate every type of mistake that will be made in the future. Many times you have to do an ad-hoc, "best guess" modification of your view - and you have to make certain you don't mess that up as well.

"Hopefully having the right logs" to fix mistakes is not sound engineering practice. It bears worth repeating: human mistakes are inevitable. As you have seen, recomputation-based algorithms have much stronger human fault-tolerance than incremental algorithms.

4.3.3 Generality of algorithm

While incremental algorithms can be faster to run, they must often be tailored to address the problem at hand. For example, you have previously seen that an incremental algorithm for computing the number of unique visitors can generate prohibitively large batch views. This cost can be offset by probabilistic counting algorithms such as HyperLogLog that store intermediate statistics to estimate the overall unique count. [1] This greatly reduces the storage cost of the batch view, but at the price of making the algorithm approximate instead of exact.

Footnote 1 We will discuss HyperLogLog further in subsequent chapters.

The gender inference query introduced in the beginning of this chapter illustrates another issue: incremental algorithms shift complexity to the "on the fly" computation. As you improve your semantic normalization algorithm, you will want to see those improvements reflected in the results of your queries. Yet, if you do the normalization as part of the precomputation, your batch view will be out of date whenever you improve the normalization. The normalization must occur during the on the fly porton of the query when using an incremental algorithm. Your view will have to contain every name seen for each person, and your on the fly code will have to renormalize each name every time a query is performed. This increases the latency of the on the fly component and could very well be too long for your application requirements.

Since a recomputation algorithm continually rebuilds the entire batch view, the structure of the batch view and the complexity of the on the fly component are both much simpler, leading to a more general algorithm.

4.3.4 Choosing a style of algorithm

Figure 4.8 summarizes this section in terms of recomputation and incremental algorithms.

	recomputation algorithms	incremental algorithms
performance	require computational effort to process the entire master dataset	require less computational resources but may generate much larger batch views
human-error tolerance	extremely tolerant of human errors, as the batch views are continually rebuilt	do not facilitate repairing errors in the batch views; repairs are ad hoc and may require estimates
generality	complexity of the algorithm is addressed during precomputation, resulting in simple batch views and low latency on the fly processing	require special tailoring; may shift complexity to on the fly query processing
conclusion	essential to supporting a robust data processing system	can increase the efficiency of your system, but only as a supplement to recompilation algorithms

Figure 4.8 Comparing recomputation and incremental algorithms.

The key takeaway is that you must *always* have recomputation versions of your algorithms. This is the only way to ensure human fault-tolerance for your system,

and human fault-tolerance is a non-negotiable requirement for a robust system. Additionally, you have the option to add incremental versions of your algorithms to make them more resource efficient. For the remainder of this chapter, we will focus solely on recomputation algorithms, though in Chapter 7 we will come back to the topic of incrementalizing the batch layer.

4.4 Scalability in the batch layer

The word *scalability* gets thrown around a lot, so let's carefully define what it means in a data systems context. Scalability is the ability of a system to maintain performance under increased load by adding more resources. Load in a Big Data context is a combination of the total amount of data you have, how much new data you receive every day, how many requests per second your application serves, and so forth.

More important than a system being scalable is a system being *linearly scalable* . A linearly scalable system can maintain performance under increased load by adding resources in proportion to the increased load. A nonlinearly scalable system, while "scalable", isn't particular useful. Suppose the number of machines you need in relation to the load on your system has a quadratic relationship, like in Figure 4.9. Then the costs of running your system would rise dramatically over time. Increasing your load tenfold would increase your costs by a hundred. Such a system is not feasible from a cost perspective.

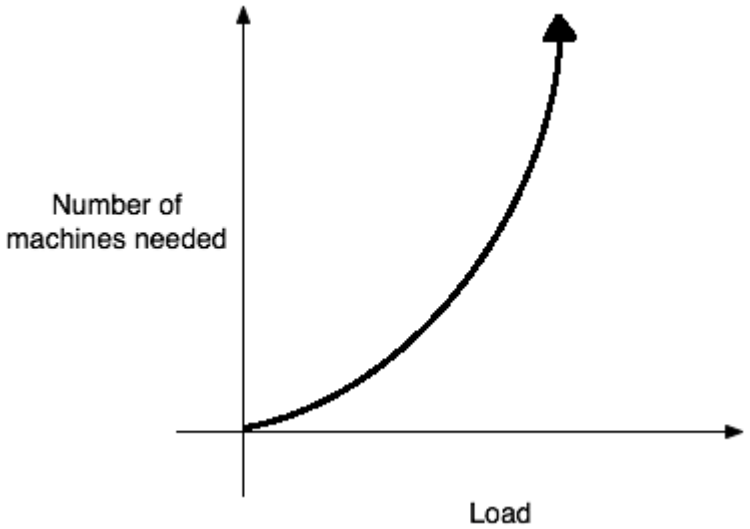


Figure 4.9 Non-linear scalability

When a system is linearly scalable, then costs rise in proportion to the load. This is a critically important property of a data system.

SIDEBAR **What scalability doesn't mean...**

Counterintuitively, a scalable system does not necessarily have the ability to *increase* performance by adding more machines. For an example of this, suppose you have a website that serves a static HTML page. Let's say that every web server you have can serve 1000 requests/sec within a latency requirement of 100 milliseconds. You won't be able to lower the latency of serving the web page by adding more machines – an individual request is not parallelizable and must be satisfied by a single machine. However, you can scale your website to increased requests per second by adding more web servers to spread the load of serving the HTML.

More practically, with algorithms that are parallelizable, you might be able to increase performance by adding more machines, but the improvements will diminish with the more machines you add. This is because of the increased overhead and communication costs associated with having more machines.

We delved into this discussion about scalability to set the framework to introduce MapReduce, a distributed computing paradigm that can be used to implement a batch layer. As we cover the details of its workings, keep in mind it is linearly scalable: should the size of your master dataset double, then twice the number of servers will be able to build the batch views with the same latency.

4.5 MapReduce: a paradigm for Big Data computing

MapReduce is a distributed computing paradigm originally pioneered by Google that provides primitives for scalable and fault-tolerant batch computation. With MapReduce, you write your computations in terms of *map* and *reduce* functions that manipulate key-value pairs. These primitives are expressive enough to implement nearly any function, and the MapReduce framework executes those functions over the master dataset in a distributed and robust manner. Such properties make MapReduce an excellent paradigm for the precomputation needed in the batch layer, but it is also a low level abstraction where expressing computations can be a large amount of work.

#In this section you will examine the concepts that drive MapReduce, #leaving the implementation details to the following section. You will begin with an #example of a MapReduce program, and then you will examine its #scalability, fault-tolerance and generality. The canonical MapReduce example is *word count*. Word count takes a dataset of text and determines the number of times each word

appears throughout the text. The map function in MapReduce executes once per line of text and emits any number of key-value pairs. For word count, the map function emits a key-value pair for every word in the text, setting the key to the word and the value to the number one.

```
function word_count_map(sentence) {
  for(word in sentence.split(" ")) {
      emit(word, 1)
  }
}
```

MapReduce then arranges the output from the map functions so that all values from the same key are grouped together. The reduce function takes the full list of values sharing the same key and emits new key-value pairs as the final output. In word count, the input is a list of "one" values for each word, and the reducer simply sums together the values to compute the count for that word.

```
function word_count_reduce(word, values) {
  sum = 0
  for(val in values) {
      sum += val
  }
  emit(word, sum)
}
```

There's a lot happening under the hood to run a program like word count across a cluster of machines, but the MapReduce framework handles most of the details for you. The intent is for you to focus on *what* needs to be computed without worrying about the details for *how* it is computed.

4.5.1 Scalability

The reason why MapReduce is such a powerful paradigm is because programs written in terms of MapReduce are inherently scalable. A program that runs on ten gigabytes of data will also run on ten petabytes of data. MapReduce automatically parallelizes the computation across a cluster of machines regardless of input size. All the details of concurrency, transferring data between machines, and execution planning are abstracted for you by the framework.

Let's walk through how a program like word count executes on a MapReduce cluster. The input to your MapReduce program is stored within a distributed filesystem such as the Hadoop Distributed Filesystem (HDFS) you encountered in the last chapter. Before processing the data, the program first determines which machines in your cluster host the blocks containing the input - see Figure 4.10.

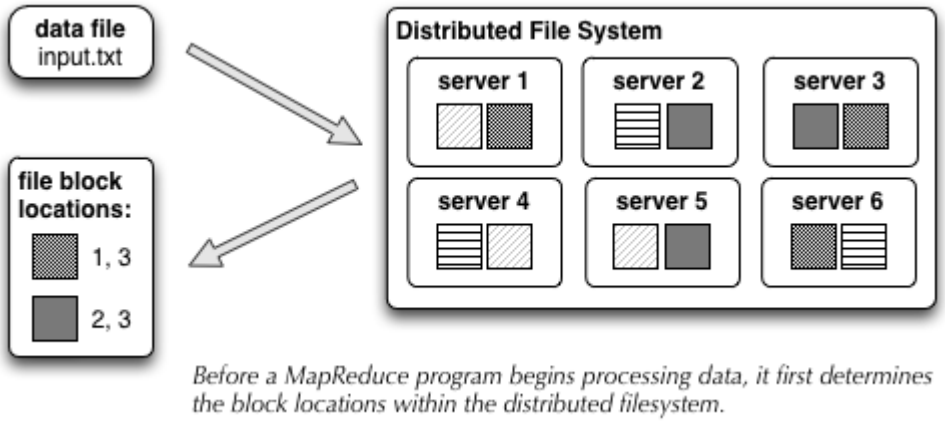


Figure 4.10 Locating the servers hosting the input files for a MapReduce program

After determining the locations of the input, MapReduce launches a number of map tasks proportional to the input data size. Each of these tasks is assigned a subset of the input and executes your map function on that data. Because the size of the code is typically much smaller than the size of the data, MapReduce attempts to assign tasks to servers that host the data to be processed. As shown in Figure 4.11, moving code to the data avoids having to transfer all that data across the network.

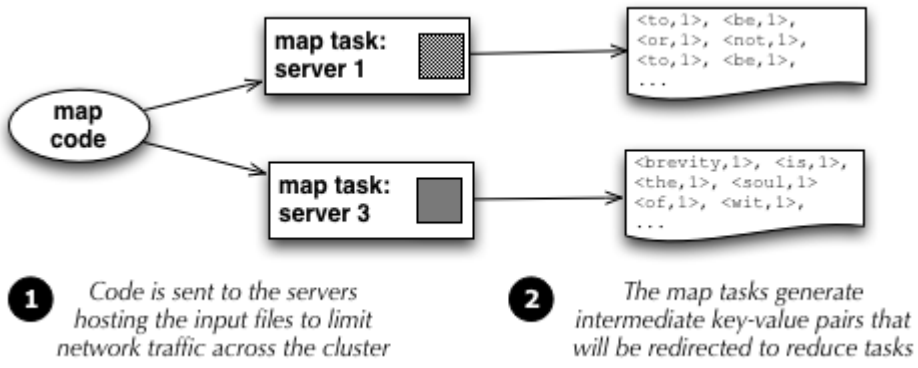
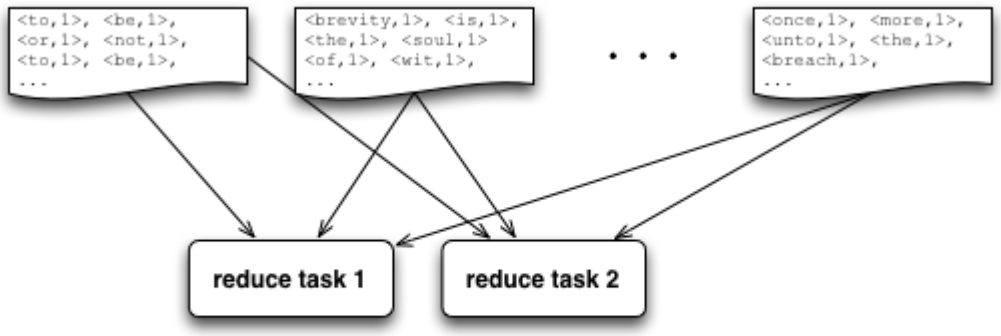


Figure 4.11 MapReduce promotes data locality, running tasks on the servers that host the input data

Like maps, there are also reduce tasks spread across the cluster. Each of these

tasks is responsible for computing the reduce function for a subset of keys generated by the map tasks. Because the reduce function requires all values associated with a given key, a reduce task cannot begin until all map tasks are complete.

Once the map tasks finish executing, each emitted key-value pair is sent to the reduce task responsible for processing that key. Therefore each map task distributes its output among all the reducer tasks. This transfer of the intermediate key-value pairs is called *shuffling* and is illustrated in Figure 4.12.



During the shuffle phase, all of the key-value pairs generated by the map tasks are distributed among the reduce tasks. In this process, all of the pairs with the same key are sent to the same reducer.

Figure 4.12 The shuffle phase distributes the output of the map tasks to the reduce tasks

Once a reduce task receives all of the key-value pairs from every map task, it sorts the key-value pairs by key. This has the effect of organizing all the values for any given key to be together. The reduce function is then called for each key and its group of values as demonstrated in Figure 4.13.

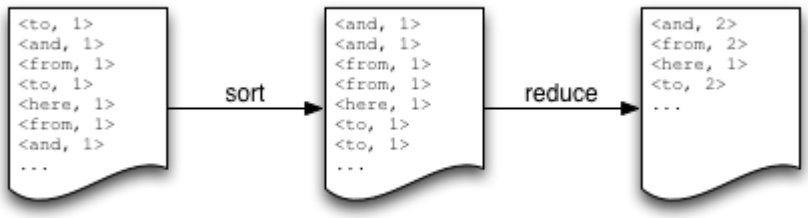


Figure 4.13 A reduce task sorts the incoming data by key, then performs the reduce function on the resulting groups of values

As you can see, there are many moving parts to a MapReduce program. The important takeaways from this overview are the following:

1. MapReduce programs execute in a fully distributed fashion with no central point of

contention
2. MapReduce is scalable: the map and reduce functions you provide are executed in parallel across the cluster
3. The challenges of concurrency and assigning tasks to machines is handled for you

4.5.2 Fault-tolerance

Distributed systems are notoriously testy. Network partitions, server crashes and disk failures are relatively rare for a single server, but the likelihood of something going wrong greatly increases when coordinating computation over a large cluster of machines. Thankfully, in addition to being easily parallelizable and inherently scalable, MapReduce computations are also fault-tolerant.

A program can fail for a variety of reasons: a hard disk can reach capacity, the process can exceed available memory, or the hardware can break down. MapReduce watches for these errors and automatically retries that portion of the computation on another node. An entire application (commonly called a *job*) will fail only if a task fails more than a configured amount of times - typically four. The idea is that a single failure may arise from a server issue, but a repeated failure is likely a problem with your code.

Since tasks can be retried, MapReduce requires that your map and reduce functions be *idempotent*. This means that given the same inputs, your functions must always produce the same outputs. It's a relatively light constraint but important for MapReduce to work correctly. An example of a non-idempotent function is one that generates random numbers. If you want to use random numbers in a MapReduce job, you need to make sure to explicitly seed the random number generator so that it always produces the same outputs.

4.5.3 Generality of MapReduce

It's not immediately obvious, but the computational model supported by MapReduce is expressive enough for to compute almost any functions on your data. To illustrate this, let's look at how you could use MapReduce to implement the batch view functions for the queries introduced at the beginning of this chapter.

IMPLEMENTING NUMBER OF PAGEVIEWS OVER TIME

```
function map(record) {
    key = [record.url, toHour(record.timestamp)]
    emit(key, 1)
}

function reduce(key, vals) {
    emit(new HourPageviews(key[0], key[1], sum(vals)))
}
```

This code is very similar to word count, but the key emitted from the mapper is a struct containing the URL and the hour of the pageview. The output of the reducer is the desired batch view containing a mapping from [URL, hour] to the number of pageviews for that hour.

IMPLEMENTING GENDER INFERENCE

```
function map(record) {
    emit(record.userid, normalizeName(record.name))   ❶
}

function reduce(userid, vals) {
    allNames = new Set()
    for(normalizedName in vals) {
        allNames.add(normalizedName)   ❷
    }
    maleProbSum = 0.0
    for(name in allNames) {
        maleProbSum += maleProbabilityOfName(name)   ❸
    }
    maleProb = maleProbSum / allNames.size()
    if(maleProb > 0.5) {   ❹
        gender = "male"
    } else {
        gender = "female"
    }
    emit(new InferredGender(userid, gender))
}
```

❶ semantic normalization occurs during the mapping stage
❷ a set is used to remove any potential duplicates
❸ average the probabilites of being male
❹ return the most likely gender

Gender inference is similarly straightforward. The map function performs the name semantic normalization, and the reduce function computes the predicted gender for each user.

IMPLEMENTING INFLUENCE SCORE

The influence score precomputation is more complex than the previous two examples and requires two MapReduce jobs to be chained together to implement the logic. The idea is that the output of the first MapReduce job is fed as the input to the second MapReduce job. The code is as follows:

```
function map1(record) {
    emit(record.responderId, record.sourceId)
}

function reduce1(userid, sourceIds) {
    influence = new Map(default=0)                    (1)
    for(sourceId in sourceIds) {
        influence[sourceId] += 1
    }
    emit([userid, topKey(influence)])
}

function map2(record) {
    emit(record[1], 1)
}

function reduce2(influencer, vals) {
    emit(new InfluenceScore(influencer, sum(vals)))   (2)
}
```

(1) the first job determines the top influencer for each user
(2) the top influencer data is then used to determine the number of people each user influences

It's typical for computations to require multiple MapReduce jobs – that just means multiple levels of grouping were required. The first job requires grouping all reactions for each user to determine that user's top influencer. The second job then groups the the records by top influencer to determine the influence scores.

When you take a step back and look at what MapReduce is doing at a fundamental level, MapReduce:

1. Arbitrarily partitions your data through the key you emit in the map phase. Arbitrary partitioning lets you connect your data together for later processing while still processing everything in parallel.

2. Arbitrarily transforms your data through the code you provide in the map and reduce phases.

It's hard to envision anything more general that could still be a scalable, distributed system. Now let's take a look at how you would use MapReduce in practice.

4.6 Word Count on Hadoop

Let's move on from using pseudocode and implement word count with an actual MapReduce framework. Hadoop MapReduce is an open-source, Java-based implementation of MapReduce that integrates with the Hadoop Distributed Filesystem that you learned about in Chapter 3. The input to the program will be a set of text files containing a sentence on each line, and the output will be written to a separate text file stored in HDFS.

You will begin with defining classes that fulfill the map and reduce functionality. Afterwards, you will learn how to tie these classes together into a complete MapReduce program using the Hadoop API.

4.6.1 Mapper

Both map and reduce functions are implemented using classes that extend from a common MapReduceBase class. A class for a map function must also implement the Mapper interface. The following listing is the a mapper class for word count:

```
public static class WordCountMapper extends MapReduceBase implements
    Mapper<LongWritable, Text, Text, IntWritable>  (1)
{
  private static final IntWritable one = new IntWritable(1);
  private Text word = new Text();  (2)

  public void map(LongWritable key, Text value,
      OutputCollector<Text, IntWritable> output,
      Reporter reporter) throws IOException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
      word.set(tokenizer.nextToken());
      output.collect(word, one);  (3)
    }
  }
}
```

1. the generic classes define the input and output key-value pairs
2. create the key and value output objects
3. emit the key-value pair

The Mapper interface is generic, templatized by the key and value classes of both input and output. For word count, the input value is a string containing a line from a text file, and the key is a long integer denoting the corresponding line number. As you would expect, the output key represents a word and the value is the constant value one. The actual implementation uses standard Java string manipulation, with the key-value pairs emitted to the OutputCollector.

SIDEBAR **Hadoop Writables**

One notable aspect of MapReduce code is the use of classes such as LongWritable, Text and IntWritable instead of standard longs, strings and integers. MapReduce gains its flexibility by manipulating key-value pairs containing any type of object, but the default serialization of Java is very inefficient - as issue that is magnified when streaming large amounts of data. Hadoop instead uses a Writable interface for serialization, and the LongWritable, Text, and IntWritable classes are wrappers to implement this interface for their corresponding data types.

4.6.2 Reducer

Similarly, a class that implements the reduce function extends MapReduceBase and implements the generic Reducer interface. The reduce function is called for each key assigned to the reduce task.

```
public static class WordCountReducer extends MapReduceBase implements
    Reducer<Text, IntWritable, Text, IntWritable>
{
  public void reduce(Text key, Iterator<IntWritable> values,
      OutputCollector<Text, IntWritable> output,
      Reporter reporter) throws IOException {
    int sum = 0;
    while (values.hasNext()) {  (1)
      sum += values.next().get();
    }
    output.collect(key, new IntWritable(sum));
  }
}
```

(1)

Hadoop provides an iterator over all values for the current key

As a rule of thumb, the reduce phase is approximately four times more expensive than the map phase. This is because the reduce step requires receiving data from all the mappers, sorting the data using the keys, then actually computing the function for each key. As such, part of making an efficient MapReduce workflow is minimizing the number of reduce steps.

4.6.3 Application

As a final step, you must define a Hadoop job to be submitted to your cluster. The job requires not only the map and reduce classes, but specifies the input and output locations, the expected file formats, and the final output types. Listed below is a basic job configuration for Word Count.

```
public class WordCount {
  public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");

    conf.setMapperClass(WordCountMapper.class);
    conf.setReducerClass(WordCountReducer.class);
    conf.setNumReduceTasks(3);     (1)

    conf.setInputFormat(TextInputFormat.class);     (2)
    conf.setOutputFormat(TextOutputFormat.class);

    conf.setOutputKeyClass(Text.class);     (3)
    conf.setOutputValueClass(IntWritable.class);

    FileInputFormat.setInputPaths(conf, new Path(args[0]));     (4)
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    JobClient.runJob(conf);     (5)
  }
}
```

(1) Hadoop can determine the number of maps from the input size, but the number of reduce tasks must be set
(2) the input and output with both use text files
(3) the reducer will emit pairs of words and integers
(4) set the HDFS locations for both input and output
(5) at last, execute the job

The Hadoop implementation of word count requires you to do considerable work that's tangential to the actual problem of counting words. There is an overabundance of type declarations. You need to know details about the filesystem and the representation of your data. A different class is used to set the input and output locations. Lastly, it's awkward that you must set the final output key and value classes separately from the output format.

These issues arise from the fact that the Hadoop API is a low level abstraction of MapReduce. Hadoop MapReduce provides access to more than a hundred configurable options. This provides you with great flexibility to tune your jobs for optimal performance, but it makes it an unwieldy framework, increasing code complexity and development effort. In the next section we will explore a more complicated example that further demonstrates both the power of MapReduce and the cumbersome nature of the Hadoop API.

4.7 The low level nature of MapReduce

The Hadoop MapReduce API is expressive, but difficulties arise even in simple examples. The code becomes further convoluted when you perform more complex operations, such as computing multiple views from a single dataset or joining two datasets.

To show how much effort is needed to use Hadoop MapReduce directly, we will write a MapReduce program that determines the relationship between the length of a word and the number of times that word appears in a set of sentences. This is only slightly more complicated than counting words, though stop words like "a" and "the" should be ignored to avoid skewing the results.

