Big Data

Principles and best practices of scalable realtime data systems

Nathan Marz

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Big Data version 4

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.

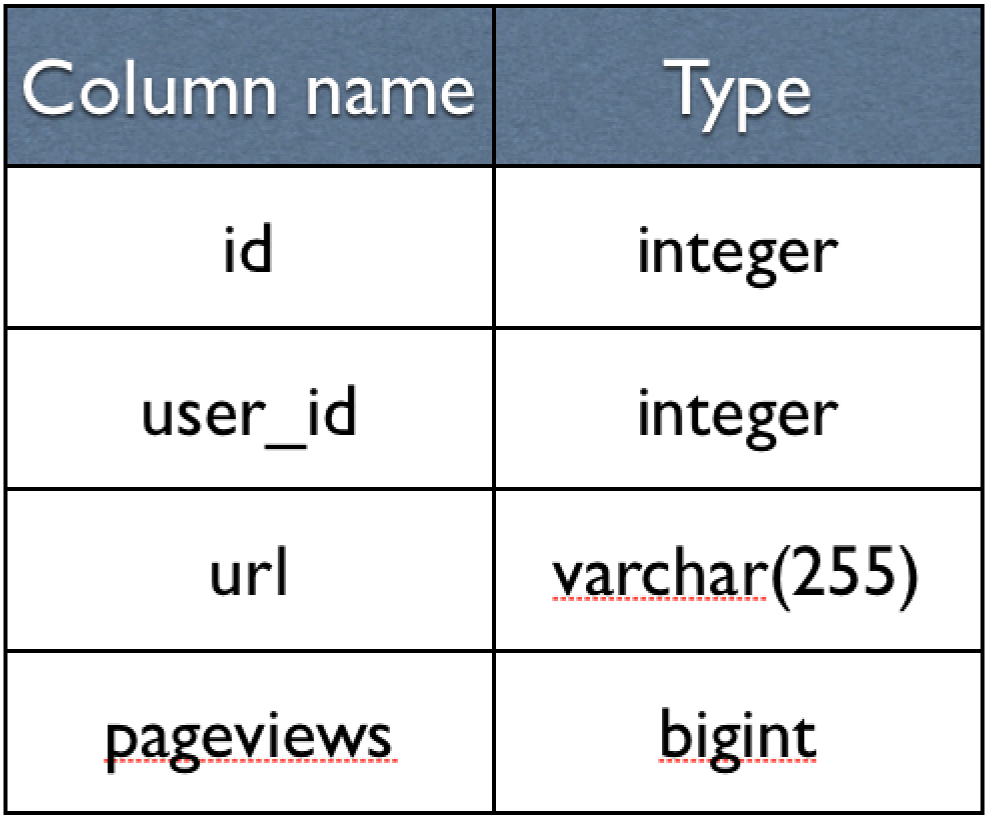
By the end of this book, you will have a thorough understanding of the principles of data systems. You will be able to use that understanding to choose the right tools for your specific application.

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.

