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 7

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.

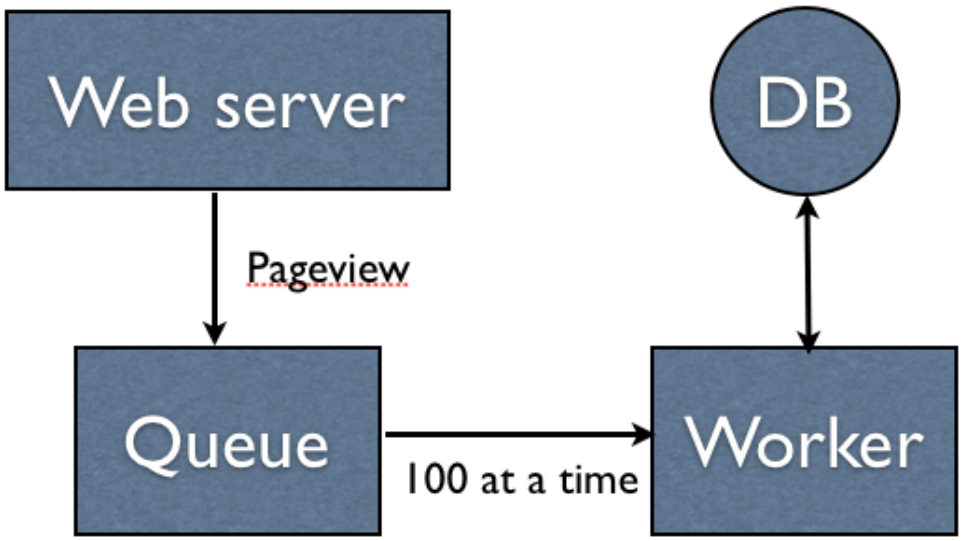


Figure 1.2 Batching updates with queue and worker

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.

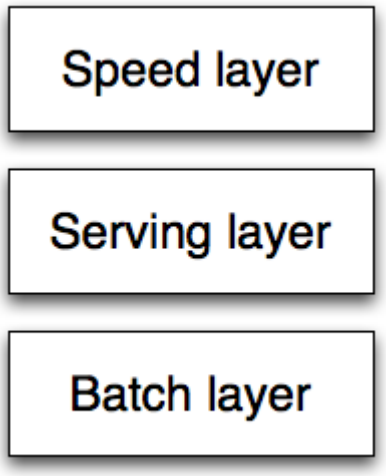


Figure 1.4 Lambda Architecture

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:

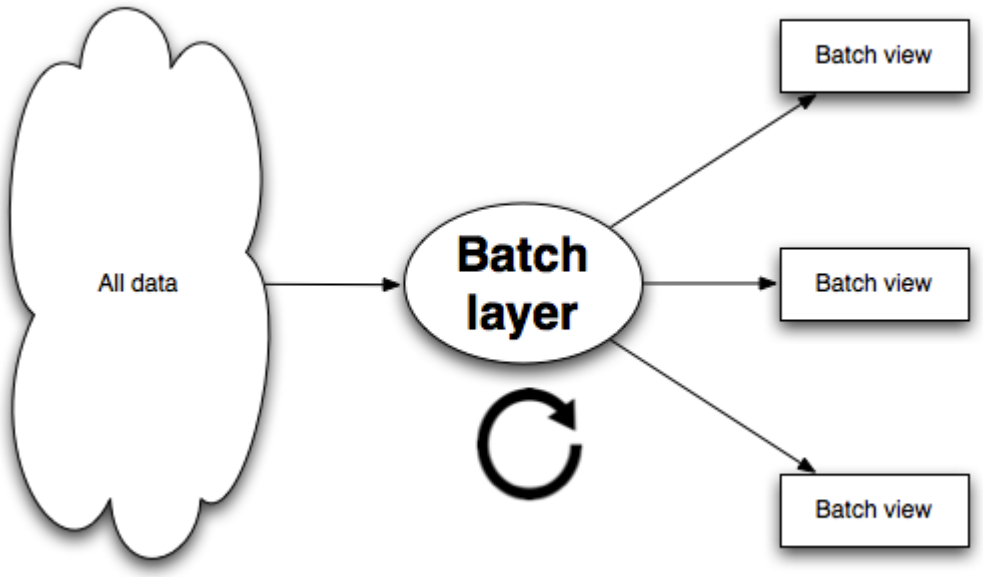


Figure 1.5 Batch layer

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

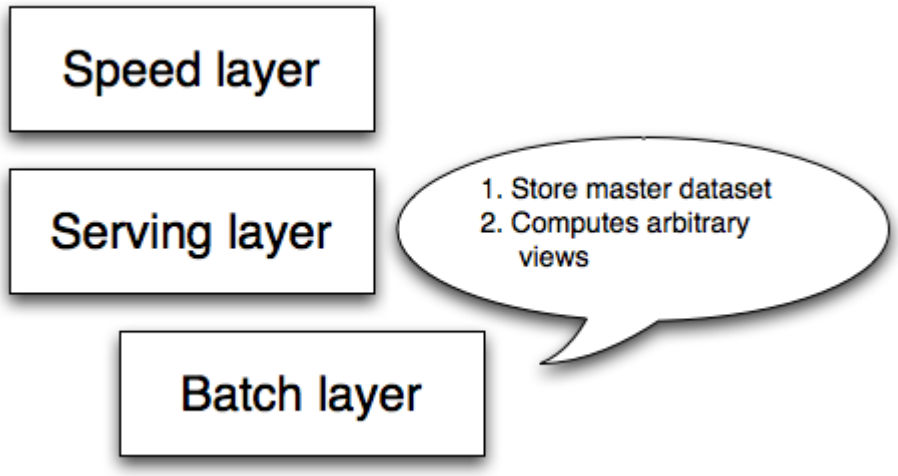


Figure 1.6 Batch layer

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.

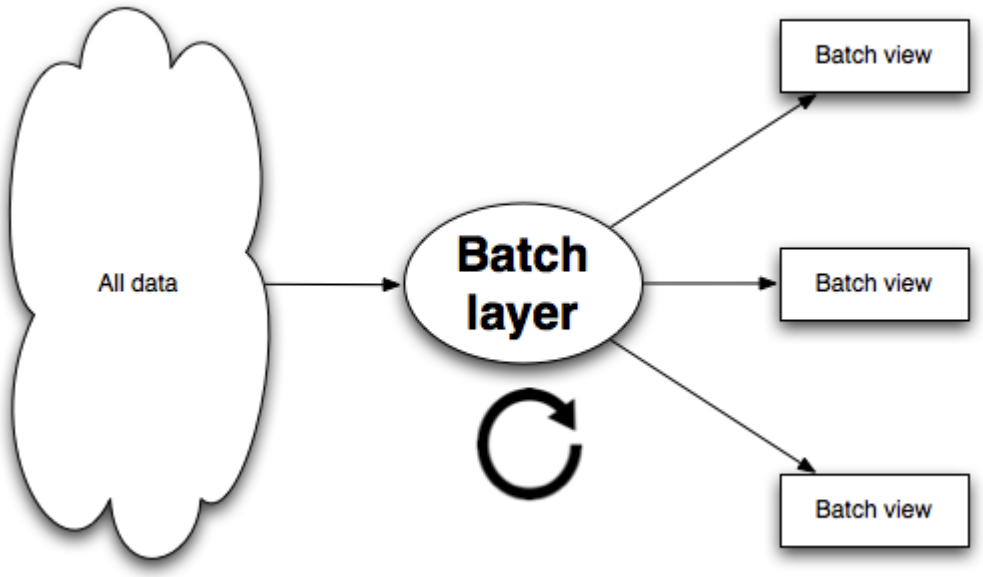


Figure 1.7 Batch layer

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

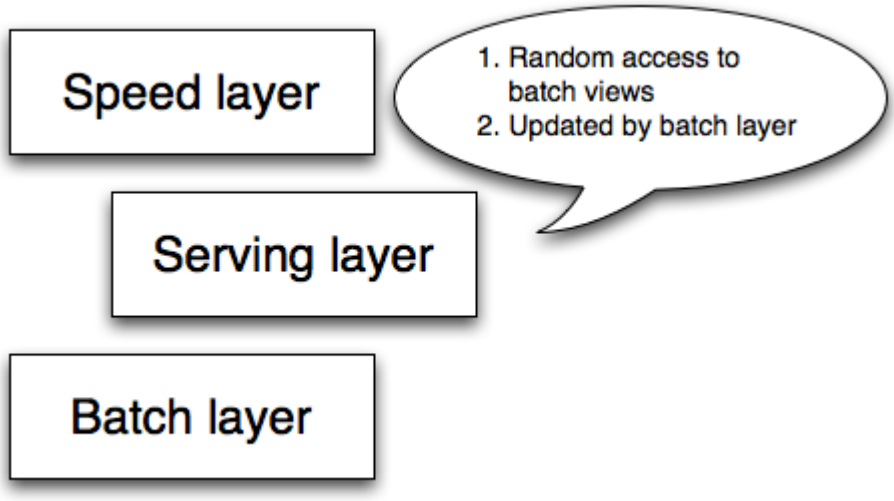


Figure 1.8 Serving layer

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

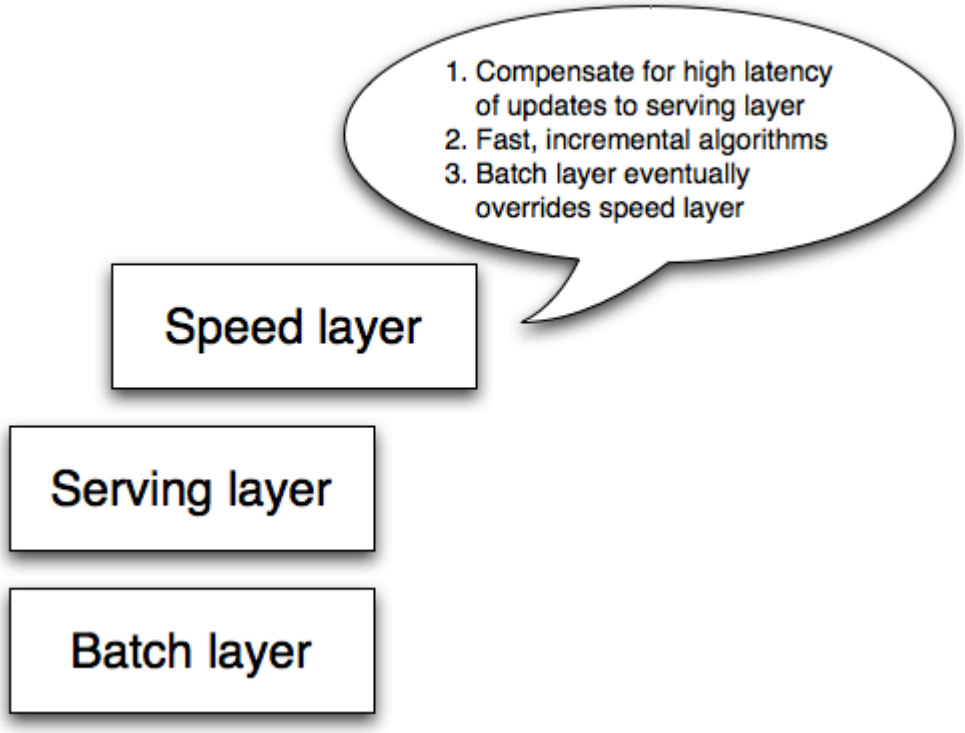


Figure 1.9 Speed layer

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.

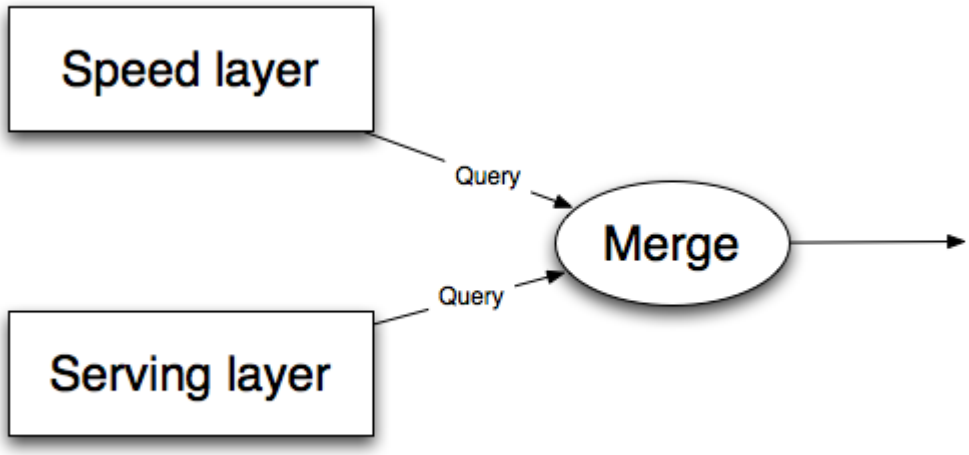


Figure 1.10 Satisfying application queries

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

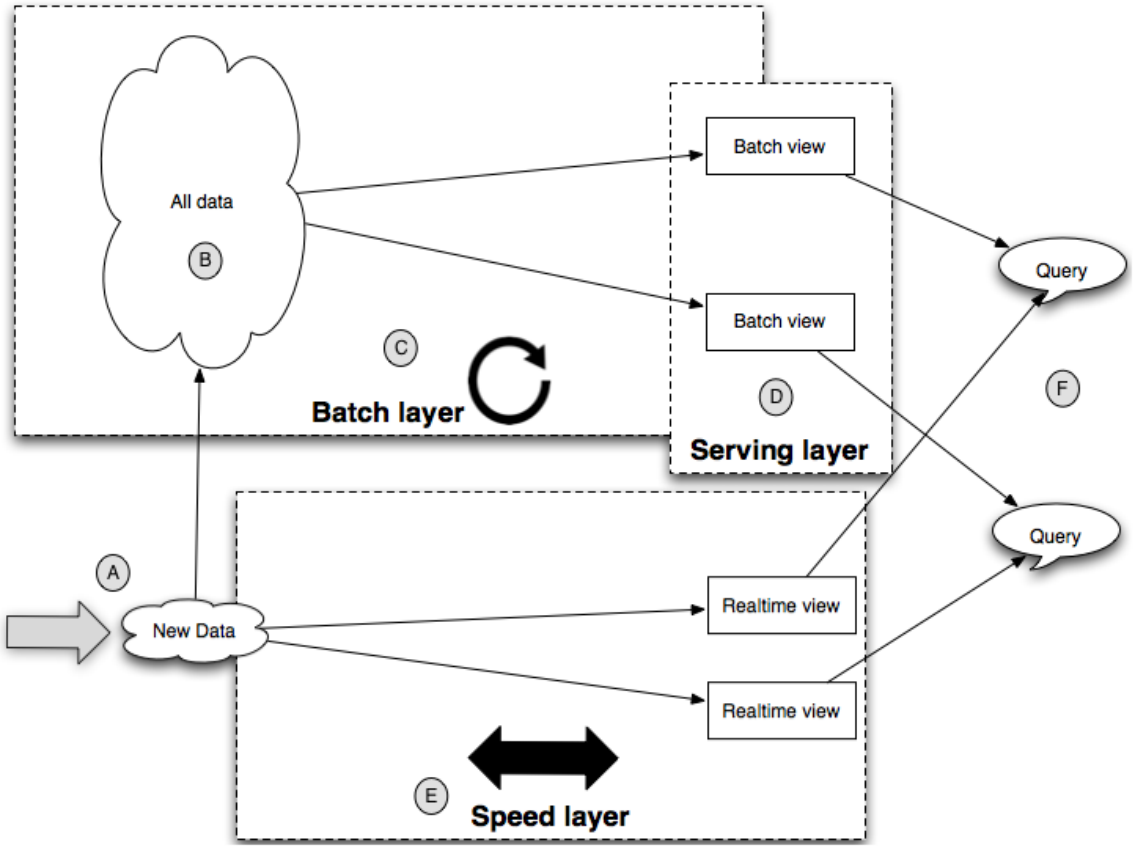


Figure 1.11 Lambda Architecture diagram

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.

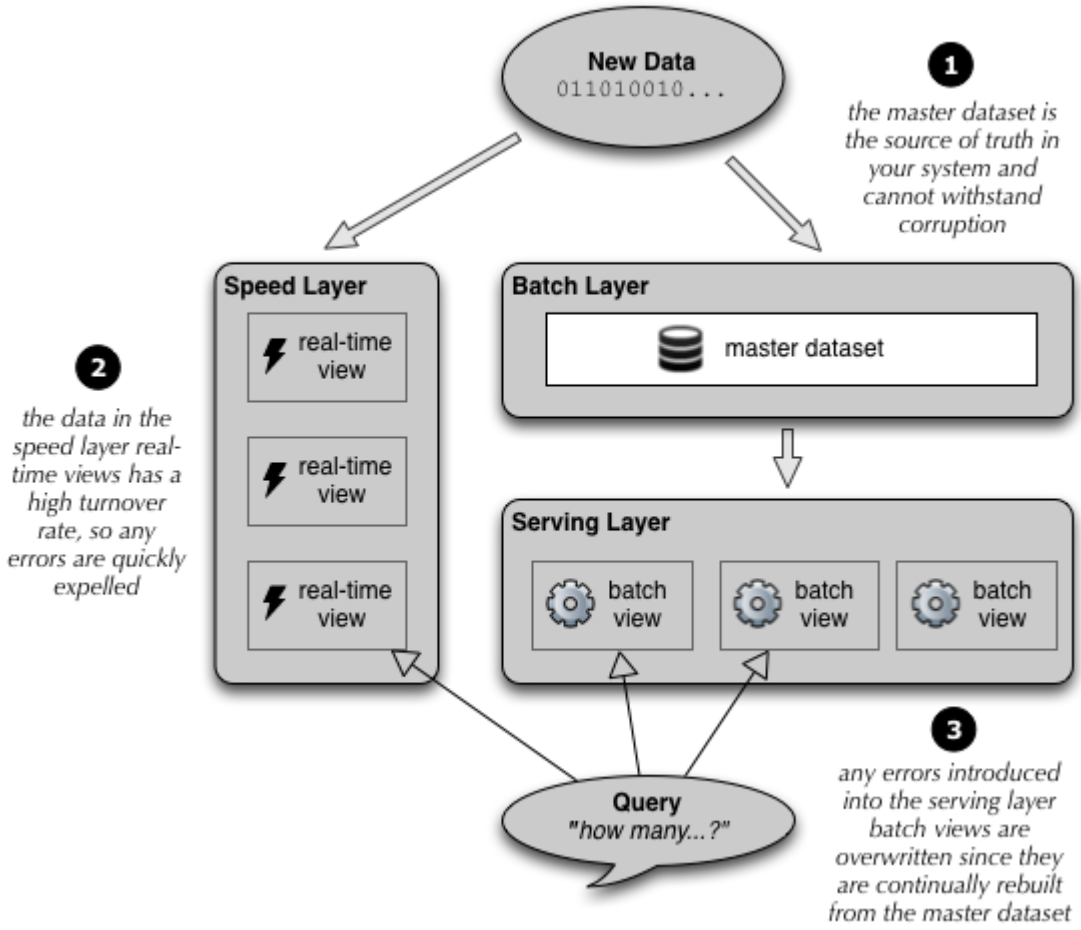


Figure 2.1 The master dataset in the Lambda Architecture serves as the source of truth of your Big Data system. Errors at the serving and speed layers can be corrected, but corruption at the master dataset is irreparable.

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.

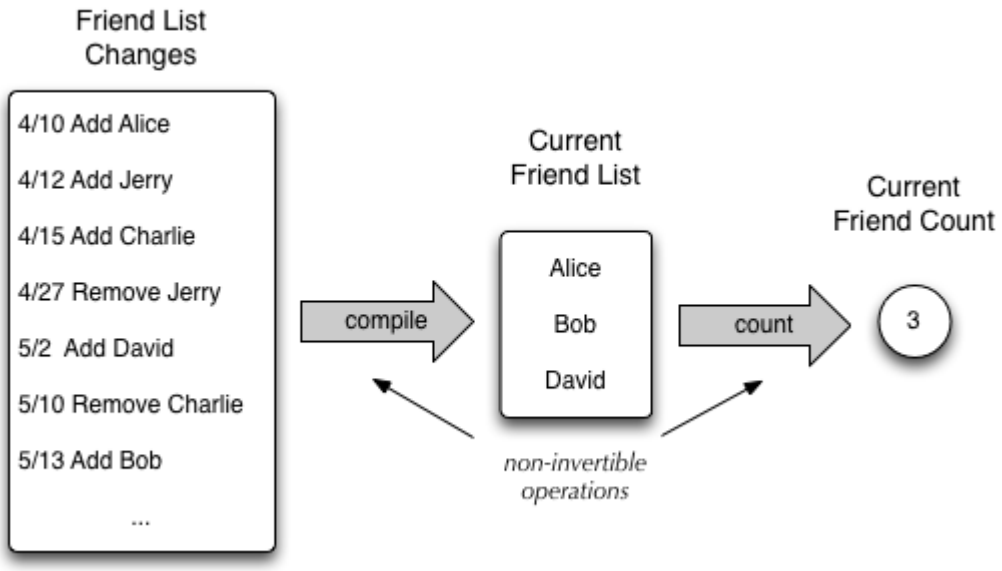


Figure 2.2 Three possible options for storing friendship information for FaceSpace. Each option can be derived from the one to its left, but it's a one way process.

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.

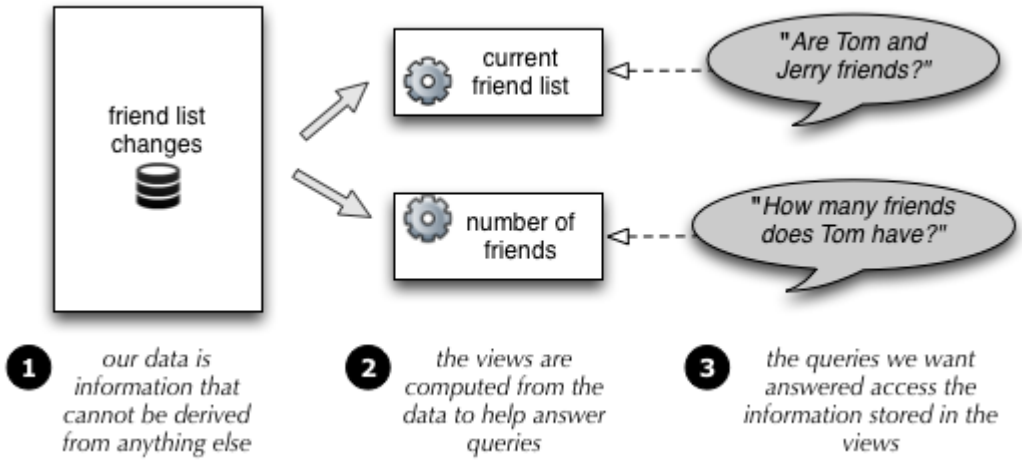


Figure 2.3 The relationships between data, views and queries.

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

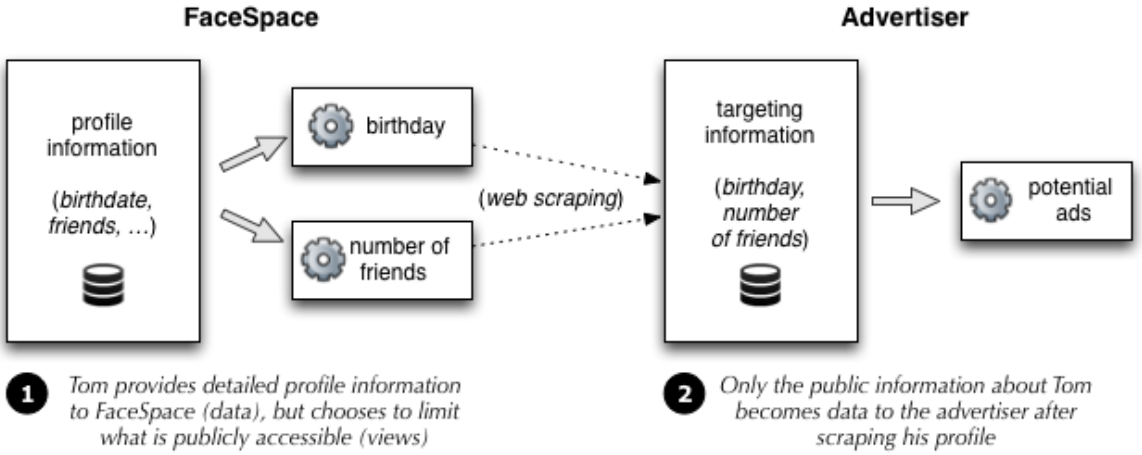


Figure 2.4 Classifying information as data or a view depends upon your perspective. As the owner of FaceSpace, Tom's birthday is a view since it is derived from the user's birthdate. However, this information is considered data to a third party advertiser.