A good way to do this query is to modify word count and add a second MapReduce job. The map phase of the first job emits <word, 1> pairs like before but also filters out stop words.

```
public static class SplitAndFilterMapper extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

  public static final Set<String> STOP_WORDS = new HashSet<String>()
  {{
     add("the");   (1)
     add("a");
  }};

  private static final IntWritable one = new IntWritable(1);   (2)
  private Text word = new Text();
```

```
  public void map(LongWritable key, Text value,
      OutputCollector<Text, IntWritable> output,
      Reporter reporter) throws IOException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
      String token = tokenizer.nextToken();
      if(!STOP_WORDS.contains(token)) {   (3)
        word.set(token);
        output.collect(word, one);
      }
    }
  }
}
```

(1) create the stop words set; only two words shown for brevity
(2) the value for emitted key-value pairs is always 1, so only need a single object for all values
(3) remove all stop words

The first reduce phase is also similar to word count, but the emitted key is the length of the word and not the word itself.

```
public static class LengthToCountReducer extends MapReduceBase
    implements Reducer<Text, IntWritable, IntWritable, IntWritable> {

  public void reduce(Text key, Iterator<IntWritable> values,
      OutputCollector<IntWritable, IntWritable> output,
      Reporter reporter) throws IOException {
    int sum = 0;
    while (values.hasNext()) {
      sum += values.next().get();
    }
    output.collect(new IntWritable(key.toString().length()),
                   new IntWritable(sum));   (1)
  }
}
```

(1) use the word length as the key

The second MapReduce job uses the results from the first job as input and emits <word length, average word count> key-value pairs as its output. The second mapper does not need to transform the data: it simply passes the key-value pairs to

the reducer.

```
public static class PassThroughMapper extends MapReduceBase
    implements Mapper<IntWritable, IntWritable,
                      IntWritable, IntWritable> {

  public void map(IntWritable key, IntWritable count,
      OutputCollector<IntWritable, IntWritable> output,
      Reporter reporter) throws IOException {
    output.collect(key, count);   (1)
  }
}
```

(1) the mapper emits all key-value pairs as read from the input

The reducer then computes the average word count for each word length.

```
public static class AverageReducer extends MapReduceBase
    implements Reducer<IntWritable, IntWritable,
                       IntWritable, DoubleWritable> {
  public void reduce(IntWritable key,
      Iterator<IntWritable> values,
      OutputCollector<IntWritable, DoubleWritable> output,
      Reporter reporter) throws IOException {
    int sum = 0;
    int count = 0;
    while (values.hasNext()) {
      sum += values.next().get();   (1)
      count +=1;
    }
    double avg = 1.0 * sum / count;
    output.collect(key, new DoubleWritable(avg));   (2)
  }
}
```

(1) keep the running count and running sum when iterating over all values for the key
(2) calculate and emit the average for the key

For the application class, each MapReduce job will be configured and executed in a separate static function. Both of these functions are very similar to the application code for word count.

```
private static void runWordBucketer(  // first job
    String input, String output) throws Exception {

  JobConf conf = new JobConf(WordFrequencyVsLength.class);
  conf.setJobName("freqVsAvg1");                                  (1)

  conf.setMapperClass(SplitAndFilterMapper.class);                (2)
  conf.setReducerClass(LengthToCountReducer.class);

  FileInputFormat.setInputPaths(conf, new Path(input));           (3)
  FileOutputFormat.setOutputPath(conf, new Path(output));

  conf.setInputFormat(TextInputFormat.class);                     (4)
  conf.setOutputFormat(TextOutputFormat.class);

  conf.setOutputKeyClass(Text.class);
  conf.setOutputValueClass(IntWritable.class);

  JobClient.runJob(conf);                                         (5)
}

private static void runBucketAverager(
    String input, String output) throws Exception { // second job

  JobConf conf = new JobConf(WordFrequencyVsLength.class);
  conf.setJobName("freqVsAvg2");                                  (6)

  conf.setMapperClass(PassThroughMapper.class);
  conf.setReducerClass(AverageReducer.class);

  FileInputFormat.setInputPaths(conf, new Path(input));
  FileOutputFormat.setOutputPath(conf, new Path(output));

  conf.setInputFormat(TextInputFormat.class);
  conf.setOutputFormat(TextOutputFormat.class);

  conf.setOutputKeyClass(IntWritable.class);
  conf.setOutputValueClass(DoubleWritable.class);

  JobClient.runJob(conf);
}
```

(1) define the first job
(2) specify the mapper and reducer classes
(3) set the input and output file paths
(4) provide classes describing how the input and output will be formatted
(5) execute the job
(6) analogously configure the second job

The last step is to define the main function to tie these two jobs together.

```
public class WordFrequencyVsLength {
  public static void main(String[] args) throws Exception {
    String tmpPath = "/tmp/" + UUID.randomUUID().toString();   (1)
    runWordBucketer(args[0], tmpPath);   (2)
    runBucketAverager(tmpPath, args[1]);
    FileSystem.get(new Configuration())   (3)
              .delete(new Path(tmpPath), true);
  }
}
```

(1) create a temporary directory so the output of the first job can be used as input for the second
(2) run the two jobs in succession
(3) remove the temporary path before completing

The most apparent difference between this example and word count is the length of the code. Although the problems are of similar complexity, the code has doubled in size. Also notice that a temporary path must be created to store the intermediate output between the two jobs. This should immediately set off alarm bells, as it is a clear indication that you're working at a low level of abstraction. You want an abstraction where the whole computation can be represented as a single conceptual unit, and details like temporary path management are automatically handled for you.

Looking at this code, many other problems become apparent. There is a distinct lack of composability in this code. You couldn't reuse much from the word count example. Moreover, the mapper of the first job does the dual tasks of splitting sentences into words and filtering out stop words. Ideally you would separate those tasks into separate conceptual units.

You have seen this pattern before: HDFS is a well-suited platform for storing the master dataset, but the low level APIs complicate the execution of simple tasks. Instead, Pail serves as a high level abstraction to provide an easy API for common tasks. Analogously, in the next chapter we will introduce JCascalog, a high level abstraction to MapReduce.

4.8 Conclusion

The MapReduce paradigm provides the primitives for precomputing query functions across all your data, and Apache Hadoop is a practical implementation of MapReduce.

However, it can be hard to think in MapReduce. Although MapReduce provides

the essential primitives of fault-tolerance, parallelization, and task scheduling, it's clear that working with the raw MapReduce API is tedious and limiting.

In the next chapter you'll explore a higher level abstraction to MapReduce called JCascalog. JCascalog alleviates the abstraction and composability problems with MapReduce that you saw in this chapter, making it much easier to develop complex MapReduce flows in the batch layer.

5 Batch layer: abstraction and composition

This chapter covers:

- Sources of complexity in data processing code
- The JCascalog API
- Applying abstraction and composition techniques to data processing

In the last chapter you saw how to compute arbitrary functions of your master dataset using the MapReduce programming paradigm. This is a key aspect of the Lambda Architecture since it determines the types of queries you can ask of your data. What makes MapReduce powerful is that in addition to supporting the computation of nearly any function, it also automatically scales your computations across a cluster of machines and executes those computations in a fault-tolerant way.

However, MapReduce is not a panacea - you also saw examples demonstrating its low level of abstraction. Coding directly against the MapReduce API can be tedious and, once written, it is often difficult to reuse your code. These drawbacks directly lead to large, complex, and hard-to-maintain codebases that greatly diminish the productivity of your development team.

Within the Lambda Architecture, the aim of this chapter is the same as the last: processing the master dataset to create views within the serving layer. Whereas we previously explored the framework for processing data stored in the batch layer, in this chapter you'll learn how to make your data processing code simple, reusable and elegant. A key point is that your data processing code is no different than any other code you write. As such, it requires good abstractions that are reusable and composable. Abstraction and composition are the cornerstones of good software engineering.

The concepts of abstraction and composition in MapReduce data processing will be illustrated using a Java library called JCascalog. After looking at an simple, end-to-end example of JCascalog, we'll examine some common sources of complexity in data processing code. Then we'll look at JCascalog in-depth and discuss how it avoids complexity and allows you to apply abstraction and composition techniques to data processing.

5.1 An illustrative example

Word count is the canonical MapReduce example, and you saw in the last chapter how to implement it using MapReduce directly. Let's take a look at how it's implemented using JCascalog, a much higher level abstraction.

For introductory purposes, we will explicitly store the input dataset - the Gettysburg address - in an in-memory list where each phrase is stored separately.

```
List SENTENCE = Arrays.asList(
  Arrays.asList("Four score and seven years ago our fathers"),
  Arrays.asList("brought forth on this continent a new nation"),
  Arrays.asList("conceived in Liberty and dedicated to"),
  Arrays.asList("the proposition that all men are created equal"),
  ...  (1)
```

(1) truncated for brevity

The following snippet is a complete JCascalog implementation of word count for this dataset.

```
Api.execute(new StdoutTap(),  (1)
            new Subquery("?word", "?count")  (2)
              .predicate(SENTENCE, "?sentence")  (3)
              .predicate(new Split(), "?sentence").out("?word")  (4)
              .predicate(new Count(), "?count"));  (5)
```

(1) query output to be written to the console
(2) specify the output types returned by the query
(3) read each sentence from our input
(4) tokenize each sentence into separate words
(5) determine the count for each word

The first thing to note is that this code is really concise! JCascalog's high level

nature may make it difficult to believe it is a MapReduce interface, but when this code is executed it runs as a MapReduce job. Upon running this code, it would print the output to your console, returning results similar to the following (partial listing for brevity):

```
RESULTS
----------
But     1
Four    1
God     1
It      3
Liberty 1
Now     1
The     2
We      2
```

Let's go through this definition of word count line by line to understand what it's doing. If every detail isn't completely clear, don't worry. We'll be going through JCascalog in much greater depth later in the chapter.

In JCascalog, inputs and outputs are defined via an abstraction called a *tap*. The tap abstraction allows results to be displayed on the console, stored in HDFS, or written to a database. The first line reads as "execute the following computation and direct the results to the console".

```
Api.execute(new StdoutTap(), ...
```

The second line begins the definition of the computation. Computations are represented via instances of the *Subquery* class. This subquery will emit a set of tuples containing two fields named ?word and ?count.

```
new Subquery("?word", "?count")
```

The next line sources the input data for the query. It reads from the SENTENCE dataset and emits tuples containing one field named ?sentence. As with outputs, the Tap abstraction allows inputs from different sources, such as in-memory values, HDFS files or the results from other queries.

```
.predicate(SENTENCE, "?sentence")
```

The fourth line splits each sentence into a set of words, giving the *Split* function the ?sentence field as input and storing the output in a new field called ?word.

```
.predicate(new Split(), "?sentence").out("?word")
```

The Split function is not part of the JCascalog API but demonstrates how new user defined functions can be integrated into queries. Its operation is defined via the following class. Its definition should be fairly intuitive; it takes in input sentences and emits a new tuple for each word in the sentence.

```
public static class Split extends CascalogFunction {
  public void operate(FlowProcess process, FunctionCall call) {
    String sentence = call.getArguments().getString(0);
    for (String word: sentence.split(" ")) {          (1)
      call.getOutputCollector().add(new Tuple(word));  (2)
    }
  }
}
```

(1) partition a sentence into words
(2) each word is emitted in its own tuple

Finally, the last line counts the number of times each word appears and stores the result in the ?count variable.

```
.predicate(new Count(), "?count"));
```

Now that you've had a taste of what a higher level abstraction for MapReduce can look like, let's take a step back and understand the reasons why having a higher level abstraction for MapReduce is so important.

5.2 Sources of complexity in data processing code

As with any code, keeping your data processing code simple is essential so that you can reason about your system and ensure correctness. Complexity in code arises in two forms: *essential complexity* that is inherent in the problem to be solved and *accidental complexity* that arises solely from the approach to the solution. By minimizing accidental complexity, your code will be easier to maintain and you will have greater confidence in its correctness.

In this section we will look at three sources of accidental complexity in data processing code: an inability to modularize, interactions with custom languages, and poorly composable abstractions. Each of these sources frequently arise when working with MapReduce, and by studying them we will identify the desired properties for a high level abstraction for data processing.

5.2.1 Inability to modularize due to performance cost

Modularizing your code is one of the keys to reducing complexity and keeping your code easy to understand. By breaking code down into components, it's easier to understand each piece in isolation and reuse functionality as needed. However, it's critical not to incur any excessive cost in performance during the process.

This performance concern is precisely one of the drawbacks in using the MapReduce API directly. As an example, suppose you want to create a variant of word count that only counts words greater than length 3 and only emits counts greater than 10. If you did this in a modular way with the MapReduce API, each transformation would be written as a separate MapReduce job as shown in Figure 5.1.

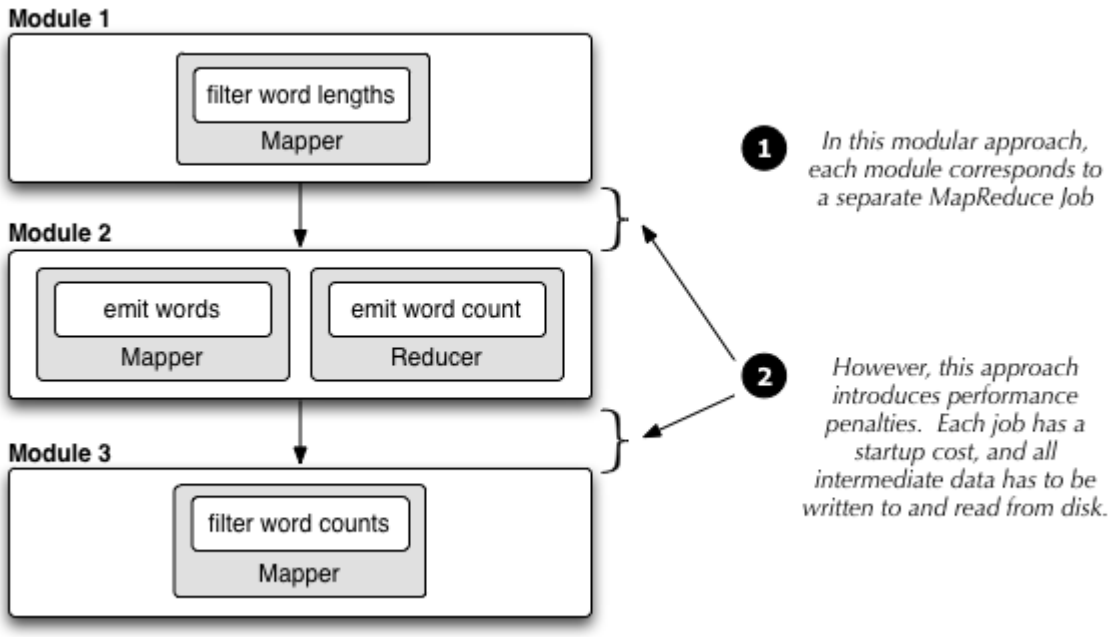


Figure 5.1 Using separate MapReduce jobs as modules imposes latency due to job startup costs and serialization of results between jobs.

Although this is a modular implementation, each MapReduce job is expensive: a job involves launching tasks, reading data to and from disk, and streaming data over the network. You need to execute your code in as few MapReduce jobs as possible to maximize performance.

A high level abstraction should disassociate the specification of the computation from how it executes. Rather than running each portion of your computation as its own job, a high level abstraction should compile the desired computation into a minimal number of MapReduce jobs, packing functions into the same Mapper or Reducer whenever possible. For example, the word count variant just described would execute as one MapReduce job, as shown in Figure 5.2.

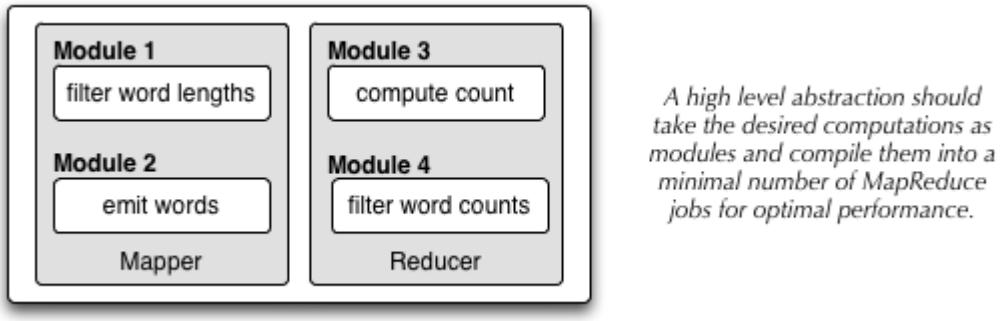


Figure 5.2 A high level abstraction should compiling modularized functionality into as few MapReduce jobs as possible

5.2.2 Custom languages

Another common source of complexity in data processing tools is the use of custom languages. Examples of this include SQL for relational databases, or Pig and Hive for Hadoop. Using a custom language for data processing, while tempting, introduces a number of serious complexity problems.

The use of custom languages introduces a language barrier that requires an interface to interact with other parts of your code. This interface is a common source of errors and an unavoidable source of complexity. As an example, SQL injection attacks take advantage of an improperly defined interface between user-facing code and the generated SQL statements for querying a relational database. Because of this interface, you have to be constantly on your guard to ensure you don't make any mistakes.

The language barrier causes all kinds of other complexity issues. Modularization can become painful – the custom language may support namespaces and functions, but ultimately these are not going to be as good as their general purpose language counterparts. Furthermore, if you want to incorporate your own business logic into queries, you must create your own user-defined functions (UDFs) and register them with the language.

Lastly, you have to coordinate switching between your general purpose language and your data processing language. For instance, you may write a query using a custom language and then want to use the Pail class from Chapter 3 to append the resulting data into an existing store. The Pail invocation is just standard Java code, so you will need to write shell scripts that perform tasks in the correct order. Because you're working in multiple languages stitched together via scripts, mechanisms like exceptions and exception handling break down – you have to check return codes to make sure you don't continue to the next step when the prior step failed.

These are all examples of accidental complexity that can be avoided completely when your data processing tool is a library for your general purpose language. You can then freely intermix regular code with data processing code, use your normal mechanisms for modularization, and have exceptions work properly. As you'll see, it's possible for a regular library to be concise and just as pleasant to work with as a custom language.

5.2.3 Poorly composable abstractions

The third common source of accidental complexity can occur when using multiple abstractions in conjunction. It's important that your abstractions can be composed together to create new and greater abstractions – otherwise you are unable to reuse code and you keep reinventing the wheel in slightly different ways.

A good example of this is the "Average" aggregator in Apache Pig (another abstraction for MapReduce). At the time of this writing, the implementation has over 300 lines of code and 15 separate method definitions. Its intricacy is due to code optimizations for improved performance and coordinates work in both map and reduce phases.

The problem with Pig's implementation is that it is reimplements the functionality of the "Count" and "Sum" aggregators without being able to reuse the code written for those aggregators. This is unfortunate because it's more code to maintain, and every time an improvement is made to Count and Sum, those changes need to be incorporated into Average as well. It is much preferred to define "Average" as the *composition* of a count aggregation, a sum aggregation, and the division function. In fact, this is exactly how you define Average in JCascalog:

```
PredicateMacroTemplate Average =
  PredicateMacroTemplate.build("?val")
    .out("?avg")
    .predicate(new Count(), "?count")
    .predicate(new Sum(), "?val").out("?sum")
    .predicate(new Div(), "?sum", "?count").out("?avg");
```

In addition to its simplicity, this definition of Average is as efficient as the Pig implementation since it reuses the previously optimized Count and Sum aggregators. We'll cover this functionality of JCascalog in depth later - the takeaway here is the importance of abstractions being composable. There are many other examples of composition which we'll be exploring throughout this chapter.

Now that you've seen some of the common sources of complexity in data processing tools, let's take a look at a library designed to avoid these pitfalls.

5.3 An Introduction to JCascalog

In the last section we saw some of the issues you encounter when writing against the MapReduce API directly. Rather than worrying about the complexity that arises from low-level interactions with MapReduce, you are much better off interacting with MapReduce at a higher level. Towards that end, in this section we will introduce JCascalog, a Java library that provides composable abstractions for expressing MapReduce computations.

Recall that the goal of this book is to illustrate the concepts of Big Data, using specific tools to ground those concepts. There are other tools that provide higher level interfaces to MapReduce - Hive, Pig and Cascading among the most popular - but many of them still have limitations in their ability to abstract and compose data processing code. We have chosen JCascalog because it was specifically written to enable new abstraction and composition techniques to reduce the complexity of batch processing.

JCascalog is a declarative abstraction where computations are expressed via logical constraints. Rather than providing explicit instructions on how to derive the desired output, you instead describe the output in terms of the input. From that description, JCascalog determines the most efficient way to perform the calculation via a series of MapReduce jobs.

If you're experienced with relational databases, JCascalog will seem to be both strange and familiar at the same time. You'll recognize familiar concepts like declarative programming, joins and aggregations, albeit in different packaging. However, it may seem different because rather than SQL, it's an API based on logic programming.

5.3.1 The JCascalog data model

To understand how JCascalog processes data, you must first understand its underlying data model. JCascalog works by manipulating and transforming *tuples* - named lists of values where each value can be any type of object. A set of consistent tuples share a *schema* which specifies how many fields are in each tuple and the name of each field. Figure 5.3 demonstrates an example set of tuples with a shared schema.

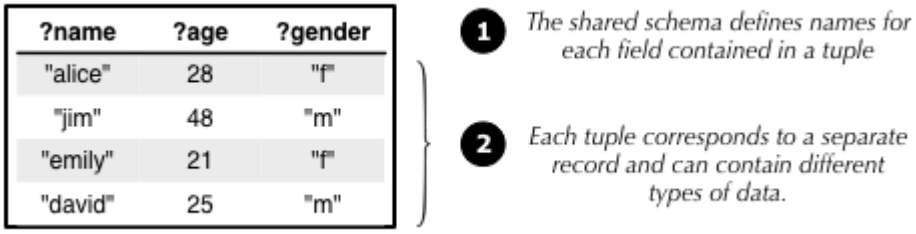


Figure 5.3 An example set of tuples with a schema describing their contents

As the figure illustrates, JCascalog's data model is similar to the "rows and columns" model of relational databases. When executing a query, JCascalog represents the initial data as tuples and transforms the input into a succession of other tuple sets at each stage of the computation.

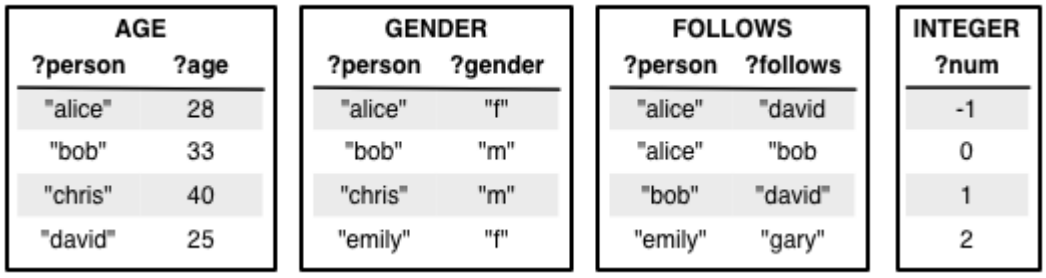
SIDEBAR **An abundance of punctuation?!**

After seeing this example and the word count example earlier, a natural question is: "what's the meaning of all those question marks?" We're glad you asked.