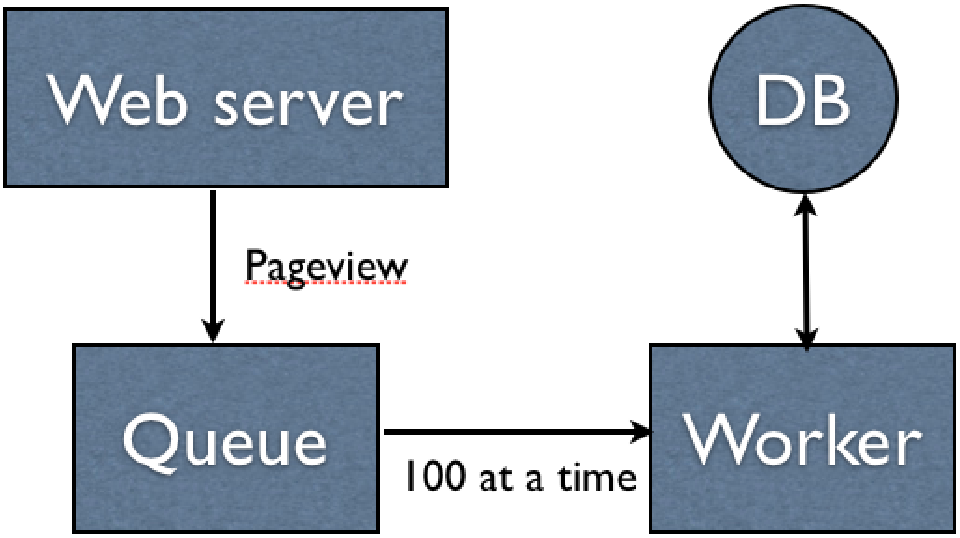


Figure 1.2 Batching updates with queue and worker

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:

query = function(all data)

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.

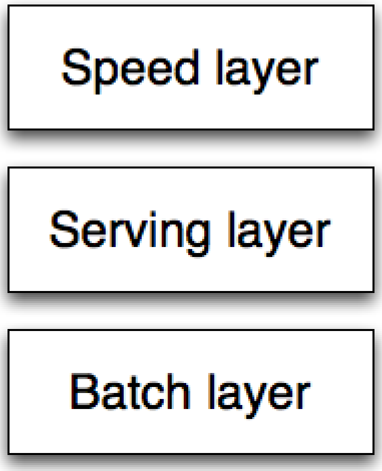


Figure 1.4 Lambda Architecture

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:

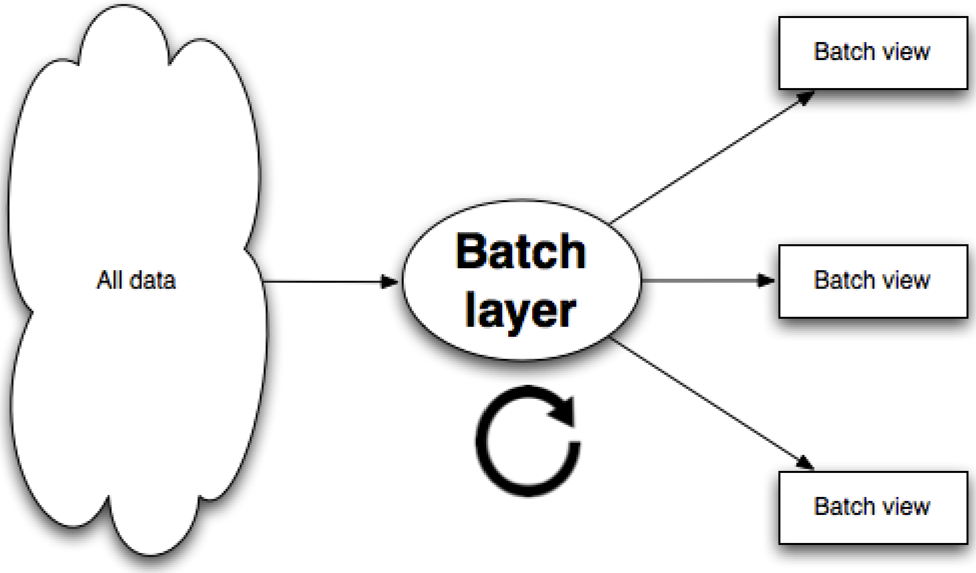


Figure 1.5 Batch layer

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

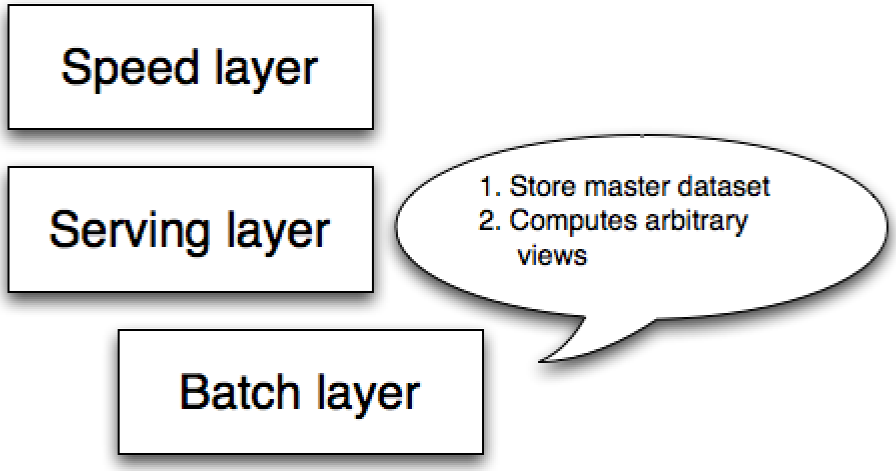


Figure 1.6 Batch layer

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.