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.

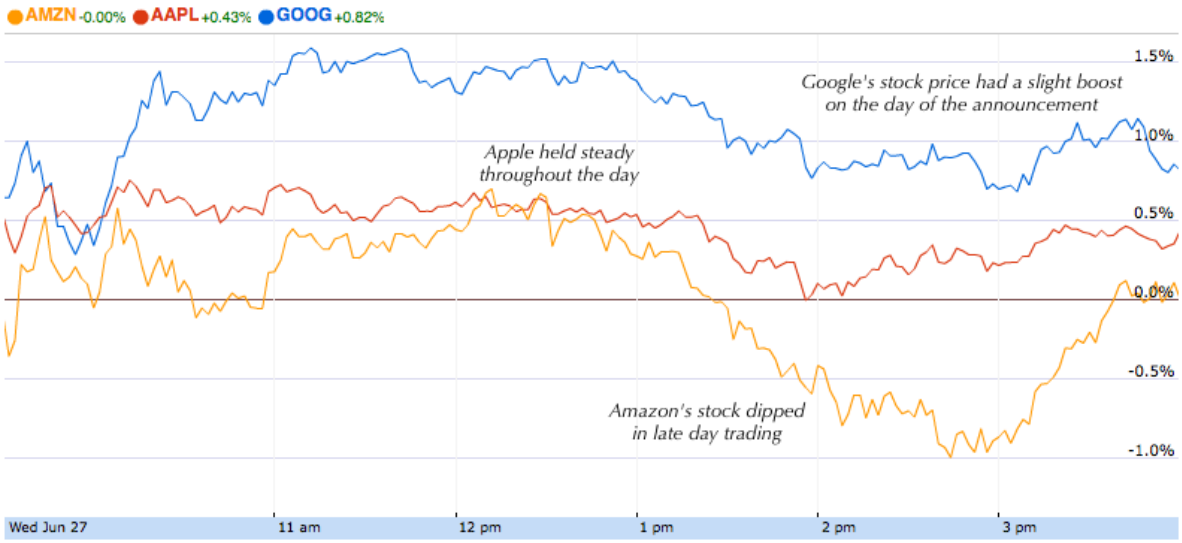


Figure 2.6 Relative stock price changes of Google, Apple and Amazon on June 27, 2012 compared to closing prices on June 26. Short term analysis isn't supported by daily records but can be performed by storing data at a finer time resolution.

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.

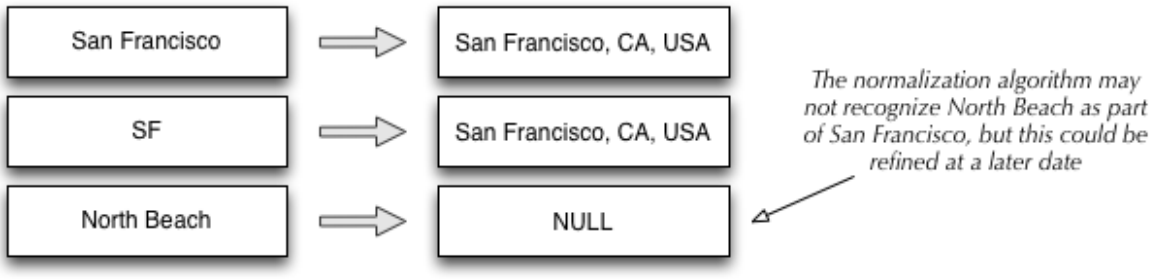


Figure 2.7 Semantic normalization of unstructured location responses to city, state and country. A simple algorithm would normalize "North Beach" to NULL if it does not recognize it as a San Francisco neighborhood.

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:

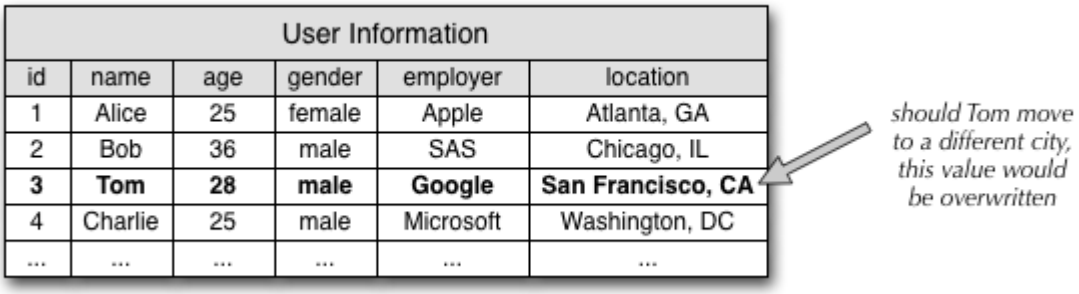


Figure 2.8 A mutable schema for FaceSpace user information. When details change - say Tom moves to Los Angeles - previous values are overwritten and lost.

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.

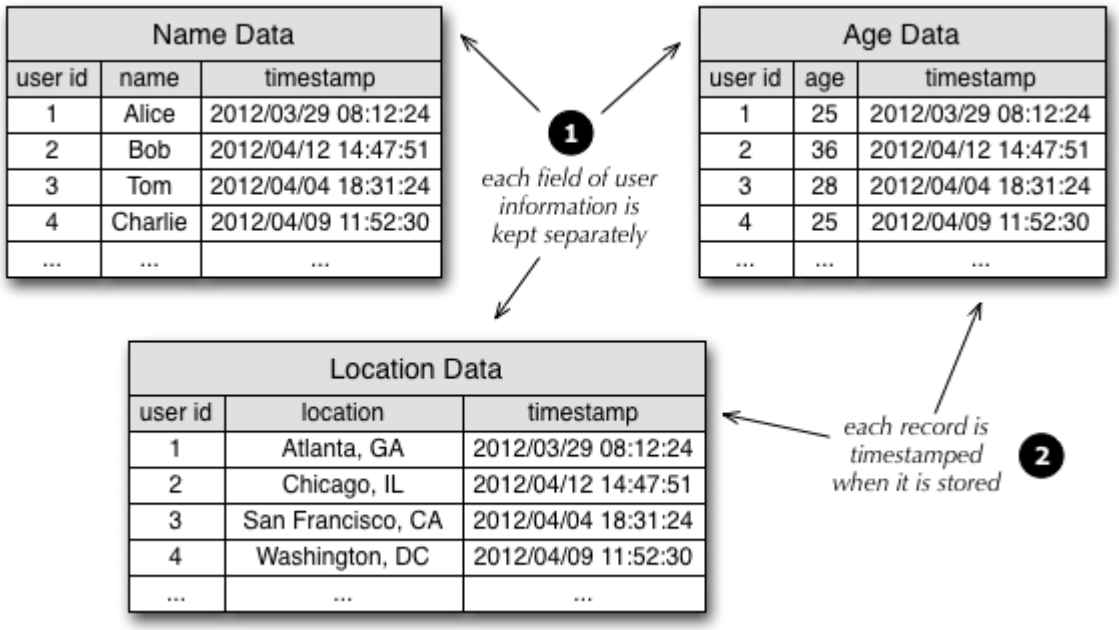


Figure 2.9 An equivalent immutable schema for FaceSpace user information. Each field is tracked in a separate table, and each row has a timestamp for when it is known to be true. (Gender and employer data are omitted for space but are stored similarly.)

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.

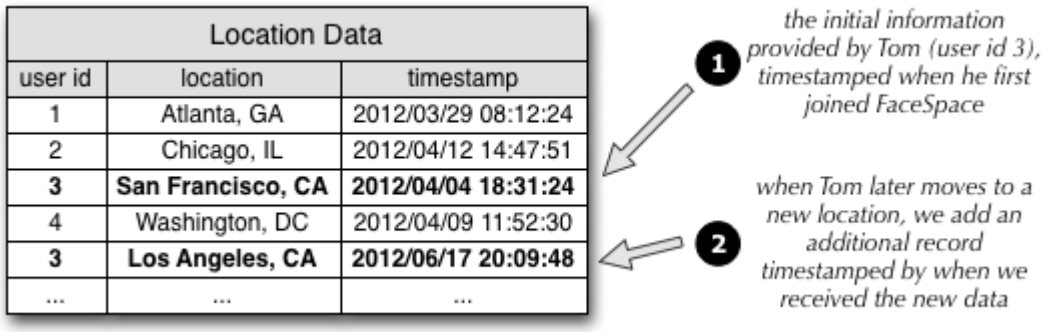


Figure 2.10 Instead of updating prexisting records, an immutable schema uses new records to represent changed information. An immutable schema thus can store multiple records for the same user.(Other tables omitted since they remain unchanged.)

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.

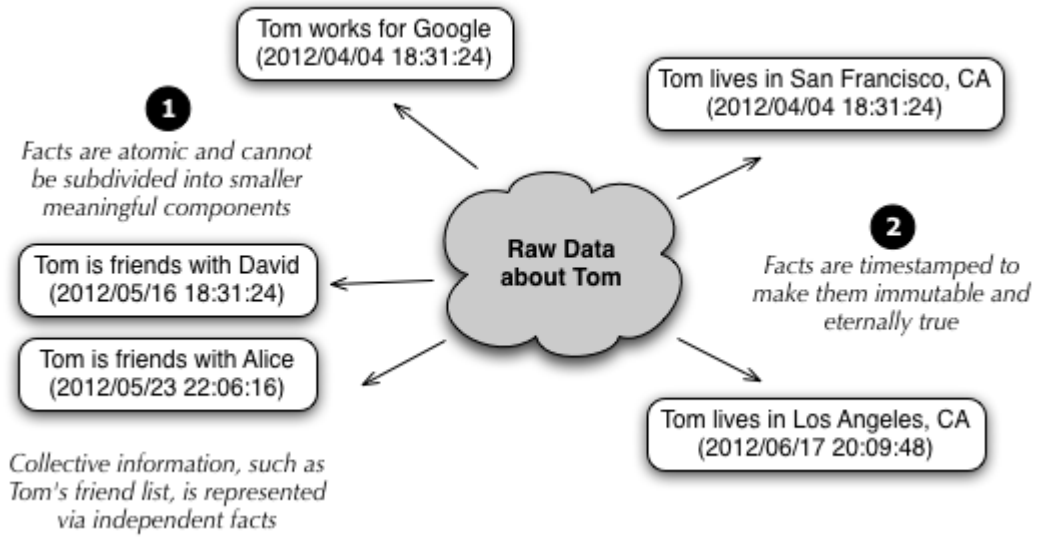


Figure 2.11 All of the raw data concerning Tom are deconstructed into time-stamped, atomic units we call facts.

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.

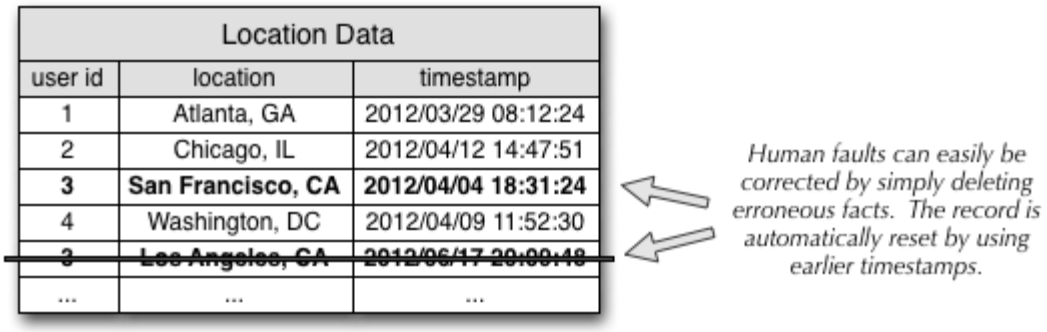


Figure 2.12 To correct for human errors, simply remove the incorrect facts. This process automatically resets to an earlier state by "uncovering" any relevant predated facts.

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

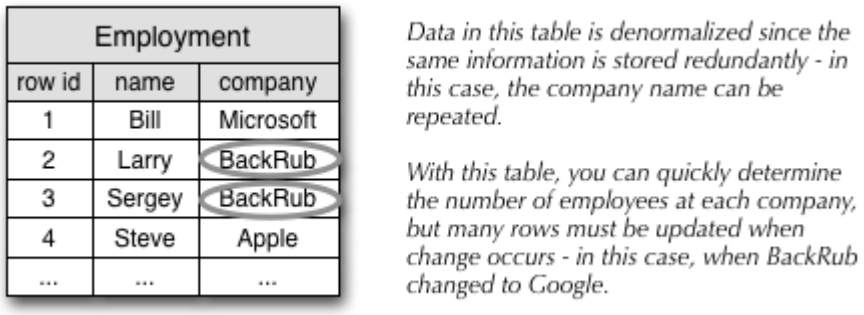
Storing one fact per record makes it easy to handle partial information about an entity without introducing NULL values into your dataset. Suppose Tom provided his age and gender but not his location or profession. Your dataset would only have facts for the known information - any "absent" fact would 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
...	...	...

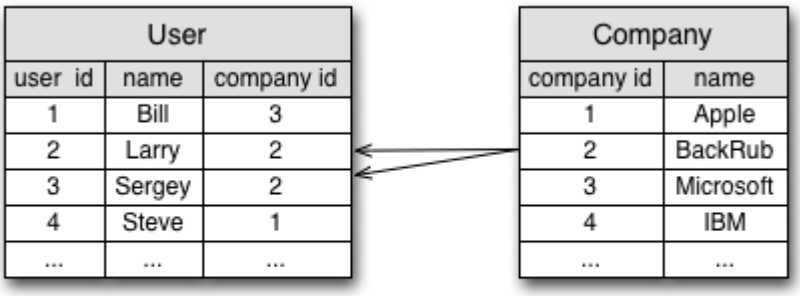
Data in this table is denormalized since the same information is stored redundantly - in this case, the company name can be repeated.

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.

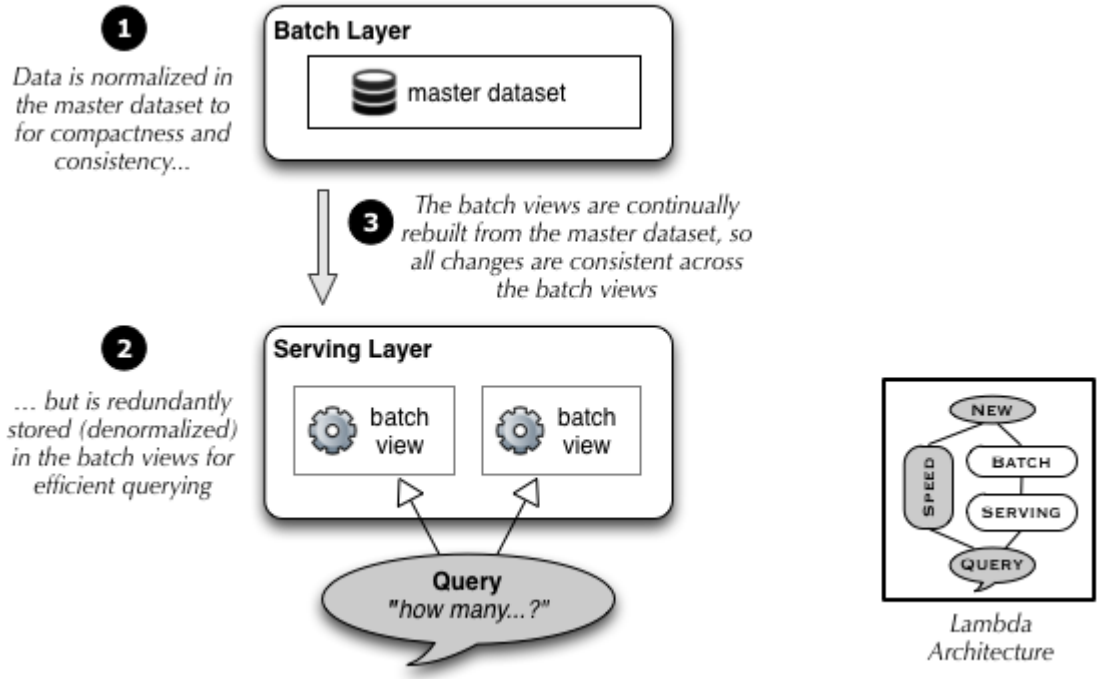


Figure 2.15 The Lambda Architecture has the benefits of both normalization and denormalization by separating objectives at different layers.

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.

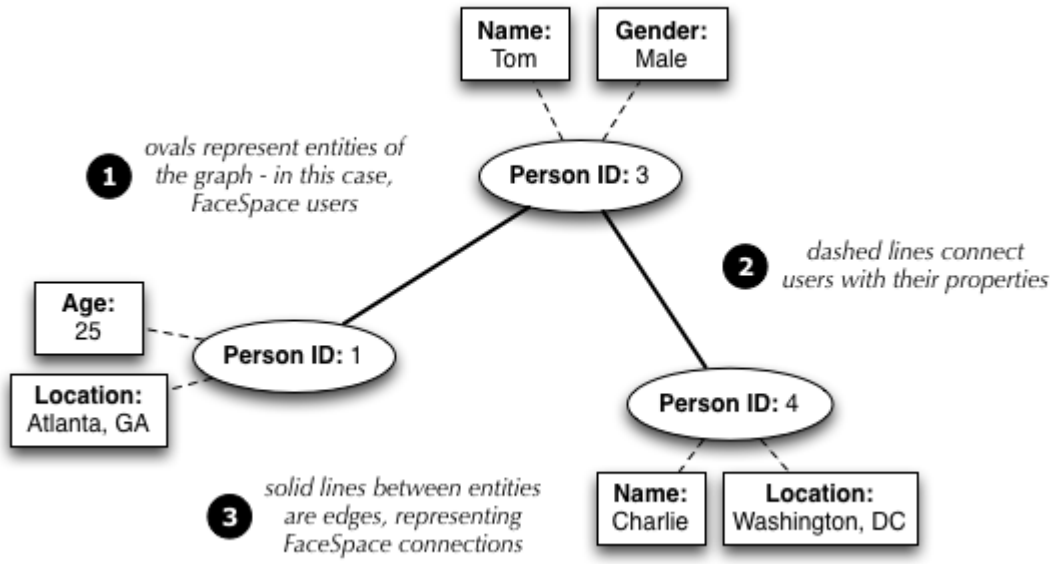


Figure 2.16 Visualizing the relationship between FaceSpace facts

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.

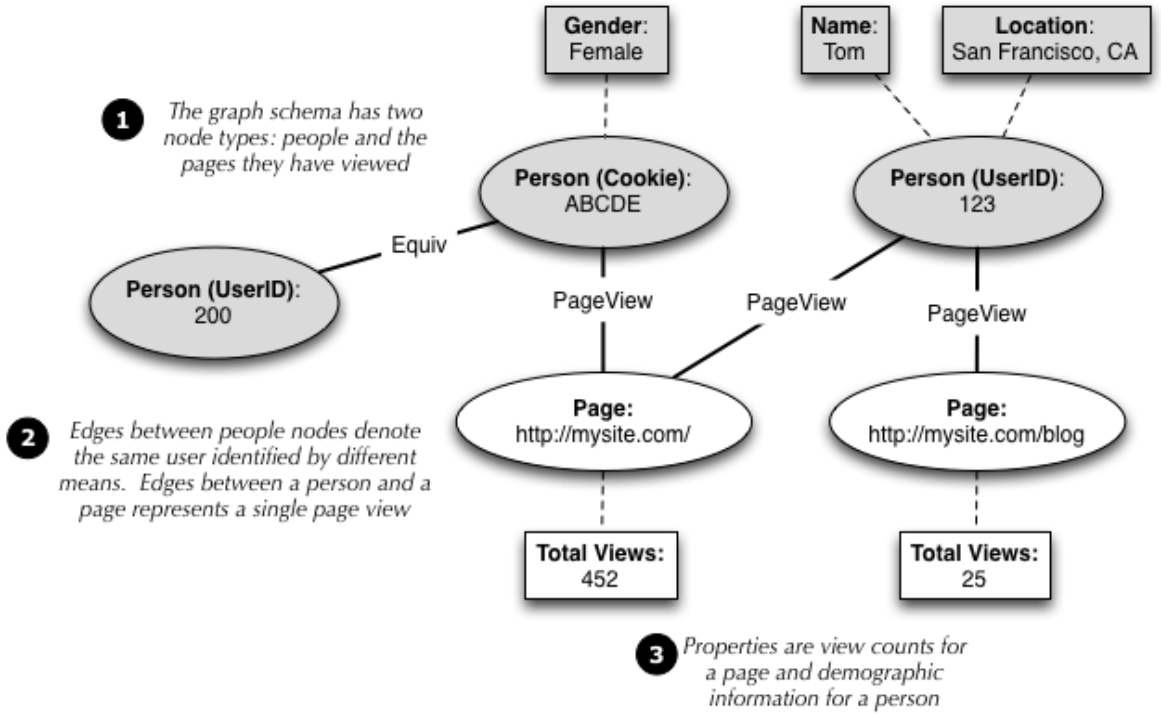


Figure 2.17 The graph schema for SuperWebAnalytics.com. There are two node types: people and edges. People nodes and their properties are slightly shaded to distinguish the two.

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. Let's recap where we are in the Lambda Architecture - see Figure 3.1.

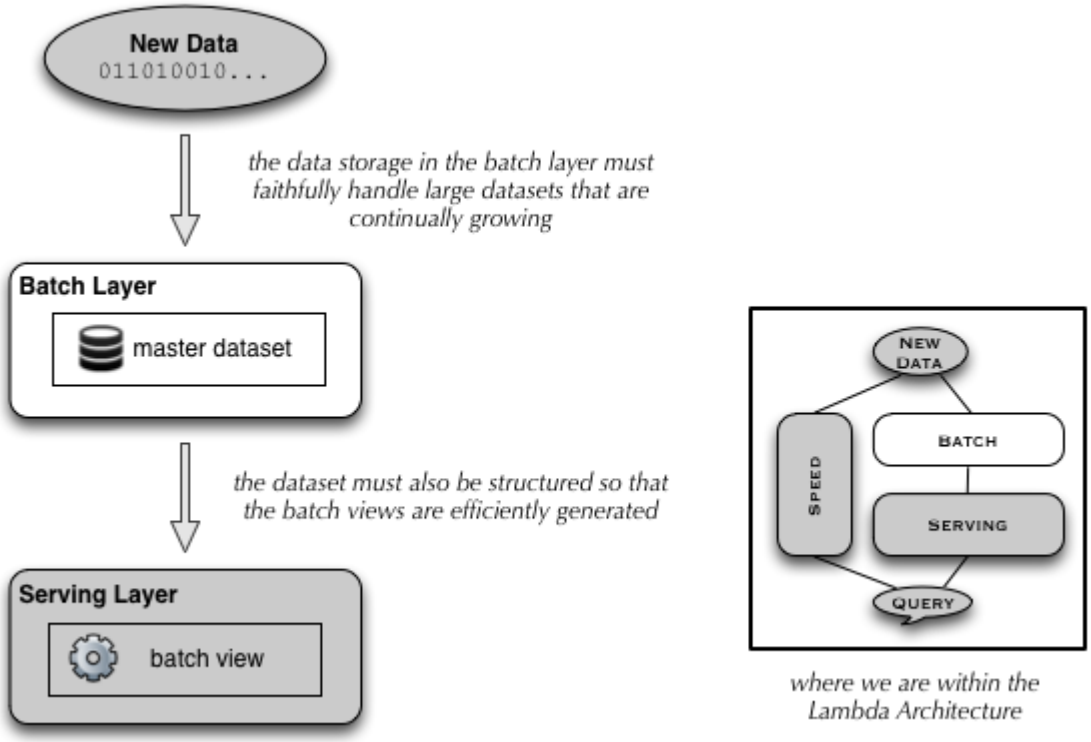


Figure 3.1 This chapter focuses on the data storage within the batch layer. This includes how the master dataset is physically stored and structured so you can efficiently write and read your data.

As with the last chapter, this chapter is all about 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.

For example, you must determine whether each piece of data is stored only once or redundantly copied across multiple locations. The latter obviously has larger storage requirements, but redundantly available data can be accessed in parallel and is resistant to hardware failures. Additionally, you can choose whether to store your data in a compressed or uncompressed format. Compressed data needs less disk space, but accessing it require additional processor effort to decompress the data to a useful form. These tradeoffs, amongst others, must be factored into your storage plan.