Fields with names starting with "?" are non-nullable. If JCascalog encounters a tuple with a null value for a non-nullable field, it is immediately filtered from the working dataset. Conversely, field names beginning with "!" may contain null values.

Additionally, field names starting with '!!' are also nullable and are needed to perform outer joins between datasets. For joins involving this kind of field names, records that do not satisfy the join condition between datasets are still included in the result set, but with null values for these fields where data is not present.

The best way to introduce JCascalog is through a variety of examples. Along with the SENTENCE dataset we encouncered earlier, we will be using a few other in-memory datasets to demonstrate the different aspects of JCascalog. Examples from these datasets are shown in Figure 5.4, with the full set available in the source code bundle that accompanies this book.



AGE

?person	?age
"alice"	28
"bob"	33
"chris"	40
"david"	25

GENDER

?person	?gender
"alice"	"f"
"bob"	"m"
"chris"	"m"
"emily"	"f"

FOLLOWS

?person	?follows
"alice"	"david
"alice"	"bob
"bob"	"david"
"emily"	"gary"

INTEGER

?num
-1
0
1
2

Figure 5.4 Example datasets we will use to demonstrate the JCascalog API: a set of people's ages, a separate set for gender, a person-following relationship such as Twitter, and a set of integers.

JCascalog benefits from a simple syntax that is capable of expressing complex queries. We'll examine JCascalog's query structure next.

5.3.2 The structure of a JCascalog query

JCascalog queries have a uniform structure consisting of a destination tap and a subquery that defines the actual computation. Consider the following example which finds all people from the AGE dataset younger than 30:

```
Api.execute(new StdoutTap(),                    ❶
            new Subquery("?person")             ❷
              .predicate(AGE, "?person", "?age")  ❸
              .predicate(new LT(), "?age", 30));
```

❶ the destination tap
❷ the output fields
❸ predicates that define the desired output

Note that instead of expressing *how* to perform a computation, JCascalog uses predicates to describe the desired output. These predicates are capable of expressing all possible operations on tuple sets - transformations, filters, joins and so forth - and can be categorized into four main types:

- A *function* predicate specifies a relationship between a set of input fields and a set of output fields. Mathematical functions such as addition and multiplication fall into this category, but a function can also emit multiple tuples from a single input.
- A *filter* predicate specifies a constraint on a set of input fields and removes all tuples that do not meet the constraint. The "less than" and "greater than" operations are examples of this type.
- An *aggregator* predicate is a function on a group of tuples. An example aggregator is to compute the average, which emits a single output for the entire group.
- A *generator* predicate is simply a finite set of tuples. A generator can either be a concrete

source of data such as an in-memory data structure or file on HDFS, or it can be the result from another subquery.

Additional example predicates are shown below in Figure 5.5:

Type	Example	Description
Generator	`.predicate(SENTENCE, "?sentence")`	*A generator that creates tuples from the SENTENCE dataset, with each tuple consisting of a single field called ?sentence*
Function	`.predicate(new Multiply(), 2, "?x").out("?z")`	*This function doubles the value of ?x and stores the result as ?z*
Filter	`.predicate(new LT(), "?y", 50)`	*This filter removes all tuples unless the value of ?y is less than 50*

Figure 5.5 Example generator, function and filter predicates. We will discuss aggregators later in the chapter, but they share the same structure.

A key design decision to JCascalog was to make all predicates share a common structure. The first argument to a predicate is the "predicate operation", and the remaining arguments are parameters for that operation. For function and aggregator predicates, the labels for the outputs are specified using the "out" method.

Being able to represent every piece of your computation via the same simple, consistent mechanism is the key to enabling highly composable abstractions. Despite their simple structure, predicates provide extremely rich semantics. This is best illustrated by examples, as shown in Figure 5.6:

Type	Example	Description
Function as filter	`.predicate(new Plus(), 2, "?x").out(6)`	*Although Plus() is a function, this predicate filters all tuples where the value of ?x ≠ 4*
Compound filter	`.predicate(new Multiply(), 2, "?a").out("?z")` `.predicate(new Multiply(), 3, "?b").out("?z")`	*In concert, these predicates filter all tuples where 2(?a) ≠ 3(?b)*

Figure 5.6 The simple predicate structure can express deep semantic relationships to describe the desired query output.

As we earlier mentioned, joins between datasets are also expressed via predicates - we'll expand on this next.

5.3.3 Querying multiple datasets

Many queries will require that you combine multiple datasets together. In relational databases this is most commonly done via a join operation, and joins exist in JCascalog as well.

Suppose you want to combine the AGE and GENDER datasets to create a new set of tuples that contains the age and gender of all people that exist in both datasets. This is a standard inner join on the "?person" field and is illustrated in Figure 5.7:

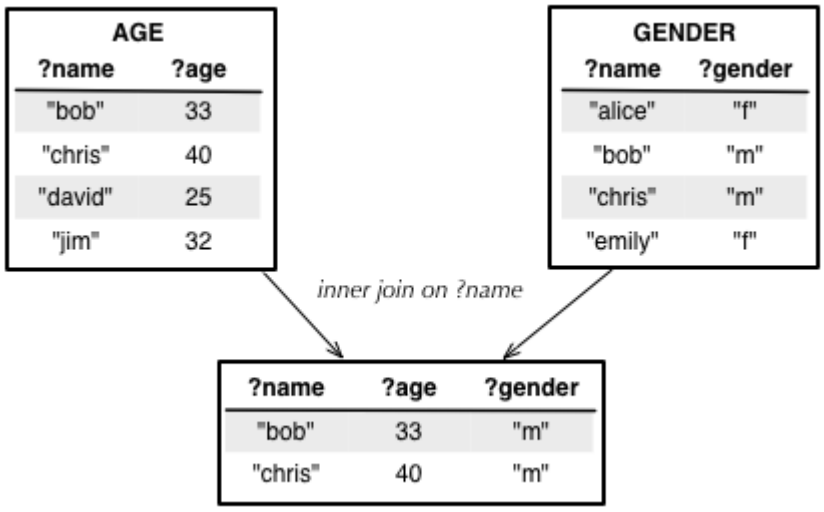


Figure 5.7 This inner join of the AGE and GENDER datasets merges the data for tuples for values of ?person that are present in both datasets.

In a language like SQL, joins are expressed *explicitly*. However, joins in JCascalog are implicit based on the variable names. Figure 5.8 highlights the differences:

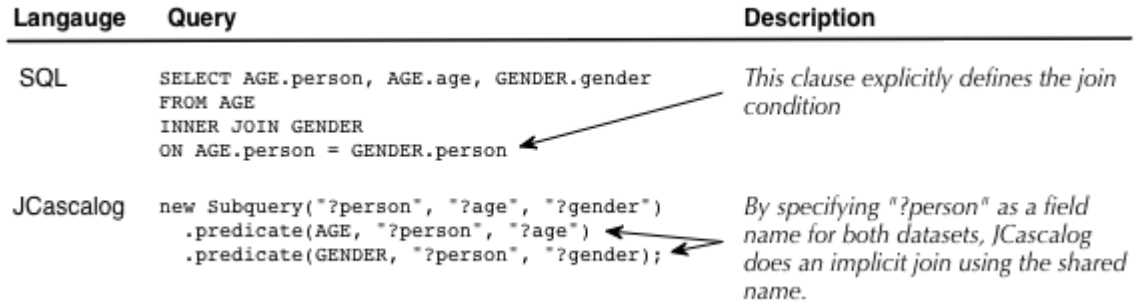


Figure 5.8 A comparison between SQL and JCascalog syntax to perform the inner join between the AGE and GENDER datasets.

For the JCascalog query, the same field name ?person is used as the output of two different generator predicates, AGE and GENDER. Since each instance of the variable must have the same value for any resulting tuples, JCascalog knows that the right way to resolve the query is to do an inner join between the AGE and GENDER datasets.

Inner joins only emit tuples for join fields that exist for all sides of the join. However, there are circumstances where you may results for records that don't exist in one dataset or the other, getting a null value for the non-existing data. These operations are called outer joins and are just as easy to do in JCascalog. Consider the join examples if Figure 5.9:

Left Outer Join

```
new Subquery("?person", "?age", "!!gender")
  .predicate(AGE, "?person", "?age")
  .predicate(GENDER, "?person", "!!gender");
```

?name	?age	?gender
"bob"	33	"m"
"chris"	40	"m"
"david"	25	null
"jim"	32	null

Full Outer Join

```
new Subquery("?person", "!!age", "!!gender")
  .predicate(AGE, "?person", "!!age")
  .predicate(GENDER, "?person", "!!gender");
```

?name	?age	?gender
"alice"	null	"f"
"bob"	33	"m"
"chris"	40	"m"
"david"	25	null
"emily"	null	"f"
"jim"	32	null

Figure 5.9 JCascalog queries to implement two types of outer joins between the AGE and GENDER datasets.

As mentioned earlier, for outer joins, JCascalog uses fields beginning with "!!" to generate null values for non-existing data. In the left outer join, a person must have an age to be included in the result set with null values introduced for missing gender data. For the full outer join, all people present in either data set are included in the results with null values for any missing age or gender data.

Besides joins, there are a few other ways to combine datasets. Occasionally you have two datasets that contain the same type of data and you want to merge them into a single dataset. For this, JCascalog provides the "combine" and "union"

functions. The "combine" function concatenates the datasets together, whereas "union" will remove any duplicate records during the combining process. Figure 5.10 illustrates the difference between the two functions:

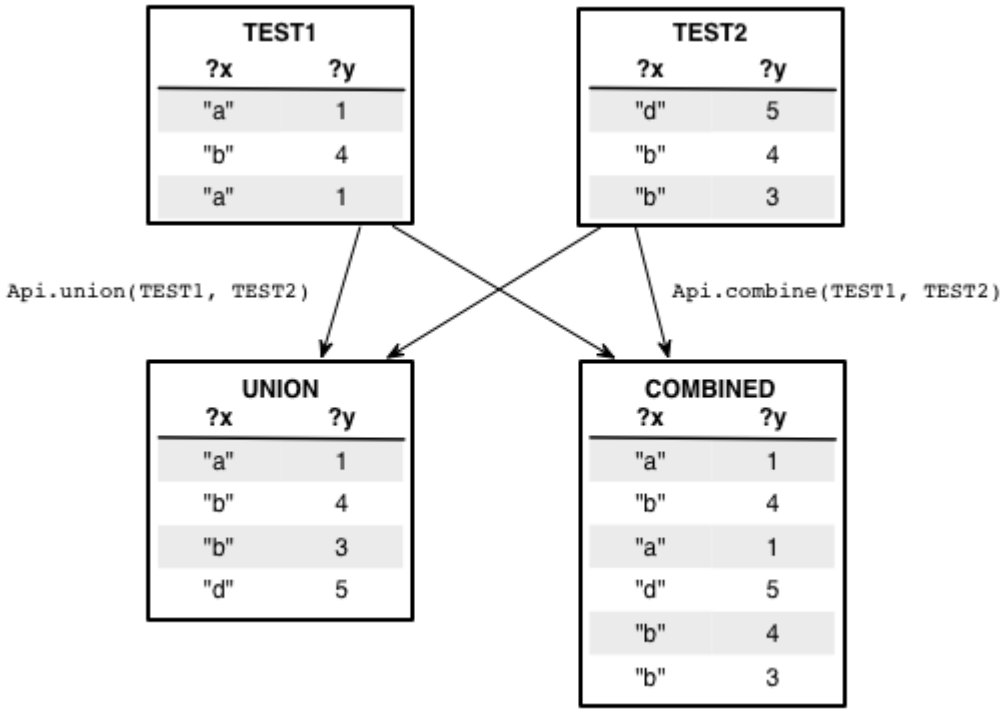


Figure 5.10 JCascalog provides two different means to merge compatible datasets: combine and union. Combine does a simple aggregation of the two sets, whereas union removes any duplicate tuples.

So far you've seen transformation that act on one tuple at a time or combine datasets together. We next cover operations that process groups of tuples.

5.3.4 Grouping and aggregators

There are many types of queries where you want to aggregate information for specific groups: "what is the average salary for different professions?" or "what age group writes the most tweets?" In SQL you explicitly state how records should be grouped and the operations to be performed on the resulting sets.

Following the implicit / explicit differences we observed with joins, there is no explicit "GROUP BY" command in JCascalog to indicate how to partition tuples for aggregation. Instead, the grouping is implicit based on the desired query output. To understand this, let's look at couple of examples. The first example uses the Count aggregator to finds the number of people each person follows.

```
new Subquery("?person", "?count")                    (1)
  .predicate(FOLLOWS, "?person", "_")                (2)
  .predicate(new Count(), "?count");                 (3)
```

(1) the output field names define all potential groupings
(2) the underscore informs JCascalog to ignore this field
(3) when executing the aggregator, the output fields imply tuples should be grouped by ?person

When JCascalog executes the count predicate, it deduces from the declared output that a grouping on "?person" must be done first. The second example is similar, but performs a couple of other operations before applying the aggregator.

```
new Subquery("?gender", "?count")                    (1)
  .predigate(GENDER, "?person", "?gender")
  .predicate(AGE, "?person", "?age")                 (2)
  .predicate(new LT(), "?age", 30)                   (3)
  .predicate(new Count(), "?count");                 (4)
```

(1) this query will group tuples by ?gender
(2) before the aggregator, the AGE and GENDER datasets are joined
(3) tuples are then filtered on ?age
(4) even though the ?person and ?age fields were used in earlier predicates, they are discarded by the aggregator since they are not included in the specified output

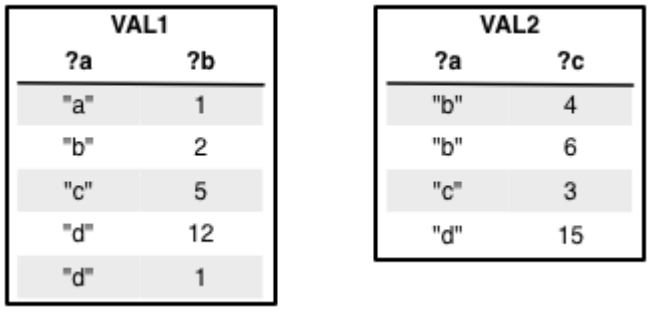
After the AGE and GENDER datasets are joined, JCascalog filters all people with age 30 or above. At this point, the tuples are grouped by gender and the count aggregator is applied.

Within the MapReduce framework, there are multiple ways aggregators can distribute the computation across the map and reduce stages. JCascalog actually supports three types of aggregators: *Aggregators*, *Buffers* and *Parallel Aggregators* . We are only introducing the notion for now, as we'll delve into the differences when we cover implementing custom predicates.

We've spoken at length about the different types of JCascalog predicates. Next, let's step through the execution of a query to see how tuple sets are manipulated at different stages of its computation.

5.3.5 Stepping though an example query

For this exercise, we'll start with two test datasets as shown in Figure 5.11:



VAL1

?a	?b
"a"	1
"b"	2
"c"	5
"d"	12
"d"	1

VAL2

?a	?c
"b"	4
"b"	6
"c"	3
"d"	15

Figure 5.11 Test data for our query execution walkthrough

We'll use the following query to explain the execution of a JCascalog query, observing how the sets of tuples change at each stage in the execution.

```
new Subquery("?a", "?avg")
  .predicate(VAL1, "?a", "?b")                                    ❶
  .predicate(VAL2, "?a", "?c")
  .predicate(new Multiply(), 2, "?b").out("?double-b")            ❷
  .predicate(new LT(), "?b", "?c")
  .predicate(new Count(), "?count")                               ❸
  .predicate(new Sum(), "?double-b").out("?sum")
  .predicate(new Div(), "?sum", "?count").out("?avg")
  .predicate(new Multiply(), 2, "?avg").out("?double-avg")        ❹
  .predicate(new LT(), "?double-avg", 50);
```

❶ generators for the test datasets
❷ pre-aggregator function and filter
❸ multiple aggregators
❹ post-aggregator predicates

At the start of a JCascalog query, the generator datasets exist in independent branches of the computation. In the first stage of execution, JCascalog applies functions, filters tuples and joins datasets until it can no longer do so. A function or filter can be applied if all the input variables for the operation are available. This stage for our query is illustrated in Figure 5.12.

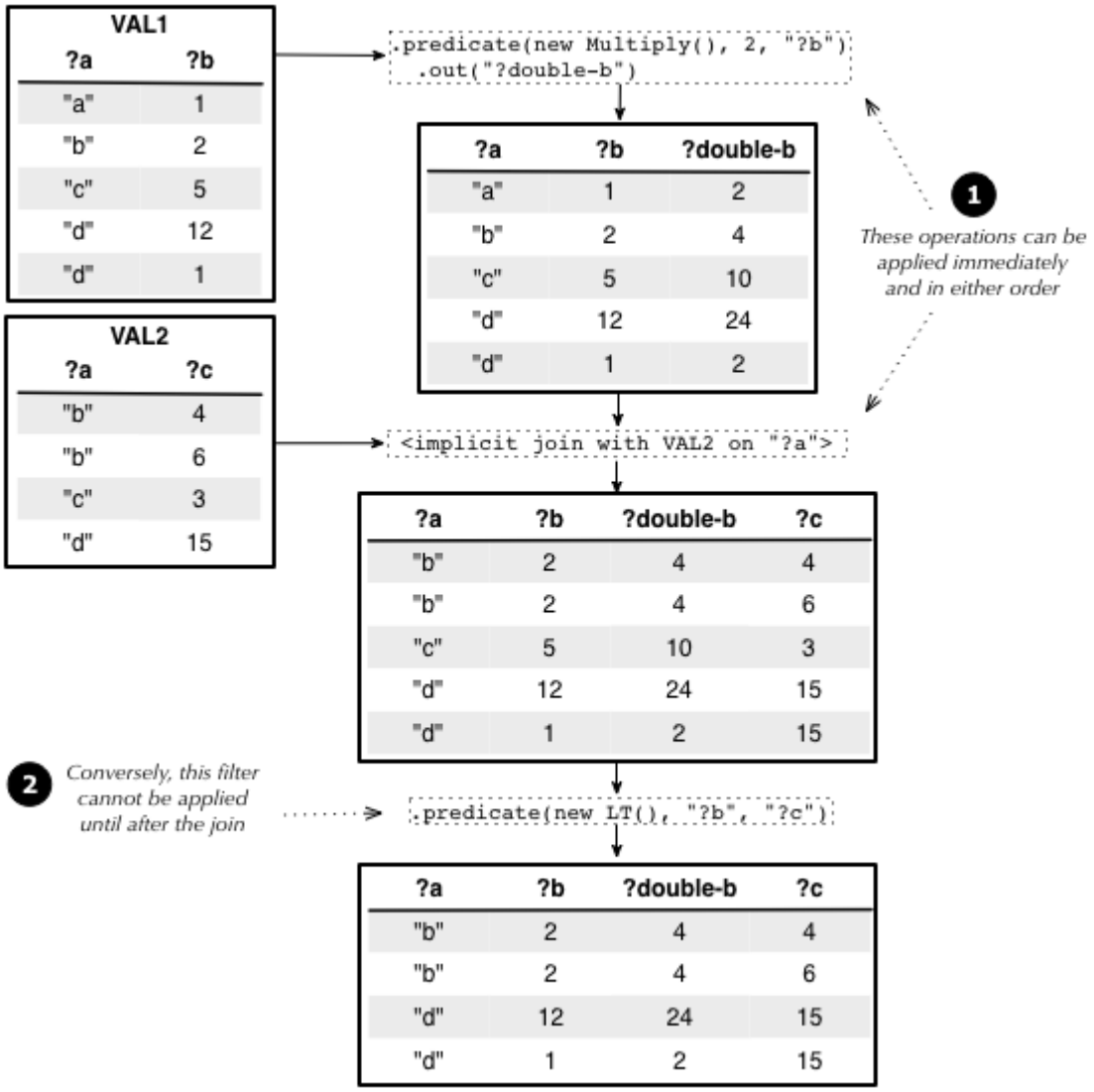


Figure 5.12 The first stage of execution entails of applying all functions, filters and joins where the input variables are available.

Note that some predicates require other predicates to be applied first. In the example, the "less than" filter could not be applied until after the join was performed.

Eventually this phase reaches a point where no more predicates can be applied because the remaining predicates are either aggregators or require variables that are not yet available. At this point, JCascalog enters the aggregation phase of the query. JCascalog groups the tuples by any variables available that are declared as output variables for the query, then applies the aggregators to each group of tuples. This is demonstrated in Figure 5.13:

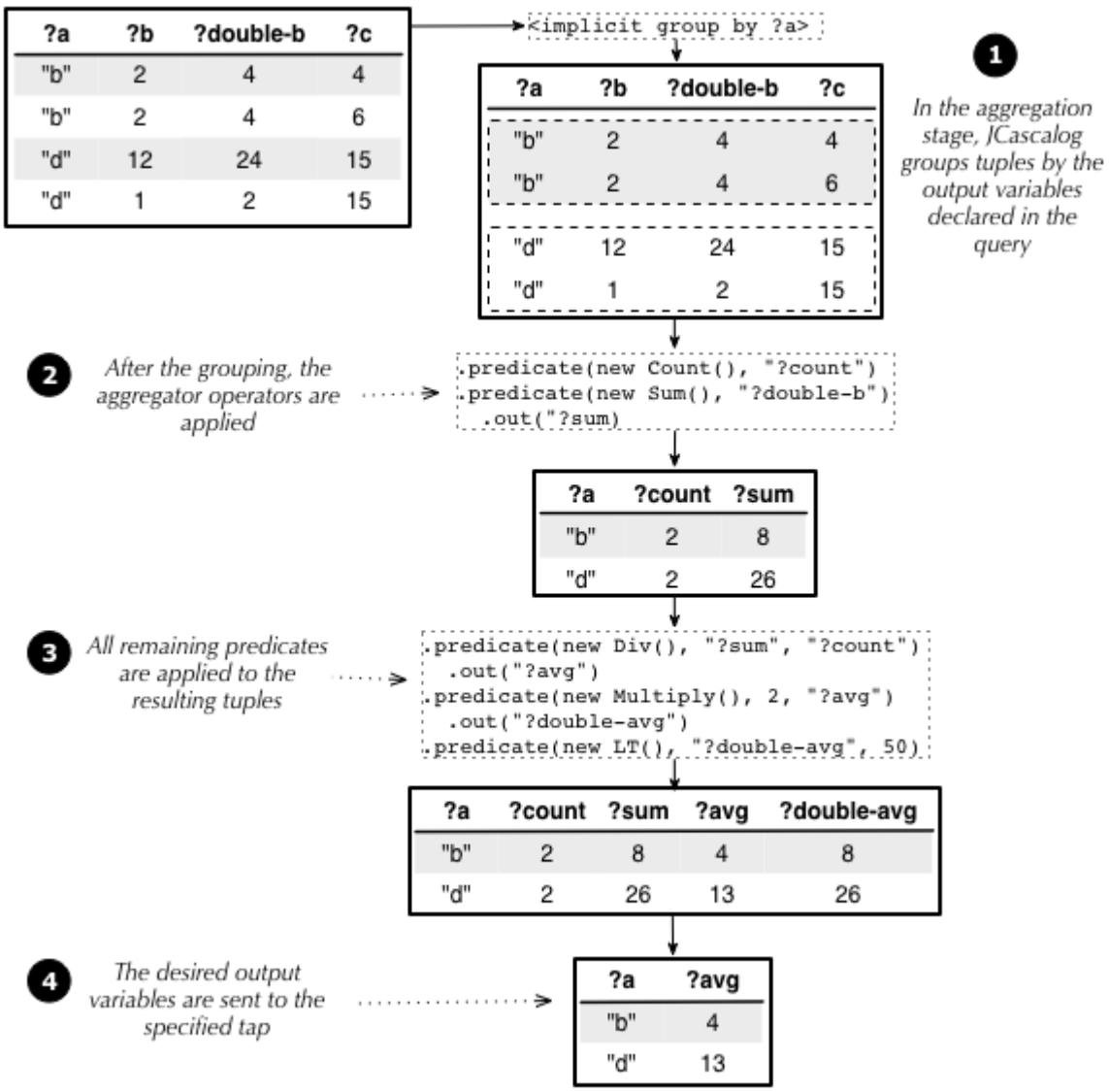


Figure 5.13 The aggregation and post-aggregation stages for the query. The tuples are grouped based on the desired output variables, then all aggregators are applied. All remaining predicates are then executed and the desired output is returned.