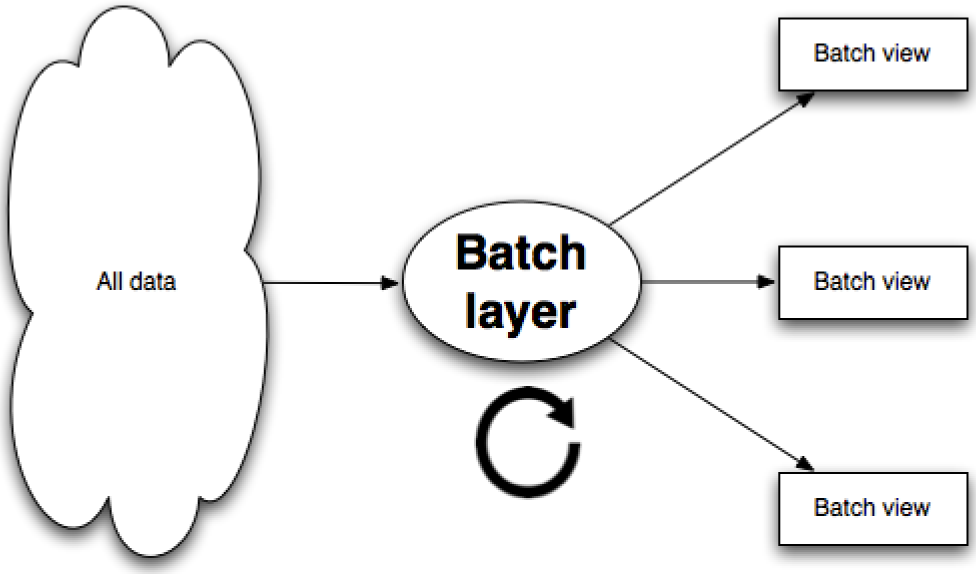


Figure 1.7 Batch layer

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

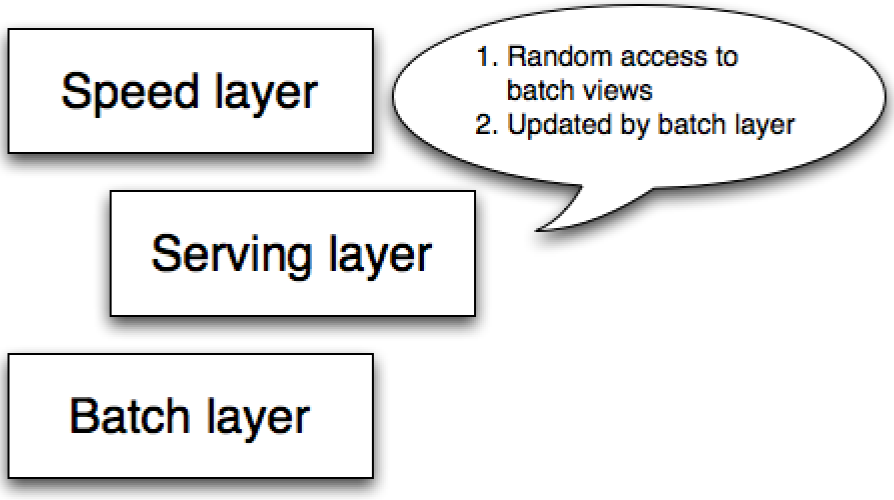


Figure 1.8 Serving layer

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

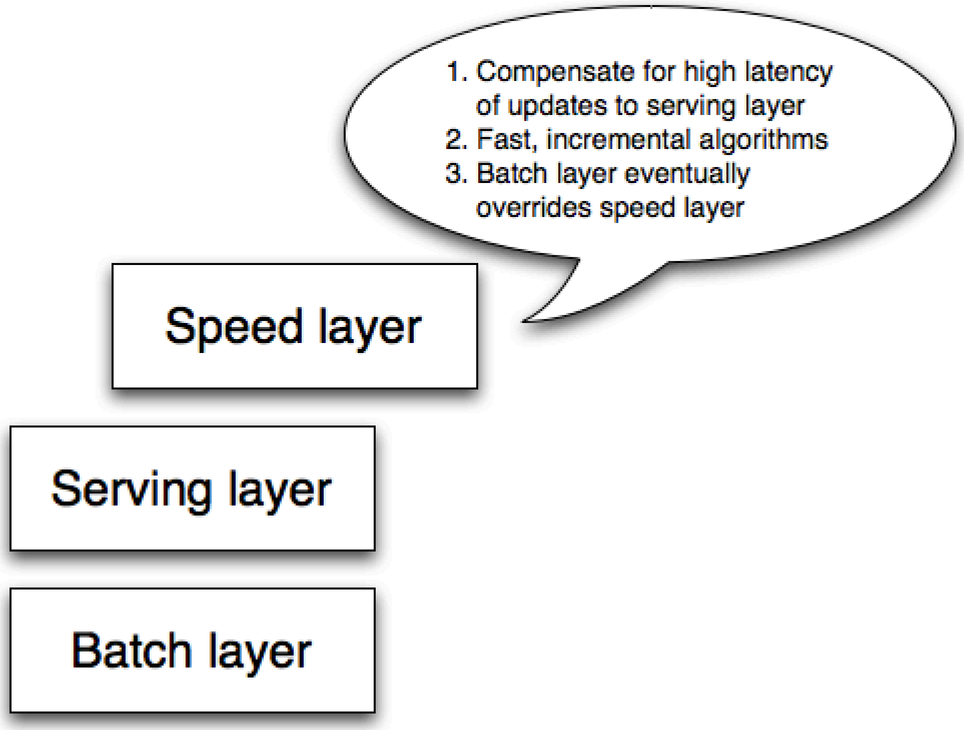

1. Compensate for high latency of updates to serving layer
2. Fast, incremental algorithms
3. Batch layer eventually overrides speed layer
Speed layer
Serving layer
Batch layer

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.