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

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 for alters - 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.

Once the master dataset is stored, the batch layer is 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

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 different alternatives for the batch storage solution. There are many viable candidates, ranging from distributed file systems to key-value stores to document-oriented databases. For our purposes, 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

Again, this book focuses on concepts. HDFS is an excellent medium to demonstrate how your master dataset could be stored, but you could use any technology that meets the checklist.

With that disclaimer, let's start talking about what HDFS actually *is*.

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 very intricate technology, 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.

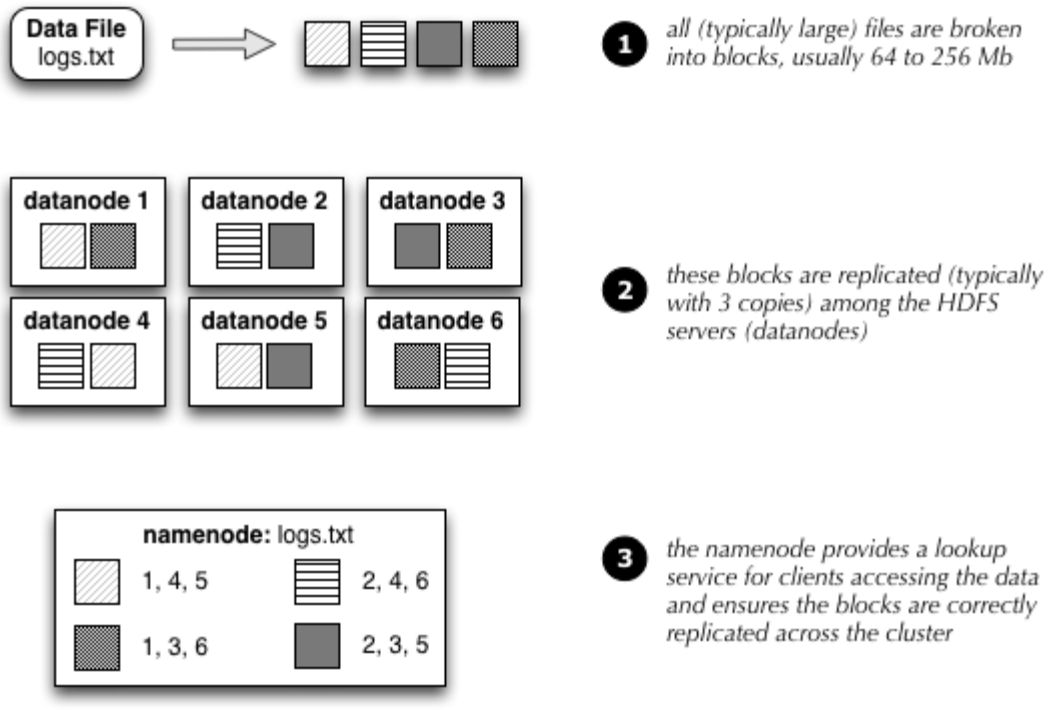


Figure 3.2 Files are chunked into blocks which are dispersed to datanodes in the cluster.

When a program needs to access a file stored in HDFS, it asks the namenode for the file's block IDs and the datanodes where each block can be found. It then contacts the datanodes directly to request the block contents. Distributing a file in this way across many nodes allows it to be easily processed in parallel. 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 optimal performance on your specific hardware. To get started quickly (and avoid the expense of obtaining servers), we suggest downloading a Hadoop virtual machine image made freely available by Cloudera (www.cloudera.com). It's useful to learn both HDFS and MapReduce, but you still need to take the time to learn the inner workings of Hadoop when setting up your own cluster.

That's all you really need to know about distributed filesystems to begin making use of them. Now let's 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 really quite 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.[1]

Footnote 1 The "hadoop fs" commands are Hadoop shell commands that interact directly with HDFS. A full list and their descriptions are available at http://hadoop.apache.org/.

```
$ hadoop fs -mkdir /logins
$ hadoop fs -put logins-2012-10-25.txt /logins
```

You can list the directory contents:

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

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)
...
```

Although it's not apparent during the upload, the process of chunking the file into blocks and distributing them among the datanodes was done behind the scenes. 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):
OK
0. blk_-1821909382043065392_1523 len=134217728
   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]
```

From the output, you can see the file has two blocks, each being replicated on

three datanodes.

Nested folders provides 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 3.3. By storing each day's information in a separate subfolder, a function can pass over data not relevant to its computation.

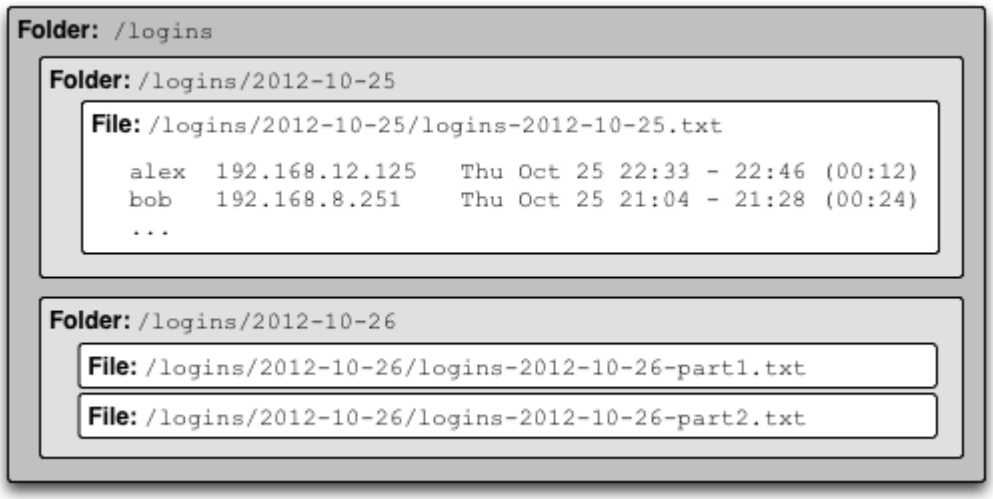


Figure 3.3 A vertical partitioning scheme for login data. By separating information for each date in a separate folder, a function can select only the folders containing data relevant to its computation.

We earlier asserted that HDFS meets the storage requirements for the batch layer, but we held off going through the checklist so we could provide more background on Hadoop. We are now more than prepared to return to our list.

3.2.3 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

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 and issues you will face when interacting with your master dataset. We'll cover these issues 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

Given that the only write operation for the batch layer is to append new data, maintaining your dataset should be an easy chore. Nevertheless, you must take caution to preserve the integrity and performance of the batch layer. When using HDFS, maintenance is essentially two operations: appending new files to the master dataset folder in a robust manner, and consolidating data to avoid small files. We'll discuss the details of each operation, paving the way to introduce a high level abstraction to accomplish these tasks.

3.3.1 Appending to the master dataset

Appending new files to the master dataset is a basic operation, but there are many potential pitfalls. To illustrate this point, consider the following rudimentary solution that uses the HDFS API.

Listing 3.1 A naive implementation to merge the contents of two 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);
    Path sourcePath = new Path(sourceDir);

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

1. get directory listing
2. move file to destination folder

Aside from the complexity due to the low-level nature of the HDFS API, this code is far from robust. Let's examine some potential failings:

1. **Non-unique filenames:** If the data to be merged is created by the same process as data already in your master dataset, it is certainly possible for the same filename to exist in both folders. Should this happen, you will overwrite - and hence lose - previously stored data.
2. **Non-standard file formats:** The data within the master dataset must be consistent in terms of its expected file format and compression type. The code in Listing 3.1 does not guarantee the files to be appended meet the standards of the master dataset.
3. **Inconsistent vertical partitioning:** Similar to maintaining a consistent file format, the code does not check or guarantee that the directory structure of the update folder follows the scheme the master dataset uses to vertically partition the data.

We cannot stress enough the importance of maintaining the integrity of your master dataset, and each of minor issues could corrupt your data.

3.3.2 Consolidating the master dataset

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 one of the intricacies of Hadoop is that computing performance is significantly degraded when the data is stored in many small files in HDFS. There can be an order of magnitude performance difference 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 each task in a MapReduce job processes a single block of a file. There is overhead to executing a task, and that overhead becomes extremely noticeable with small files. This property of MapReduce means you will want to consolidate your data should small files be added to 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.

3.3.3 The need for a high level abstraction

When building a Big Data system, you want to be spending your time using your data - not worrying about maintaining it. Instead of handling the meticulous details of appending and consolidating data, it would be much more preferable to have high-level functions as shown in Listing 3.2.

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();
  }
}
```

This example does the append in one line of code, and the consoliation in another. It throws an exception if either operation is invalid for any reason. This

code uses an abstraction called Pail that makes it dramatically easier to work with data than using files and folders directly. Let's dive into using it.

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. Specifically, Pail frees you from having to think about file formats and greatly reduces the complexity of your storage code. It makes it easy to vertically partition a dataset and provides a dead-simple API for appends and consolidation.

Under the hood, Pail is just a Java library that uses the standard APIs provided by Hadoop. As you witnessed in Listing 3.1, these APIs are quite low-level. Pail provides a high-level API that isolates you from the complexity of Hadoop's internals. As the name suggests, Pail uses *pails*, abstractions of 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 the integrity of the master dataset.

Let's dive right into the code to 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 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 example byte arrays.

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

If 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   pail.meta
```

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, a reader of 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 conflicts.

The other file in the directory stores the metadata for the pail. Let's examine its contents:

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

The metadata describes both the contents of the pail and how it is stored. This default metadata file shows that the data is stored using Hadoop's SequenceFile format, but there is no information about the type of records contained in the pail. We next cover how to store real objects in Pail, not just binary records.

3.4.2 Serializing objects into pails

To store Java objects in a pail, you must provide Pail with instructions for serializing and deserializing your objects to and from binary data. Let's return to the earlier example of storing the logins on a server. The following listing is a simplified class containing the user name and the timestamp of the login time.

```
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 implement the PailStructure interface to describe how serialization should be performed.

Listing 3.3 Implementing the PailStructure interface

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

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

  public byte[] serialize(Login login) {
    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) {
    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) {
    return Collections.EMPTY_LIST;
  }

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

By passing a 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());
  TypedRecordOutputStream out = loginPail.openWrite();
  out.writeObject(new Login("alex", 1352679231));
  out.writeObject(new Login("bob", 1352674216));
  out.close();
}
```

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) {
    System.out.println(l.userName + " " + l.loginUnixTime);
  }
}
```

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 managing files yourself. 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. To create a partitioned directory structure for a pail, you must implement two additional methods of the PailStructure interface. Pail will use these methods to enforce its structure and automatically map records to their correct subdirectory, whether those records are written into a pail in a MapReduce job or via Pail's interface.

The following code demonstrates how to partition Login objects so that records are grouped by the login date.

Listing 3.4 A vertical partitioning scheme for Login records

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

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

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

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));
  os.writeObject(new Login("david", 1352788472));
  os.close();
}
```

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:
d8c0822b-6caf-4516-9c74-24bf805d565c.pailfile

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

Returning to Listing 3.4, the "getTarget" method takes a record and returns the subdirectory where the record belongs. Pail uses this method to route a record to the correct location whenever a record is written to it. The "isValidTarget" method returns whether a directory can possibly be a valid target for a record. This method is used by Pail to understand the structure of the pail in an aggregate way.

Pail is smart about enforcing the structure of a pail and guarantees its structure

is never violated. Imagine trying to manage the vertical partitioning without the pail abstraction. It is all too easy to forget that two datasets are partitioned differently and accidentally do an append. Similarly, it would be quite easy to mistakenly violate the partitioning structure when consolidating your data. Besides providing a very simple interface for these operations, Pail protects you from making these kinds of mistakes.

3.4.5 Pail file formats and compression

Pail stores records across many files throughout 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. The beauty of Pail is that once you specify the file format, it will remember and automatically make use of that information. Any additional data added to that pail will automatically use that file format.

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 easily processed in parallel via MapReduce, and has support for automatically compressing the records in the file.

Pail formats can take additional options to influence the format. For example, 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.TYPE_ARG,
              SequenceFileFormat.TYPE_ARG_BLOCK);
  options.put(SequenceFileFormat.CODEC_ARG,
              SequenceFileFormat.CODEC_ARG_GZIP);
  LoginPailStructure struct = new LoginPailStructure();
  Pail compressed = Pail.create("/tmp/compressed",
                          new PailSpec("SequenceFile", options, struct));
}
```

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

```
root:/ $ cat /tmp/compressed/pail.meta
```

```
---
format: SequenceFile
structure: manning.LoginPailStructure
args:
  compressionCodec: gzip
  compressionType: block
```

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.

That concludes our whirlwind tour through Pail. It is a useful and powerful abstraction for interacting with your data in the batch layer, but most importantly it allows you to focus on using your data.

3.5 Pail Structure for SuperWebAnalytics.com

When we last left SuperWebAnalytics.com, we had just finished developing a graph schema for the data using Thrift. We've provided a partial listing of the resulting schema below.

Listing 3.5 Two unions from the SuperWebAnalytics.com Thrift graph schema

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

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

Pail can be used to store these data objects. In particular, the Thrift field ids provide a natural way to vertically partition the data. Figure 3.4 shows what the pail structure looks like on the filesystem, using the field ids as folder names. Each edge has its own subfolder, and each property is nested two levels deep: the first level indicating the property group and the second level the specific property within that group.

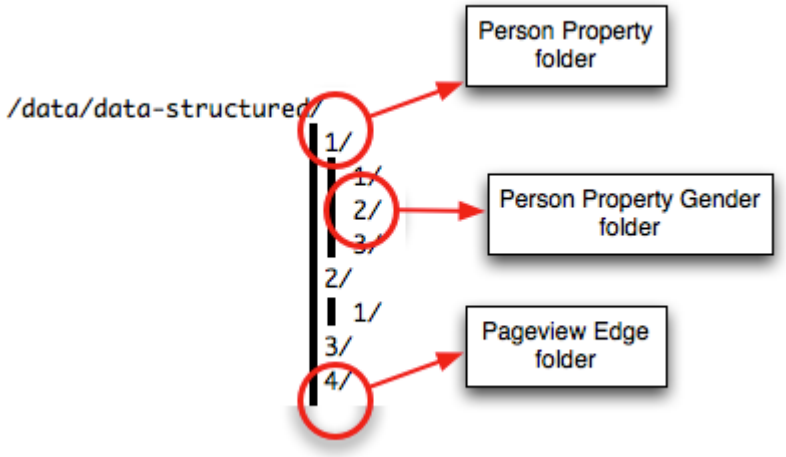


Figure 3.4 Physical structure to vertically partition the SuperWebAnalytics.com data

The code implementing this structure is a bit complex, so we'll break it down in stages. Throughout this section, don't worry so much 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. That being said, you first will need an abstract class useful for storing Thrift objects in Pails.

Listing 3.6 A generic abstract pail structure for serializing Thrift objects

```
public abstract class ThriftPailStructure<T extends Comparable>
  implements PailStructure<T>
{
  protected abstract T createThriftObject();

  private transient TDeserializer des;

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

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

  private transient TSerializer ser;
```

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

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

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

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

The next step is to create a concrete implementation for storing the Data objects of the SuperWebAnalytics.com schema.

Listing 3.7 A concrete implementation for Data objects

```
package manning.tap;

import manning.schema.Data;


public class DataPailStructure extends ThriftPailStructure<Data> {

    @Override
    protected Data createThriftObject() {
        return new Data();
    }

    public Class getType() {
        return Data.class;
    }

}
```

We will create a SplitPailDataStructure subclass to implement the vertical partitioning. All of the following listings are extracted from this class.

To accommodate the heterogeneity of storing both edges and properties, we first introduce an interface to capture the subpail logic.

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

Since edges don't have subpails, their implementation is trivial.

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

Properties are much more complex. The following code inspects the Thrift schema to build a map between Thrift field ids and the corresponding property types. This map is then used for vertical partitioning of the subpails.

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

  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());
  }

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

```
      valueId = getIdForClass(propMeta, valClass);

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

  public boolean isValidTarget(String[] dirs) {
    if(dirs.length<2) return false;
    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());
     }
  }
}
```

We can now present the remainder of the SplitDataPailStructure class. Similar to the PropertyStructure, it also inspects the Thrift schema to create a mapping between field ids and edges / properties.

Listing 3.8 Pail structure for SuperWebAnalytics.com

```
public class SplitDataPailStructure extends DataPailStructure {

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

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

```
      throw new RuntimeException(e);
    }
  }

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

  @Override
  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) return false;
      else return s.isValidTarget(dirs);
    } catch(NumberFormatException e) {
      return false;
    }
  }

  @Override
  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);
    validFieldMap.get(id).fillTarget(ret, du.getFieldValue());
    return ret;
  }
}
```

You can use "SplitDataPails" to easily vertically partition the SuperWebAnalytics.com dataset or easily read subsets of the data for batch processing. This functionality will be used heavily in the upcoming chapters when doing fine-grained processing of the data.

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 MapReduce and Batch Processing

This chapter covers:

- Computing functions on the batch layer
- Splitting a query into a precomputed portion and an on-the-fly computed portion
- Recomputation vs. incremental algorithms
- What scalability means
- The MapReduce paradigm
- Using Hadoop MapReduce

In the last two chapters, you learned how to form a data model for your data and store that data in a scalable way. The data that you store in the batch layer is called the master dataset and is the source of truth for your entire data system.

Let's take a step back and review how the Lambda Architecture works at a high level. The goal of a data system is to answer arbitrary questions about all your data. Any question you might want to ask of your dataset can be implemented as a function that takes in all of your data as input. Ideally, you could run these functions on the fly whenever you want to query your dataset. Unfortunately, running a function that takes your entire dataset as input will take a very long time to run, so it will take a very long time to get your answer. You need a different strategy if you want to be able to answer queries quickly.

In the Lambda Architecture, you precompute your queries so that at read time you can get your results quickly. The Lambda Architecture has three layers: the batch, serving, and speed layers. In the batch layer, you run functions of your entire dataset to precompute your queries. Because it is running functions of the entire dataset, the batch layer is slow to update. However, the fact that it is looking at all the data at once means the batch layer can precompute *any* query. You should view

the high latency of the batch layer as an **advantage**, as the batch layer has the time needed to do deep analyses of the data and connect diverse pieces of data together.

The serving layer is a database that is updated by the batch layer and serves the precomputed queries (called "views") with low latency reads. The speed layer compensates for the high latency of the batch/serving layers by filling in the gap for any data that hasn't made it through the batch and serving layers yet. Applications satisfy queries by reading from the serving layer view and the speed layer view and merging the results together.

Now that you know how to store data in the batch layer, the next step is learning how to compute arbitrary functions on that data. We will start with some motivating examples of queries that will be used to illustrate the concepts of computation on the batch layer. Then you'll learn in detail how the data flow through the batch layer works: how you precompute indexes which the application layer uses to complete the queries. You'll see the tradeoffs between recomputation algorithms, the style of algorithm emphasized in the batch layer, and incremental algorithms, the kind of algorithms typically used with RDBMS's. 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 the batch layer.

4.1 Motivating examples

Let's consider some example queries to motivate the theory 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 if you could run a function that takes in the entire master dataset as input. Later on we will modify these implementations to be precomputed rather than computed 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 to a URL over any range of hours. This query can be written in pseudocode like so:

```
function pageviewsOverTime(masterDataset, url,
                           startHourTime, endHourTime) {
    pageviews = 0
    for(record in masterDataset) {
        if(record.url == url &&
           record.time >= startHourTime &&
```

```
            record.endTime <= endHourTime) {
            pageviews += 1
            }
    }
    return pageviews
}
```

To compute this query with a function over the entire dataset, you simply iterate through every record and keep a counter of all the pageviews for that URL that fall within that range. Then you return the value of the counter.

4.1.2 Gender inference

The next example query operates over a dataset of name records and determines whether each person in the dataset is male or female. The algorithm first does semantic normalization on the name, doing things like converting "Bob" to "Robert" and "Bill" to "William". The algorithm makes use of a model that provides the probability of a gender for every name. That model is retrained once per month using a combination of machine learning and human intervention (human intervention to label a sample of the name data). The gender inference algorithm looks like this:

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

An interesting aspect of this query is that the results of the query 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 "personId" and "reactedToPersonId" field, indicating that "personId" retweeted or replied to "reactedToPersonId"'s tweet. 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 someone's influencer score is as follows:

```
function influence_score(masterDataset, personId) {
    // first, compute amount of influence between all pairs of people
    influence = new Map()
    for(record in masterDataset) {
        curr = influence.get(record.personId) || new Map(default=0)
        curr[record.reactedToPersonId] += 1
        influence[record.personId] = curr
    }

    // then, count how many people for which `personId` is
    // the top influencer
    score = 0
    for(entry in influence) {
        if(topKey(entry.value) == personId) {
            score += 1
        }
    }
    return score
}
```

In this code, the "topKey" function is mocked out 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 used is the top influencer.

4.2 Balancing precomputation and on-the-fly computation

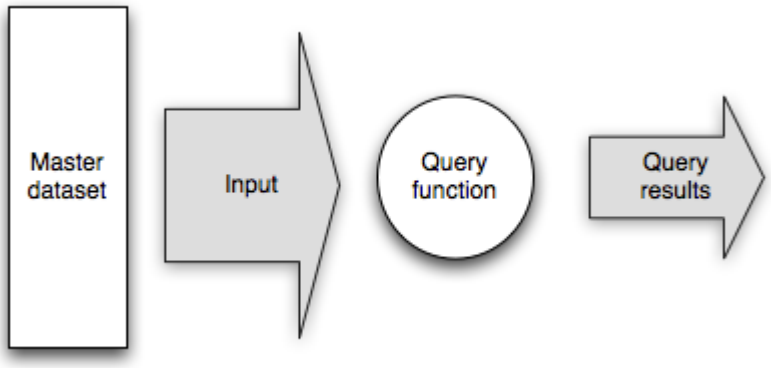


Figure 4.1 Computing queries by running function on the master dataset directly

In the Lambda Architecture, the batch layer precomputes queries into a set of batch views so that the queries can be resolved with low latency. Rather than compute queries by running a function on the master dataset, as in Figure 4.1, queries are computed by looking at the information in the precomputed batch views, as in Figure 4.2.

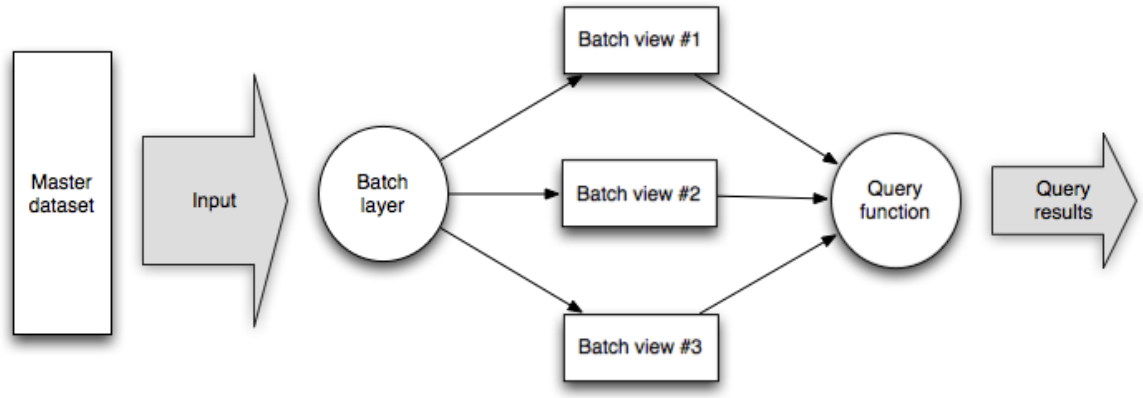


Figure 4.2 Computing queries by running function on precomputed batch views

You can't always precompute **everything**. Consider the pageviews over time query as an example. Suppose you tried to precompute every possible query – you would need to precompute the answer for every possible range of hours for every URL. However, the number of ranges of hours in a given range can be huge. For example, in a one year period, there are about 380 million distinct ranges of hours. So to precompute the query, you would need to precompute and index 380 million values **for every URL**. This is infeasible and will not be a workable solution.

Instead, what you can do is precompute an intermediate result and then do some computation on the fly to complete queries based on those intermediate results. For the pageviews over time query, you can precompute the number of pageviews for every hour of time for each URL. This is illustrated in Figure 4.3.