After the aggregation phase, all remaining functions and filters are applied. The end of this phase drops any variables from the tuples that are not declared in the output fields for the query.

SIDEBAR **A verbose explanation**

You may have noticed that this example computes an average by doing a count, sum, and division. This was solely for the purposes of illustration – these operations can be abstracted into an "Average" aggregator, as we glanced at earlier in this chapter.

You may have also noticed that some variables are never used after a point yet still remain in the resulting tuple sets. For example, the ?b variable is not used after the LT predicate is applied, yet is still grouped along with the other variables. In reality, JCascalog will drop any variables once they're no longer needed so that they are not serialized or transferred over the network.

You've now seen how to use predicates to construct arbitrarily complex queries that filter, join, transform and aggregate your data. We next demonstrate how to extend JCascalog with customized predicates.

5.3.6 Custom predicate operations

While JCascalog has a rich library of built-in predicate operations, you will frequently need to create additional predicate types to implement your business logic. Toward this end, JCascalog exposes simple interfaces to define new filters, functions and aggregators.

FILTERS

We begin with filters. A filter predicate requires a single method named "isKeep" that returns true if the input tuple should be kept and false if it should be filtered. The following is a filter that keeps all tuples where the input is greater than 10.

```
public static class GreaterThanTenFilter extends CascalogFilter {
  public boolean isKeep(FlowProcess process, FilterCall call) {
    return call.getArguments().getInteger(0) > 10;  (1)
  }
}
```

(1) obtain the first element of the input tuple and treat the value as an integer

FUNCTIONS

Next up are functions. Like filters, a function predicate implements a single method - in this case named "operate". A function takes in a set of input and then emits zero or more tuples as output. Here's a simple function that implements its input value by one:

```
public static class IncrementFunction extends CascalogFunction {
  public void operate(FlowProcess process, FunctionCall call) {
    int v = call.getArguments().getInteger(0);   (1)
    call.getOutputCollector().add(new Tuple(v + 1));   (2)
  }
}
```

(1) obtain the value from the input tuple
(2) emit a new tuple with the incrememnted value

Figure 5.14 shows the result of applying this function to a set of tuples.

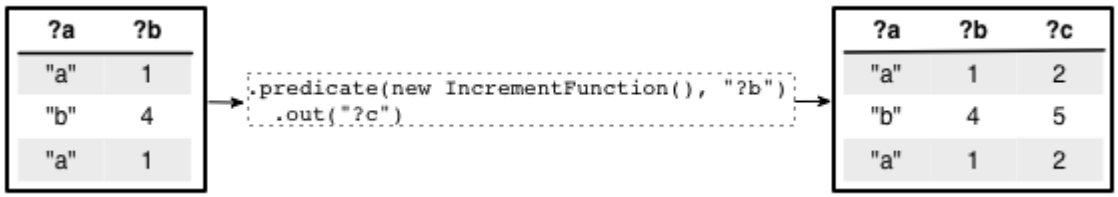


Figure 5.14 The IncrementFunction predicate applied to some sammple tuples

Recall earlier that a function can act as a filter if it emits zero tuples for a given tuple. Here's a function that attempts to parse an integer from a string, filtering out the tuple if the parsing fails:

```
public static class TryParseInteger extends CascalogFunction {
  public void operate(FlowProcess process, FunctionCall call) {
    String s = call.getArguments().getString(0);   (1)
    try {
      int i = Integer.parseInt(s);
      call.getOutputCollector().add(new Tuple(i));   (2)
    }
    catch(NumberFormatException e) {}   (3)
  }
}
```

1. regard input value as a string
2. emit value as integer if parsing succeeds
3. else emit nothing if parsing fails

Figure 5.15 illustrates this function applied to a tuple set. You can observe that one tuple is filtered by the process.

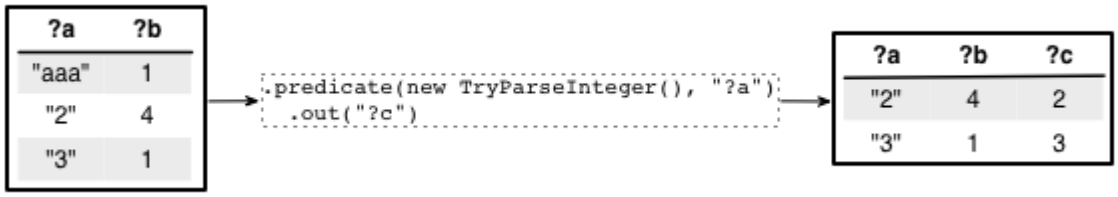


Figure 5.15 The TryParseInteger function filters rows that where ?a cannot be converted to an integer value

Finally, if a function emits multiple output tuples, each output tuple is appended to its own copy of the input arguments. As an example, here is the Split function from word count:

```
public static class Split extends CascalogFunction {
  public void operate(FlowProcess process, FunctionCall call) {
    String sentence = call.getArguments().getString(0);
    for(String word: sentence.split(" ")) {          (1)
      call.getOutputCollector().add(new Tuple(word));   (2)
    }
  }
}
```

1. for simplicity, split into words using a single white space
2. emit each word as a separate tuple

Figure 5.16 shows the result of applying this function to a set of sentences. You can see that each input sentence gets duplicated for each word it contains.

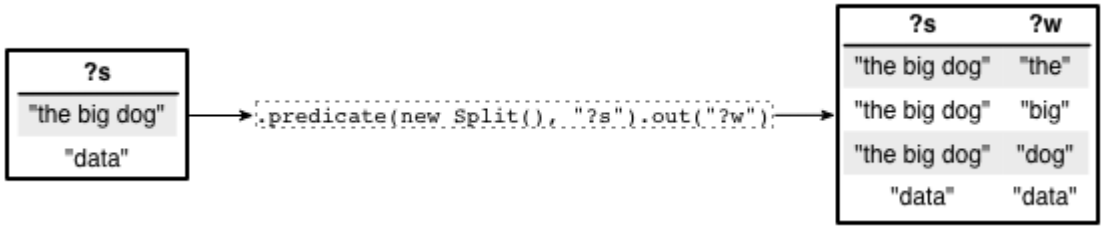


Figure 5.16 The Split function can emit multiple tuples from a single input tuple

AGGREGATORS

The last class of customizable predicate operations are aggregators. As we mentioned earlier, there are three types of aggregrators, each with different properties regarding composition and performance.

Perhaps rather obviously, the first type of aggregator is literally called an *Aggregator*. An Aggregator looks at one tuple at a time for each tuple in a group, adjusting some internal state for each observed tuple. The following is an implementation of sum as an Aggregator:

```
public static class SumAggregator extends CascalogAggregator {
  public void start(FlowProcess process, AggregatorCall call) {
    call.setContext(0);  (1)
  }

  public void aggregate(FlowProcess process, AggregatorCall call) {
    int total = (Integer) call.getContext();
    call.setContext(total + call.getArguments().getInteger(0));  (2)
  }

  public void complete(FlowProcess process, AggregatorCall call) {
    int total = (Integer) call.getContext();
    call.getOutputCollector().add(new Tuple(total));  (3)
  }
}
```

(1) initialize the aggregator internal state
(2) called for each tuple; updates the internal state to store the running sum
(3) once all tuples are processed, emit a tuple with the final result

The next type of aggregator is called a *Buffer*. A buffer receives an iterator to the entire set of tuples for a group. Here's an implementation of sum as a Buffer:

```
public static class SumBuffer extends CascalogBuffer {
  public void operate(FlowProcess process, BufferCall call) {
    Iterator<TupleEntry> it = call.getArgumentsIterator();  (1)
    int total = 0;
    while(it.hasNext()) {
      TupleEntry t = it.next();
      total+=t.getInteger(0);
    }
    call.getOutputCollector().add(new Tuple(total));  (2)
  }
}
```

1. the tuple set is accessible via an iterator
2. a single function iterate overs all tuples and emits the output tuple

Buffers are easier to write than aggregators since you only must implement one method rather than three. However, unlike buffers, aggregators can be chained in a query. *Chaining* means you can compute multiple aggregations at the same time for the same group. Buffers cannot be used along with any other aggregator type, but Aggregators can be used with other Aggregators.

In the context of the MapReduce framework, both Buffers and Aggregators rely on reducers to perform the actual computation for these operators. This is illustrated in Figure 5.17:

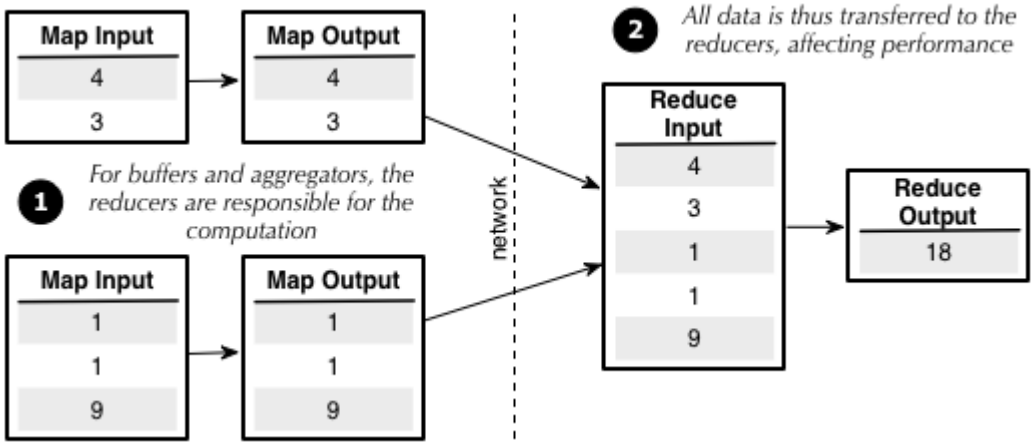


Figure 5.17 Execution of sum Aggregator and sum Buffer at the MapReduce level

JCascalog packs together as many operations as possible into map and reduce tasks, but these operators are solely performed by reducers. This necessitates a network intensive approach since all data for the computation must flow from the mappers to the reducers. Furthermore, if there were only a single group (e.g. counting the number of tuples in a dataset), all the tuples will have to be sent to a single reducer for aggregation, defeating the purpose of using a parallel computation system.

Fortunately, the last type of aggregator operation can do aggregations more scalably and efficiently. A *Parallel Aggregator* performs an aggregation incrementally by doing partial aggregations in the map tasks. Figure 5.18 shows the division of labor for sum when implemented as a Parallel Aggregator.

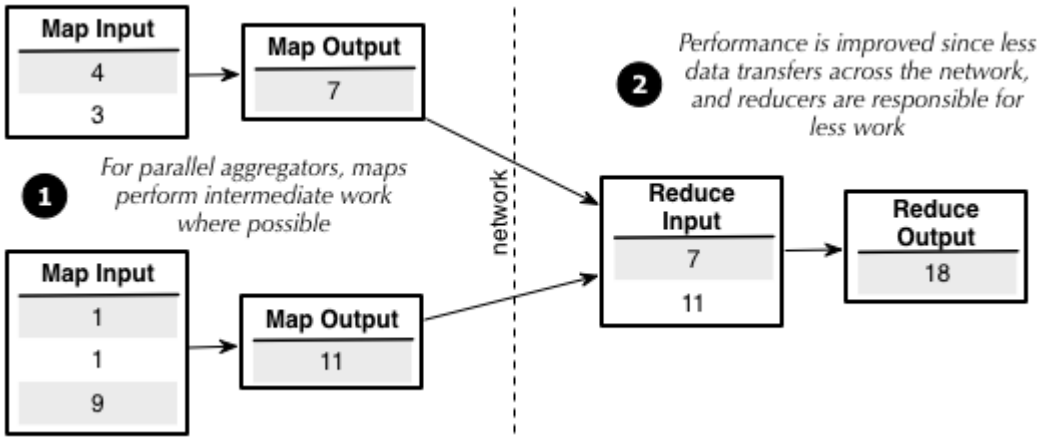


Figure 5.18 Execution of a sum Parallel Aggregator at the MapReduce level

When an aggregator is implemented as a Parallel Aggregator, it therefore executes much more efficiently. To write your own Parallel Aggregator, you must implement two functions:

- the *init* function maps the arguments from a single tuple to a partial aggregation for that tuple, and
- the *combine* function specifies how to combine two partial aggregations into a single aggregation value.

The following listing implements sum as a Parallel Aggregator.

```
public static class SumParallel implements ParallelAgg {
  public List<Object> init(List<Object> input) {
    return input;   (1)
  }

  public List<Object> combine(List<Object> input1,
    List<Object> input2) {
    int val1 = (Integer) input1.get(0);
    int val2 = (Integer) input2.get(0);
    return Arrays.asList((Object) (val1 + val2));   (2)
  }
}
```

(1) for sum, the partial aggregation is just the value in the argument
(2) to combine two partial aggregations, simply sum the values

Parallel Aggregators can be chained with other Parallel Aggregators or regular Aggregators. However, when chained with regular Aggregators, Parallel Aggregators are unable to do partial aggregations in the map tasks and will act like

regular Aggregators.

You've now seen all the abstractions that comprise JCascalog subqueries: predicates, functions, filters and aggregators. The power of these abstractions lies is how they promote reuse and composability; let's now take a look at the various composition techniques possible with JCascalog.

5.4 Composition

During our discussion on minimizing accidental complexity in your data processing code, we emphasized that abstractions should be composable to create new and greater functionalities. This philosophy is pervasive throughout JCascalog. In this section we will cover composing abstractions via combining subqueries, predicate macros, and functions to dynamically create both subqueries and macros. These techniques take advantage of the fact that there's no barrier between the query tool and the general purpose programming language, allowing you to manipulate your queries in a very fine-grained way.

5.4.1 Combining subqueries

Subqueries are the basic unit of abstraction in JCascalog, for they represent an arbitrary view on any number of data sources. One of the most powerful features of subqueries is that they can be addressed as data sources for other subqueries. Just as you break down a large program into many functions, this allows you to similarly deconstruct large queries.

Let's look at an example to find all the records from the FOLLOWS dataset where each person in the record follows more than 2 people:

```
Subquery manyFollows = new Subquery("?person")                    ❶
                         .predicate(FOLLOWS, "?person", "_")      ❷
                         .predicate(new Count(), "?count")
                         .predicate(new GT(), "?count", 2);       ❸

Api.execute(new StdoutTap(),
            new Subquery("?person1", "?person2")
              .predicate(manyFollows, "?person1")                 ❹
              .predicate(manyFollows, "?person2")
              .predicate(FOLLOWS, "?person1", "?person2"));       ❺
```

❶ the first subquery determines all people that follow more than 2 others
❷ consider only the follower, not the source
❸ count the number of people each user follows and keep those with count greater

than 2
4. use the results of the first subquery as the source for this subquery
5. only keep records where follower and source are both present in the result of the first subquery

Subqueries are lazy – nothing is computed until Api.execute is called. In the previous example, even though the *manyFollows* subquery is defined first, no MapReduce jobs are launched until the Api.execute call is made.

Here's another example of a query that requires multiple subqueries. This query extends word count by finding the number of words that exist for each computed word count:

```
Subquery wordCount = new Subquery("?word", "?count")   (1)
                       .predicate(SENTENCE, "?sentence")
                       .predicate(new Split(), "?sentence").out("?word")
                       .predicate(new Count(), "?count");

Api.execute(new StdoutTap(),
            new Subquery("?count", "?num-words")
              .predicate(wordCount, "_", "?count")   (2)
              .predicate(new Count(), "?num-words"));   (3)
```

1. the basic word count subquery
2. the second subquery only requires the count for each word
3. determine the number of words for each count value

Combining subqueries is a powerful paradigm to express complex operations using simple components. This power is made more accessible since functions can generate subqueries directly, as we will discuss next.

5.4.2 Dynamically created subqueries

One of the most common techniques when using JCascalog is to write functions that create subqueries dynamically. That is, you write regular Java code that constructs a subquery according to some parameters. You have previously witnessed the advantages of using subqueries as data sources of other subqueries, and generating subqueries dynamically makes it easier to access these benefits.

For example, suppose you have text files on HDFS representing transaction data: an ID for the buyer, an ID for the seller, a timestamp, and a dollar amount. The data is JSON-encoded and looks like this:

```
{"buyer": 123, "seller": 456, "amt": 50, "timestamp": 1322401523}
{"buyer": 1009, "seller": 12, "amt": 987, "timestamp": 1341401523}
{"buyer": 2, "seller": 98, "amt": 12, "timestamp": 1343401523}
```

You may have a variety of computations you want to run on this data, but each of your queries share a common need of parsing the data from the text files. A useful utility function would take an HDFS path and return a subquery that parses the data at that location:

```
public static class ParseTransactionRecord extends CascalogFunction {   (1)
  public void operate(FlowProcess process, FunctionCall call) {
    String line = call.getArguments().getString(0);
    Map parsed = (Map) JSONValue.parse(line);   (2)
    call.getOutputCollector().add(new Tuple(parsed.get("buyer"),   (3)
                                            parsed.get("seller"),
                                            parsed.get("amt"),
                                            parsed.get("timestamp")));
    }
}

public static Subquery parseTransactionData(String path) {   (4)
  return new Subquery("?buyer", "?seller", "?amt", "?timestamp")
    .predicate(Api.hfsTextline(path), "?line")   (5)
    .predicate(new ParseTransactionRecord(), "?line")   (6)
    .out("?buyer", "?seller", "?amt", "?timestamp");
}
```

(1) the subquery needs a Cascalog function to perform the actual parsing
(2) an external library converts the JSON to a map
(3) the desired map values are translated into a single tuple
(4) a regular Java function dynamically generates the subquery
(5) generate a tap from the provided HDFS path
(6) call the custom JSON parsing function

Once defined, you can use this abstraction for any query over the dataset. For example, here's a query that computes the number of transactions for each buyer:

```
public static Subquery buyerNumTransactions(String path) {
  return new Subquery("?buyer", "?count")
    .predicate(parseTransactionData(path), "?buyer", "_", "_", "_")   (1)
    .predicate(new Count(), "?count");
}
```

❶ disregard all fields but the buyer

This is a very simple example of creating subqueries dynamically, but it illustrates how subqueries can be composed together in order to enable abstracting away pieces of a more complicated computation. Let's look at another example in which the number of predicates is dynamic according to the arguments.

Suppose you have a set of retweet data with each record denoting a retweet of some other tweet, and you want find all chains of retweets of a certain length. That is, for a chain of length 4, you want to know all retweets of retweets of retweets of tweets. The original dataset consists of pairs of tweet identifiers. Notice that you can transform these pairs into chains of length 3 by joining the dataset with itself. Similarly, you can then find chains of length 4 by joining the length 3 chains with the original pairs. To illustrate, here's a query that returns chains of length 3 given an input generator of pairs:

```
public static Subquery chainsLength3(Object pairs) {
  return new Subquery("?a", "?b", "?c")
    .predicate(pairs, "?a", "?b")
    .predicate(pairs, "?b", "?c");
}
```

An additional join finds all chains of length 4:

```
public static Subquery chainsLength4(Object pairs) {
  return new Subquery("?a", "?b", "?c", "?d")
    .predicate(pairs, "?a", "?b")
    .predicate(pairs, "?b", "?c")
    .predicate(pairs, "?c", "?d");
}
```

To generalize this process to find chains of any length, you need a function that generates a subquery with the correct number of predicates and variables. This can be accomplished by writing some fairly simple Java code:

```
public static Subquery chainsLengthN(Object pairs, int n) {
  List<String> genVars = new ArrayList<String>();
  for(int i=0; i<n; i++) {
    genVars.add(Api.genNullableVar());  ❶
  }
```

```
  Subquery ret = new Subquery(genVars);
  for(int i=0; i<n-1; i++) {
    ret = ret.predicate(pairs, genVars.get(i), genVars.get(i+1));  (2)
  }
  return ret;
}
```

(1) generate unique nullable output variables
(2) loop to define the required number of joins

An interesting note about this function is that it's not specific to retweet data: in fact, it can take any subquery or source of data containing pairs and return a subquery that computes chains.

Let's look at one more example of a dynamically created subquery. Suppose you want to draw a random sample of N elements from a dataset of unknown size. The simplest strategy to accomplish this in a distributed and scalable way is with the following algorithm:

1. Generate a random number for every element
2. Find the N elements with the smallest random numbers

JCascalog has a built-in aggregator named "Limit" for performing the second step. Limit uses a strategy similar to parallel aggregators where it finds the smallest N elements on each map task, then combines the results from all the map tasks to find the N smallest elements overall. The following code implements this strategy to draw a random sample:

```
public static Subquery fixedRandomSample(Object data, int n) {
  List<String> inputVars = new ArrayList<String>();
  List<String> outputVars = new ArrayList<String>();
  for(int i=0; i < Api.numOutFields(data); i++) {  (1)
    inputVars.add(Api.genNullableVar());  (2)
    outputVars.add(Api.genNullableVar());
  }

  String randVar = Api.genNullableVar();  (3)
  return new Subquery(outputVars)
    .predicate(data, inputVars)
    .predicate(new RandLong(), randVar)  (4)
    .predicate(Option.SORT, randVar)  (5)
    .predicate(new Limit(n), inputVars).out(outputVars);  (6)
}
```

1. introspect the input dataset to determine the correct number of input and output fields
2. generate separate fields for input and output variables
3. create a separate field to hold the random values
4. use the JCascalog RandLong function to append each input tuples with a random value
5. perform secondary sorting on the random values
6. use the Limit aggregator to find N random tuples from the dataset

The brilliant aspect of this algorithm is its scalability: it parallelizes the computation of the fixed sample without ever needing to centralize all the records in one place.

When writing JCascalog queries, you will notice that certain combinations of predicates are frequently used together. In these situations it is simpler and more efficient to express the collective functionality with a single operation. We next delve into how JCascalog supports this ability by use of predicate macros.

5.4.3 Predicate macros

A predicate macro is an operation that JCascalog expands to another set of predicates. Since JCascalog represents all operations as predicates, macros can create powerful abstractions by composing predicates together, whether they are aggregators, filters or functions.