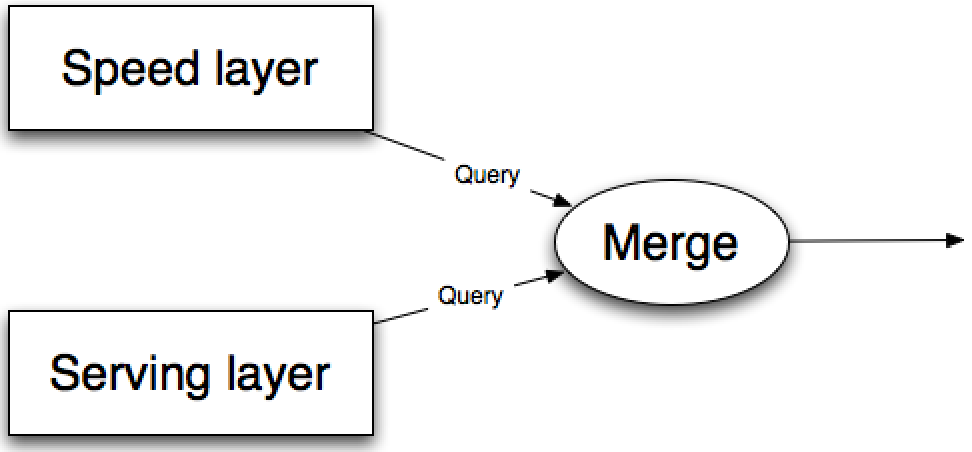


Figure 1.10 Satisfying application queries

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

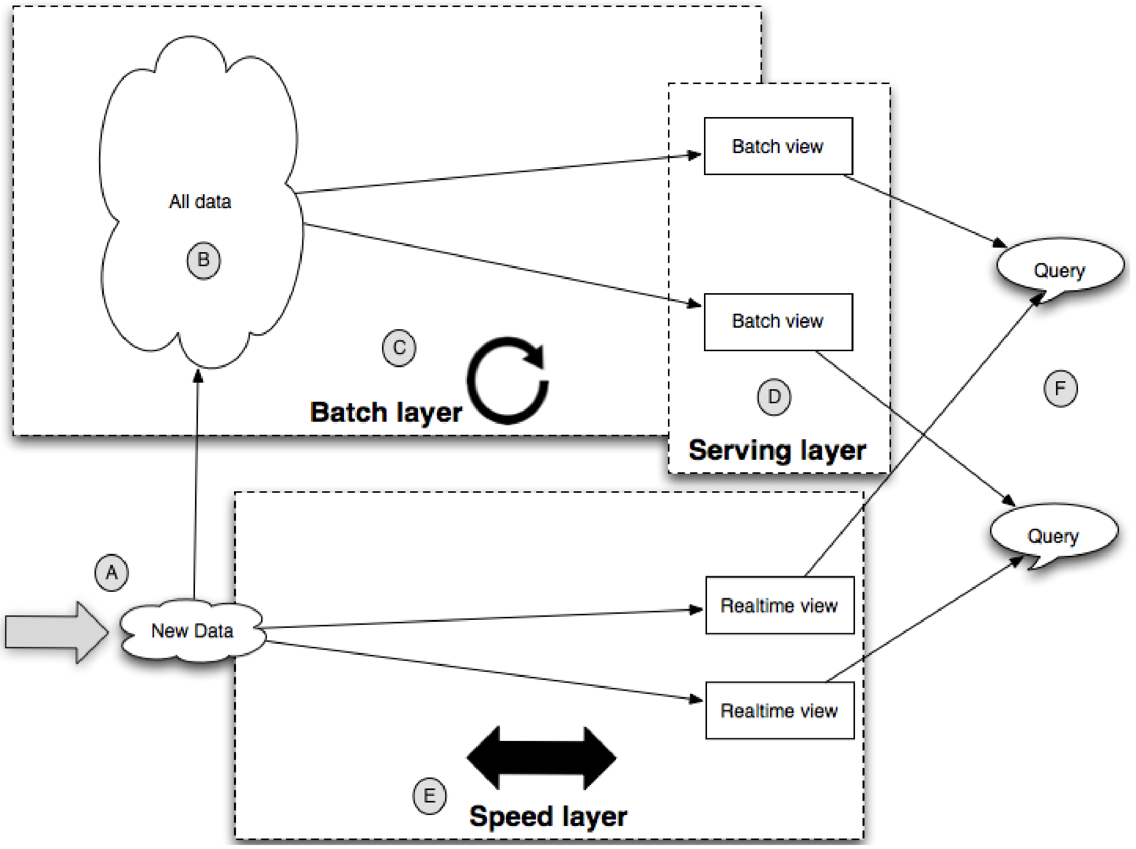


Figure 1.11 Lambda Architecture diagram

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:

- Fact-based data model
- Immutability
- Comparison to relational data model
- Graph schemas
- Implementing a fact-based data model with 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 for data systems. 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 realtime.

At the core of the Lambda Architecture is the master dataset, illustrated 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 databases and speed layer databases, 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.