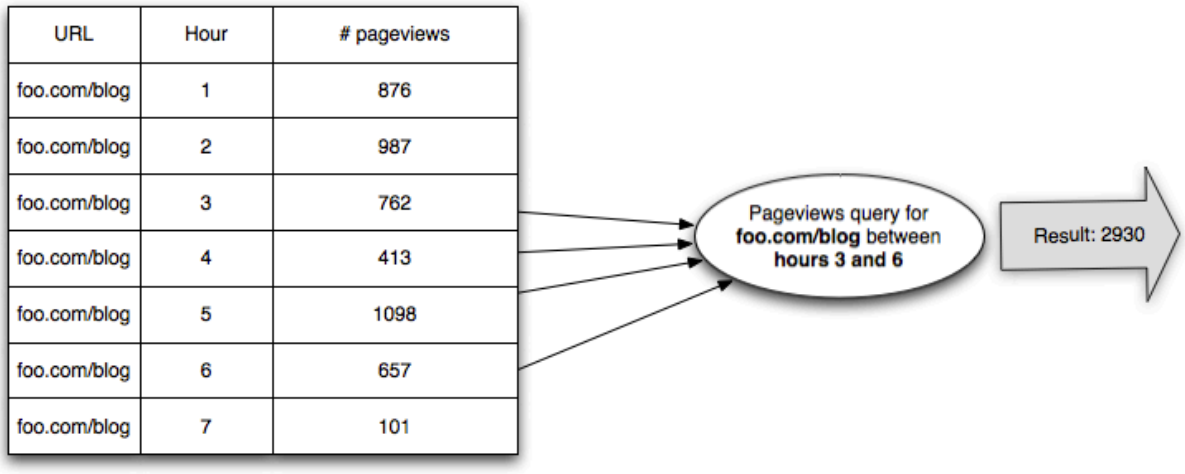


Figure 4.3 Computing number of pageviews by querying an indexed batch view

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

Designing a batch layer for a query involves striking a balance between what will be precomputed and what will be computed on the fly to complete the query. By doing a little bit of computation on the fly to complete queries, you save yourself from having to precompute unreasonably enormous indices. The key is that you need to precompute enough such that the query can be completed quickly.

4.3 Recomputation algorithms vs. incremental algorithms

The batch layer emphasizes recomputation algorithms over incremental algorithms. A recomputation algorithm computes its results by running a function on the raw data. When new data arrives, a recomputation algorithm updates the views by throwing away the old views and *recomputing* the function from scratch. An incremental algorithm, on the other hand, updates its views directly when new data arrives.

As a basic example, consider a query for the total number of records in your master dataset. Suppose you've already computed this before, and since then you've accumulated some amount of new data. A recomputation algorithm would update

the count by appending the new data into the master dataset and recounting all the records from scratch. In pseudocode, this looks like:

```
function globalCountRecomputation(masterDataset, newData) {
    append(masterDataset, newData)
    return numRecords(masterDataset)
}
```

An incremental algorithm, on the other hand, would count the number of new data records and add it to the existing count, like so:

```
function incrementalCountRecomputation(masterDataset, newData) {
    count = getCurrentCount()
    newCount = count + numRecords(newData)
    setCurrentCount(newCount)
    return newCount
}
```

You might be wondering why would you ever use a recomputation algorithm when you can use a vastly more efficient incremental algorithm. In reality, there are some important tradeoffs between the two styles of algorithms. The tradeoffs can be broken down in three categories: performance, human fault-tolerance, and the generality of the algorithm.

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. An incremental algorithm looks at the new data and the current state of the batch view to do its update. For something like 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 it needs to do an update can be multiple orders of magnitude higher than an incremental algorithm.

On the other hand, the size of the batch view for an incremental algorithm can be significantly larger than the corresponding batch view for a batch algorithm. This is because the view needs to be formulated in a way such that it can be incrementally updated. Consider these two examples of when the batch view for an

incremental algorithm is significantly larger than the corresponding recomputation-based view:

1. A query that computes the average number of pageviews for a URL in any particular domain: A recomputation algorithm would store in the batch view a map from the domain to the average. This is not possible with an incremental algorithm, since without knowing the number of records that went into computing that average, you can't incrementally update the average. So an incremental view would need to store both the average and the count for each domain, not just the average. In this case the incremental view is larger than the recomputation-based view by a constant factor.
2. A query that computes the number of unique visitors to every URL: A recomputation algorithm simply stores a map from domain to the unique count. An incremental algorithm, on the other hand, needs to know the full set of visitors for a URL to incrementally update it with new visitors. So the incremental view would need to contain the full set of visitors for each URL, which could potentially make the view nearly as large as the master dataset! In this case, the incremental view is much, much larger than the recomputation-based view.

4.3.2 Human fault-tolerance

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 by 2 for each record instead of by 1.

If your algorithm is recomputation-based, then all you have to do is fix the algorithm, redeploy the code, and your batch view will become correct the next time the batch layer runs. This is because the recomputation-based algorithm recomputes the batch view from scratch.

If your algorithm is incremental, then fixing your view isn't so simple. The only way to fix your view is to determine what records were overcounted, how many records there are like that, and then modify your view by decrementing it by the number of records that were overcounted. Accomplishing this with a high degree of confidence that you got it right is not always possible! Hopefully you have detailed logging that lets you determine what was overcounted and what wasn't. However, given that you can't really predict what mistakes will be made in the future, it's likely you won't be able to determine exactly what records were overcounted. Many times the best you can do is an estimate. Then you have to do an ad-hoc modification of your view, so you have to make sure you don't mess that up either.

Depending on "hopefully having the right logs" to be able to fix human mistakes is not sound engineering practice. Remember, human mistakes are inevitable. The lifetime of a data system is extremely long: bugs can and will be deployed to production during that time period. As you have seen, recomputation-based algorithms have much stronger human fault-tolerance properties than incremental algorithms.

4.3.3 Generality of algorithm

You saw that even though incremental algorithms can be faster to run, they can incur an enormous cost in the size of the batch view created. Oftentimes you work around this by modifying the incremental version of the algorithm to be approximate. For example, when computing uniques, you might use a bloom filter to store the set of users rather than an actual set (a bloom filter is a compact data structure that represents a set of users, but testing for membership sometimes gives false positives). This can greatly reduce the storage cost with the tradeoff that the results in the view are slightly inaccurate.

Some algorithms are particularly difficult to incrementalize. Consider the gender inference query introduced in the beginning of this chapter. As you improve your semantic normalization algorithm for names, you're going to want to see that reflected in the results of your gender inference queries. Yet, if do the normalizations as part of the precomputation, as soon as you update the normalization algorithm your entire view is completely out of date. The only way to make the view incremental is to move the normalization to happen during the "on-the-fly" portion of the query. Your view will have to contain every name ever seen for each person, and your on-the-fly code will have to re-normalize every single name every time a query is done. Semantic normalization operations tend to be somewhat slow, so trying to incrementalize the view has a serious latency cost for each time you resolve the query. The latency cost of the on-the-fly computation could very well be too much for your application requirements.

A recomputation algorithm, of course, can do the normalization during the precomputation, since if the algorithm changes it will rerun it on the entire dataset.

4.3.4 Choosing a style of algorithm

You must always be able to perform a full recompute of your batch views from your master dataset. 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. On top of that, you can optionally add incremental versions of your algorithms to make them more resource-efficient. Typically the incrementalized versions of an algorithm are similar to the recomputation version, although not always. For the remainder of this chapter, we will focus on just recomputation algorithms; in chapter 7 we will come back to the topic of incrementalizing the batch layer. The key takeaway for now is that you must always have recomputation versions of your algorithms, and on top of that you *might* add incremental versions for performance.

4.4 Scalability in the batch layer

The word "scalability" gets thrown around a lot, so let's carefully define what it actually 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 on.

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 4.4. Then the costs of running your system would rise dramatically over time. Increasing your load 10x would increase your costs 100x. Such a system is not feasible from a cost perspective.

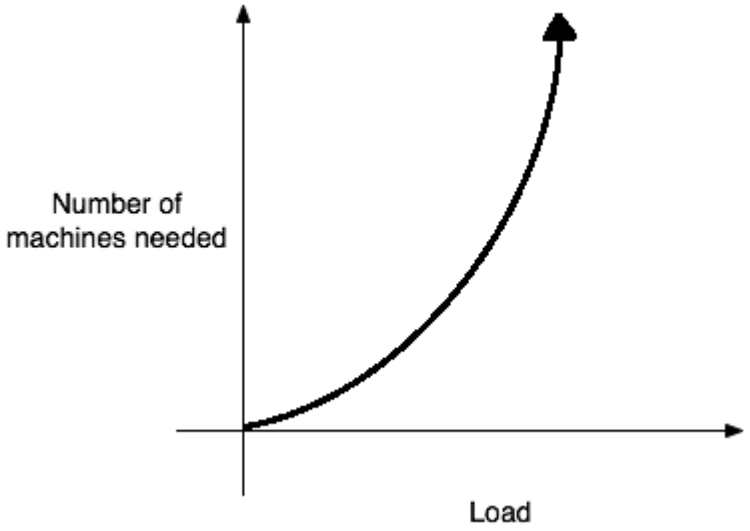


Figure 4.4 Non-linear scalability

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

Counterintuitively, a scalable system does not necessarily have the ability to *increase* performance (like lower latency of requests) 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 your latency requirements (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.

Let's now take a look at MapReduce, a distributed computing paradigm that can be used to implement a batch layer.

4.5 MapReduce

MapReduce is a distributed computing paradigm originally pioneered by Google that provides primitives for scalable and fault-tolerant batch computation. MapReduce's primitives are general enough to allow you to implement nearly any query function, making it a perfect paradigm for the precomputation needed in the batch layer.

With MapReduce, you write your computations in terms of "map" and "reduce" functions, and MapReduce will automatically execute that program across a cluster of computers.

Let's look at an example of a MapReduce program and then we'll look at how MapReduce scales and is fault-tolerant.

The canonical MapReduce example is "word count". "Word count" takes as input a dataset of sentences and emits as output the number of times each word appears across all sentences.

The "map" function in MapReduce executes once for each input record and emits any number of key/value pairs. The "map" function for word count looks like this:

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

This map function is applied once for every sentence in the input dataset. It emits a key/value pair for every word in the sentence, setting the key to the word and the value to the number one.

MapReduce then sorts your key/value pairs and runs the "reduce" function on each group of values that share the same key. The reduce function for word count looks like this:

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

The reduce function is applied for every group of values that were emitted with the same key. So in word count, the reduce function receives a list of "one" values for every word. To finish the computation, the reduce function for word count simply sums together the "one" values to compute the count for that word.

There's a lot happening underneath the hood to get a program like word count to work across a cluster of machines, but at a high level, that's the interface you use to write your programs.

4.5.1 Scalability

The reason why MapReduce is such a powerful paradigm is because programs written in terms of MapReduce are inherently scalable. The same program can run on ten gigabytes of data as can run on ten petabytes of data. MapReduce automatically parallelizes the computation across a cluster of machines. All the details of concurrency, transferring data between machines, and what to run where 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 files containing sentences. MapReduce operates on data stored in a distributed filesystem, like HDFS, the distributed filesystem you learned about in Chapter 3. Recall that the contents of a file in a distributed filesystem are spread across the cluster across many machines.

First, MapReduce executes the map function across all your sentences. It launches a number of "map tasks" proportional to the amount of data you have across your files. Each spawned task is responsible for processing a subset of one file. MapReduce assigns tasks to run on the same machine as the data they're processing, when possible. The idea is that it's much more efficient to move code to data than to move data to code. Moving code to the data avoids having to transfer all that data across the network.

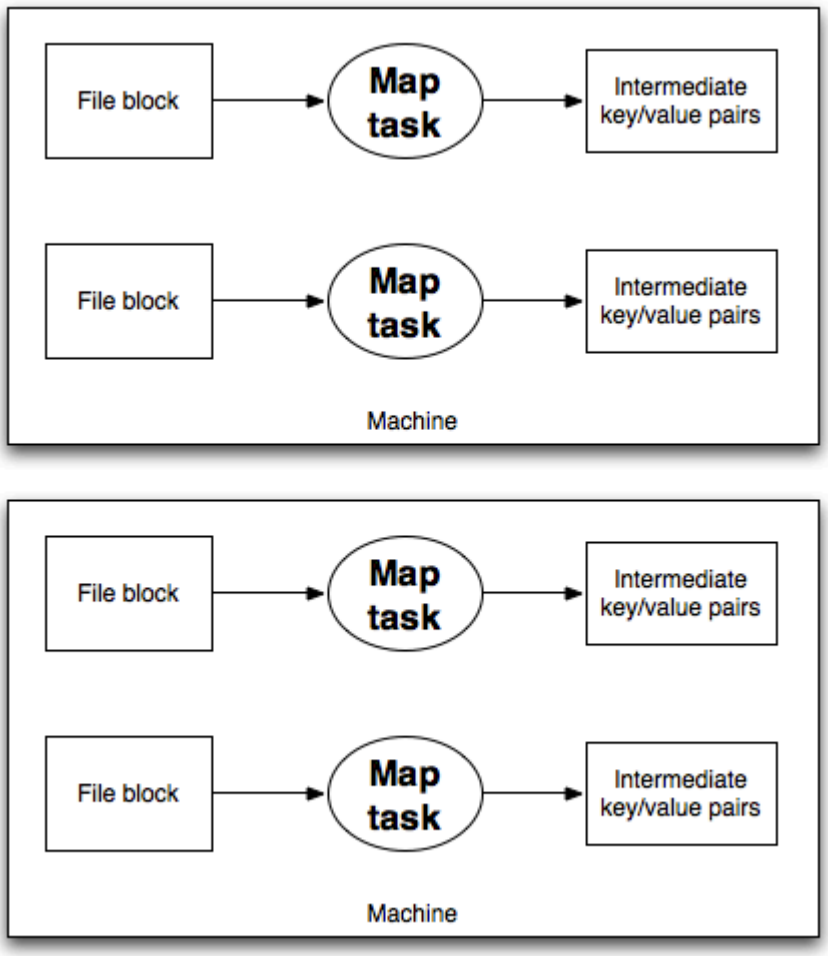


Figure 4.5 Map phase

Each "map task" runs the same code – the code that you provided in your map function. Each task produces a set of intermediate key/value pairs, as shown in Figure 4.5. The key/value pairs are sent to the "reducer tasks" which are responsible for executing the reduce function. Like the map tasks, the reduce tasks are spread across the cluster. Each reduce task is responsible for computing the reduce function for a subset of the keys. Each key/value pair emitted by a map task is sent to the reduce task responsible for that key, so each map task ends up sending key/value pairs to all the reduce tasks. Transferring the intermediate key/value pairs to the correct reducers is called the "shuffle phase" and is illustrated in Figure 4.6.

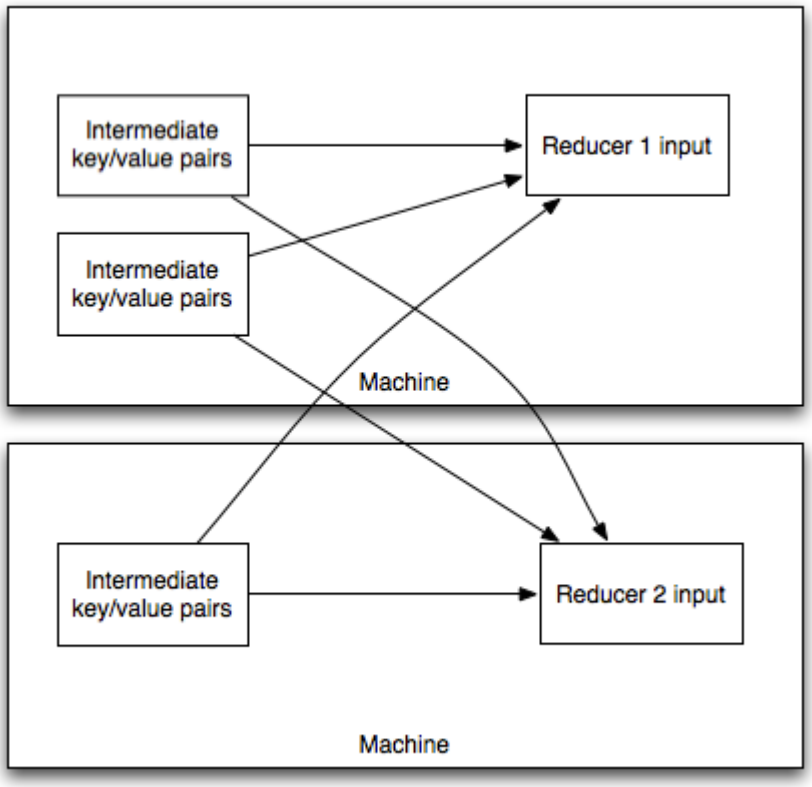


Figure 4.6 Shuffle phase

Once a reduce task receives key/value pairs from each map task, it sorts the key/value pairs by key, as shown in Figure ref:sortphase. This is called the "sort phase" and has the effect of organizing all the values for any given key to be together.

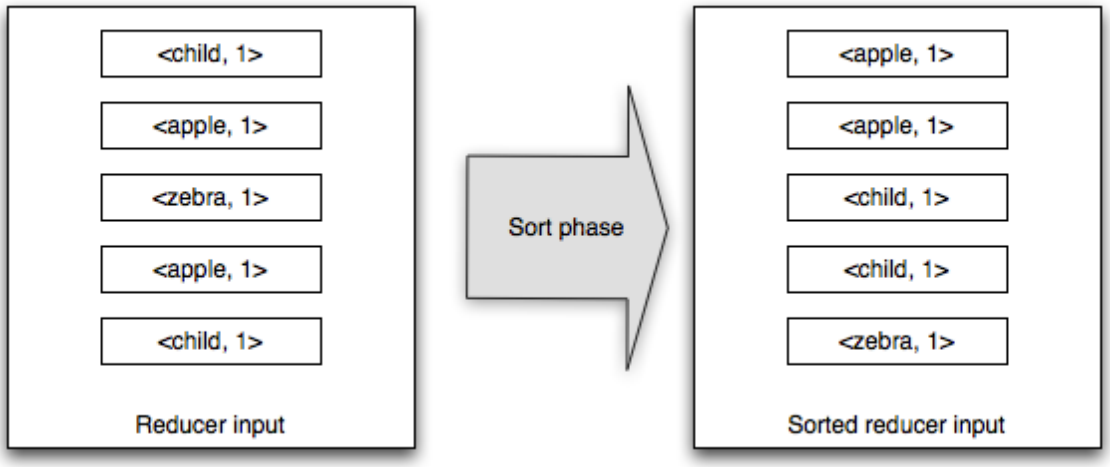


Figure 4.7 Sort phase

Finally, in the "reduce phase", the reducer scans through each group of values, executing the reduce function on each group. The output of the reduce function is stored in the distributed filesystem in another set of files, as diagrammed in Figure 4.8.

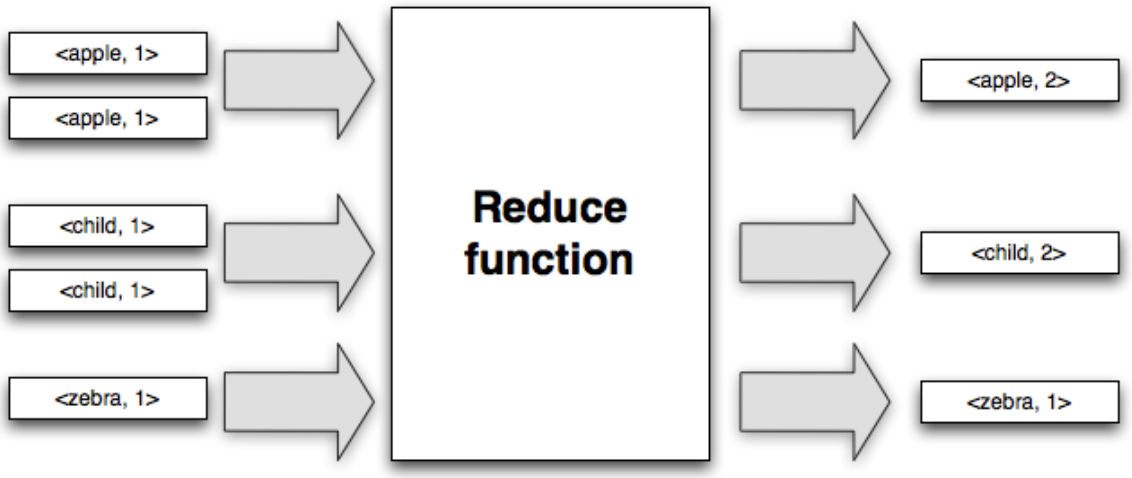


Figure 4.8 Reduce phase

As you can see, there's a lot going on to execute a MapReduce program. What's important to takeaway from this overview of how MapReduce works is the following:

1. MapReduce programs execute in a fully distributed fashion: there's no central point of contention
2. MapReduce is scalable: the "map" and "reduce" functions you provide are executed in parallel across the cluster
3. All the challenges of concurrency and assigning tasks to machines is handled for you

4.5.2 Fault-tolerance

In addition to being trivially parallelizable, MapReduce computations are fault-tolerant. A MapReduce cluster is made up of a collection of individual machines working together to run your computations. If one of the machines involved in a computation goes down, MapReduce will automatically retry that portion of the computation on another node.

Machines can fail for a variety of reasons: the hard disk can fill up, the hardware can break down, or the computation load can get too high. MapReduce watches for these errors and retries any affected tasks. An entire job will fail only if a task fails more than a configured amount of times (by default, 4). The idea is that

if a task fails once, it's probably random. If a task keeps failing repeatedly, it's most 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 produces random numbers. If you wanted to use random numbers in a MapReduce job, you would 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 exposed by MapReduce is general enough for computing nearly arbitrary functions on your data. To illustrate this, let's take a 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

As explained earlier, the batch view for number of pageviews over time contains a mapping from [url, hour] to number of pageviews for that hour. Some on-the-fly computation will be required to complete the query at read time.

The MapReduce logic to compute this batch view is as follows:

```
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)))
}
```

As you can see, it's similar to how word count works. The key emitted by the map function is just a little bit different.