You've already seen one example of a predicate macro with the definition of Average from the beginning of this chapter. Let's look at that definition once more:

```
PredicateMacroTemplate Average =
  PredicateMacroTemplate.build("?val")    (1)
    .out("?avg")    (2)
    .predicate(new Count(), "?count")    (3)
    .predicate(new Sum(), "?val").out("?sum")    (4)
    .predicate(new Div(), "?sum", "?count").out("?avg");    (5)
```

1. use a template to define a macro with one input variable
2. the macro returns a single output
3. average expands to three predicates: count, sum and div
4. temp variables store the results of the aggregation
5. divide aggregate results to compute final output

Average consists of three predicates composed together: a count aggregation, a

sum aggregation, and a division function. Figure 5.19 demonstrates how Average is called and its resulting expansion:

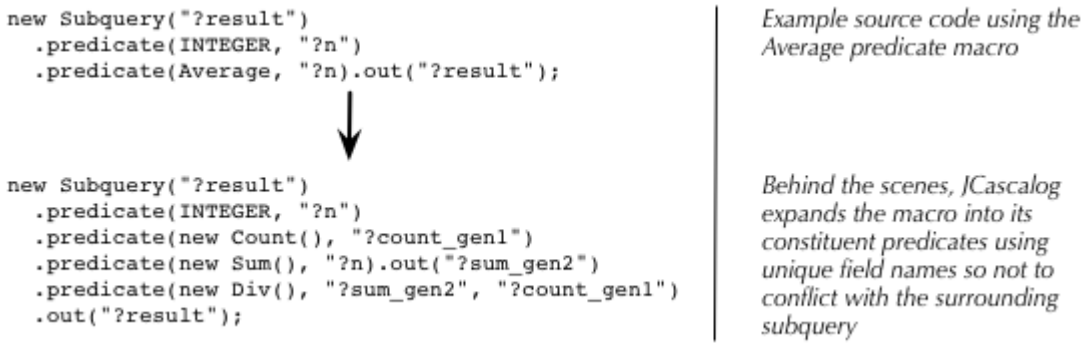


Figure 5.19 Predicate macros provide powerful abstractions to write simple queries that JCascalog automatically expands into the constituent predicates

The definition of Average uses a JCascalog template to specify simple compositions of predicates when the number of input and output fields are fixed. However, not everything can be specified with a template. For example, suppose you wanted to create a predicate macro that computes the number of distinct values for a given set of variables, like so:

```
new Subquery("?distinct-followers-count")
  .predicate(FOLLOWS, "?person", "_")
  .predicate(new DistinctCount(), "?person")   ❶
  .out("?distinct-followers-count");
```

❶ the desired macro counts the number of distinct followers

This subquery determines the number of distinct users that follow at least one other person. Unlike calculating the average of a single variable, you could potentially calculate distinct counts for variable sets of any size. Predicate macros can support this generality of arbitrary sets, but you must define the macro manually instead of using a template.

Towards this end, you first need to define an aggregator that performs the actual computation. This aggregator must work even if the number of tuples for a group is so large that it could not be contained in memory. To solve this problem, you can make use of a feature called *secondary sorting* that sorts the group of

tuples before being processed by the aggregator. Once sorted, the aggregator only increments the distinct count if the current tuple is different from its predecessor. The code to perform the aggregation is provided below:

```
public static class DistinctCountAgg extends CascalogAggregator {
  static class State {                                          (1)
    int count = 0;
    Tuple last = null;
  }

  public void start(FlowProcess process, AggregatorCall call) {
    call.setContext(new State());                               (2)
  }

  public void aggregate(FlowProcess process, AggregatorCall call) {
    State s = (State) call.getContext();                        (3)
    Tuple t = call.getArguments().getTupleCopy();
    if(s.last==null || !s.last.equals(t)) {
      s.count++;                                                (4)
    }
    s.last = t;                                                 (5)
  }

  public void complete(FlowProcess process, AggregatorCall call) {
    State s = (State) call.getContext();
    call.getOutputCollector().add(new Tuple(s.count));          (6)
  }
}
```

(1) internal state to track the current count and the previously seen tuple
(2) for each group, initialize the tracking state
(3) when processing a tuple, retrieve the current state
(4) increase the distinct count only if the current tuple differs from the previous one
(5) always update the last seen tuple in the state
(6) when all tuples of the group have been processed, emit the distinct count

DistinctCountAgg contains the logic to compute the unique count given a sorted input; unsurprisingly, JCascalog has an Option.SORT predicate to specify how to sort the tuples for each group. The following listing demonstrates how you define the sort and compute the distinct count by hand:

```
public static Subquery distinctCountManual() {
  return new Subquery("?distinct-followers-count")
    .predicate(FOLLOWS, "?person", "_")
```

```
    .predicate(Option.SORT, "?person")  ❶
    .predicate(new DistinctCountAgg(), "?person")
    .out("?distinct-followers-count");
```

❶ sorts the tuple by

Of course, you would much prefer a macro so that you don't have to specify the sort and aggregator each time you want to do a distinct count. The most general form of a predicate macro is a function that takes a list of input fields, a list of output fields, and returns a set of predicates. The following is the definition of DistinctCount as a regular PredicateMacro:

```
public static class DistinctCount implements PredicateMacro {
  public List<Predicate> getPredicates(Fields inFields,
                                       Fields outFields) {  ❶
    List<Predicate> ret = new ArrayList<Predicate>();
    ret.add(new Predicate(Option.SORT, inFields));  ❷
    ret.add(new Predicate(new DistinctCountAgg(),
                          inFields,
                          outFields));  ❸
    return ret;
  }
}
```

❶ the input and output fields are determined when the macro is used within a subquery
❷ groups are sorted by the provided input fields
❸ for this macro distinct count emits a single field, but the general macro form supports multiple outputs

5.4.4 Dynamically created predicate macros

You previously saw how regular Java functions can dynamically create subqueries, so it is no great surprise that you can do the same with predicate macros. This is an extremely powerful technique that showcases the advantages of having your query tool just be a library for your general purpose programming language.

Consider the following query:

```
new Subquery("?x", "?y", "?z")
  .predicate(TRIPLETS, "?a", "?b", "?c")  ❶
  .predicate(new IncrementFunction(), "?a").out("?x")  ❷
```

```
  .predicate(new IncrementFunction(), "?b").out("?y")
  .predicate(new IncrementFunction(), "?c").out("?z");
```

1. read a dataset containing triples of numbers
2. return a new triplet where each field is incremented

Although a simple query, there is considerable repetition since it must explicitly apply the IncrementFunction to each field from the input data. It would be nice to have a macro that eliminates this repetition, like so:

```
new Subquery("?x", "?y", "?z")
  .predicate(TRIPLETS, "?a", "?b", "?c")
  .predicate(new Each(new IncrementFunction()), "?a", "?b", "?c")
  .out("?x", "?y", "?z");
```

Rather than repeatedly using the IncrementFunction, the Each macro applies the function to the specified input fields and generates the desired output. The expansion of the macro matches the three separate predicates in the original query. The Each macro...

```
public static class Each implements PredicateMacro {
  Object _op;

  public Each(Object op) {
    _op = op;   (1)
  }

  public List<Predicate> getPredicates(Fields inFields,
                                       Fields outFields) {
    List<Predicate> ret = new ArrayList<Predicate>();
    for(int i=0; i<inFields.size(); i++) {
      Object in = inFields.get(i);
      Object out = outFields.get(i);
      ret.add(new Predicate(_op,
                            Arrays.asList(in),
                            Arrays.asList(out)));   (2)
    }
    return ret;
  }
}
```

1. Each is parameterized with the predicate operation to use
2. the macro creates a predicate for each given input/output field pair

Let's look at another example of a dynamic predicate macro. We earlier defined the IncrementFunction as its own function that increments its argument, but in reality it is simply the Plus function with one argument set to "1". A useful macro would generate new predicates by abstracting away the partial application of a predicate operation. We could then define the Increment operation like this:

```
Object Increment = new Partial(new Plus(), 1);
```

As you can see, Partial is a predicate macro that fills in some of the input fields. It allows you to rewrite the query that increments the triplets as so:

```
new Subquery("?x", "?y", "?z")
  .predicate(TRIPLETS, "?a", "?b", "?c")
  .predicate(new Each(new Partial(new Plus(), 1)), "?a", "?b", "?c")
  .out("?x", "?y", "?z");
```

After expanding all the predicate macros, this query translates to:

```
new Subquery("?x", "?y", "?z")
  .predicate(TRIPLETS, "?a", "?b", "?c")
  .predicate(new Plus(), 1, "?a").out("?x")
  .predicate(new Plus(), 1, "?b").out("?y")
  .predicate(new Plus(), 1, "?c").out("?z");
```

The definition of Partial is straightforward:

```
public static class Partial implements PredicateMacro {
  Object _op;
  List<Object> _args;

  public Partial(Object op, Object... args) {
    _op = op;
    _args = Arrays.asList(args);
  }

  public List<Predicate> getPredicates(Fields inFields,
                                       Fields outFields) {
    List<Predicate> ret = new ArrayList<Predicate>();
    List<Object> input = new ArrayList<Object>();
    input.addAll(_args);
    input.addAll(inFields);
```

```
    ret.add(new Predicate(_op, input, outFields));
    return ret;
  }
}
```

The predicate macro simply prepends any provided input fields to the input fields specified when the subquery is created. As you can see, dynamic predicate macros gives you great power to manipulate the construction of your subqueries.

5.5 Conclusion

The way you express your computations is crucially important in order to avoid complexity, prevent bugs, and increase productivity. The main techniques for fighting complexity are abstraction and composition, and it's important that your data processing tool encourage these techniques rather than make them difficult.

In the next chapter, we will tie things together by showing how to use JCascalog along with the graph schema from Chapter 2, and the Pail from Chapter 3, to build out the batch layer for SuperWebAnalytics.com. These examples will be more sophisticated than what you saw in this chapter and show how the abstraction and composition techniques you saw in this chapter apply towards a realistic use case.

6 Batch layer: Tying it all together

This chapter covers:

- Building a batch layer end to end
- Ingesting new data into the master dataset
- Practical examples of precomputation
- Using a Thrift-based schema, Pail, and JCascalog together

In the last few chapters you've learned all the pieces of the batch layer: formulating a schema for your data, storing a master dataset, and running computations on your data at scale with a minimum of complexity. In this chapter you'll see how to tie all these pieces together into a coherent batch layer.

There's no new theory espoused in this chapter. Our goal is to reinforce the concepts of the prior chapters by going through an end to end example of a batch layer. There's value in seeing how the theory maps to the nitty gritty details of a non-trivial example.

You will see how to create the batch layer for our running example SuperWebAnalytics.com. SuperWebAnalytics.com is complex enough so that you'll be able to follow along with the creation of a fairly sophisticated batch layer, but it's not too complex as to lose you in the details. As you'll see, the various batch layer abstractions you've seen throughout the book fit together nicely and the batch layer for SuperWebAnalytics.com will be quite elegant.

After reviewing the product requirements for SuperWebAnalytics.com, we will give a broad overview of what the batch layer for it must accomplish and what should be precomputed for each batch view. Then you'll see how to implement each portion of the batch layer using the Thrift schema, Pail, and JCascalog. The batch views generated will be unindexed – that final piece of indexing the batch

views so that they can be read in a random-access manner will be covered in the next chapter about the serving layer.

6.1 SuperWebAnalytics.com batch layer overview

We will be building out the batch layer for SuperWebAnalytics.com to support the computation of three different queries. Recall from Chapter 4 that the goal of the batch layer is precompute views such that the queries can be satisfied with low latency. Also recall that there's a balance to strike between the size of the view generated and the amount of on-the-fly computation that will be necessary at query-time. Let's look at the queries SuperWebAnalytics.com will support and then determine the batch views that are necessary to support those queries.

6.1.1 Queries

SuperWebAnalytics.com will support three different kinds of queries:

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

One important aspect of the data schema that makes the second two queries more challenging (and more interesting) is the way people are modeled. The schema developed in Chapter 2 models people in one of two ways: as the user id of a logged in user or via a cookie identifier on the browser. A person may visit the same site under different identifiers, their cookie may change if they clear the cookie, and a person may even sign up with multiple user ids. The schema handles this by allowing for "Equiv" edges to be written which indicate that two different user ids are actually the same person. The "Equiv graph" for a person can be arbitrarily complex, as shown in Figure 6.1. So to accurately compute the second two queries, you need to do analysis on the data to determine which pageviews belong to the same person but exist under different identifiers.

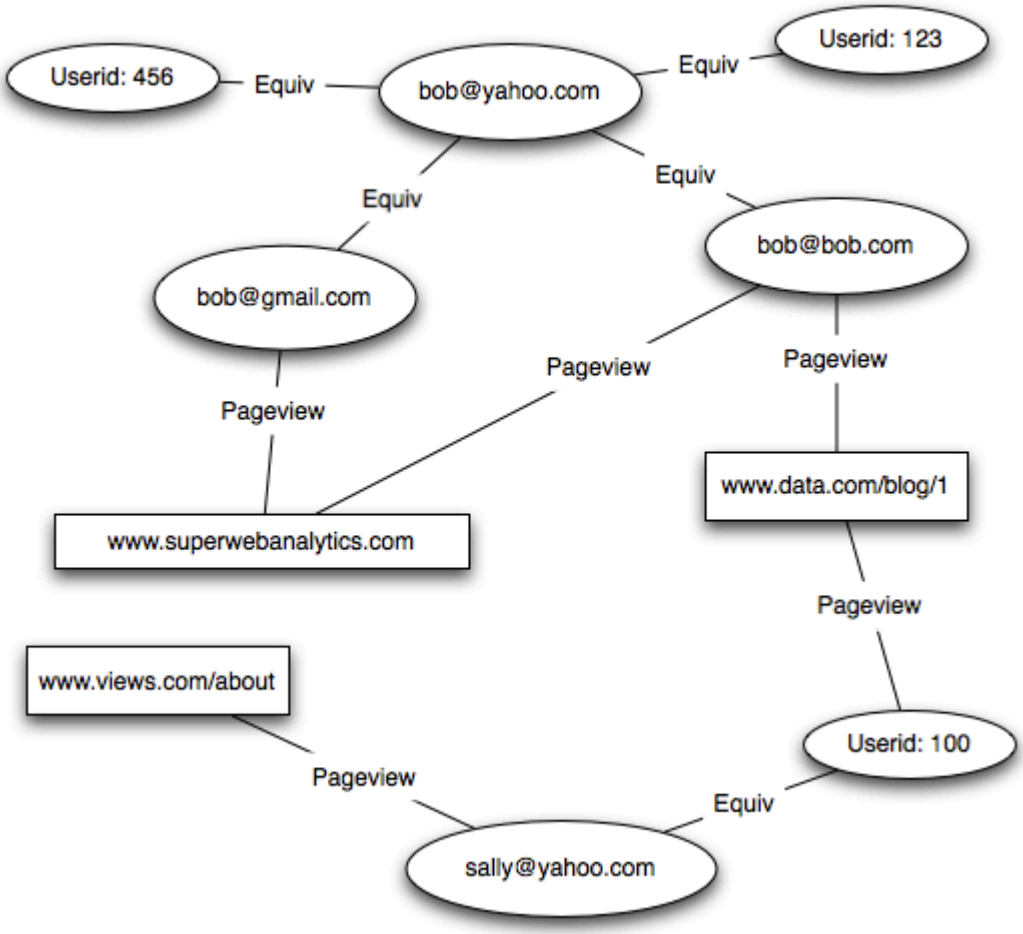


Figure 6.1 Examples of different pageviews for same person being captured for different user ids

6.1.2 Batch views

Let's now go over the batch views needed to satisfy each query. The key to each batch view is striking a balance between how much of the query is precomputed and how much on-the-fly computation will be required at query time.

PAGEVIEWS OVER TIME

We wish to be able to retrieve the number of pageviews for a URL for any range of time to hour granularity. This is the same query as discussed in Chapter 4. As discussed in Chapter 4, precomputing the value for every possible range of time is infeasible, as that would require 380 million precomputed values for every URL for one year of time, an unmanageable number. Instead, you can precompute a smaller amount and require more computation to be done at query-time.

The simplest approach is to precompute the number of pageviews for every URL and hour bucket. You would end up with a batch view that looks like Figure

6.2.

URL	Hour	# pageviews
foo.com/blog	1	876
foo.com/blog	2	987
foo.com/blog	3	762
foo.com/blog	4	413
foo.com/blog	5	1098
foo.com/blog	6	657
foo.com/blog	7	101

Figure 6.2 Precompute pageviews at hourly granularity

Then, to resolve a query, you retrieve the value for every hour bucket in the time range and sum the values together.

However, there is a problem with this approach. The query gets slower the larger the range of time. Finding the number of pageviews for a one year time period would require about 8760 values to be retrieved from the view and summed together. As many of those values are going to be served from disk, that can cause the latency of queries with larger ranges to be substantially higher than queries of small ranges.

Fortunately, the solution is simple. Instead of precomputing only at the hour granularity, you can also precompute at coarser granularities like day, seven day ("week") and twenty-eight day ("month") granularities. Let's see why this improves things by looking at an example.

Suppose you want to compute the number of pageviews from March 3rd at 3am through September 17th at 8am. If you only used hour granularity, this query would require retrieving and summing together the values for 4805 hour buckets. You can substantially reduce the number of values you need to retrieve by making use of the coarser granularities. The idea is that you can retrieve the values for each month between March 3rd and September 17th, and then add or subtract smaller granularities to get to the desired range. This idea is illustrated in Figure 6.3.

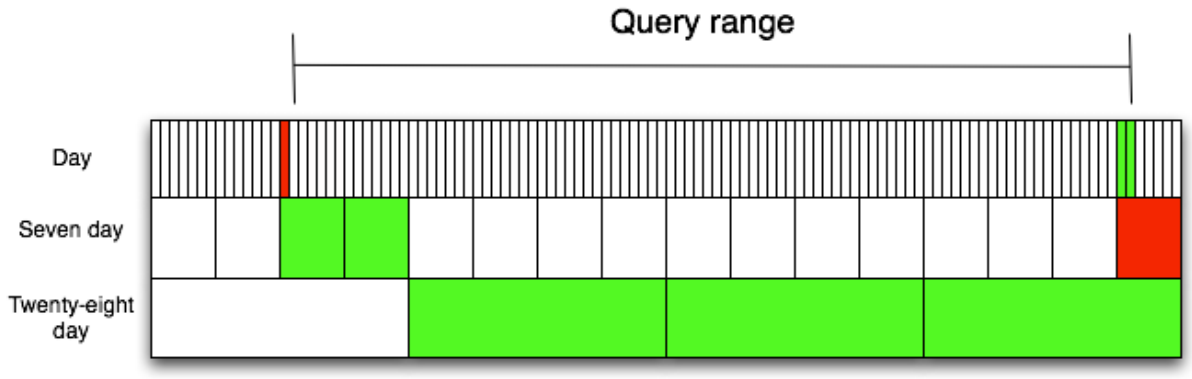


Figure 6.3 Optimizing pageviews over time query with coarser granularities. For this example range, green means to add that value into the result, red means to subtract that value.

For this query, only 26 values need to be retrieved, a much smaller number and almost a 200x improvement!

You might be wondering about how expensive it is to compute the day, seven day, and twenty-eight day granularities in addition to the hour granularity. The great thing is that there's hardly any cost to it at all! Let's look at the numbers for how many time buckets there are for each granularity in a one year period in Figure 6.4.

Granularity	Number buckets in one year
Hour	8760
Day	about 365
Seven day	about 52
Twenty-eight day	about 13

Figure 6.4 Number of buckets in a one year period for each granularity

Adding up the numbers, the day, seven day, and twenty-eight day granularities require an additional 430 values to be precomputed for every URL for a one year period. That's only a 5% increase in precomputation for a 200x improvement in the amount of work that needs to be done at query-time for large ranges. So the tradeoff is most certainly worth it.

UNIQUE VISITORS OVER TIME

The next query is unique visitors over time. Implementing unique visitors over time seems like it should be similar to pageviews over time, but there's one key difference. Whereas you can get the total number of pageviews for a two hour period by adding the number of pageviews for each individual hour together, you can't do the same for uniques. This is because a unique count represents the size of a *set* of elements, and there may be overlap between the sets for each hour. So you can't just add the counts for the two hours together, as that would double-count people who are incorporated into the count for both hours.

The only way to compute the number of uniques with perfect accuracy over any time range is to compute the unique count on the fly. This requires random access to the set of visitors for each URL for each hour time bucket. This is doable, but expensive, as essentially your entire master dataset needs to be indexed.

Alternatively, you can use an approximation algorithm that sacrifices some accuracy to vastly decrease the amount that needs to be indexed in the batch view. An example of an algorithm that can do this for distinct counting is the HyperLogLog algorithm. HyperLogLog only requires information on the order of one kilobyte to be indexed for every URL and hour bucket to estimate set cardinalities up to one billion with only a 2% error rate. [1] We don't wish to lose you in the details of HyperLogLog, so we will be using a library that implements the HyperLogLog algorithm. The library has an interface like the following:

Footnote 1 The HyperLogLog algorithm is described in the research paper at this link: http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf

```
interface HyperLogLog {
  long cardinality();
  void add(Object o);
  HyperLogLog merge(HyperLogLog... otherSets);
}
```

Each HyperLogLog object represents a set of elements and supports adding new elements to the set, merging with other HyperLogLog sets, and retrieving the size of the set.

Otherwise, going the HyperLogLog route makes the uniques over time query very similar to the pageviews over time query. The key differences are that a somewhat larger value is computed for each URL and time bucket rather than just

a simple count, and instead of summing counts together to get the total number of pageviews, the HyperLogLog merge function is used to combine time buckets to get the unique count. Like pageviews over time, we will compute HyperLogLog sets for seven day and twenty eight day granularities to reduce the amount of work that needs to be done at query time.

BOUNCE RATE ANALYSIS

The final query is determining the bounce rate for every domain. The batch view for this query is simple: just a map from domain to counts of the number of bounced visits and the total number of visits. The bounce rate is the number of bounced visits divided by the total number of visits.

The key to precomputing these values is defining what a "visit" is. We will define two pageviews as being part of the same visit if they are from the same user to the same domain and are separated by less than half an hour. A visit is a bounce if it only contains one pageview.

6.2 Workflow overview

Now that the specific requirements for the batch views are understood, let's determine what the batch workflow should be at a high level. The workflow is illustrated in Figure 6.5.

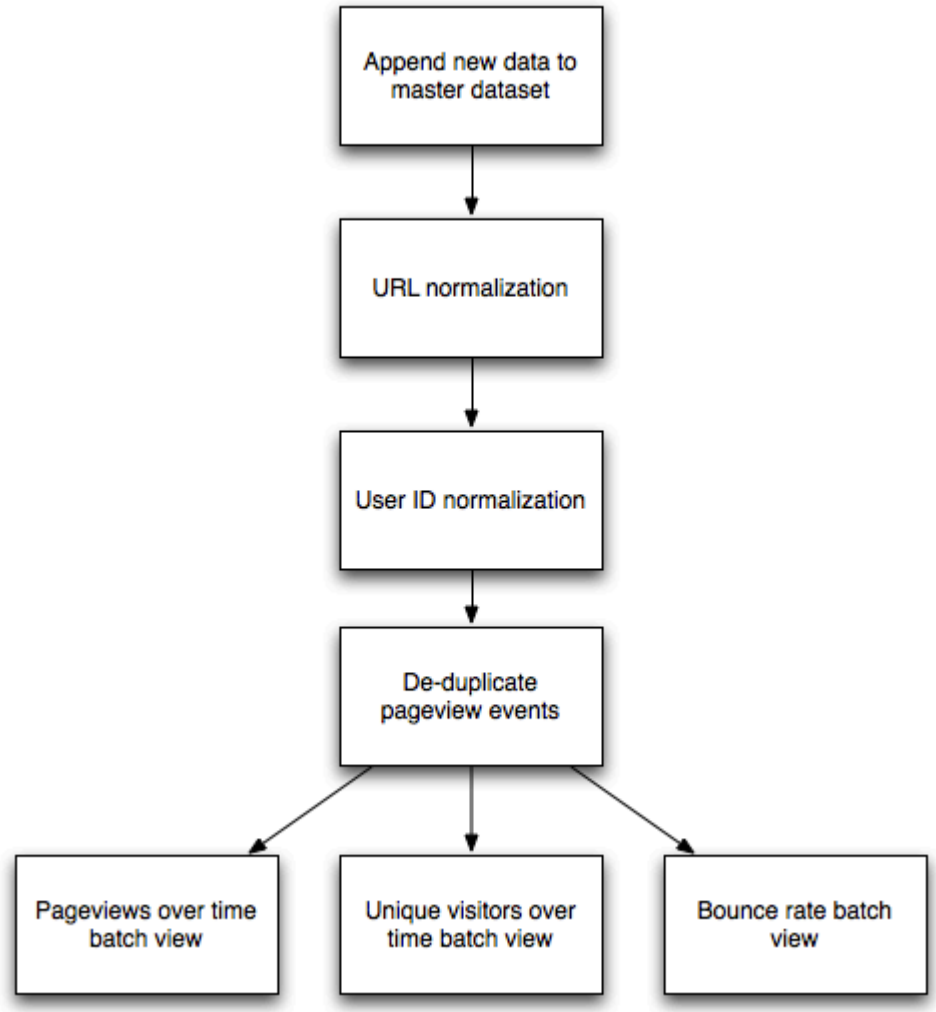


Figure 6.5 Batch workflow for SuperWebAnalytics.com

At the start of the batch layer is a single folder on the distributed filesystem that contains the master dataset. As it's very common for computations to only use a small subset of all the different properties and edges available, the master dataset is a Pail structured by the type of data. This means each property and edge is stored in a different subfolder. Let's say that the master dataset is stored at the path "/data/master".

A separate Pail at the location "/data/new" stores new data that hasn't been incorporated into the batch views yet. When someone wants to add data to the system, they create a new file in the "new data Pail" with the new data units. The first step of the workflow is to append the contents of the new data Pail into the master dataset Pail, and then delete from the new data Pail everything that was copied over.

The next two steps are normalization steps to prepare the data for computing the batch views. The first normalization step accounts for the fact that there can be

many variants of the same URL for the same web location. For example, the URLs "www.mysite.com/blog/1?utmcontent=1" and "http://mysite.com/blog/1" refer to the same web location. So the first normalization step normalizes all URLs to the same format so that it's future computations can aggregate the data.

The second normalization step accounts for the fact that data for the same person can exist under different user identifiers. Different user identifiers are marked as belonging to the same person using equiv edge dataunits. In order to run computations about "visits" and "visitors", a single user id needs to be selected for each person. The second normalization step will walk the equiv graph to accomplish this. As the batch views only make use of the pageviews data, only the pageviews data will be converted to use the selected user ids.

The next step de-duplicates the pageview events. Recall from Chapter 2 the advantages of data having the property of "identifiability", where a piece of data contains the information to uniquely identify the event. It's perfectly valid for the same pageview event to exist multiple times as outlined in Chapter 2. De-duplicating the pageviews makes it easier to compute the batch views, as you then have the constraint of having exactly one record for each pageview.

The final step is to run computations on the normalized data to compute the batch views described in the previous section.

Note that this workflow is a pure recomputation workflow. Every time new data is added, the batch views are recomputed from scratch. In a later chapter, you'll learn about how in many cases you can incrementalize the batch layer so that you don't always have to recompute from scratch. However, it's absolutely essential to have the pure recomputation workflow at hand, because you need the capability to recompute from scratch in case a mistake is made and the views get corrupted.

6.3 Preparing the workflow

A small preparation step is needed before we get to implementing the workflow itself. Many pieces of the workflow manipulate objects from the Thrift schema: such as Data objects, PageViewEdge's, PageID's, and PersonID's. Hadoop needs to know how to serialize and deserialize these objects during jobs so that it can get objects from one machine to another. Hadoop lets you register serializers via the config, and Hadoop will automatically figure out which serializer to use when it encounters an unfamiliar object (like one of the SuperWebAnalytics.com Thrift objects).

The cascading-thrift project has a serializer implementation for Thrift objects that you can make use of. Registering it is done as follows:

```
Map conf = new HashMap();
String sers = "backtype.hadoop.ThriftSerialization";
sers += ",";
sers += "org.apache.hadoop.io.serializer.WritableSerialization";
conf.put("io.serializations", sers);
Api.setApplicationConf(conf);
```

This code tells Hadoop to use both the default serializer (WritableSerialization) and the Thrift serializer (ThriftSerialization). The config is set globally and will be used by every job launched by the program executing the batch workflow.

6.4 Ingesting new data

Let's now see how to implement the first step of the workflow: getting new data into the master dataset Pail. The first problem to solve here is a synchronization one: you need to get the contents of the new data Pail into the master dataset Pail and then delete whatever you appended from the new data Pail. Suppose you did the following (leaving out the details on the actual append for the moment):

```
appendNewDataToMasterDataPail(masterPail, newDataPail);
newDataPail.clear();
```

There's a race condition in this code: more data will be written to the new data Pail as the append is running, so clearing the new data Pail after appending will cause that data written during the append job to be lost.

Fortunately, the solution is very simple. Pail provides methods "snapshot" and "deleteSnapshot" to solve this problem. "snapshot" makes a copy of the Pail in a new location, and "deleteSnapshot" deletes exactly what's in the snapshot Pail from the original Pail. So the following code is safe:

```
Pail snapshotPail = newDataPail.snapshot(
                      "/tmp/swa/newDataSnapshot");
appendNewDataToMasterDataPail(masterPail, snapshotPail);
newDataPail.deleteSnapshot(snapshotPail);
```

This code ensures that the only data removed from the new data Pail is data that was successfully appended to the master dataset Pail.

Also note that the code creates intermediate data as part of the workflow: this example creates a snapshot at the path "/tmp/swa/newDataSnapshot". The path

"/tmp/swa" will be used as a working space for intermediate data throughout the workflow. So at the very start of the workflow you should run the following code to ensure that working space is clean when the workflow begins:

```
FileSystem fs = FileSystem.get(new Configuration());
fs.delete(new Path("/tmp/swa"), true);
fs.mkdirs(new Path("/tmp/swa"));
```

The next problem to solve is the actual append of the new data into the master dataset. The new data Pail is unstructured: each file within the Pail can contain data units of all property types and edges. Before that data can be appended into the master dataset, it needs to be re-organized to be structured the same way the master dataset Pail is structured (by property or edge type). Reorganizing a Pail to have a new structure is called "shredding".

In order to shred, you need to be able to read and write from Pails via JCascalog queries. Recall that the abstraction for sinking and sourcing data from a JCascalog query is called a "Tap". The dfs-datastores-cascading project provides a tap implementation called "PailTap" that provides the integration you need to read and write from Pails via JCascalog.

The PailTap is easy to use. To create a tap reading all the data from a Pail, you do this:

```
Tap source = new PailTap("/tmp/swa/snapshot");
```

PailTap automatically detects the type of records being stored and deserializes them for you. So when used on a Pail storing objects from your Thrift schema, you will receive Thrift Data objects when reading from this tap. You can use this tap in JCascalog query like so:

```
new Subquery("?data")
        .predicate(source, "_", "?data");
```

The tap emits two fields into a query. The first field is the relative path within the pail where the record is stored. We'll never need this value for any of the examples in this book, so we can ignore that field. The second field is the deserialized record from the Pail.

PailTap also supports reading a subset of the data within the Pail. For Pails

using the SplitDataPailStructure from Chapter 3, you can construct a PailTap that reads only Equiv edges as follows:

```
PailTapOptions opts = new PailTapOptions();
opts.attrs = new List[] {
                new ArrayList<String>() {{
                   add("" + DataUnit._Fields
                                .EQUIV
                                .getThriftFieldId());
                }}
                };

Tap equivs = new PailTap("/tmp/swa/snapshot", opts);
```

Since we'll make use of this functionality quite a bit we'll wrap this code into a function as follows:

```
public static PailTap attributeTap(
        String path,
        final DataUnit._Fields... fields) {
    PailTapOptions opts = new PailTapOptions();
    opts.attrs = new List[] {
                    new ArrayList<String>() {{
                       for(DataUnit._Fields field: fields) {
                           add("" + field.getThriftFieldId());
                       }
                    }}
                    };

    return new PailTap(path, opts);
}
```

When sinking data from queries into brand new Pails, you need to make sure to set up the PailTap to know what kind of records you'll be writing to it. You do this by setting the *spec* field to contain the appropriate PailStructure. To create a Pail that shreds the data units by attribute, you can just use the SplitDataPailStructure from Chapter 3, like so:

```
public static PailTap splitDataTap(String path) {
    PailTapOptions opts = new PailTapOptions();
    opts.spec = new PailSpec(
                  (PailStructure) new SplitDataPailStructure());
    return new PailTap(path, opts);
}
```

Now let's see how to use PailTap and JCascalog to implement the shredding part of the workflow.

Your first attempt to shred might look something like this:

```
PailTap source = new PailTap("/tmp/swa/snapshot");
PailTap sink = splitDataTap("/tmp/swa/shredded");

Api.execute(sink,
        new Subquery("?data")
            .predicate(source, "_", "?data"));
```

Logically, this query is correct. However, when you try to run this on a large input dataset on Hadoop, you'll find that you get strange errors at runtime. You'll see things like NameNode errors and file handle issues.

What you've run into are limitations in Hadoop itself. As discussed in Chapter 4, Hadoop does not play well with lots of small files. And the problem with this query is that it creates an enormous amount of small files.

To see why, you have to understand how this query executes. Because there is no aggregation or joining in this query, it executes as a map-only job. Normally this would be great, as the reduce step is far more expensive than the map step. In this case, you run into problems because there are many more mappers than reducers.

The map step of a MapReduce job scales the number of map tasks based on the size of the input. Generally there will be one map task per block of data (usually 128MB). Since this is a map-only job, the map tasks are responsible for emitting their output. Since this job is shredding the input data into a file for each type of data, and since the different types of data are mixed together in the input files, each map task will create a number of files equal to the total number of properties and edges. If your schema has 100 different properties and edges, and your input data is 2.5 terabytes, then the total number of output files will be about one million. Hadoop can't handle that many small files.

You can solve this problem by artificially introducing a reduce step into the computation. Unlike mappers, you can explicitly control the number of reducers via the job config. So if you ran the shredding job on 2.5 terabytes of data with 100 reducers, you would end up with 10000 files, a much more manageable number.

The code for forcing the query to use reducers looks like this:

```
PailTap source = new PailTap("/tmp/swa/snapshot");
```

```
PailTap sink = splitDataTap("/tmp/swa/shredded");
Subquery reduced = new Subquery("?rand", "?data")
    .predicate(source, "_", "?data-in")
    .predicate(new RandLong(), "?rand")
    .predicate(new IdentityBuffer(), "?data-in").out("?data");
Api.execute(
    sink,
    new Subquery("?data")
        .predicate(reduced, "_", "?data"));
```

This code assigns a random number to each data record, and then uses an identity aggregator to get each data record to the reducer. It then projects out the random number and executes the computation.

After the query finishes, you can reduce the number of files even further by consolidating the shredded pail, like so:

```
Pail shreddedPail = new Pail("/tmp/swa/shredded");
shreddedPail.consolidate();
```

Now that the data is shredded and the number of files has been minimized, you can finally append it into the master dataset Pail, like so:

```
masterPail.absorb(shreddedPail);
```

That ends the ingestion portion of the workflow.

6.5 URL Normalization

The next step of the workflow is to normalize all URLs in the master dataset. We will accomplish this by creating a copy of the master dataset in the scratch area that normalizes all URLs across all data objects.

The query for this is very simple. The query requires a custom function that implements the normalization logic. The function takes in a Data object and emits a normalized Data object. The code for the normalization function is shown below (this is a very rudimentary implementation of normalization intended just for demonstration purposes):

```
public static class NormalizeURL extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        Data data = ((Data) call.getArguments()
                      .getObject(0)).deepCopy();
        DataUnit du = data.get_dataunit();
```

```
        if(du.getSetField() == DataUnit._Fields.PAGE_VIEW) {
            normalize(du.get_page_view().get_page());
        } else if(du.getSetField() ==
                  DataUnit._Fields.PAGE_PROPERTY) {
            normalize(du.get_page_property().get_id());
        }
        call.getOutputCollector().add(new Tuple(data));
    }

    private void normalize(PageID page) {
        if(page.getSetField() == PageID._Fields.URL) {
            String urlStr = page.get_url();
            try {
                URL url = new URL(urlStr);
                page.set_url(url.getProtocol() + "://" +
                        url.getHost() + url.getPath());
            } catch(MalformedURLException e) {
            }
        }
    }

}
```

You can then use this function to implement the URL normalization portion of the query like so:

```
Tap masterDataset = new PailTap("/data/master");
Tap outTap = splitDataTap("/tmp/swa/normalized_urls");

Api.execute(outTap,
    new Subquery("?normalized")
        .predicate(masterDataset, "_", "?raw")
        .predicate(new NormalizeURL(), "?raw")
            .out("?normalized"));
```

That's all there is to URL normalization.

6.6 User ID normalization

The next step is selecting a single user ID for each person. This is the most sophisticated portion of the workflow as it involves a fully distributed iterative graph algorithm. Yet it will only require a little over a hundred lines of code to accomplish.

User IDs are marked as belonging to the same person via Equiv edges. If you were to visualize the equiv edges for a dataset, you would see something like Figure 6.6.

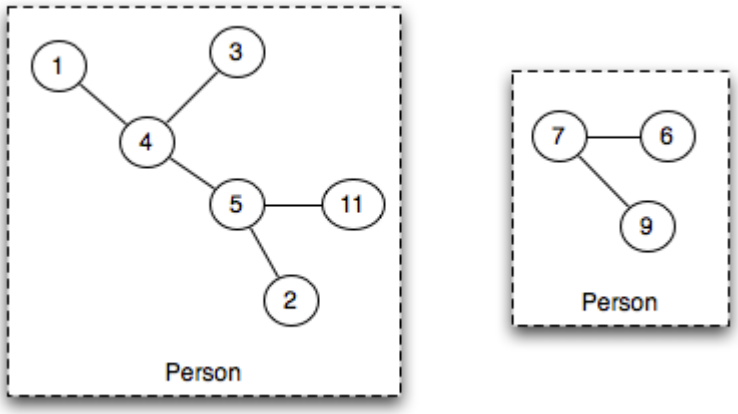


Figure 6.6 Example equiv graph

For each person you need to select a single user ID and produce a mapping from the original user IDs to the user ID selected for that person, as shown in Figure 6.7.

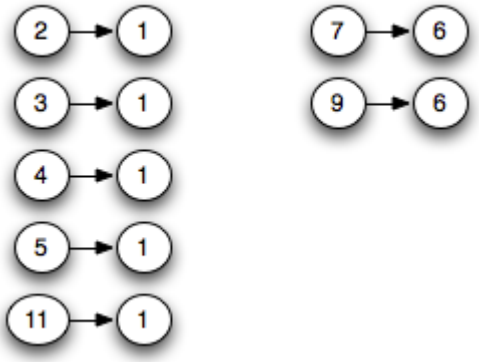


Figure 6.7 Mapping from user IDs to a single userID from each set

We will accomplish this by transforming the original equiv graph so that it is of the form in Figure 6.7. So our example would transform into something looking like Figure 6.8, where every user ID for a person points to a single user ID for that person.

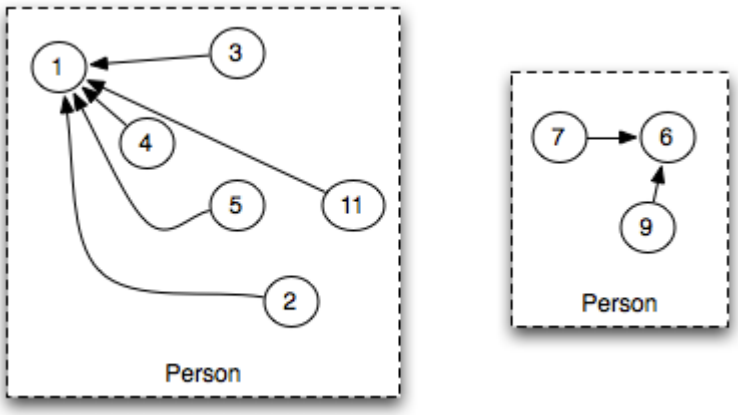


Figure 6.8 Original equiv graph transformed so that all nodes in a set point to a single node

Now this idea must be mapped into a concrete algorithm that can run scalably on top of MapReduce. In all the MapReduce computations you've seen so far, only a single query had to be executed to get the output. For this algorithm, it's impossible to get the desired results in just a single query. Instead, you can take an iterative approach where each query transforms the graph to get closer to the desired structure in Figure 6.8. After enough steps, you'll get the desired results.

Once you have an algorithm that transforms the graph to be closer to the desired result, you then run it over and over until no more progress is made. This is called reaching a "fixed point", where the output of an iteration is the same as the input. When this point is reached, then you know that the graph is in the desired structure.

In each step, each node will look at all the nodes connected directly to it. It will then move the edges to all point to the smallest node among the nodes it's connected to. How this works for a single node is illustrated in Figure 6.9.

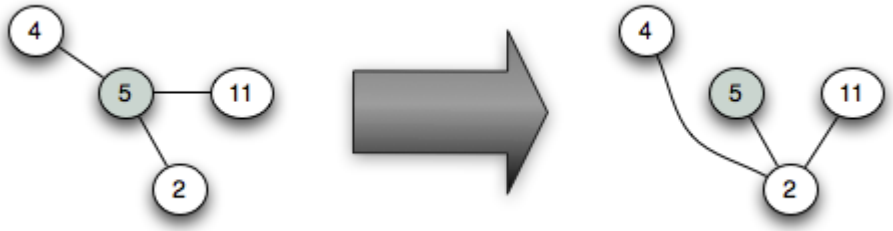


Figure 6.9 How the edges around a single node move in a single iteration

Thrift provides a natural ordering for all Thrift structures, so you can make use of that to order the PersonID's.

Let's see how this algorithm works on the equiv graph from Figure 6.6. Figure

6.10 shows how the graph gets transformed until it reaches fixed point.

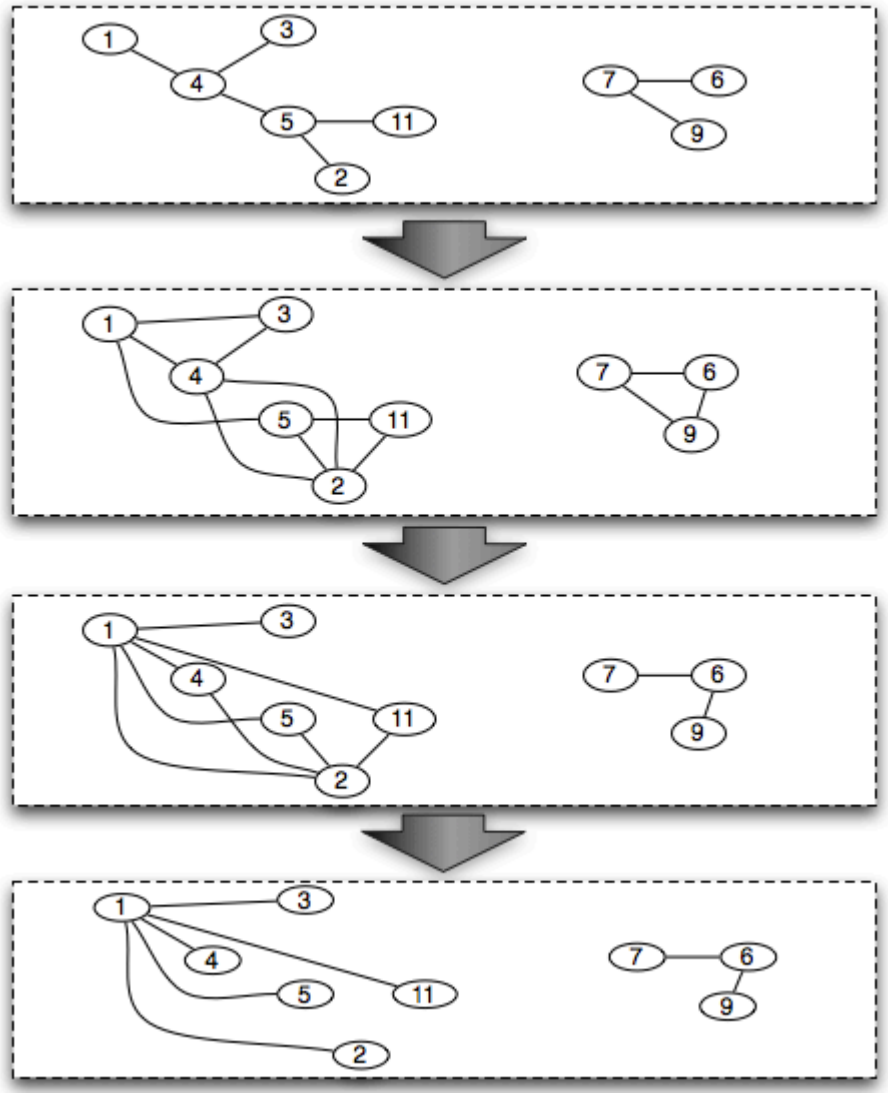


Figure 6.10 Running algorithm until fixed point

Now let's translate this to actual code. Each iteration will be stored in a new output folder on the distributed filesystem, using the template "/tmp/swa/equivs{iteration number}" for the path. The output of each iteration will just be 2-tuples, where the first field is the smaller of the two PersonID's.