IMPLEMENTING GENDER INFERENCE

Gender inference is similarly straightforward:

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

function reduce(userid, vals) {
```

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

The map function performs the name normalization, and the reduce function computes the probability of being male or female.

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.personId, record.reactedToPersonId)
}

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

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

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

The first MapReduce job is specified with the functions map1 and reduce1, while the second stage is specified with the functions map2 and reduce2. It's

typical for computations to require multiple MapReduce jobs – that just means multiple levels of grouping were required.

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

1. Arbitrarily partition your data (through the key you emit in the map phase). Arbitrary partitioning lets you connect your data together any way you want while still processing everything in parallel.
2. Arbitrarily transform your data (through the code you provide in the map and reduce phases)

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

4.6 Using Hadoop MapReduce

Hadoop MapReduce is an open-source implementation of MapReduce. Hadoop MapReduce integrates with the Hadoop Distributed Filesystem (HDFS) that you learned about in Chapter 3. HDFS is optimized for the kinds of streaming access patterns you use when doing MapReduce computations, where large amounts of data are read at once. Let's look at how to use Hadoop to write and execute MapReduce computations.

You should know up front that thinking in terms of MapReduce can be difficult. If Hadoop feels like a low level abstraction, that's because it is a low level abstraction. In the next chapter you'll learn about JCascalog, a higher level abstraction over Hadoop that makes implementing MapReduce workflows much easier. However, to use these abstractions effectively, it's important to understand what's going on under the hood.

In addition to the "map" and "reduce" functions we covered above, you have to tell a Hadoop program what data to read and where to write the results. These are called, respectively, the "Input Reader" and "Output Writer". Every MapReduce program requires these four components:

- Input Reader
- Mapper
- Reducer
- Output Writer

As a developer, you're responsible for providing each of these. Let's examine

each of the components in detail.

4.6.1 Input Reader

The input reader specifies how to get the input data for a MapReduce computation. While an input reader can read data from anywhere, you'll almost always be reading data from the distributed filesystem, as the distributed filesystem is built for doing these kinds of batch computations. An input reader tells MapReduce how to split up the input data so that it can be processed in parallel, and provides a "record reader" so that MapReduce can read records into the map function.

MapReduce records are always represented as key/value pairs. The input reader produces key/value pairs, mappers emit key/value pairs, and reducers emit key/value pairs. The key/value data model can be confusing as it forces you to choose two values for each data record. Sometimes it's not clear what that second value should be; to deal with this some Input Readers set the value to be null or something arbitrary.

The key/value pairs produced by the input format are supplied as input to the mapper. The mapper is where the magic starts to happen.

4.6.2 Mapper

As discussed earlier, a mapper is a function that accepts a record and emits any number of key/value pairs. The map function is run in parallel across the cluster on all the splits of data provided by the input reader.

The key emitted by a mapper decides which reducer the emitted key/value pair goes to. You saw an example of this with the word count example, where the key is the word and the value is the number "1". As an example, take the word "banana". Every key/value pair with "banana" as its key will find its way to a single reducer.

4.6.3 Reducer

The reducer is called once for each key and its corresponding group of values and emits key/value pairs. In the word count example, the reduce function is called for each word with the word and a list of 1's. The reducer simply sums up each of the "1" values to produce the count for that word.

As a general rule of thumb, the reduce phase is about 4x more expensive than the map phase. This is because to do the reduce step, all the data from the mappers must be transferred across the network, sorted, and then computed on. As you'll soon see, reduce functions are necessary for doing joins and aggregations, so part of making an efficient MapReduce workflow is minimizing the number of reduces.

4.6.4 Output Writer

The output writer is responsible for storing the key/value pairs emitted by the reducer. An output writer might write each key/value pair into a text file, into a binary file, or into a writable database. Often, output data will be stored on the distributed filesystem for later use by other MapReduce jobs. Many applications require multiple MapReduce jobs to be chained together. In these cases, the output can be safely discarded after another MapReduce job's input reader has consumed all data.

4.7 Word Count on Hadoop

Let's look at how you'd implement word count on top of Hadoop. Instead of using pseudocode, as we were doing before, you'll see how to implement word count on an actual MapReduce framework. The input to the program will be a set of text files containing a sentence on each line. The goal of the job is to emit, for each word, the number of times that word appears across every input sentence. The output will be emitted to a text file.

It's not important to understand this code in depth, only to see how cumbersome it is. In the next chapter, you'll explore a higher level abstraction to MapReduce that makes writing these computations much, much simpler.

Listing 4.1 Word count implemented using Hadoop

```
public class WordCount {
    public static class Map extends MapReduceBase implements
        Mapper<LongWritable, Text, Text, IntWritable> {
      private final static IntWritable one = new IntWritable(1);
      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()) {
          word.set(tokenizer.nextToken());
          output.collect(word, one);
        }
      }
    }

    public static class Reduce 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()) {
          sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
      }
    }

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

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

      conf.setMapperClass(Map.class);
      conf.setReducerClass(Reduce.class);

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

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

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

The Map and Reduce classes should be easy to follow; they implement the map and reduce functions as described earlier in this chapter. The main method configures and launches the job. Main specifies the classes for the InputFormat, Mapper, Reducer, and OutputFormat, as well as specifying the types for the input and output data. It sets the path for the input data and the path where the output should go using static methods on FileInputFormat and FileOutputFormat. Finally, the main function launches the job by passing the job configuration to the JobClient.

Hadoop's implementation of WordCount forces you do a lot of extra "stuff" that's tangential to the actual problem of counting words. There's an overabundance of type declarations. You have to know quite a bit about the filesystem you're working with and the representation of the data. It's strange that you can't parameterize the input format and output format directly, instead having to set parameters in some other class.

This is the canonical introduction to Hadoop and is already quite painful.

4.8 MapReduce is a low level of abstraction

To show how much work it is to use Hadoop MapReduce directly, let's 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. To avoid skewing the results stop words like "a" and "the" should be ignored from the computation. This is only a slightly more complicated problem than counting words.

A good way to do this query is to modify the word count example with a second MapReduce job. The first job will be modified to emit <word length, count> key/value pairs from the reducer for non-stop words. The map phase emits <word, 1> pairs like before but also filters out stop words. The reduce phase sums together the 1's and then emits the length of the word and the count as its result.

The second MapReduce job takes as input the <word length, word count> key/value pairs from the first job and emits <word length, average word count> key/value pairs as its result. The mapper doesn't have to do anything: it simply passes the key/value pairs it receives as-is to the reducer. The reducer then takes the average of the word counts it sees to produce the average word count for each word length.

Now let's look at the actual Hadoop code:

Listing 4.2 Word frequency vs. word length implemented using Hadoop

```
public class WordFrequencyVsLength {
    public static final Set<String> STOP_WORDS = new HashSet<String>()
    {{
       add("the");
       add("a");
       // add more stop words here
    }};

    public static class SplitterAndFilter extends MapReduceBase
       implements Mapper<LongWritable, Text, Text, IntWritable> {
      private final static IntWritable one = new IntWritable(1);
      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)) {
              word.set(token);
              output.collect(word, one);
```

```
      }
    }
  }
}

public static class LengthToCount 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));
  }
}

public static class AverageMap extends MapReduceBase
    implements Mapper<IntWritable, IntWritable,
                      IntWritable, IntWritable> {
  private final static IntWritable length = new IntWritable();

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

public static class AverageReduce 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();
      count +=1;
    }
    double avg = 1.0 * sum / count;
    output.collect(key, new DoubleWritable(avg));
  }
}

private static void runWordBucketer(
        String input,
        String output) throws Exception {
  JobConf conf = new JobConf(WordFrequencyVsLength.class);
  conf.setJobName("freqVsAvg1");
```

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

      conf.setMapperClass(SplitterAndFilter.class);
      conf.setReducerClass(LengthToCount.class);

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

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

      JobClient.runJob(conf);
    }

    private static void runBucketAverager(
            String input,
            String output) throws Exception {
      JobConf conf = new JobConf(WordFrequencyVsLength.class);
      conf.setJobName("freqVsAvg2");

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

      conf.setMapperClass(AverageMap.class);
      conf.setReducerClass(AverageReduce.class);

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

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

      JobClient.runJob(conf);
    }

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

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's a clear indication that you're working at a low level of abstraction. You want to work with an abstraction where the whole computation can be represented as a single conceptual unit, and things like temporary path management are handled for you.

Looking at this code, a lot of other problems become apparent. There's a

distinct lack of composability in this code. You couldn't reuse much from the word count example. The mapper of the first job does the dual tasks of splitting sentences into words and filtering out stop words. You really would like to separate those tasks into separate conceptual units. Likewise, the reducer of the second job is doing a count aggregation, a sum aggregation, and then a division function to produce the average. All these operations could be represented as separate functions, yet it's not clear how to compose them together using the raw MapReduce API.

This code is also coupled to the types of data its dealing with as well as the file formats of the inputs and outputs. It's a lot of work to make the code generic. Things get much more complex when you try to do a complex operation like a join between two datasets.

Finally, imagine that you want to do more than one thing with an input dataset, like compute multiple views from one set of data. Each view requires its own sequence of MapReduce jobs, yet the sequences of jobs are independent and can be run in parallel. To do this with the raw MapReduce API, you'd have to manually spawn threads for each view and handle the coordination yourself. This quickly becomes a nightmare.

4.9 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 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. What makes MapReduce powerful is that it's general enough for computing nearly any function, it automatically scales your computations across a cluster of machines, and it executes your computations in a fault-tolerant way.

You also saw some examples for why MapReduce is a low level of abstraction, how it can be tedious to code in and make it difficult to reuse code. Using MapReduce directly leads to large, difficult to maintain codebases that greatly diminishes the productivity of your development team.

In this chapter you'll learn how to fight complexity and make data processing code simple and elegant. A key point is that your data processing code is no different than any other code you write. Like any other code, 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 look at 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:

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

The first thing to note is that this code is really concise! It's so much higher level than MapReduce it may be hard to believe that it's an interface to MapReduce, but when this is executed it runs as a MapReduce job. The query reads its input from a variable called "SENTENCE", which is a dataset where each record is a line from the Gettysburg address. The definition of "SENTENCE" looks like this (only a portion shown for brevity):

```
    public static List SENTENCE = Arrays.asList(
Arrays.asList("Four score and seven years ago our fathers " +
            "brought forth on this continent a new nation"),
Arrays.asList("conceived in Liberty and dedicated to the " +
            "proposition that all men are created equal"),
Arrays.asList("Now we are engaged in a great civil war testing " +
            "whether that nation or any nation so"),
Arrays.asList("conceived and so dedicated can long endure We are " +
            "met on a great battlefield of that war"),
```

As you can see, it's just an in-memory list of records. If you ran this code, it would print the results to standard output, showing something like this (only showing a portion of the output 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. The first line says "Execute the following computation and put the results into standard output". Outputs are defined via an abstraction called a *Tap* and can go anywhere. To modify this query to write the results to text files on HDFS at the "*data/results" directory, you would replace /new StdoutTap()* with *Api.hfsTextline("/data/results").*

The next 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.

The next line sources the input data into the query. It reads from the SENTENCE dataset and emits tuples containing one field called ?sentence. Like how outputs can be written anywhere, the inputs to JCascalog are specified via the same *Tap* abstraction. To read sentences from text files at "*data/sentences" on HDFS, you would replace SENTENCE with /Api.hfsTextline("/data/sentences").*

The next line splits each sentence into a set of words, giving the *Split* function the ?sentence field as input and storing the output in the ?word field. The Split operation is defined via the following class:

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

This definition should be pretty intuitive; it takes in input sentences and emits a new tuple for each word in the sentence. We'll discuss custom operations in greater depth later in this chapter.

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

This code really captures the essence of word count by boiling it down to the fundamental pieces. If every detail isn't completely clear, don't worry. We'll be going through JCascalog in much greater depth later in the chapter.

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

Like any other code, keeping your data processing code simple is essential so that you can reason about your system and achieve correctness. Let's take a look at three common sources of complexity in data processing code: an inability to modularize due to the performance cost, custom languages, and abstractions that don't compose well.

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 functions, it's easier to understand each piece in isolation and reuse functionality as needed. In order to use modularization techniques effectively, it's critical that the act of modularizing your code not incur an unreasonable performance cost.

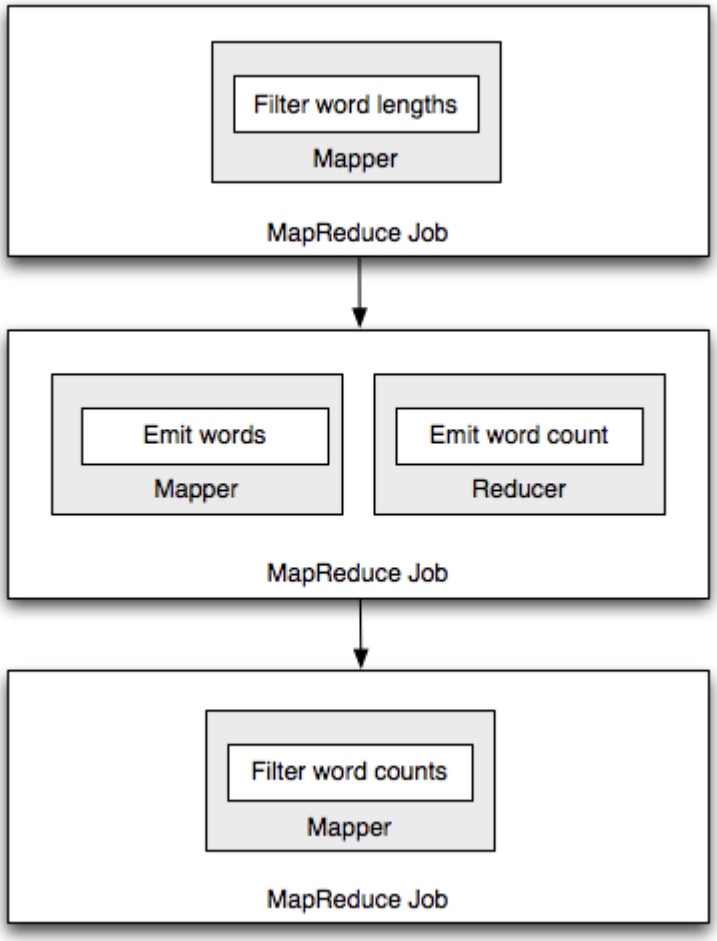


Figure 5.1 Modularized MapReduce jobs

One of the reasons why using the MapReduce API directly is such a bad idea is

because it's hard to modularize without incurring a major performance penalty. Suppose, for example, you want to do 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, you would treat each transformation independently and write them as their own MapReduce job, as shown in Figure 5.1. However, each MapReduce jobs is expensive: each job involves launching tasks, reading data to and from disk, and streaming data over the network. To maximize performance, you need to execute your code in as few Mapreduce jobs as possible.

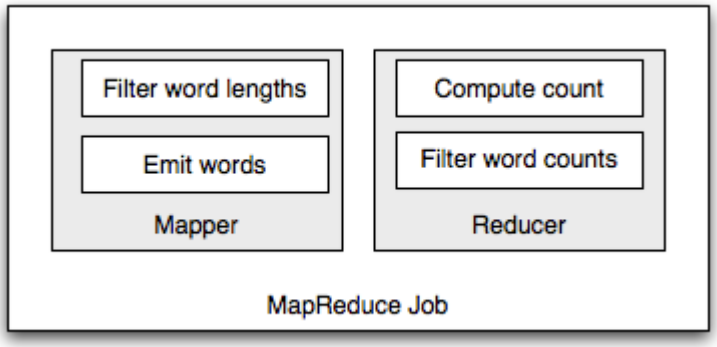


Figure 5.2 Compiling modularized functionality into as few MapReduce jobs as possible

A high level abstraction disassociates the specification of your computation from how it executes. Rather than run each portion of your computation as its own job, a high level abstraction compiles your computation into as few MapReduce jobs as possible, packing in different functions into the same Mapper or Reducer as much as possible. For example, the word count variant just described would execute as one MapReduce job, as shown in Figure 5.2.

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, does introduce a number of serious complexity problems.

A well-known symptom of some of the complexity incurred by using a custom language for data processing are SQL injection attacks. SQL injection attacks are a major problem, with high profile instances of it occurring every year. In 2011, for example, HBGary, a *technology security firm,* was breached via a SQL injection attack.

You have to ask yourself: why do SQL injection attacks occur? At a fundamental level, the cause is due to a programmer dynamically creating a SQL query from some general purpose language and the programmer failing to properly escape user input. SQL injection attacks occur due to the barrier of trying to use two independent languages together.

Of course, tools have been built to help avoid SQL injection attacks: parameterized SQL being the notable example. And if used properly, these tools do prevent attacks. However, at the end of the day, the tools exist and the attacks still happen. The language barrier is an unavoidable source of complexity, and you have to be constantly on your guard to make sure you don't make a mistake.

The barrier causes all sorts of other complexity issues. Modularization can become painful – if you're lucky the data processing language has facilities for namespacing and functions, but these are ultimately not going to be as good as what you have in your general purpose language. It's more work to insert your own business logic intro your queries: you have to go through a "User-defined function" (UDF) interface and register your objects with the language so that it can see them.

Another problem is the fact that you have to coordinate switching between your general purpose language and your data processing language. For instance, you may write a job using the data processing language that produces some output, and then you want to use the Pail class from Chapter 3 to append that data into an existing store. The Pail invocation is just regular Java code, so you have to write shell scripts that invoke things in the right order. Because you're working in multiple languages stitched together via scripts, basic things 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" – complexity that's not fundamental to the problem you're trying to solve but rather caused by the tools you're using. All of this complexity is avoided completely when your data processing tool is a library for your general purpose language. Then you can freely intermix regular code with data processing code, use your normal mechanisms for modularization, and have exceptions work properly. And 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 Abstractions that don't compose well

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 definition of the "Average" aggregator in Apache Pig (another abstraction for MapReduce). The implementation is over 300 lines of code and has 15 method definitions as part of its implementation. The reason why it's so complex is because it's a type of aggregator that can be optimized by doing part of the aggregation work in the map phase, so the implementation has to coordinate how that works. The implementation contains code for doing partial counts and sums in the mapper, and then more code to combine the partial counts and sums and finally divide them at the end.

The problem with Pig's definition of Average is that it's reimplementing the existing "Count" and "Sum" aggregators. Pig is unable to reuse the work that went into 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 ported to Average's versions as well. What you really want to do is define "Average" as the *composition* of a count aggregation, a sum aggregation, and the division function. In fact, this is exactly how you can define Average in JCascalog:

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

This definition of Average is as efficient as the Pig implementation since it's able to reuse the already optimized Count and Sum aggregators. Additionally, it's much simpler. Don't worry about fully understanding this code quite yet – we'll discuss it in depth later in the chapter. 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 JCascalog, a tool that avoids these sources of complexity.

5.3 Why JCascalog?

Recall that the goal of this book is to illustrate the concepts of Big Data, using specific tools to ground the concepts. We will not be doing a survey of all the Big Data tools available. That said, let's take a moment to understand why we will be looking at JCascalog to understand data processing concepts rather than another tool.

There are lots of tools that provide higher level interfaces to MapReduce, such as Pig, Hive, and Cascading. I, the author, used these tools for years. While these tools are massive improvements over using MapReduce directly, you do run into fundamental limitations with these tools in your ability to abstract and compose your data processing code. We've discussed some of these already. I wrote JCascalog specifically to resolve these limitations and enable new abstraction and composition techniques to reduce the complexity of batch processing.

If you're experienced with relational databases, JCascalog will seem to be both very different and very familiar at the same time. It will seem different because it's an API rather than a language and based on logic programming instead of SQL. It will seem familiar because concepts like declarative programming, joins, and aggregations are still there, albeit in a different packaging.

5.4 Basics of JCascalog

JCascalog is a Java library that exposes composable abstractions for expressing MapReduce computations. The JCascalog API is a form of "logic programming", where computations are expressed via logical constraints. Don't let that term scare you though! Logic programming is a very natural fit for this kind of processing and isn't hard to grasp. It fits very well with the data model you learned in Chapter 2. You learned about how data is the rawest information you have, and everything else is derived from that data. In that sense, you can think of data as the "axioms" in your system – information you hold to be true for no other reason than it exists – and your queries as logical deductions upon those axioms. Logic programming excels at expressing these kinds of logical deductions.

JCascalog is a declarative abstraction. Rather than describe how to create your output from your input, you instead describe your output in terms of your input and JCascalog determines the most efficient way to execute that via a series of MapReduce jobs.

A computation in JCascalog is called a "query". Queries read and write data from input and output "taps". "Taps" are an abstraction that know how to read from

and write to arbitrary data sources, whether in-memory, HDFS, or databases.

We will be using in-memory datasets to demonstrate JCascalog in the coming examples. The playground datasets are in the manning.example.Playground class in the source code bundle that comes with this book.

5.4.1 Data model

JCascalog works by manipulating and transforming sets of tuples. A tuple is a named list of values, where each value can be any type of object. A set of tuples has a "schema" associated with it which specifies how many fields are in each tuple and the name of each field. Figure 5.3 shows an example of a set of tuples. JCascalog's data model is similar to the "rows and columns" model of relational databases. A tuple is like a row, and a field is like a column.

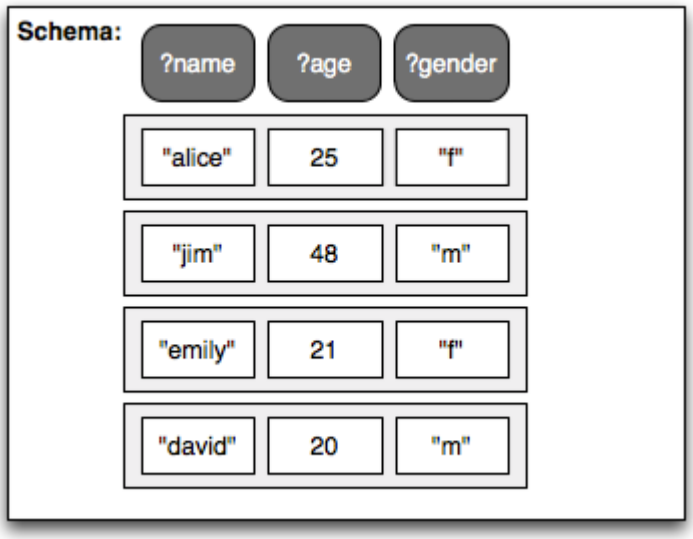


Figure 5.3 Set of tuples with schema

A "set of tuples" is represented in JCascalog in one of two ways:

1. A base set of tuples, such as an in-memory data structure or a set of files on HDFS.
2. A transformation of a set of tuples: this is called "subquery". Subqueries are not materialized until you execute them. For instance, the word count example in the beginning of this chapter contained a subquery that represented a set of [?word, ?count] tuples that are computed based on a set of [?sentence] tuples.

5.4.2 Structure of a query

JCascalog queries have a very uniform structure. Consider the following example which finds all people from the AGE dataset younger than 30:

```
new Subquery("?person")
```

```
    .predicate(AGE, "?person", "?age")
    .predicate(new LT(), "?age", 30);
```

The Api.execute function is used to execute a query. The first argument specifies what to do with the output, and the second argument is the "subquery" that defines the actual computation. A subquery has two pieces: first, it specifies the schema for the tuples it emits. This subquery emits one-tuples with one field called "?person". Next, it has a sequence of "predicates" which define and constrain those output variables.

Predicates are at the heart of JCascalog. Everything is represented via predicates: sourcing data, functions, filters, aggregations, joins, grouping, secondary sorting, and so on. Consider this example predicate:

```
.predicate(new Multiply(), 2, "?x").out("?z")
```

All predicates have the exact same structure. The first argument to a predicate is the "predicate operation". The predicate operation in this example is the "multiply" function. Next, you have a set of input fields and a set of output fields. The input fields are the input to your operation, and the output fields capture the output of your operation. So this example multiples the "?x" variable by 2 and stores the result in the "?z" variable.

Fields can be either constants or variables. Variables are any string prefixed with a "?" or "!". The only difference between the two types of variables is that "?" variables are not allowed to be null. So if Cascalog encounters a tuple where that variable would be set to null, it instead filters out that tuple.

Sometimes you don't need input fields for a predicate. For example, predicates that source data only have output fields. Here's the predicate that emits a "?sentence" variable from the SENTENCE dataset:

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

On the other hand, some predicates don't require any output fields. For example, here's a predicate that filters out all "?age" values that are less than 30:

```
.predicate(new LT(), "?age", 30)
```

When you only specify one set of fields, JCascalog knows whether they are input or output fields based on the type of the predicate operation specified. Since the SENTENCE dataset is a source of data, JCascalog knows the set of fields for that predicate are output fields. And since LT is a filter, JCascalog knows the set of fields for that predicate are input fields.

5.4.3 Predicates as constraints

While predicates look a lot like functions, with inputs and outputs, they have richer semantics than that. One way to look at predicates is as a set of constraints. Consider these three examples:

```
.predicate(new Plus(), 2, "?x").out(6)

.predicate(new Multiply(), 2, "?a").out("?z")
.predicate(new Multiply(), 3, "?b").out("?z")

.predicate(new Multiply(), "?x", "?x").out("?x")
```

The first predicate can be read as "When you add two to ?x you should get six back". So even though this function predicate takes in ?x as an input field, it's really acting as a constraint on ?x. It will only keep tuples for which ?x is equal to 4.

The second example contains two predicates, which can be read as "When you multiply ?a by 2 you should get some value ?z, and when you multiply ?b by 3 you get the same value ?z back". Again, even though ?a and ?b are inputs to these predicates these two predicates create a constraint on ?a and ?b, only keeping tuples where 2 * ?a = 3 * ?b.

Finally, the last example says "When you multiply ?x by itself, you should get the same value ?x back". This will only be true when ?x is 0 or 1, so all other values of ?x are filtered out.

There are four main types of predicates in JCascalog: functions, filters, aggregators, and generators.

A function specifies a constraint between a set of inputs fields and a set of output fields. Addition and multiplication are examples of this kind of predicates.

A filter specifies a constraint on a set of inputs fields. The "less than" and "greater than" operations are examples of this kind of predicate.

An aggregator is a function on a group of tuples. Example aggregators are "count", which emits a single value representing the number of tuples in the group,

or "sum", which adds together one particular field across all tuples and emits a single value representing the sum as the result.

Finally, a generator is a finite set of tuples. A generator can either come from a concrete source of data, like an in-memory data structure or files on HDFS, or it can be another subquery.

As you can see, all predicates, whether function, filter, aggregator, or generator, look exactly the same. Being able to represent every piece of your computation via the same simple, consistent mechanism is the key to enabling highly composable abstractions, as we'll explore more later on.

Predicates and subqueries are the basic concepts of JCascalog. Let's see how these concepts fit together for representing arbitrary computations by going through a sequence of examples. Figure 5.4 shows excerpts from some of the in-memory datasets that will be used in the upcoming examples.

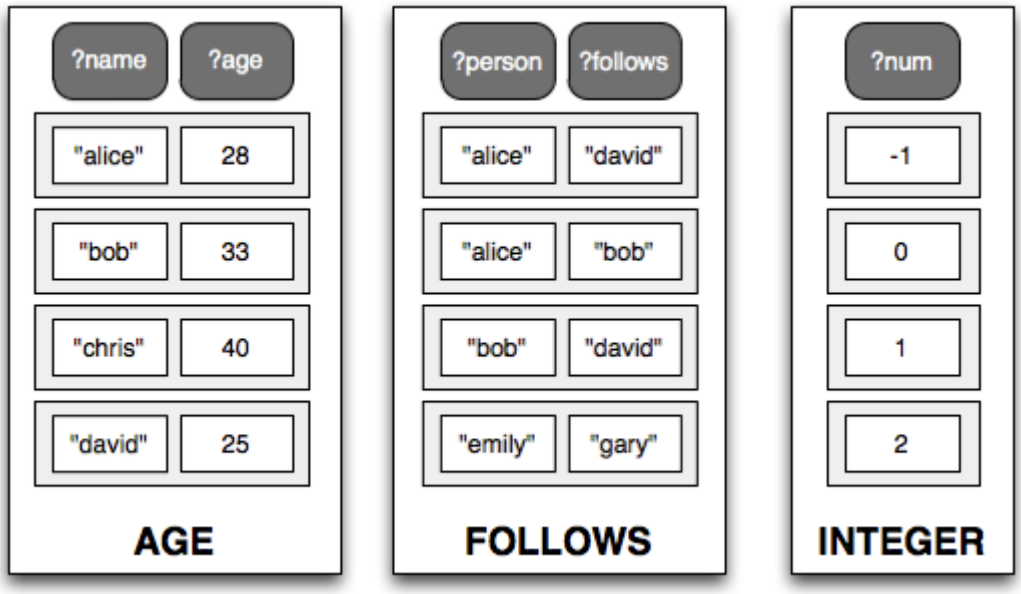


Figure 5.4 Excerpts from playground datasets

This first query finds all people from the AGE dataset:

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

The query emits tuples with one field called ?person. The query binds ?person to the first field of each tuple emitted by AGE and binds ?age to the second field of each tuple emitted by AGE. ?person and ?age here represent every tuple in the AGE dataset, so any operations using those fields will be applied to each AGE

tuple. In this case, no operations are done, and since only ?person is defined as an output variable for the query, all that happens is the ?age field is dropped from the output.

The next query finds all people from the AGE dataset who are 25:

```
new Subquery("?person")
    .predicate(AGE, "?person", 25);
```

This query is nearly the same as the last query, except instead of binding the second field of AGE to a variable a constant is placed there instead. When a constant is placed in the position of the output of a generator or operation, it acts as a filter. So in this case, any ?person who is not paired with an age equal to 25 in the AGE dataset is filtered out. Being able to use constants as filters is very convenient.

The next query finds all people from the AGE dataset who are less than 30 years old.

```
new Subquery("?person")
    .predicate(AGE, "?person", "?age")
    .predicate(new LT(), "?age", 30);
```

Unlike the past two examples, this example contains two predicates. The first predicate is the same generator you've seen before, binding the ?person and ?age variables to the contents of the AGE dataset. The second predicate applies the LT filter (which stands for "less than") to the arguments ?age and 30, filtering out any tuples whose ?age value is greater than or equal to 30. LT is a useful predicate operation that comes built in to JCascalog – you'll see later how to define your own predicate operations.

The next query is the same as the last, except it emits the ?age value along with each ?person:

```
new Subquery("?person", "?age")
    .predicate(AGE, "?person", "?age")
    .predicate(new LT(), "?age", 30);
```

So while the last query emitted 1-tuples, this query emits 2-tuples.

The next query emits all people from the AGE dataset along with the double of their age:

```
new Subquery("?person", "?double-age")
    .predicate(AGE, "?person", "?age")
    .predicate(new Multiply(), "?age", 2).out("?double-age");
```

This is an example of a function predicate, which defines a relationship between a set of input fields and a set of output fields. Here ?double-age is bound to the multiplication of ?age and the constant 2.

The next query is slightly more involved than the next query, emitting every person from the AGE dataset along with their age doubled and incremented:

```
new Subquery("?person", "?double-age-plus-1")
    .predicate(AGE, "?person", "?age")
    .predicate(new Plus(), "?double-age", 1)
        .out("?double-age-plus-1")
    .predicate(new Multiply(), "?age", 2).out("?double-age");
```

?double-age-plus-1 is bound to be one plus ?double-age, and ?double-age is bound to be twice the ?age value. Note that the ordering of predicates in JCascalog queries doesn't matter. JCascalog chooses the ordering based on which variables have been defined so far. So in this query, JCascalog can't apply the Plus predicate until ?double-age has been defined, which is dependent on the Multiply predicate being run. JCascalog detects these dependencies between variables and orders the execution of predicates accordingly.

The next query finds all values from the INTEGER dataset which when multiplied by themselves, equal themselves.

```
new Subquery("?n")
    .predicate(INTEGER, "?n")
    .predicate(new Multiply(), "?n", "?n").out("?n");
```

This query, of course, only emits the values 0 and 1.

Predicate operations can take variable numbers of input fields, as demonstrated by the next query which emits all values from the INTEGER dataset which equal themselves when cubed:

```
new Subquery("?n")
    .predicate(INTEGER, "?n")
    .predicate(new Multiply(), "?n", "?n", "?n").out("?n");
```

?n is passed into the Multiply operation here three times, as opposed to the last query which passed it in twice.

5.4.4 Combining independent datasets

Many queries require you to combine multiple datasets together. There are a few ways to combine datasets. One of the most common ways is with a *join*. *Joins* are common with query languages for relational databases and exist in JCascalog as well.

Suppose you have an AGE dataset, like we've discussed, and a GENDER dataset containing person names and their genders. Now suppose you want to create a new set of tuples which contains the age and gender for all people who exist in both the AGE and GENDER datasets. This is called an *inner join*. An *inner join* combines tuples from each dataset which have matching values for the *join fields*. So doing an inner join of AGE and GENDER on the ?name field for each dataset produces the output shown in Figure 5.5.

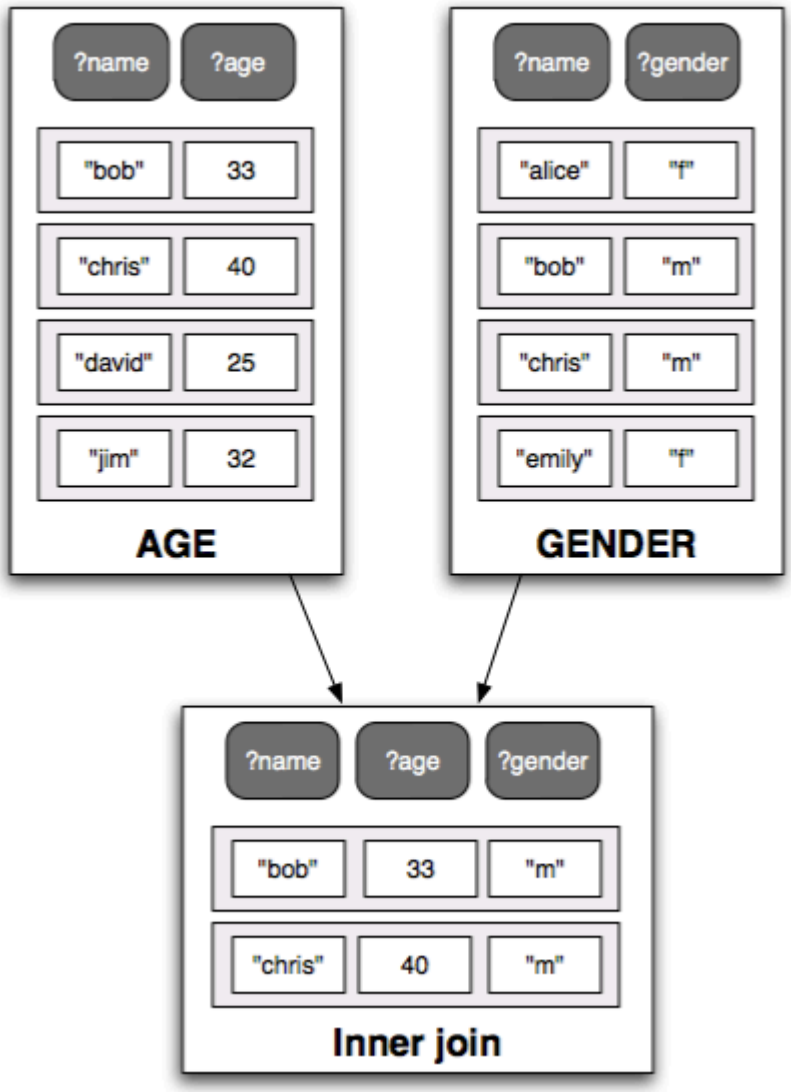


Figure 5.5 Inner join of AGE and GENDER datasets

Note that the output dataset only contains results for people who exist in both the AGE and GENDER datasets.

In a language like SQL, joins are expressed *explicitly*. With SQL doing the join from Figure 5.5 might look something like:

```
SELECT AGE.person, AGE.age, GENDER.gender
FROM AGE
INNER JOIN GENDER
ON AGE.person=GENDER.person
```

In JCascalog, joins are implicit based on the variable names. The above SQL query looks like the following in JCascalog:

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

JCascalog looks at this query and sees that the same variable ?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.

Here's another example of an inner join in JCascalog. This query finds all the people that "emily" follows who are male:

```
new Subquery("?person")
    .predicate(FOLLOWS, "emily", "?person")
    .predicate(GENDER, "?person", "m");
```

The implicit join along with the ability to use constants as filters makes this query read very naturally.

Inner joins only emit tuples for join fields which exist across all sides of the join. So for the age+gender query, you'll only get back people who exist in both datasets. But you may want results for people who don't exist in one dataset or the other, getting a null value for the non-existent age or gender. This is called an "outer join", and is just as easy to do in JCascalog. Consider these examples:

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

JCascalog treats variables beginning with "!!" specially. Those variables are set to null during joins when tuples from that dataset don't match the join fields from the dataset being joined against. In the first example, if a person has an age but not a gender, they will still have a record emitted – their gender will be set to null. But since ?age is still a normal variable, People with genders but no age will be filtered out.

In the second example, any person in either the AGE or GENDER dataset will have a tuple emitted, since neither !!age nor !!gender require matching against tuples from the other dataset.

Here's a more sophisticated usage of a join. This query finds all people who follow someone younger than themselves:

```
new Subquery("?person1", "?person2")
    .predicate(FOLLOWS, "?person1", "?person2")
    .predicate(AGE, "?person1", "?age1")
    .predicate(AGE, "?person2", "?age2")
    .predicate(new LT(), "?age2", "?age1");
```

This query joins the AGE dataset against the FOLLOWS dataset twice, once for each side of the follows relationship. The LT predicate is applied once the joins complete to filter out any follows where someone follows someone of the same age or older.

Besides joins, there's a few other ways to combine datasets. Sometimes you have two datasets which represent the same thing and you want to combine them into the same dataset. For this, JCascalog provides the "combine" and "union" functions. "combine" just concatenates the datasets together, whereas "union" will remove any duplicate records during the combining process. Figure 5.6 illustrates the difference between the two functions.

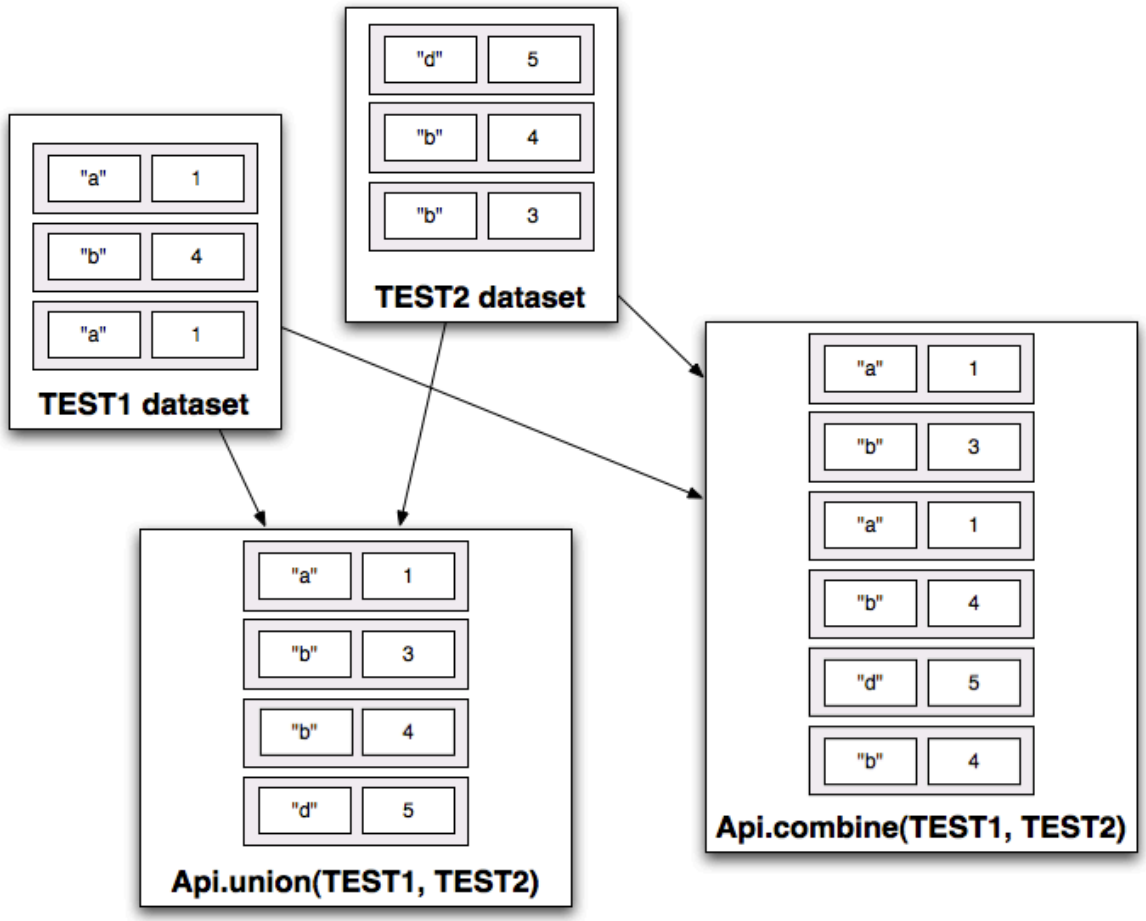


Figure 5.6 Union and combine functions

5.4.5 Aggregation

So far you've seen how to do transformations that look at one tuple at a time or combine datasets together. What's left is aggregations – operations that look at groups of tuples at a time.

Here are two examples of aggregations. The first computes the total number of people less than 30 years old, and the second computes the number of people each person follows:

```
new Subquery("?count")
        .predicate(AGE, "_", "?age")
        .predicate(new LT(), "?age", 30)
        .predicate(new Count(), "?count");
new Subquery("?person", "?count")
        .predicate(FOLLOWS, "?person", "_")
        .predicate(new Count(), "?count");
```

Unlike SQL, there's no explicit "GROUP BY" command to indicate how to split up the tuples for aggregation. Instead, the grouping is implicit based on the variable names. To understand this, let's look at the phases of execution of a

JCascalog query.

We'll use the following query in order to explain how the execution of a JCascalog subquery works. Then we'll come back to the above aggregation examples. At each step of execution you'll see how the sets of tuples change.

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

At the start of the execution of a query are a set of datasets to transform. The datasets are represented by the *generator* predicates in the subquery. This query contains two generator predicates, VAL1 and VAL2. Suppose the VAL1 and VAL2 datasets contain the data shown in Figure 5.7.

Figure 5.7 Start of query plan

At the start of a JCascalog query, the generator datasets exist in independent branches of the computation.

In the first phase of execution, JCascalog applies functions, applies filters, and joins together datasets until it can no longer do so. A function or filter can be applied to a branch if all the input variables for the operation are are available in that branch. For example, the *Multiply* predicate can be applied to the VAL1 generator because the generator contains the ?b variable. After the function is

applied, that branch of execution has the ?b and ?double-b variables available, as indicated in Figure 5.8. However, the LT predicate cannot be applied, since it requires both the ?b and ?c variables to be available in the same branch.

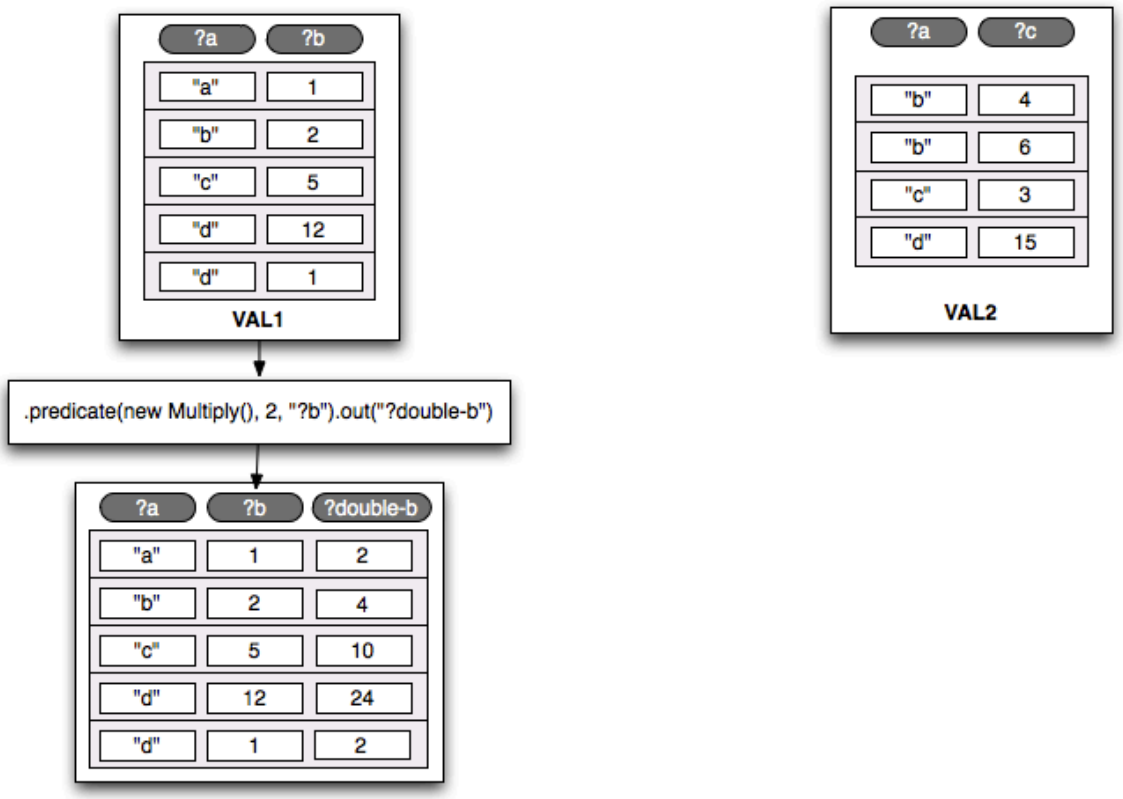


Figure 5.8 Applying function to a branch of query plan

A join between two branches is possible if they have any intersecting variables. When the two branches join, the resulting dataset contains all variables from both sides of the join. Once the VAL1 and VAL2 branches join together in this example, all the variables needed to apply the LT predicate are available, and the LT predicate can be applied. This is shown in Figure 5.9.

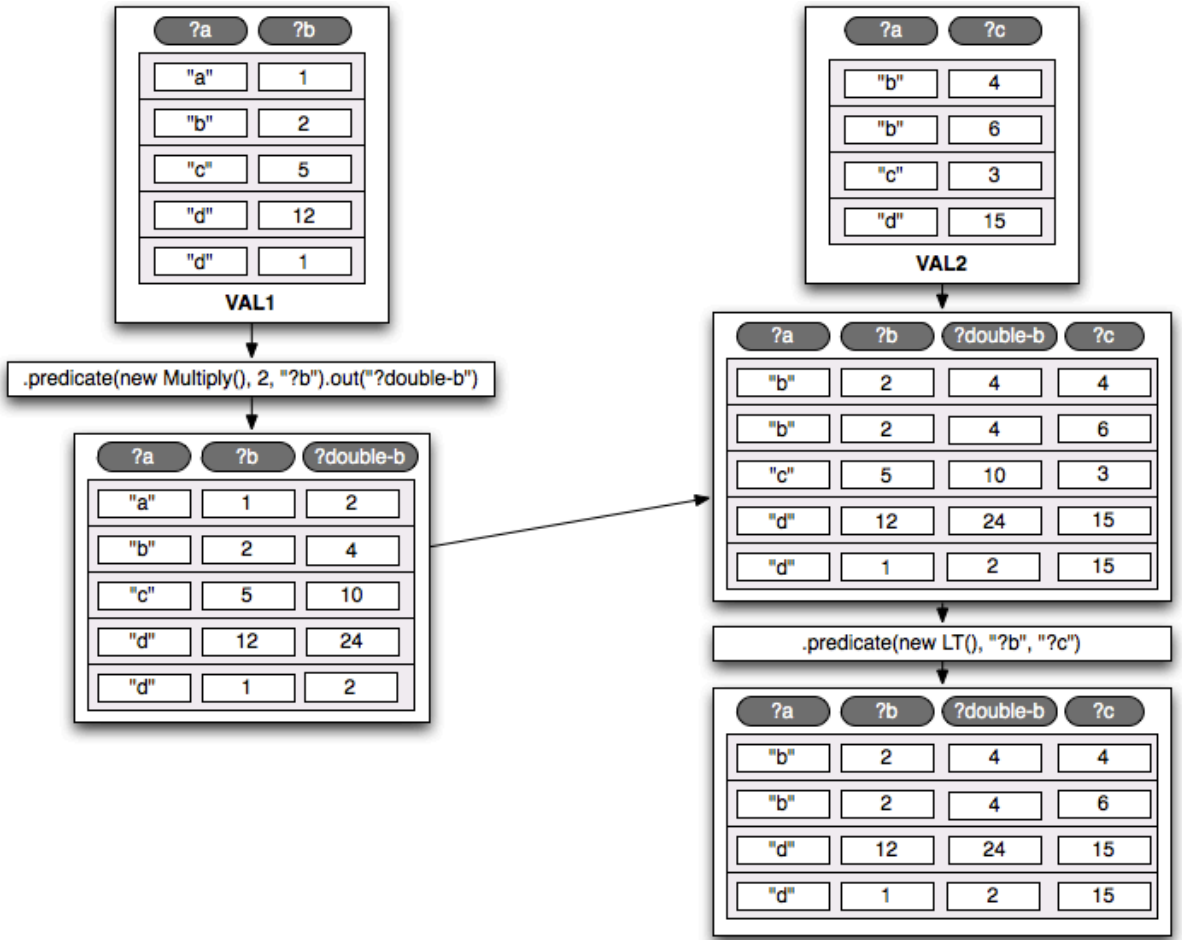


Figure 5.9 Join and applying a filter

Eventually the phase reaches a point where no more predicates can be applied because the predicates left are either aggregators or require variables that don't exist yet in the branch. 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. In this example, the only variable that matches this constraint is the ?a variable, so JCascalog groups the dataset by ?a. The splitting up of tuples into groups is shown in Figure 5.10. If none of the available variables have been declared as output variables, JCascalog groups all tuples into a single group.

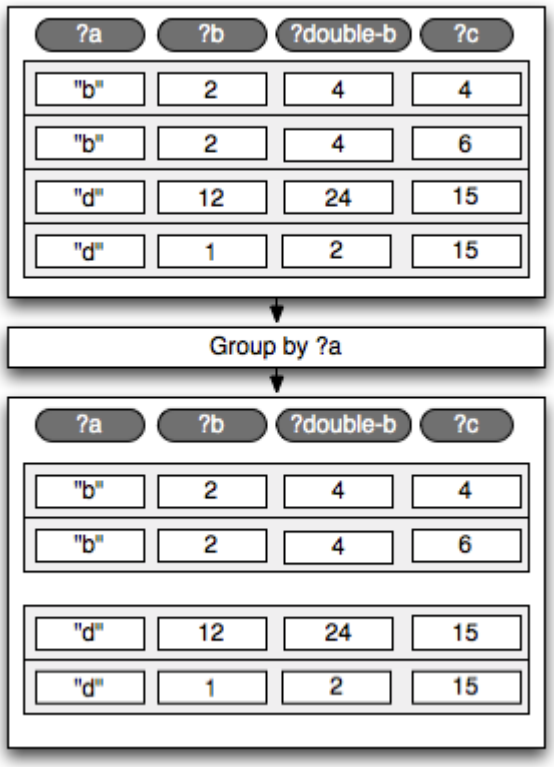


Figure 5.10 Grouping of tuples during aggregation phase

Next, JCascalog applies the aggregators to each group of tuples. In this case, the Count and Sum aggregators are applied to each group. The Count predicate emits the number of tuples in each group, and the Sum predicate emits the Sum of the ?double-b field for each group. The result of applying the aggregators to the grouped tuples is shown in Figure 5.11.