The first step is to transform the Thrift equiv Data objects into 2-tuples. Here's a custom function that does the translation:

```
public static class EdgifyEquiv extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        Data data = (Data) call.getArguments().getObject(0);
        EquivEdge equiv = data.get_dataunit().get_equiv();
        call.getOutputCollector().add(
                new Tuple(equiv.get_id1(), equiv.get_id2()));
```

```
    }
}
```

And here's the query that uses that function to create the first set of 2-tuples:

```
Tap equivs = attributeTap("/tmp/swa/normalized_urls",
                          DataUnit._Fields.EQUIV);
Api.execute(Api.hfsSeqfile("/tmp/swa/equivs0"),
        new Subquery("?node1", "?node2")
            .predicate(equivs, "_", "?data")
            .predicate(new EdgifyEquiv(),
                       "?node1", "?node2"));
```

Now let's look at the portion of the code that implements a single iteration of the algorithm. This code does two things. It emits the new set of edges, as discussed, and it also marks which edges are new or which are the same as the last iteration. When a node is the smallest of all nodes surrounding it, then the emitted edges are the same. Otherwise, some of the emitted edges are new. Here's the code for the query portion:

```
Tap source = (Tap) Api.hfsSeqfile(
            "/tmp/swa/equivs" + (i - 1));
Tap sink = (Tap) Api.hfsSeqfile("/tmp/swa/equivs" + i);
Subquery iteration = new Subquery(
        "?b1", "?node1", "?node2", "?is-new")
        .predicate(source, "?n1", "?n2")
        .predicate(new BidirectionalEdge(), "?n1", "?n2")
            .out("?b1", "?b2")
        .predicate(new IterateEdges(), "?b2")
            .out("?node1", "?node2", "?is-new");
iteration = (Subquery) Api.selectFields(iteration,
        new Fields("?node1", "?node2", "?is-new"));
Subquery newEdgeSet = new Subquery("?node1", "?node2")
        .predicate(iteration, "?node1", "?node2", "_")
        .predicate(Option.DISTINCT, true);
```

There are two subqueries defined here. "iteration" contains the resulting edges from doing one step of the algorithm and contains the marker for which edges are new. "newEdgeSet" projects away that flag from "iteration" and uniques the tuples to produce the result for the next iteration.

The bulk of the logic is in producing the "iteration" subquery. It has to do two things. First, for each node, it has to get all the nodes connected to it together into a single function. Then it has to emit the new edges. In order to accomplish the first

part, the subquery groups the stream by one of the nodes in the edges. Before it does that, it emits every edge in both directions, so that the edge [123, 456] will exist as both [123, 456] and [456, 123] in the set of tuples. This ensures that when the grouping happens by one of the fields that every node connected to that node is brought into the function. The "BidirectionalEdge" custom function accomplishes this:

```
public static class BidirectionalEdge extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        Object node1 = call.getArguments().getObject(0);
        Object node2 = call.getArguments().getObject(1);
        if(!node1.equals(node2)) {
            call.getOutputCollector().add(
                new Tuple(node1, node2));
            call.getOutputCollector().add(
                new Tuple(node2, node1));
        }
    }
}
```

Finally, the "IterateEdges" function implements the logic that emits the new edges for the next iteration. It also marks edges as new appropriately:

```
public static class IterateEdges extends CascalogBuffer {
    public void operate(FlowProcess process, BufferCall call) {
        PersonID grouped = (PersonID) call.getGroup()
                                         .getObject(0);
        TreeSet<PersonID> allIds = new TreeSet<PersonID>();
        allIds.add(grouped);
        Iterator<TupleEntry> it = call.getArgumentsIterator();
        while(it.hasNext()) {
            allIds.add((PersonID) it.next().getObject(0));
        }

        Iterator<PersonID> allIdsIt = allIds.iterator();
        PersonID smallest = allIdsIt.next();
        boolean isProgress = allIds.size() > 2 &&
                             !grouped.equals(smallest);
        while(allIdsIt.hasNext()) {
            PersonID id = allIdsIt.next();
            call.getOutputCollector().add(
                    new Tuple(smallest, id, isProgress));
        }
    }
}
```

The next part of the algorithm needs to know if there were new edges so as to

determine when a fixed point has been reached. This is a straightforward subquery to find all edges whose "is new" flag was set to true:

```
Tap progressEdgesSink = (Tap) Api.hfsSeqfile(
        "/tmp/swa/equivs" + i + "-new");
Subquery progressEdges = new Subquery("?node1", "?node2")
        .predicate(iteration, "?node1", "?node2", true);
```

The rest of the algorithm involves running the iterations in a loop until fixed point is reached. This is accomplished by the following code:

```
int i = 1;
while(true) {
  Tap progressEdgesSink = runUserIdNormalizationIteration(i);

  if(!new HadoopFlowProcess(new JobConf())
          .openTapForRead(progressEdgesSink)
          .hasNext()) {
      break;
  }
  i++;
}
```

The "openTapForRead" function used there is an easy way to get access to the tuples in a Tap via regular Java code. As you can see there, it checks to see whether there are any tuples in that Tap. If there is, then at least one new edge was created, so fixed point has not been reached yet. Otherwise, fixed point has been reached and it can stop doing iterations.

The last thing to do to complete this workflow step is to change the PersonID's in the pageview data to use the PersonID's that have been selected. Since it's perfectly valid for a PersonID to not exist in any equiv edges, meaning it was never found to belong to the same person as any other PersonID, any PersonID's in the pageview data that are not mapped to a different PersonID should just remain the same.

This last step implements this transformation by doing a join from the pageview data to the final iteration of edges. It looks like this:

```
Tap pageviews = attributeTap("/tmp/swa/normalized_urls",
        DataUnit._Fields.PAGE_VIEW);
Tap newIds = (Tap) Api.hfsSeqfile("/tmp/swa/equivs" + i);
Tap result = splitDataTap(
                "/tmp/swa/normalized_pageview_users");
```

```
Api.execute(result,
        new Subquery("?normalized-pageview")
            .predicate(newIds, "!!newId", "?person")
            .predicate(pageviews, "_", "?data")
            .predicate(new ExtractPageViewFields(),
                       "_", "?person", "_")
            .predicate(new MakeNormalizedPageview(),
                "!!newId", "?data")
                .out("?normalized-pageview"));
```

Notice the usage of the "!!newId" variable to do an outer join. That variable will be null if "?person" didn't join against anything from the edge set.

There are two custom functions used here. The first, "ExtractPageViewFields", extracts the URL, PersonID, and timestamp for every pageview. It's defined to be more general purpose than needed for this algorithm because we'll make use of it later. Here's the definition of the function:

```
public static class ExtractPageViewFields
    extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        Data data = (Data) call.getArguments().getObject(0);
        PageViewEdge pageview = data.get_dataunit()
                                    .get_page_view();
        if(pageview.get_page().getSetField() ==
           PageID._Fields.URL) {
            call.getOutputCollector().add(new Tuple(
                    pageview.get_page().get_url(),
                    pageview.get_person(),
                    data.get_pedigree().get_true_as_of_secs()
                    ));
        }
    }
}
```

Finally, the "MakeNormalizedPageview" function takes in a pageview Data object and the new PersonID, and it emits a pageview Data object with an updated PersonID. Here's the definition:

```
public static class MakeNormalizedPageview
    extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        PersonID newId = (PersonID) call.getArguments()
                                        .getObject(0);
        Data data = ((Data) call.getArguments().getObject(1))
                                                .deepCopy();
        if(newId!=null) {
            data.get_dataunit().get_page_view().set_person(newId);
```

```
        }
        call.getOutputCollector().add(new Tuple(data));
    }
}
```

If the new PersonID is null, then it didn't join against the edge set and the PersonID should remain as is.

That concludes the user ID normalization portion of the workflow. This part of the workflow is a great example of why it's so useful to have the tool for specifying the MapReduce computations just be a library for your general purpose programming language. A lot of the logic, such as using a while loop and checking for fixed point, were just done as normal Java code.

6.7 De-duplicate pageviews

The final preparation step prior to computing the batch views is de-duplicating the pageview events. This is a trivial query to write:

```
Tap source = attributeTap(
                "/tmp/swa/normalized_pageview_users",
                DataUnit._Fields.PAGE_VIEW);
Tap outTap = splitDataTap("/tmp/swa/unique_pageviews");

Api.execute(outTap,
      new Subquery("?data")
          .predicate(source, "?data")
          .predicate(Option.DISTINCT, true));
```

Since this computation only operates over pageviews, the source tap selects only pageviews to read from the input.

6.8 Computing batch views

The data is now ready for the computation of the batch views. The computation of each of the batch views is a completely independent query, and we'll go through them one by one.

The outputs of these batch views will just be flat files. In the next chapter, you'll learn about how to index the batch views so that they can be queried in a random access manner.

6.8.1 Pageviews over time

As outlined in the beginning of the chapter, the pageviews over time batch view should aggregate the pageviews for each URL at hourly, daily, seven-day, and twenty-eight day granularities.

The approach we'll take is to first roll up the pageviews at an hourly granularity. This has the effect of vastly reducing the size of the data, likely by many orders of magnitude (since thousands of pageviews – or more – exist in a single hour). From there, we'll roll up the hourly granularity into the rest of the granularities. The latter roll up will be much faster than the first roll up, due to the relative size of the inputs.

Let's start with creating a subquery that rolls up the pageviews to an hourly granularity. You'll need a function that transforms a timestamp into an hour bucket for this query, which is defined like so:

```
public static class ToHourBucket extends CascalogFunction {
    private static final int HOUR_SECS = 60 * 60;

    public void operate(FlowProcess process, FunctionCall call) {
        int timestamp = call.getArguments().getInteger(0);
        int hourBucket = timestamp / HOUR_SECS;
        call.getOutputCollector().add(new Tuple(hourBucket));
    }
}
```

Here's the query that does the roll up using those custom functions:

```
Tap source = new PailTap("/tmp/swa/unique_pageviews");

Subquery hourlyRollup = new Subquery(
    "?url", "?hour-bucket", "?count")
    .predicate(source, "?pageview")
    .predicate(new ExtractPageViewFields(), "?pageview")
        .out("?url", "_", "?timestamp")
    .predicate(new ToHourBucket(), "?timestamp")
        .out("?hour-bucket")
    .predicate(new Count(), "?count");
```

That's all there is to it: this is a very straightforward query.

The next subquery rolls up the hourly roll-ups into all the granularities needed to complete the batch view. You'll need another custom function to transform an hour bucket into buckets for all the granularities. This custom function emits two fields: the first is one of the strings "h", "d", "w", or "m" indicating hourly, daily, weekly, or monthly granularity, and the second is the numerical value of the time bucket. Here's the custom function:

```
public static class EmitGranularities extends CascalogFunction {
```

```
    public void operate(FlowProcess process, FunctionCall call) {
        int hourBucket = call.getArguments().getInteger(0);
        int dayBucket = hourBucket / 24;
        int weekBucket = dayBucket / 7;
        int monthBucket = dayBucket / 28;

        call.getOutputCollector().add(new Tuple("h", hourBucket));
        call.getOutputCollector().add(new Tuple("d", dayBucket));
        call.getOutputCollector().add(new Tuple("w", weekBucket));
        call.getOutputCollector().add(new Tuple("m",
                                                monthBucket));
    }
}
```

Then computing the rollups for all the granularities is just a simple sum:

```
new Subquery(
    "?url", "?granularity", "?bucket", "?total-pageviews")
    .predicate(hourlyRollup, "?url", "?hour-bucket", "?count")
    .predicate(new EmitGranularities(), "?hour-bucket")
        .out("?granularity", "?bucket")
    .predicate(new Sum(), "?count").out("?total-pageviews");
```

That's it! You're done with the pageviews over time batch view.

6.8.2 Unique visitors over time

The batch view for unique visitors over time contains a HyperLogLog set for every time granularity tracked for every URL. It is essentially the same computation as done to compute pageviews over time, except instead of aggregating counts you aggregate HyperLogLog sets.

You'll need two new custom operations to do this query. The first is an aggregator that constructs a HyperLogLog set from a sequence of user ids:

```
public static class ConstructHyperLogLog extends CascalogBuffer {
    public void operate(FlowProcess process, BufferCall call) {
        HyperLogLog hll = new HyperLogLog(8000);
        Iterator<TupleEntry> it = call.getArgumentsIterator();
        while(it.hasNext()) {
            TupleEntry tuple = it.next();
            hll.offer(tuple.getObject(0));
        }
        try {
            call.getOutputCollector().add(
                new Tuple(hll.getBytes()));
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
```

```
    }
}
```

The next one is another custom aggregator that is used to combine the HyperLogLog sets for hourly granularities into HyperLogLog sets for coarser granularities:

```
public static class MergeHyperLogLog extends CascalogBuffer {
    public void operate(FlowProcess process, BufferCall call) {
        Iterator<TupleEntry> it = call.getArgumentsIterator();
        HyperLogLog curr = null;
        try {
            while(it.hasNext()) {
                TupleEntry tuple = it.next();
                byte[] serialized = (byte[]) tuple.getObject(0);
                HyperLogLog hll = HyperLogLog.Builder.build(
                                        serialized);
                if(curr==null) {
                    curr = hll;
                } else {
                    curr = (HyperLogLog) curr.merge(hll);
                }
            }
            call.getOutputCollector().add(
                new Tuple(curr.getBytes()));
        } catch (IOException e) {
            throw new RuntimeException(e);
        } catch(CardinalityMergeException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Here's how you use these operations to compute the batch view. Note the similarity to the pageviews over time query.

```
public static void uniquesView() {
    Tap source = new PailTap("/tmp/swa/unique_pageviews");

    Subquery hourlyRollup =
            new Subquery("?url", "?hour-bucket", "?hyper-log-log")
                .predicate(source, "?pageview")
                .predicate(
                    new ExtractPageViewFields(), "?pageview")
                    .out("?url", "?user", "?timestamp")
                .predicate(new ToHourBucket(), "?timestamp")
                    .out("?hour-bucket")
                .predicate(new ConstructHyperLogLog(), "?user")
                    .out("?hyper-log-log");
    new Subquery(
```

```
        "?url", "?granularity", "?bucket", "?aggregate-hll")
        .predicate(hourlyRollup,
                   "?url", "?hour-bucket", "?hourly-hll")
        .predicate(new EmitGranularities(), "?hour-bucket")
            .out("?granularity", "?bucket")
        .predicate(new MergeHyperLogLog(), "?hourly-hll")
            .out("?aggregate-hll");
}
```

It's possible to abstract away the common parts between this query and the pageviews over time query into its own function. We'll leave that as an exercise for the reader.

SIDEBAR **Optimizing the HyperLogLog batch view further**

The implementation we've shown uses the same size for every HyperLogLog set: 1000 bytes. The HyperLogLog set needs to be that large in order to get a reasonably accurate answer for URLs which may receive millions or hundreds of millions of visits. However, most websites using SuperWebAnalytics.com won't get nearly that many pageviews, so its wasteful to use such a large HyperLogLog set size for them.

To optimize the batch views further, you could look at the total pageview count for URLs on that domain and tune the size of the HyperLogLog set accordingly. Using this approach you can vastly decrease the space needed for the batch view, at the cost of some complexity in the view generation code.

6.8.3 Bounce rate analysis

The final batch view computes the bounce rate for each URL. As outlined in the beginning of the chapter, we'll compute two values for each domain: the total number of visits, and the number of bounced visits.

The key part of this query is tracing the visits that each person made as they browsed the internet. An easy way to do this is to look at all the pageviews a person made for a particular domain in order of which they made the pageview. While you walk through the pageviews, you look at the time difference between the pageviews to determine if they are part of the same visit or not. If a visit contains only one pageview, it counts as a bounced visit.

To do this in a JCascalog query, you need two custom operations. The first extracts a domain from a URL:

```
public static class ExtractDomain extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        String urlStr = call.getArguments().getString(0);
        try {
            URL url = new URL(urlStr);
            call.getOutputCollector().add(
                new Tuple(url.getAuthority()));
        } catch(MalformedURLException e) {
        }
    }
}
```

The next is a custom aggregator that iterates through a sorted list of pageviews and counts the number of visits and the number of bounces for that user on that domain. This aggregator looks like the following:

```
public static class AnalyzeVisits extends CascalogBuffer {
    private static final int VISIT_LENGTH_SECS = 60 * 15;

    public void operate(FlowProcess process, BufferCall call) {
        Iterator<TupleEntry> it = call.getArgumentsIterator();
        int bounces = 0;
        int visits = 0;
        Integer lastTime = null;
        int numInCurrVisit = 0;
        while(it.hasNext()) {
            TupleEntry tuple = it.next();
            int timeSecs = tuple.getInteger(0);
            if(lastTime == null ||
                    (timeSecs - lastTime) > VISIT_LENGTH_SECS) {
                visits++;
                if(numInCurrVisit == 1) {
                    bounces++;
                }
                numInCurrVisit = 0;
            }
            numInCurrVisit++;
        }
        if(numInCurrVisit==1) {
            bounces++;
        }
        call.getOutputCollector().add(new Tuple(visits, bounces));
    }
}
```

Combining these functions you can then compute the number of visits and bounces for each user on each domain:

```
Tap source = new PailTap("/tmp/swa/unique_pageviews");
```

```
Subquery userVisits =
        new Subquery("?domain", "?user",
                     "?num-user-visits", "?num-user-bounces")
            .predicate(source, "?pageview")
            .predicate(
                new ExtractPageViewFields(), "?pageview")
                .out("?url", "?user", "?timestamp")
            .predicate(new ExtractDomain(), "?url")
                .out("?domain")
            .predicate(Option.SORT, "?timestamp")
            .predicate(new AnalyzeVisits(), "?timestamp")
                .out("?num-user-visits", "?num-user-bounces");
```

Finally, to compute the number of visits and bounces in aggregate for each domain, you simply sum together the user visits information:

```
new Subquery("?domain", "?num-visits", "?num-bounces")
    .predicate(userVisits, "?domain", "_",
               "?num-user-visits", "?num-user-bounces")
    .predicate(new Sum(), "?num-user-visits")
        .out("?num-visits")
    .predicate(new Sum(), "?num-user-bounces")
        .out("?num-bounces");
```

That's it! That completes the recomputation-based batch layer for SuperWebAnalytics.com.

6.9 Conclusion

The batch layer for SuperWebAnalytics.com is just a few hundred lines of code, yet the business logic involved is quite sophisticated. The various abstractions used fit together well – there was a fairly direct mapping from what we wanted to accomplish at each step and how we accomplished it. Here and there, hairy details popped up due to the nature of the toolset – notably Hadoop's small files problem – but these weren't hard to overcome.

As we've indicated a few times, what you saw developed in this chapter is a recomputation-based workflow, where the batch views are always recomputed from scratch. There's a large class of problems for which you can incrementalize the batch layer and make it much more resource-efficient: you'll see how to do this in a later chapter.

You should now have a good feel for how flexible the batch layer is. It's really easy to extend the batch layer to compute new views: each stage of the workflow is free to run an arbitrary function on all the data. This means the batch layer is inherently prepared to adapt to changing customer and application requirements.

7 Serving layer

This chapter covers:

- Tailoring views to the queries they will serve
- The normalization vs. denormalization problem
- Advantages of batch-writable, random-read, no random-write databases
- ElephantDB as an example of a serving layer database

You've learned so far how to precompute arbitrary views for any dataset by making use of batch computation. However, in order for those views to be useful, you need to be able to query them with low latency. That's where the serving layer comes in. The serving layer indexes the views and provides interfaces so that the views can be queried quickly. The serving layer is tightly tied to the batch layer, as the batch layer is responsible for providing the serving layer with updated views.

In the Lambda Architecture, the serving layer is the last piece of the batch portion of the architecture. The views produced by the batch layer and served by the serving layer will always be out of date due to the high latency nature of batch computation. That's fine though since the speed layer will compensate for any data not accounted for in the serving layer.

The serving layer is an area where the tooling has not caught up with the theory. It wouldn't be hard to build a general purpose serving layer implementation – in fact, it would be many orders of magnitude easier than building any of the existing NoSQL databases – just no one has done it yet. We will present the full theory behind making a simple, scalable, fault-tolerant, and general purpose serving layer and then use the best tooling available to demonstrate the concepts of the serving layer.

In the roadmap for learning about the serving layer, you will

- learn the requirements for the serving layer as dictated by the Lambda Architecture
- learn about choosing indexing strategies in order to minimize latency, resource usage, and variance
- see how the serving layer solves the age-old normalization vs. denormalization problem
- learn a simple architecture for a serving layer database
- learn a practical implementation of the serving layer in ElephantDB

Let's start by seeing the key issues you face when choosing how to structure a serving layer view.

7.1 Key issues in the serving layer

Like the batch layer, the serving layer is distributed among many machines for scalability. The serving layer indexes can be created, loaded, and served in a fully distributed fashion. There are two main performance metrics you must consider when choosing how you structure your serving layer indexes: throughput and latency. Let's look at an example to see the key issues you face when structuring the serving layer and how those affect the key metrics.

Consider the example of pageviews over time that we've looked at a few times before. The goal is to be able to get the number of pageviews for each hour in a particular range of hours. Let's look at a slightly simpler example where the only granularity produced is an hour granularity. The view produced by the batch layer would look something like Figure 7.1.

URL	Bucket	Pageviews
foo.com/blog/1	0	10
foo.com/blog/1	1	21
foo.com/blog/1	2	7
foo.com/blog/1	3	38
foo.com/blog/1	4	29
bar.com/post/a	0	178
bar.com/post/a	1	91
bar.com/post/a	2	568

Figure 7.1 Pageviews over time batch view with only hour granularity

One way to index this view would be to use a key/value indexing strategy by considering the pair of [URL, hour] as the key and the number of pageviews as the value. The indexes would then be partitioned by the key, so different hours for the same URL would be on different partitions. Different partitions can exist on different servers, so getting a range of hours for a single URL would require fetching values from many servers in your serving layer.

While this design works in principle, it will face serious issues with both latency and throughput. Let's start with latency. Because the values for a particular URL are spread all around your cluster, you will need to query many servers to get a range of hours. While you can parallelize the fetches, the query will only be as fast as the slowest server. The problem is that there will be variance among the response times of servers on your cluster. Perhaps one server is slightly more loaded than the other servers at that moment or perhaps it's in the middle of garbage collection. Suppose the query needs to hit 3 servers. The distribution of latencies on the servers may look something like Figure 7.2.

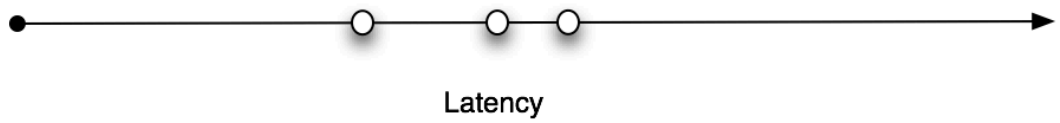


Figure 7.2 Latency distribution of three servers

Now suppose the query needs to hit 20 servers. Then the distribution of latencies may look something like Figure 7.3.

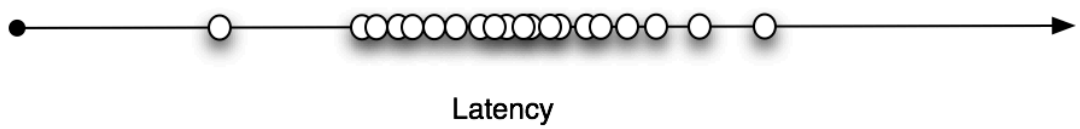


Figure 7.3 Latency distribution of twenty servers

The more servers a query needs to touch, then the higher latency the query will be because the worst server will likely have even higher latency. Variance turns the worst case performance of one server into the common case performance of queries. And the more servers you have serving a view, the worse the latency gets. This is a serious problem for achieving good latency for the pageviews over time query.

Another problem with the key/value strategy for the pageviews over time query is poor throughput, particularly if you're using disks and not SSDs. As this view represents the majority of the data you have, it is quite large. So queries will be I/O bound. The dominant cost of retrieving the value for a single key will be disk seeks. Since you have to do a disk seek per key, and since a single query may be fetching dozens of values or more, every query requires many disk seeks. There's a limit to the number of disk seeks you can do per second since your cluster is comprised of a finite number of disks. Suppose the average number of keys fetched per query is 20, the number of disks in the cluster is 100, and the average number of disk seeks possible per second per disk is 500. Then your cluster can only handle 2500 queries per second, which should seem quite low given that number of disks.

Let's take a look at a different indexing strategy that has much better latency

and throughput characteristics. The idea is to colocate all the pageview information for a single URL on the same partition and to store it sequentially. Then fetching the pageviews will require a single seek and a scan rather than a seek per time bucket. Scans are ultra cheap relative to seeks, so this is far more resource efficient. Additionally, since only a single server needs to be contacted per query, you are no longer subject to the variance issues of the previous query. The layout of the index for this strategy might look something like Figure 7.4.

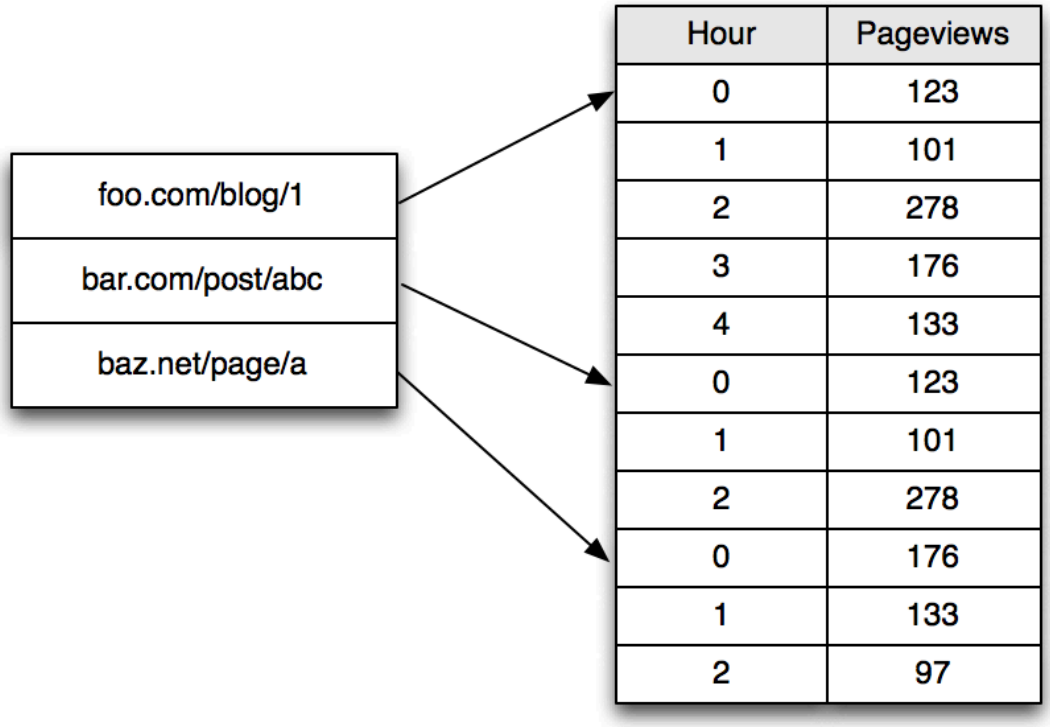


Figure 7.4 Sorted index

As you can see, the way you structure your serving layer indexes has drastic effects on the performance of your queries. The key is that the Lambda Architecture allows you to tailor the serving layer for the queries they serve and achieve optimal results.

7.2 How the serving layer solves the normalization vs. denormalization problem

The serving layer solves one of the biggest long-standing problems in the relational database world: the normalization vs. denormalization problem. Let's understand this problem via a simple example.

Suppose you are storing location information on people in a relational database with tables that look like Figure 7.5. Locations are keyed by an identifier and each person uses one of those identifiers to indicate their location. In order to get the specific location information for someone, like city, a join must be done between the person table and the location table. This is an example of a fully normalized schema as no information is stored redundantly.

ID	Name	Location ID
1	Sally	3
2	George	1
3	Bob	3

Location ID	City	State	Population
1	New York	NY	8.2M
2	San Diego	CA	1.3M
3	Chicago	IL	2.7M

Figure 7.5 Normalized schema

Suppose getting a person's city and state is an extremely common operation in your application. Joins aren't cheap, and suppose you decide that you need better performance from that operation. The only way to get rid of the join would be to redundantly store the city and state information in the person table so you don't have to do the join anymore for that operation. The schema in this case would look like Figure 7.6. This technique of redundantly storing information to avoid joins is called "denormalization" and is very common when using relational databases.

ID	Name	Location ID	City	State
1	Sally	3	Chicago	IL
2	George	1	New York	NY
3	Bob	3	Chicago	IL

Location ID	City	State	Population
1	New York	NY	8.2M
2	San Diego	CA	1.3M
3	Chicago	IL	2.7M

Figure 7.6 Denormalized schema

Clearly, denormalization is not an ideal solution. Now it's up to you, the application developer, to make sure all the copies of the location information stay consistent. This raises all sorts of uncomfortable questions like: what happens if the different copies of a city field become inconsistent? What are the semantics of your data in this case? And remember, mistakes are inevitable in long-lived systems so given enough time the fields will become inconsistent.

The normalization vs. denormalization problem is a case of unacceptable tradeoffs. You want to store your data fully normalized, but you are unable to for performance reasons. So instead you have to take on a huge complexity burden yourself.

Fortunately, in the Lambda Architecture, the split between the master dataset and the serving layer solves the normalization vs. denormalization problem. You can normalize your master dataset to your heart's content since the master dataset is read in bulk and you don't need to design the schema to optimize for reads. The serving layer is completely tailored to the queries they serve, so you can do whatever's necessary there to get maximal performance. Moreso, wthe optimizations you can do in the serving layer go far beyond just denormalization.

Besides pre-joining data, you can also do do all the aggregation and transformation needed to maximize performance. Additionally, you have the flexibility to index the data in the optimal way for your queries.

Finally, there's the question of consistency in the Lambda Architecture between the master dataset and the serving layer. It's absolutely true that information will be redundantly stored between the two. But the key is that the serving layer is defined as a function(master dataset) so it's easy to reason about. And if a mistake is made that leads to inconsistency, you can recompute the serving layer from scratch to regain consistency.

7.3 Requirements of serving layer database

The Lambda Architecture dictates a certain set of requirements on a serving layer database. However, even more interesting than the requirements are what serving layer databases don't require. Let's start with the requirements.

- Batch writable: Indexes for a serving layer viewed are produced from scratch. When a new version of the view is available, it must be possible to swap out the old version of the view for the new version.
- Scalable: A serving layer database must be scalable so that it can handle views of arbitrary size. Like the distributed filesystems and MapReduce computation framework we've already discussed, this means it must run distributed across many machines.
- Random reads: A serving layer database must support random reads where indexes are used to quickly get access to a portion of the view. This requirement is necessary to have low latency on queries.
- Fault-tolerant: Since a serving layer database is distributed, it must be tolerant to machines failing.

These requirements should be intuitive. The one requirement missing from this list – something that is standard on all databases you're accustomed to – is "random writes". A serving layer database does not require the ability to modify small portions of the view. This is completely irrelevant to the function of serving layer databases in the Lambda Architecture, where the views are produced in bulk.

This is an amazing result since random writes cause most of the complexity in databases – and even more complexity in distributed databases. To understand this complexity consider one particularly nasty detail of how random-write databases work: the use of write-ahead logs and compaction. Since modifying a disk index for every operation would be too expensive for the operation of a database, databases instead wait until many operations have been done on them so that the disk index can be modified in bulk. The pending operations are written to a sequential write ahead log so that if the database crashes it can replay those events

on startup. The process of applying events from the write ahead log to the on disk index is called "compaction".

Compaction is a fairly intensive operation. During compaction, the server will experience substantially higher load on the CPU and disks. This can lead to very variable performance. Databases like HBase and Cassandra are notorious for requiring careful configuration to avoid problems or server lockups during compaction.

Compaction is just one of the many complexities taken on by a database when it must support random writes. Another piece of complexity is the need for basic synchronization between reads and writes so that half-written values are never read. When a database doesn't have random writes, it can optimize the read path and get better performance than a random read/write database.

Another good indicator of the complexity difference between a database that allows random writes and one that doesn't is lines of code. ElephantDB, a database built specifically to be a serving layer database, is only a few thousand lines of code. HBase and Cassandra, two popular distributed read/write database, are hundreds of thousands of lines of code. Lines of code isn't normally a good complexity metric, but in this case the staggering difference should be telling.

A database that's less complex is more predictable since it fundamentally does fewer things. It's less likely to have bugs and will be easier to operate. Since the serving layer views represent the overwhelming majority of your data, the fact that the serving layer databases are far simpler is a great property to have for an architecture.

To be clear, random writes do exist in the Lambda Architecture, but they are isolated in the speed layer. Random writes are used there to achieve low latency updates. Since that's not a concern in the serving layer, random writes are not required.

7.4 Example of serving layer database: ElephantDB

Let's look at an example of a database built specifically to be used as a serving layer database, ElephantDB. We'll review the basic architecture of ElephantDB and how it meets the requirements outlined in the previous section. Afterwards we'll review the API of using ElephantDB.

ElephantDB is a key/value serving layer database where both keys and values are byte arrays. The main operation it supports is "multiGet" where the values are retrieved for any set of keys.

An ElephantDB index is partitioned into a fixed number of shards. A key is

assigned to a shard using a pluggable "ShardingScheme" that is a function from key to the shard that key should be on. One common sharding scheme is to choose the target shard by hashing the key and modding it by the number of shards. This is a common technique that ensures an even distribution of keys among shards while allowing readers to know which shard has the value for a key solely based on the key itself. However, later on you'll see cases where we'll want to use a different sharding scheme.

Each shard of a view is a local key/value index that by default uses BerkeleyDB. The actual storage engine is configurable and can be any single machine key/value indexing engine.

There are two aspects to ElephantDB: view creation and view serving. View creation occurs in a MapReduce job at the end of the batch layer, and the generated partitions are stored in the distributed filesystem. Views are served by a dedicated ElephantDB cluster that knows how to load new shards and talk to clients who are performing random read operations.

7.4.1 View creation

The input to the MapReduce job that creates ElephantDB view shards is a set of key/value pairs. The number of reducers is set to the number of ElephantDB shards, and the keys are partitioned to reducers using the sharding scheme. This has the effect of every reducer being responsible for producing exactly one shard of an ElephantDB view. Each shard is then indexed (e.g. into a BerkeleyDB index) and uploaded into the distributed filesystem.

Note that when shards are produced they are not sent directly to the ElephantDB servers that will be serving them. That would be a bad design because then the servers that are serving live traffic wouldn't be able to control their own load. Instead, the ElephantDB servers pull the shards from the DFS at a throttled rate that allows them to maintain their performance guarantees to clients.

7.4.2 View serving

An ElephantDB cluster is comprised of some number of servers that shares the work of serving the shards. Each server serves a subset of the shards. For example, if there are 64 shards and 8 ElephantDB servers, each ElephantDB server will serve 8 shards.

ElephantDB also supports replication, where each shard is replicated across some number of server nodes. For example, with 64 shards, 8 servers, and a replication factor of 3, each server will serve 24 shards and each shard will exist

across 3 different servers. Replication makes the cluster tolerant to losing machines, allowing full access to the entire view even when machines are lost. Of course, only so many machines can be lost before portions of the view will become unavailable, but replication makes this event far less likely. Replication is illustrated in Figure 7.7.

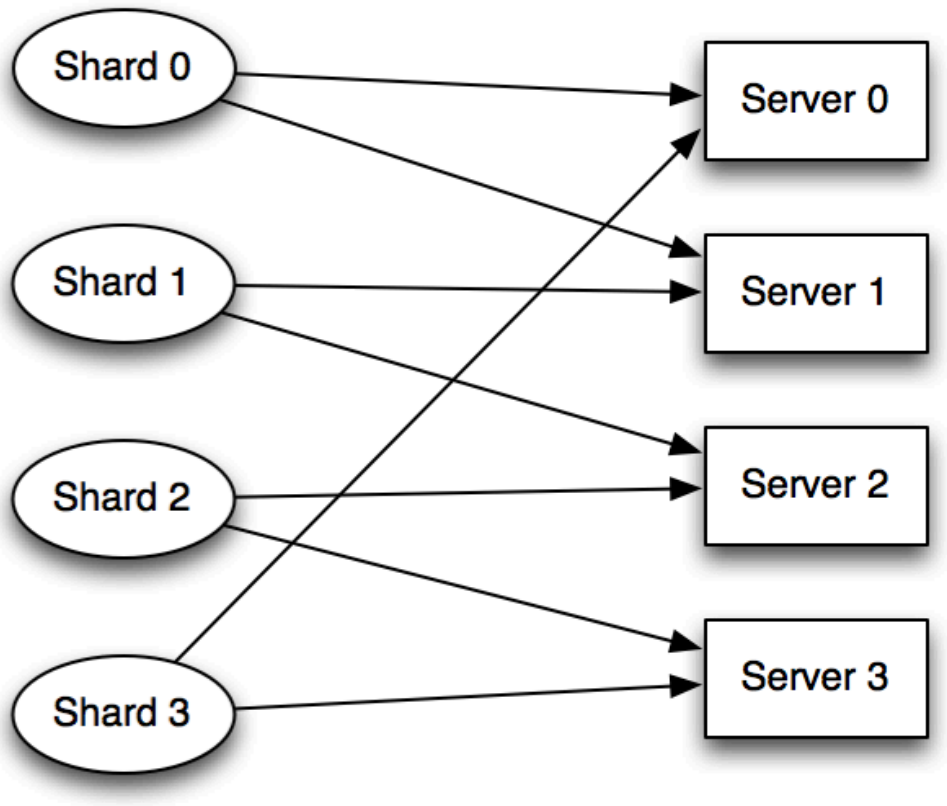


Figure 7.7 Replication

When an ElephantDB server detects a new version of a shard on the distributed filesystem, it does a throttled download of the new partition. The download is throttled so as not to saturate the I/O of the machine since it's serving live reads. It then swaps that new partition with the old one and deletes the old one.

7.4.3 Using ElephantDB

Using ElephantDB is straightforward. There are three parts to cover: creating ElephantDB shards, setting up an ElephantDB cluster that can serve requests, and using a client to query an ElephantDB cluster.

CREATING ELEPHANTDB SHARDS

Let's start with creating a set of ElephantDB shards using JCascalog. All you need to do is create an ElephantDB tap and then it will take care of everything else: the MapReduce job to correctly partition the keys, creating each index, and uploading each index to the distributed filesystem. Here's how you create an ElephantDB tap that uses BerkeleyDB as the storage engine, has 32 shards, and uses hash mod partitioning:

```
EDB.makeKeyValTap("/path/on/dfs/to/put/output",
          new DomainSpec(new JavaBerkDB(),
          new HashModScheme(),
          32));
```

Then, if you have a JCascalog subquery that contains key/value pairs, creating the ElephantDB view is as simple as executing that subquery into the tap:

```
Api.execute(tap, subquery);
```

SETTING UP AN ELEPHANTDB CLUSTER

Setting up an ElephantDB cluster is also straightforward. There are two configurations for an ElephantDB server: the local configuration which specifies server-specific properties and how to talk to the rest of the cluster, and the global configuration which lists information that's needed by every server in the cluster.

A basic local configuration looks like this:

```
{:local-root "/data/elephantdb"
 :hdfs-conf {"fs.default.name"
               "hdfs://namenode.yourcompany.com:8020"}
 :blob-conf {"fs.default.name"
               "hdfs://namenode.yourcompany.com:8020"}}
```

This lists, in order of configs, where to store downloaded shards locally, the address of the distributed filesystem which stores the shards, and the address of the distributed filesystem which stores the global configuration.

A basic global configuration looks like this:

```
{ :replication 2
  :hosts ["edb1.mycompany.com"
            "edb2.mycompany.com"
            "edb3.mycompany.com"]
  :port 3578
  :domains {"bounce_rate" "/data/output/bounce_rate"
            "pv_over_time" "/data/output/pageviews_over_time"
            "uniques_over_time" "/data/output/uniques_over_time"
            }
}
```

This lists the replication factor for each view, the port each server should take requests off of, all the hostnames of all ElephantDB servers in the cluster, and finally all the views served by this cluster. A single ElephantDB cluster can serve many domains and the config is a map from domain name to the path on HDFS where the shards for that domain are stored and updated through MapReduce.

QUERYING AN ELEPHANTDB CLUSTER

ElephantDB exposes a simple Thrift API for issuing queries. After connecting to an ElephantDB server, you can issue queries like so:

```
client.get("nameOfDomain", key);
```

The server will communicate with other servers in the cluster for any keys that are not stored locally.

7.5 Building serving layer for SuperWebAnalytics.com

Let's now see how to create optimized ElephantDB views for each query in SuperWebAnalytics.com. Let's begin with the pageviews over time view. At the end of Chapter 6 we produced a view that looks like Figure 7.8.

URL	Granularity	Bucket	Pageviews
foo.com/blog/1	h	0	10
foo.com/blog/1	h	1	21
foo.com/blog/1	h	2	7
foo.com/blog/1	w	0	38
foo.com/blog/1	m	0	38
bar.com/post/a	h	0	213
bar.com/post/a	h	1	178
bar.com/post/a	h	2	568

Figure 7.8 Pageviews over time batch view

In order to make this suitable for ElephantDB, which requires byte array keys and values, we'll encode the url, granularity, and time bucket into the key. The value will just be the number of pageviews. Here's a JCascalog subquery that transforms the results from the batch layer into key/value pairs appropriate for ElephantDB:

```
Subquery toEdb =
    new Subquery("?key", "?value")
        .predicate(pageviewBatchView,
           "?url", "?granularity", "?bucket", "?total-pageviews")
        .predicate(new ToUrlBucketedKey(),
            "?url", "?granularity", "?bucket")
            .out("?key")
        .predicate(new ToSerializedLong(), "?total-pageviews")
            .out("?value");
```

Here are the corresponding functions to do the serialization:

```
public static class ToUrlBucketedKey
    extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        String url = call.getArguments().getString(0);
        String gran = call.getArguments().getString(1);
        Integer bucket = call.getArguments().getInteger(2);
        String keyStr = url + "/" + gran + "-" + bucket;
        try {
            call.getOutputCollector().add(
                new Tuple(keyStr.getBytes("UTF-8")));
```

```
        } catch(UnsupportedEncodingException e) {
            throw new RuntimeException(e);
        }
    }
}
public static class ToSerializedLong
    extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        long val = call.getArguments().getLong(0);
        ByteBuffer buffer = ByteBuffer.allocate(8);
        buffer.putLong(val);
        call.getOutputCollector().add(
            new Tuple(buffer.array()));
    }
}
```

Next is creating the ElephantDB tap. In order to avoid the variance problem discussed at the beginning of the chapter, we'll create a custom ShardingScheme to ensure that all the key/value pairs for a single URL are on a single shard. To do this, we'll shard by doing a hash/mod on only the URL portion of each key, like so:

```
private static String getUrlFromSerializedKey(byte[] ser) {
    try {
        String key = new String(ser, "UTF-8");
        return key.substring(0, key.lastIndexOf("/"));
    } catch(UnsupportedEncodingException e) {
        throw new RuntimeException(e);
    }
}
public static class UrlOnlyScheme implements ShardingScheme {
    public int shardIndex(byte[] shardKey, int shardCount) {
        String url = getUrlFromSerializedKey(shardKey);
        return url.hashCode() % shardCount;
    }
}
```

Then, creating the tap and producing the shards is as easy as:

```
Api.execute(EDB.makeKeyValTap(
                "/outputs/edb/pageviews",
                new DomainSpec(new JavaBerkDB(),
                               new UrlOnlyScheme(),
                               32)),
            toEdb);
```

The pageviews over time view would have benefited from a more advanced serving layer database that explicitly stored all the time buckets in order on disk so

as to avoid disk seeks, as discussed in the beginning of the chapter. Such a serving layer database does not exist at the time of this writing, though such a database would not be hard to create.

The next query is the unique pageviews over time query. The code for this is almost identical to pageviews over time, except the HyperLogLog sets are already byte arrays so don't require serialization. This view uses the same sharding scheme as the pageviews over time view in order to avoid the variance problem. Here's the code that produces the ElephantDB shards for unique pageviews over time:

```
Subquery toEdb =
    new Subquery("?key", "?value")
        .predicate(uniquesView,
           "?url", "?granularity", "?bucket", "?value")
        .predicate(new ToUrlBucketedKey(),
            "?url", "?granularity", "?bucket")
            .out("?key");
Api.execute(EDB.makeKeyValTap(
                "/outputs/edb/uniques",
                new DomainSpec(new JavaBerkDB(),
                               new UrlOnlyScheme(),
                               32)),
            toEdb);
```

Finally, the last view is a map from domain to number of visits and number of bounces. All that's needed in this case is serialization code that produces a compound value containing the number of visits and number of bounces. Since queries fetch one domain at a time, there are no variance issues to worry about and normal hash/mod sharding can be used. Here's the code that produces the final view:

```
public static class ToSerializedString
    extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        String str = call.getArguments().getString(0);

        try {
            call.getOutputCollector().add(
                new Tuple(str.getBytes("UTF-8")));
        } catch(UnsupportedEncodingException e) {
            throw new RuntimeException(e);
        }
    }
}
public static class ToSerializedLongPair
    extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
```

```
        long l1 = call.getArguments().getLong(0);
        long l2 = call.getArguments().getLong(1);
        ByteBuffer buffer = ByteBuffer.allocate(16);
        buffer.putLong(l1);
        buffer.putLong(l2);
        call.getOutputCollector().add(new Tuple(buffer.array()));
    }
}
public static void bounceRateElephantDB(Subquery bounceView) {
    Subquery toEdb =
        new Subquery("?key", "?value")
            .predicate(bounceView,
               "?domain", "?bounces", "?total")
            .predicate(new ToSerializedString(),
                "?domain").out("?key")
            .predicate(new ToSerializedLongPair(),
                "?bounces", "?total").out("?value");
    Api.execute(EDB.makeKeyValTap(
                    "/outputs/edb/bounces",
                    new DomainSpec(new JavaBerkDB(),
                                   new HashModScheme(),
                                   32)),
                toEdb);
}
```

As you can see, integrating into the serving layer is almost no work at all.

7.6 Conclusion

You saw in this chapter the fundamental concepts of the serving layer and an example of a serving layer database in ElephantDB. ElephantDB is by no means a completely generic serving layer database as it only supports a key/value indexing model. With a different or more generic serving layer database, the serving layer for SuperWebAnalytics.com could be optimized even further. It wouldn't be hard to build such a serving layer database – or a library to make it easy to make customized serving layer databases – just no one has done it yet.

Now that you understand the batch and the serving layers, next up is learning the final piece of the Lambda Architecture: the speed layer. The speed layer will compensate for the serving layer's high latency of updates and allow queries to be up to date with current data.