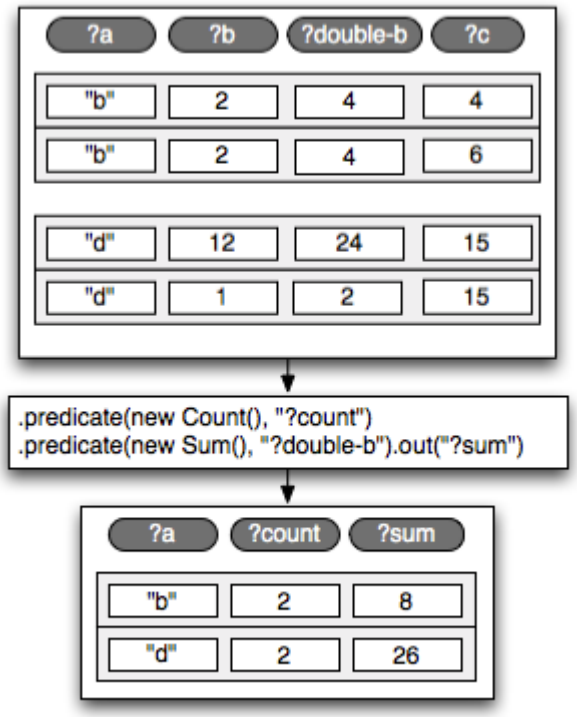


Figure 5.11 Aggregation phase

The final phase is the post-aggregation phase, in which all remaining functions and filters are applied to the one remaining branch of execution. The end of this phase drops any variables from the tuples that are not declared in the output fields for the query. The transformation of the branch of execution from the end of the aggregation phase to the end of the query is shown in Figure 5.12.

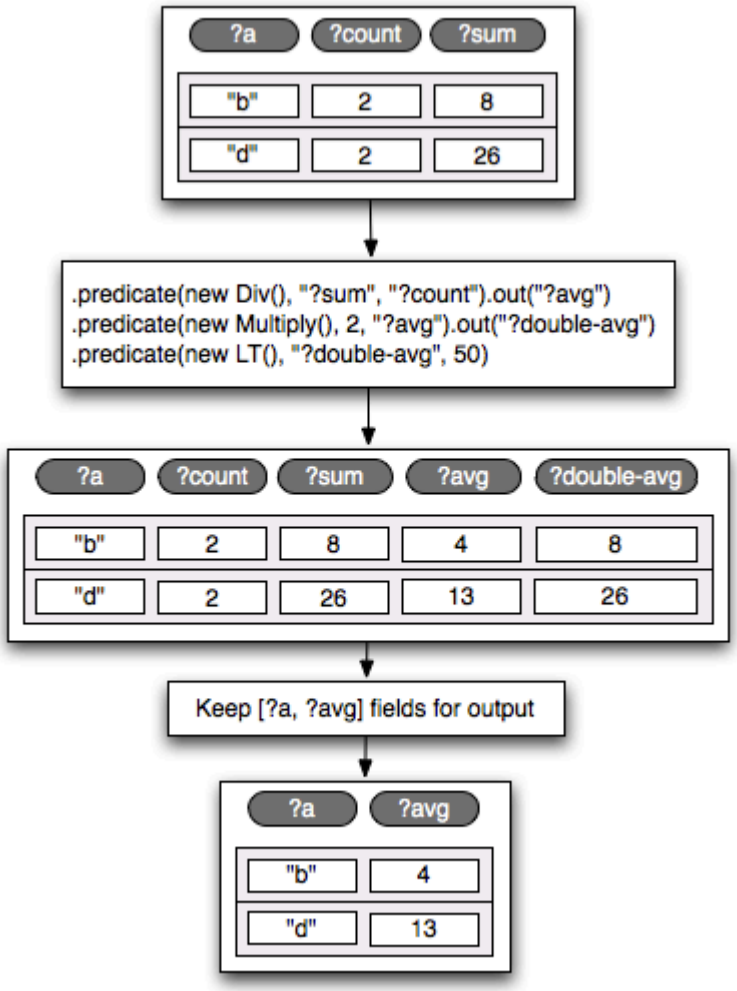


Figure 5.12 Post-aggregation phase

You may have noticed that this example computes an average by doing a count, sum, and division. This was just for the purposes of illustration – these operations can be abstracted into an "Average" aggregator, as we glanced at earlier in this chapter and will look at more deeply later on. You may have also noticed that some variables are never used after a point yet still remain in the branches of execution. For example, the ?b variable is not used after the LT predicate is applied, and 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. This is important for performance.

Lets look back at the first two aggregation examples. The first finds the number of people in the AGE dataset less than 30 years old:

```
new Subquery("?count")
      .predicate(AGE, "_", "?age")
      .predicate(new LT(), "?age", 30)
      .predicate(new Count(), "?count");
```

In the first phase of execution, the age dataset tuples are filtered down to only the tuples with ?age less than 30. The "_" field indicates to ignore that field of AGE since it's never used. Then, the aggregation phase begins. Since the only declared output variable is ?count, and doesn't yet exist, JCascalog treats all tuples as a single group. The Count aggregator is then applied against this single group, producing the number of people less than 30 years old.

The second example finds the number of people each person follows.

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

The first phase of execution is extremely simple, as there is only a single generator and no operations. The result of the first phase is just the contents of the FOLLOWS dataset. The aggregation phase then begins. The dataset is grouped by ?person since it's a declared output variable and is available. The Count aggregator is then applied to each group, producing the number of people each ?person follows.

Next let's look at how you do more complex queries which require multiple subqueries.

5.4.6 Layering subqueries

Many computations require more than one subquery to specify. One of the most powerful features of subqueries is that they act just like any other source of data, so they can be used in other subqueries as if you were reading an in-memory dataset or files from HDFS.

Let's look at an example. Here's a query that finds all the records from the FOLLOWS dataset where each person in the record follows more than 2 people:

```
Subquery manyFollows =
        new Subquery("?person")
            .predicate(FOLLOWS, "?person", "_")
            .predicate(new Count(), "?count")
            .predicate(new GT(), "?count", 2);
Api.execute(new StdoutTap(),
  new Subquery("?person1", "?person2")
    .predicate(manyFollows, "?person1")
    .predicate(manyFollows, "?person2")
    .predicate(FOLLOWS, "?person1", "?person2"));
```

The *manyFollows* variable is assigned a subquery that represents all people who follow more than two people. The next query makes use of *manyFollows* by

joining against it twice, once for each side of each FOLLOWS record.

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")
            .predicate(SENTENCE, "?sentence")
            .predicate(new Split(), "?sentence").out("?word")
            .predicate(new Count(), "?count");
Api.execute(new StdoutTap(),
    new Subquery("?count", "?num-words")
        .predicate(wordCount, "?word", "?count")
        .predicate(new Count(), "?num-words"));
```

The query groups by ?word to compute the ?count, and then it groups by ?count to compute ?num-words.

Subqueries are lazy – nothing is computed until Api.execute is called. So in the previous example, no jobs are launched until the Api.execute call.

Subqueries are the basic unit of abstraction in JCascalog. They represent an arbitrary view on any number of sources of data. Like how you decompose a large program into many functions, you decompose large queries into many subqueries.

5.4.7 Custom predicate operations

You've seen how to use predicates to construct arbitrarily complex queries that filter, join, transform, and aggregate your data. While JCascalog has lots of useful built-in predicate operations, you'll usually have lots of custom predicate operations containing your business logic. This is the last major piece of JCascalog to learn. JCascalog exposes interfaces for defining new filters, functions, and aggregators.

FILTERS

Let's start with filters. A filter contains one method called "isKeep" that returns true if the input tuple should be kept and false if it should be filtered out. For example, here's a filter that keeps all tuples where its input is greater than 10:

```
public static class Greater10Filter extends CascalogFilter {
    public boolean isKeep(FlowProcess process, FilterCall call) {
        return call.getArguments().getInteger(0) > 10;
    }
}
```

The input arguments are obtained via the *FilterCall#getArguments* method and

contains the input fields specified when the predicate is used.

FUNCTIONS

Next up are functions. Like a filter, a function takes in a set of input arguments. A function then emits zero or more tuples as output. Here's a simple function that increments its input value by one:

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

The function simply takes the input field, adds one to it, and emits the new value as a new tuple. Figure 5.13 shows the result of applying this function to a set of tuples.

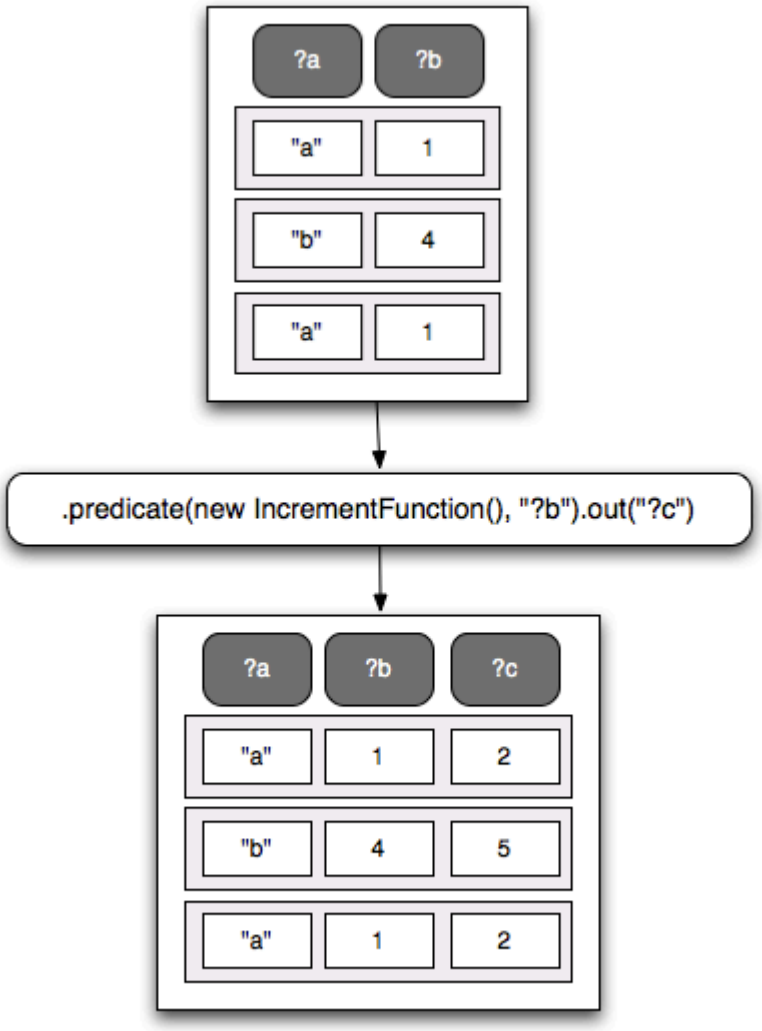


Figure 5.13 Applying a function to a set of tuples

If a function emits zero tuples, it filters out the input tuple. For example, here's a function that tries to parse an integer from a string, and filters out the tuple if it fails to parse the number:

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

Figure 5.14 illustrates what happens when this function is run on a set of tuples. You can see that "aaa" gets filtered out.

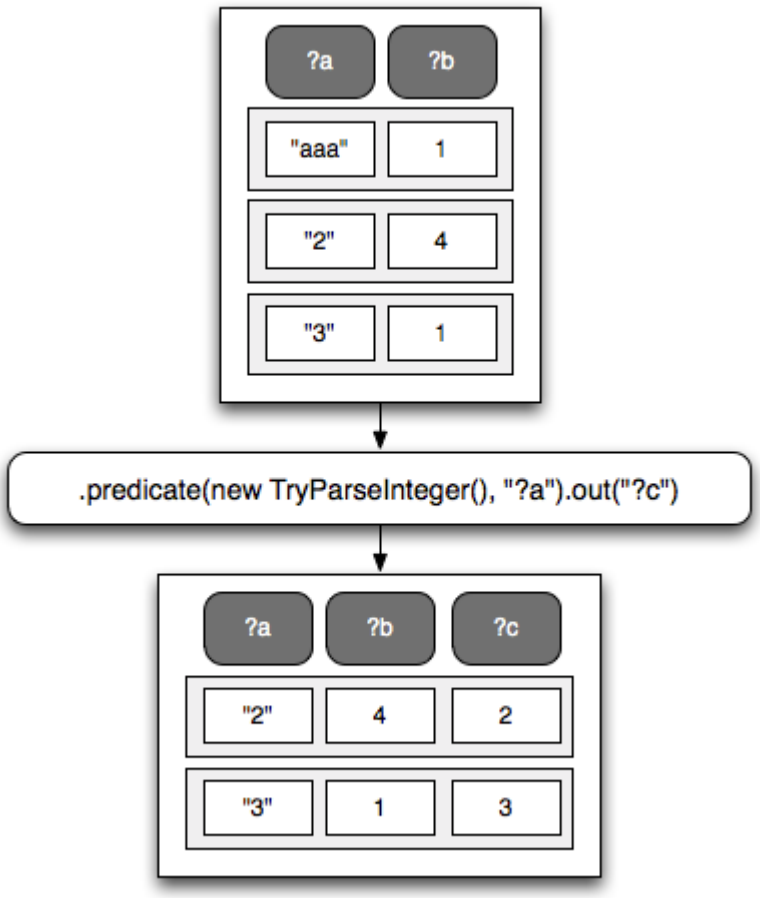


Figure 5.14 A function that also acts as a filter

Finally, if a function emits multiple output tuples, each output tuple is appended to its own copy of the input arguments. For 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(" ")) {
            call.getOutputCollector().add(new Tuple(word));
        }
    }
}
```

Figure 5.15 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. Though in word count, JCascalog will immediately drop the sentence field after splitting because it is no longer needed for the rest of the query.

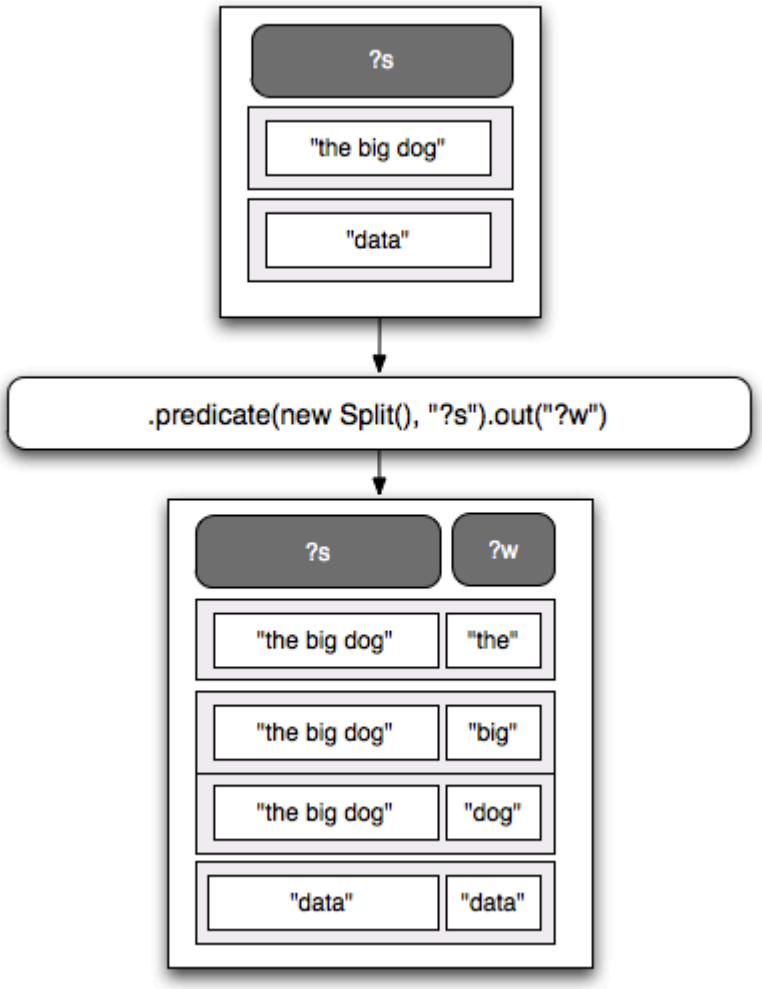


Figure 5.15 A function that that emits multiple tuples

AGGREGATORS

The last class of predicate operations are aggregators. There are three variants of aggregators you can define, with different properties around composition and performance.

The first kind of aggregator is called an *Aggregator*. An *Aggregator* looks at one tuple at a time for each tuple in a group, accumulating some state for the group. Here's how you would implement the sum aggregator as an *Aggregator*:

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

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

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

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

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

The code simply iterates through each input tuple, accumulating the sum of the first fields.

Buffers are easier to write than aggregators, since you only have to deal with one method rather than three methods. 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 buffers or aggregators, but Aggregators can be used along with other Aggregators. In the following code sample, the first subquery is invalid because SumBuffer cannot be used with other aggregators. However, the second query is valid since SumAggregator can be used with other aggregators.

```
Subquery invalidQuery =
        new Subquery("?sum", "?count")
            .predicate(INTEGER, "?n")
            .predicate(new SumBuffer(), "?n").out("?sum")
            .predicate(new Count(), "?count");
Subquery validQuery =
        new Subquery("?sum", "?count")
            .predicate(INTEGER, "?n")
            .predicate(new SumAggregator(), "?n").out("?sum")
            .predicate(new Count(), "?count");
```

Buffers and aggregators work by getting all the tuples for a group together on the same machine and running the aggregation operation on them. Different groups of tuples can be processed by different machines, so the aggregation distributes the processing of groups across the cluster. So the aggregation has two phases:

1. Transfer tuples across the network to group the tuples
2. Run aggregation function on each group

Figure 5.16 shows what the processing looks like for the SumAggregator at the MapReduce level.

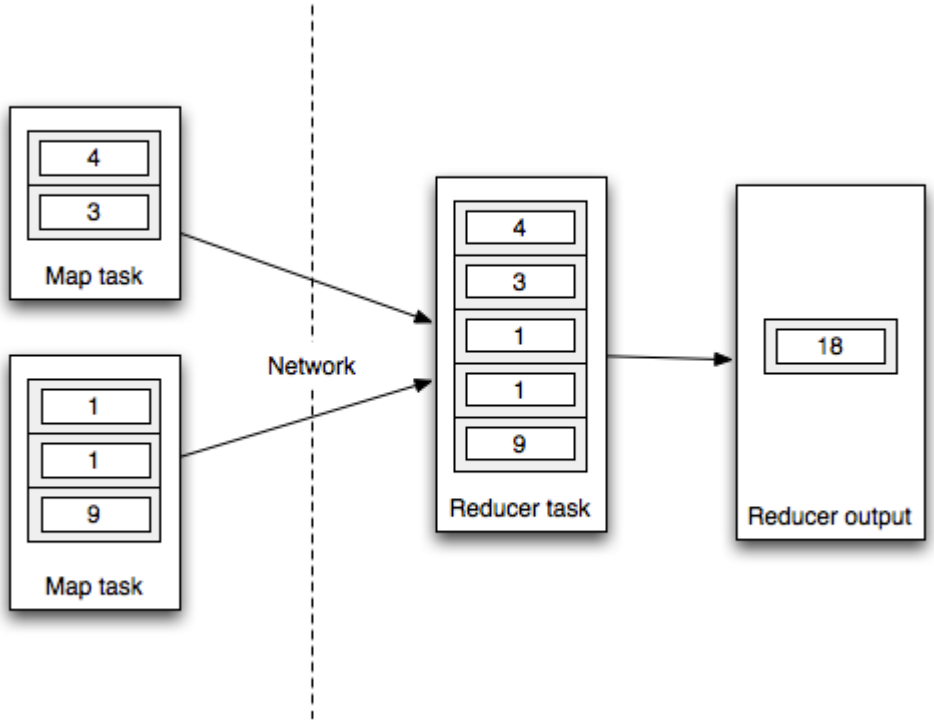


Figure 5.16 Execution of aggregators and buffers at MapReduce level

Now suppose you wanted to run the sum aggregator on the entire set of tuples as one group. Since there's only one group, this is going to be extremely inefficient and unscalable. All the tuples will have to be sent to a single machine for aggregation, defeating the purpose of using a parallel computation system.

Fortunately, there's one more type of aggregator operation that can do global aggregations like this scalably and efficiently. A *parallel aggregator* builds up an aggregation incrementally by doing partial aggregations before centralizing the computation on a single machine. Let's go through an example to understand how this works. Here's are the definitions of Sum and Count as parallel aggregators:

```
public static class SumParallel implements ParallelAgg {
    public void prepare(FlowProcess process, OperationCall call) {
    }
    public List<Object> init(List<Object> input) {
        return input;
    }
    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));
    }
}
public static class CountParallel implements ParallelAgg {
    public void prepare(FlowProcess process, OperationCall call) {
```

```
    }
    public List<Object> init(List<Object> input) {
        return Arrays.asList((Object) 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));
    }
}
```

Parallel aggregators are defined as two functions. The "init" function maps the arguments from a single tuple to a partial aggregation for that tuple. For Sum, the partial aggregation of a single tuple is simply the value in the argument, whereas for Count the partial aggregation of a single tuple is the value "1" (since the count of a single tuple is 1).

The "combine" function specifies how to combine two partial aggregations into a single aggregation value. For both Sum and Count this is just the addition operation.

When an aggregator is defined as a parallel aggregator, it executes much more efficiently. Figure 5.17 illustrates how a global sum aggregation is performed on a set of tuples. The tuples on each mapper task are reduced to a single partial aggregation value, whereupon there is very little work left to transfer the partial aggregations to a single machine and compute the aggregation result.

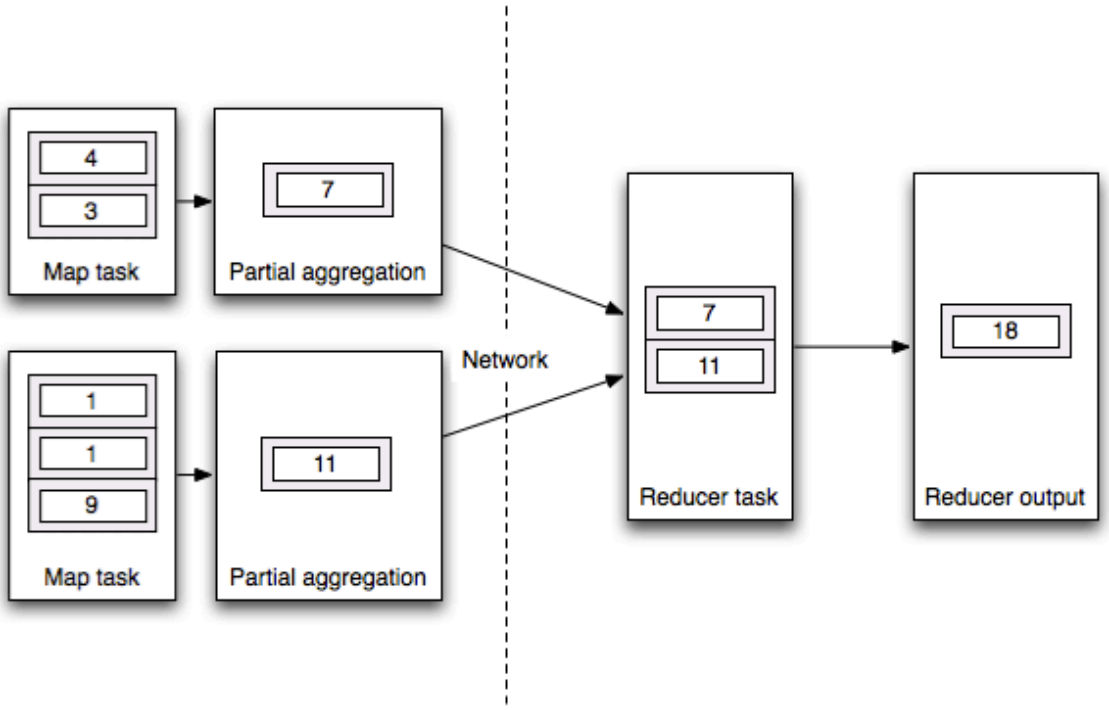


Figure 5.17 Execution of parallel aggregation at MapReduce level

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

5.5 Composition

There are three main composition techniques we'll look at: predicate macros, functions that return dynamically created subqueries, and functions that return dynamically created predicate macros. These techniques take advantage of the fact that there's no barrier between the query tool and your general purpose programming language. You're able to manipulate your queries in a very fine-grained way.

5.5.1 Predicate macros

A predicate macro is a predicate operation that expands to another set of predicates. You've already seen one example of a predicate macro in the definition of Average from the beginning of this chapter. Let's take a look at that definition again:

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

The first line of the definition says that this operation takes in one variable called ?val as input, and produces one variable called ?avg as output. Average then consists of three predicates composed together: a count aggregation, a sum aggregation, and a division function. The intermediate variables ?count and ?sum are used to store the results of the aggregation before dividing them to compute the result.

Here's an example usage of Average:

```
new Subquery("?result")
      .predicate(INTEGER, "?n")
      .predicate(Average, "?n").out("?result");
```

When JCascalog sees that a predicate is a predicate macro, it expands it into its constituent predicates, like so:

```
new Subquery("?result")
      .predicate(INTEGER, "?n")
      .predicate(new Count(), "?count_gen1")
      .predicate(new Sum(), "?n").out("?sum_gen2")
      .predicate(new Div(), "?sum_gen2", "?count_gen1", "?result");
```

You can see that the ?n and ?result variables from the Subquery replace the ?val and ?avg variables used in definition of Average. And the ?count and ?sum intermediate vars are given unique names to guarantee they don't conflict with any other variables used in the surrounding Subquery.

This definition of Average is done as a PredicateMacroTemplate, which makes it easy to specify simple compositions like Average by specifying the composition as a template. Not everything can be specified with a template, however. For example, suppose you wanted to specify an aggregator that computes the unique count of a set of variables, like so:

```
new Subquery("?unique-followers-count")
      .predicate(FOLLOWS, "?person", "_")
      .predicate(new DistinctCount(), "?person")
          .out("?unique-followers-count");
```

Now suppose that you want this aggregator to work even if the number of tuples for a group is extremely large, large enough that it would cause memory issues to keep an in-memory set to compute the unique count. What you can do is make use of a feature called "secondary sorting" that can sort your group before it's given as input to your aggregator function. Then, to compute the distinct count, your code just needs to keep track of the last tuple seen in the group to determine whether or not to increment the count or not. The code to do the sorting and aggregation looks like this:

```
public static class DistinctCountAgg extends CascalogAggregator {
    static class State {
        int count = 0;
        Tuple last = null;
    }

    public void start(FlowProcess process, AggregatorCall call) {
        call.setContext(new State());
    }

    public void aggregate(FlowProcess process, AggregatorCall call) {
        State s = (State) call.getContext();
        Tuple t = call.getArguments().getTupleCopy();
        if(s.last==null || !s.last.equals(t)) {
            s.count++;
        }
        s.last = t;
    }

    public void complete(FlowProcess process, AggregatorCall call) {
        State s = (State) call.getContext();
        call.getOutputCollector().add(new Tuple(s.count));
    }
}

public static Subquery distinctCountManual() {
    return new Subquery("?unique-followers-count")
          .predicate(FOLLOWS, "?person", "_")
          .predicate(Option.SORT, "?person")
          .predicate(new DistinctCountAgg(), "?person")
              .out("?unique-followers-count");
}
```

DistinctCountAgg contains the logic to compute the unique count given a

sorted input, and the Option.SORT predicate tells JCascalog how to sort the tuples for each group.

Of course, you don't want to have to specify the sort and aggregator each time you want to do a distinct count. What you want to do is compose these predicates together into a single abstraction. However, you can't do this with a PredicateMacroTemplate, like we used for Average, since any number of variables could be used for the distinct count.

The most general form of a predicate macro is as a function that takes in a list of input fields, a list of output fields, and returns a set of predicates. Here's the definition of DistinctCount as a regular PredicateMacro:

```
public static class DistinctCount implements PredicateMacro {
    public List<Predicate> getPredicates(Fields inFields,
            Fields outFields) {
        List<Predicate> ret = new ArrayList<Predicate>();
        ret.add(new Predicate(Option.SORT, inFields));
        ret.add(new Predicate(new DistinctCountAgg(),
                              inFields, outFields));
        return ret;
    }
}
```

The input fields and output fields come from what is specified when the PredicateMacro is used within a subquery.

Predicate macros are powerful because everything in JCascalog is represented uniformly as predicates. Predicate macros can thus arbitrarily compose predicates together, whether they're aggregators, secondary sorts, or functions.

5.5.2 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.

This technique is powerful because subqueries can be used like any other source of data – they are simply a source of tuples, like how files on HDFS are a source of tuples. So you can use subqueries to break apart a larger query into subqueries, with each subquery handling an isolated portion of the overall computation.

Suppose, for example, 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 that data, such as number of transactions within a time period, the total amount sold by each seller, or the total amount bought by each buyer. Each of these queries needs to do the same work of parsing the data from the text files, so you'd like to abstract that into its own subquery. What you need is a function that takes in a path on HDFS and returns a subquery that parses the data at that path. You can write this function like this:

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

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

You can then reuse this abstraction for each query. For example, here's the query which computes the number of transactions for each buyer:

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

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 indicating that some tweet is a retweet of some other tweet. You want to be able to query for all chains of retweets of a certain length. So for a chain of length 4, you want to know all retweets of retweets of retweets of tweets.

You start with pairs of tweet identifiers. The basic observation needed for transforming pairs into chains is to recognize that you can find chains of length 3 by joining pairs against themselves. Then you can find chains of length 4 by joining your chains of length 3 against your original pairs. For example, 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");
}
```

And here's how you find 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 for chains of arbitrary length, you need a function that generates a subquery, setting up the appropriate number of predicates and setting the variable names correctly. This can be accomplished by writing some fairly simple Java code to create the subquery:

```
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());
    }
    Subquery ret = new Subquery(genVars);
    for(int i=0; i<n-1; i++) {
        ret = ret.predicate(pairs, genVars.get(i), genVars.get(i+1));
    }
    return ret;
}
```

The function first generates unique variable names to be used within the query

using the helper Api.getNullableVar function from JCascalog. It then iterates through the variables, creating predicates to set up the joins that will return the appropriate chains. Another interesting note about this function is that it's not specific to retweet data: in fact, it can take as input 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 be able to get a random sample of some fixed number of elements from any set of data. For instance, you want to get a random 10000 elements out of 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 record
2. Find the N elements with the smallest random numbers

JCascalog has a built-in aggregator called "Limit" for doing #2: Limit uses a strategy similar to parallel aggregators to find the smallest N elements. "Limit" will find the smallest N elements on each map task, then combine all the results from each map task to find the overall smallest N elements.

Here's a function that uses Limit to find a random N elements from a dataset of arbitrary size:

```
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++) {
        inputVars.add(Api.genNullableVar());
        outputVars.add(Api.genNullableVar());
    }
    String randVar = Api.genNullableVar();
    return new Subquery(outputVars)
            .predicate(data, inputVars)
            .predicate(new RandLong(), randVar)
            .predicate(Option.SORT, randVar)
            .predicate(new Limit(n), inputVars).out(outputVars);
}
```

The function uses the Api.numOutfields introspection function to determine the number of fields in the input dataset. It then generates variables so that it can put together a subquery that represents the logic to do a fixed random sample. The RandLong function comes with JCascalog and simply generates a random long value. The secondary sort tells the Limit aggregator how to determine what the

smallest elements are, and then Limit does the heavy lifting of finding the smallest records.

The cool thing about this algorithm is its scalability: it's able to parallelize the computation of the fixed sample without ever needing to centralize all the records in one place. And it was easy to express the algorithm, since using regular Java code you can construct a subquery to do the fixed sample for any input set of data.

5.5.3 Dynamically created predicate macros

Like how you can write functions that dynamically create subqueries, you can also create predicate macros dynamically. This is an extremely powerful technique that really 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")
    .predicate(new IncrementFunction(), "?a").out("?x")
    .predicate(new IncrementFunction(), "?b").out("?y")
    .predicate(new IncrementFunction(), "?c").out("?z");
```

This query reads a dataset containing triplets of numbers, and increments each field. There's some repetition in this query, since it has to explicitly apply the IncrementFunction to each field from the input data. It would be nice if you could eliminate 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 repeat the use of IncrementFunction over and over, it's better if you could tell JCascalog to apply the function to each input/output field pair, automatically expanding that predicate to the three predicates in the first example. This is exactly what predicate macros are good at, and you can define Each quite simply like so:

```
public static class Each implements PredicateMacro {
    Object _op;

    public Each(Object op) {
        _op = op;
    }
```

```
    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)));
        }
        return ret;
    }
}
```

This definition of Each is parameterized with the predicate operation to use, and then it creates a predicate for each input/output field pair that it's given. The number of predicates generated is dynamic depending on the number of fields specified in the subquery predicate.

Let's look at another example of a dynamic subquery. We've already defined the IncrementFunction which adds the value one to its argument. We defined IncrementFunction as its own function, but in reality it's really just the Plus function with one argument filled in to "1". It would be nice if you could abstract away the partial application of a predicate operation into its own operation, and defined Increment as something like this:

```
public static Object Increment = new Partial(new Plus(), 1);
```

Partial is a predicate macro that fills in some of the input fields. It allows you to rewrite the query that increments the triplets like this:

```
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 expands 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 let you manipulate the construction of your subqueries in a very fine-grained way. In all of these composition techniques – predicate macros, dynamic subqueries, and dynamic predicate macros – regular Java code is used to create clean, reusable abstractions.

5.6 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.

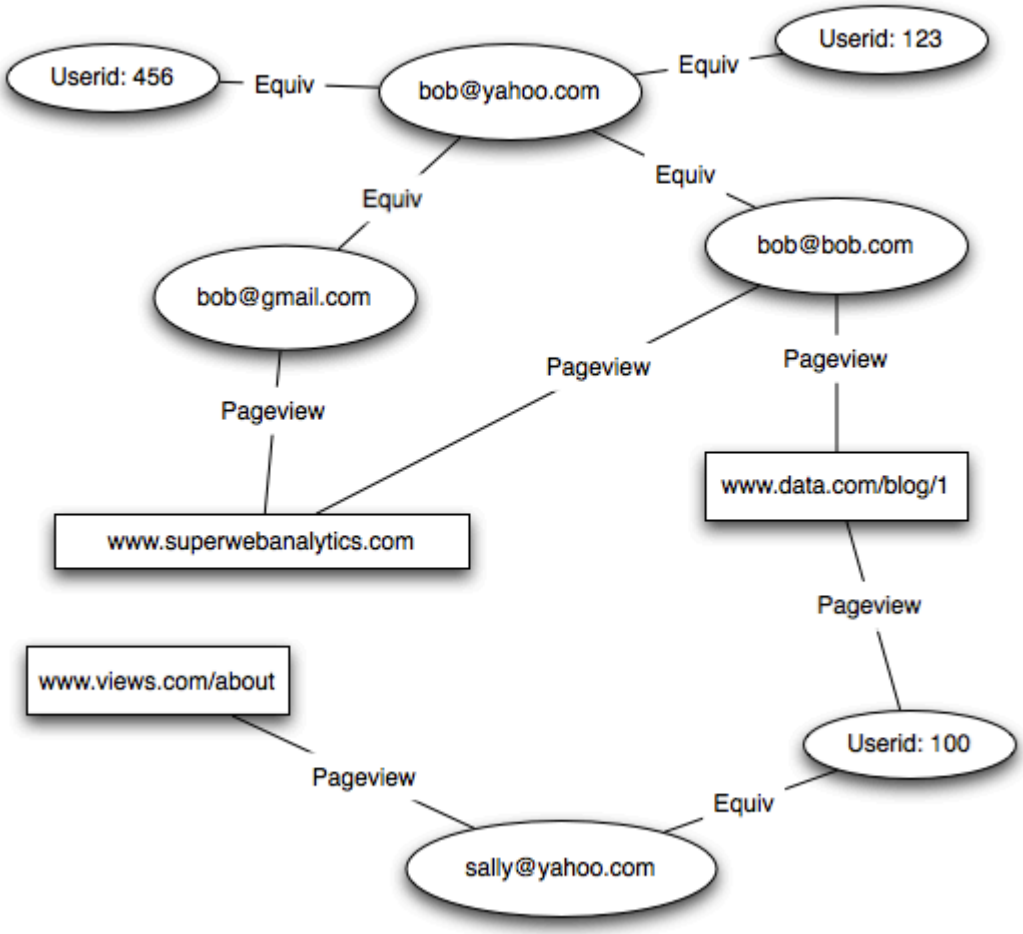


Figure 6.1 Examples of different pageviews for same person being captured for different user ids

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.

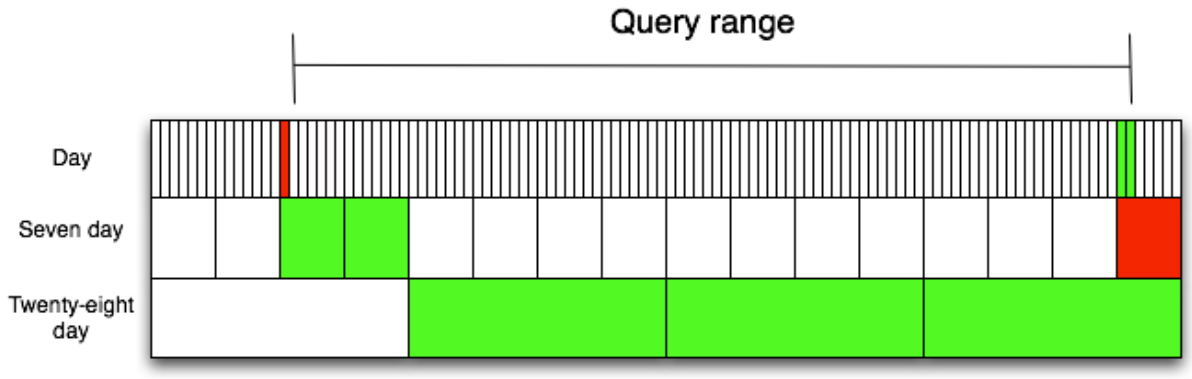


Figure 6.3 Optimizing pageviews over time query with coarser granularities. For this example range, green means to add that value into the result, red means to subtract that value.

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.

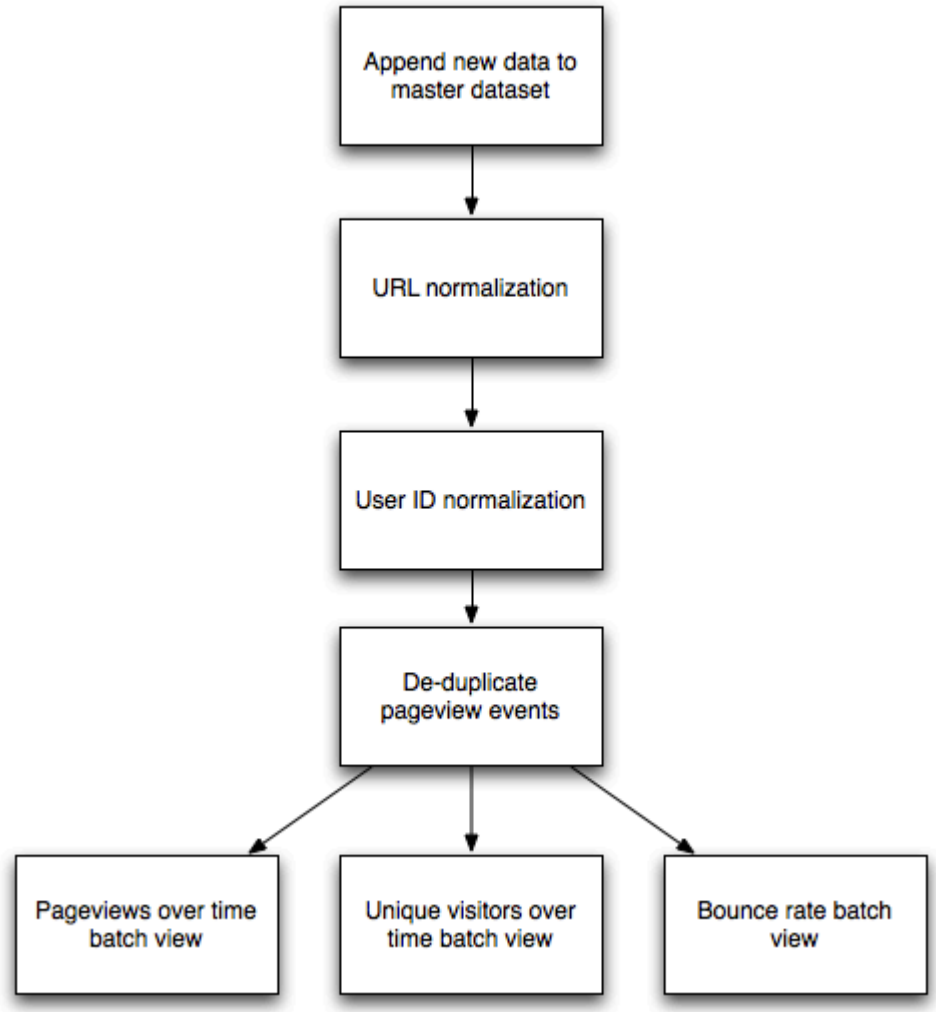


Figure 6.5 Batch workflow for SuperWebAnalytics.com

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.

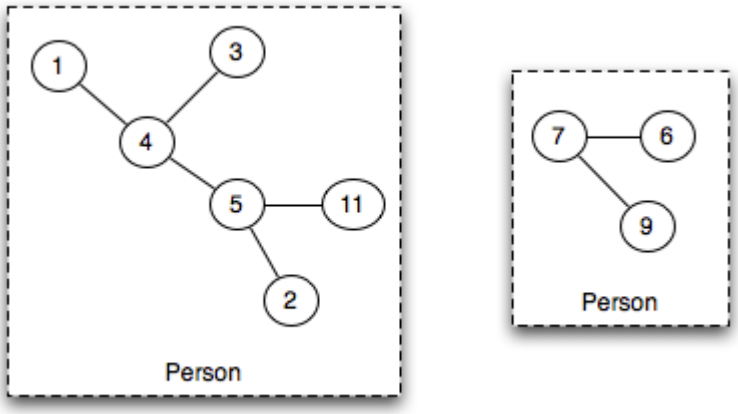


Figure 6.6 Example equiv graph

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.

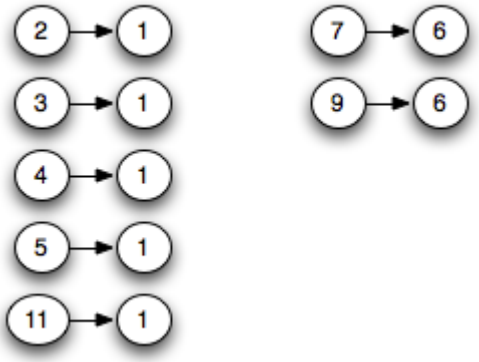


Figure 6.7 Mapping from user IDs to a single userID from each set

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.

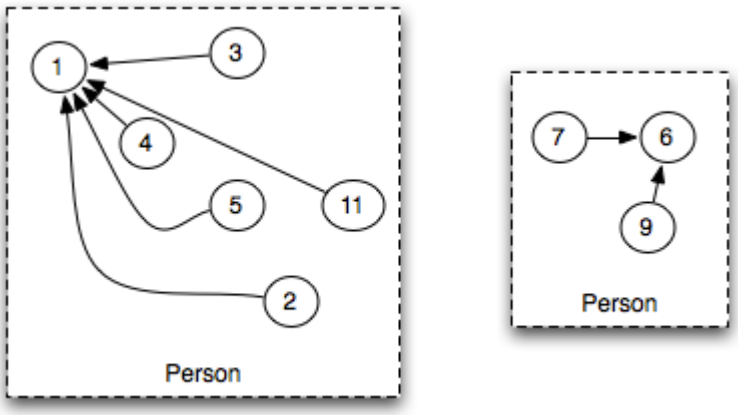


Figure 6.8 Original equiv graph transformed so that all nodes in a set point to a single node

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.

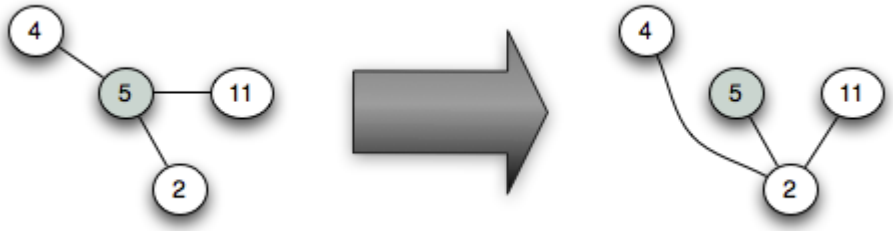


Figure 6.9 How the edges around a single node move in a single iteration

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.

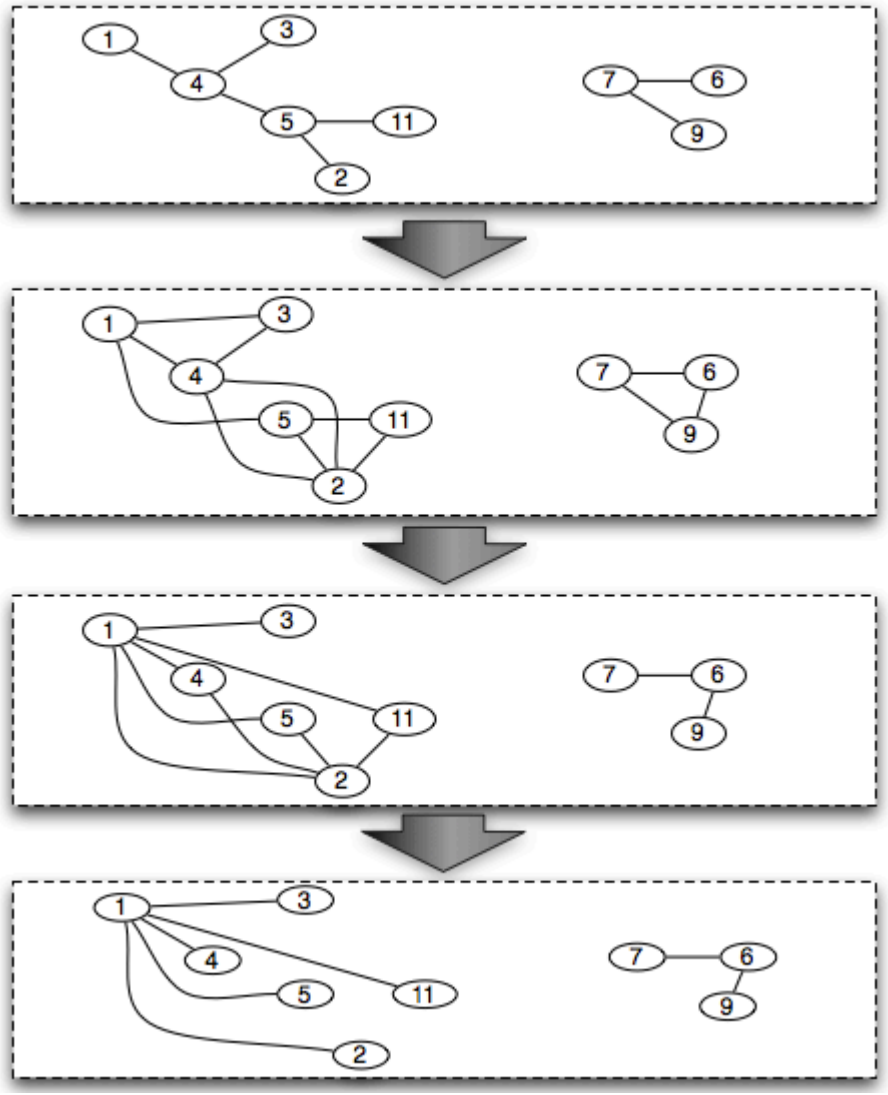


Figure 6.10 Running algorithm until fixed point

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.