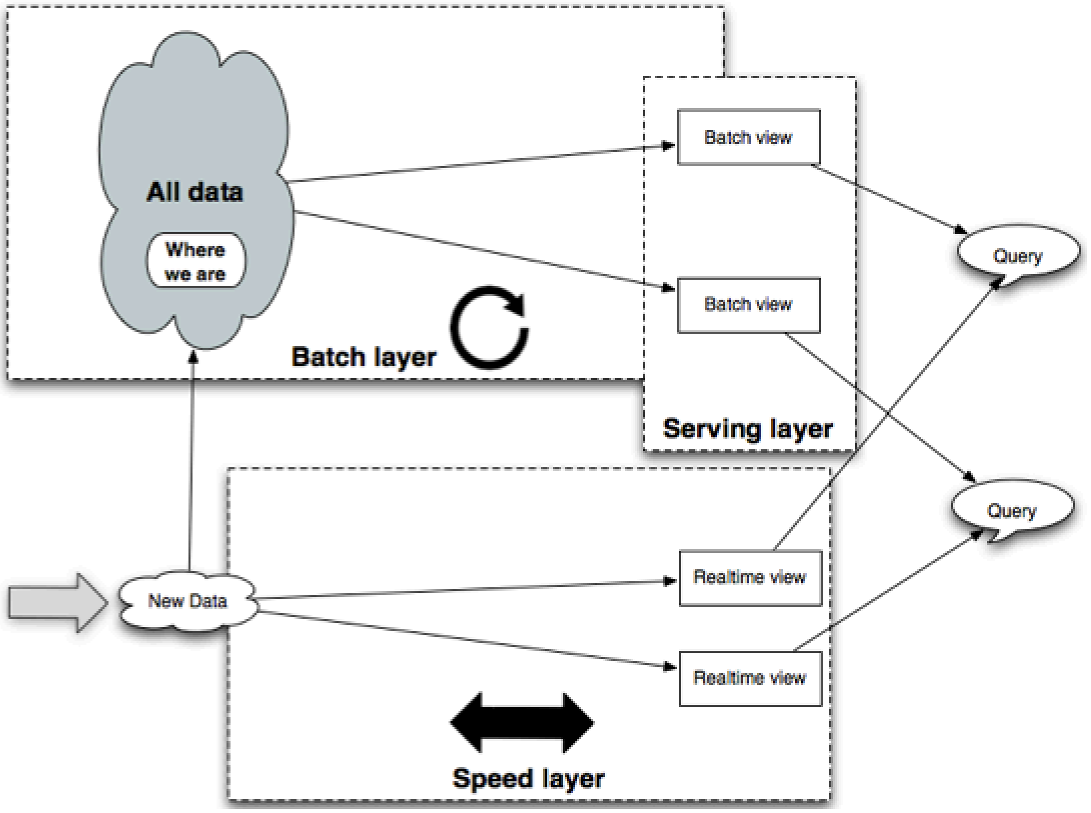


Figure 2.1 Where we are

The master dataset is the only part of the Lambda Architecture that cannot withstand corruption. Machine faults and human error inevitably happen, so you must carefully engineer the master dataset to avoid corruption in those cases. The master dataset must be resilient to machines get overloaded, disks breaking, and the power going out. It must be resilient to humans making mistakes and deploying bugs to production. Both machine fault-tolerance and human fault-tolerance are essential to the health of a long running data system.

There are two pieces to the master dataset: the data model to use, and how to physically store it. This chapter is about designing a data model for a master dataset and what properties such a data model should have. You will learn about physically storing a master dataset in the next chapter.

You are going to learn a data model for the master dataset called the "fact-based model". There are similarities between this model and the relational model, but there are some key differences. After a careful discussion of the essential properties of the master dataset, we'll define the fact-based model. Then you will see a practical implementation of the fact-based model using graph schemas with a tool called Apache Thrift.

2.1 The properties of data

In the first chapter we briefly discussed some of the distinguishing properties of data. Let's review that and then look more deeply at the properties the master dataset should have.

A data system answers questions about information you've acquired in the past. Not all the information you deal with is equal -- some information is derived from other bits of information. Consider, for example, a social network profile that displays the number of friends that person has. The friend count information derives from the list of friends which derives from all the times friends were added and removed to that person's profile. You can view information as a dependency tree as illustrated in Figure 2.2.

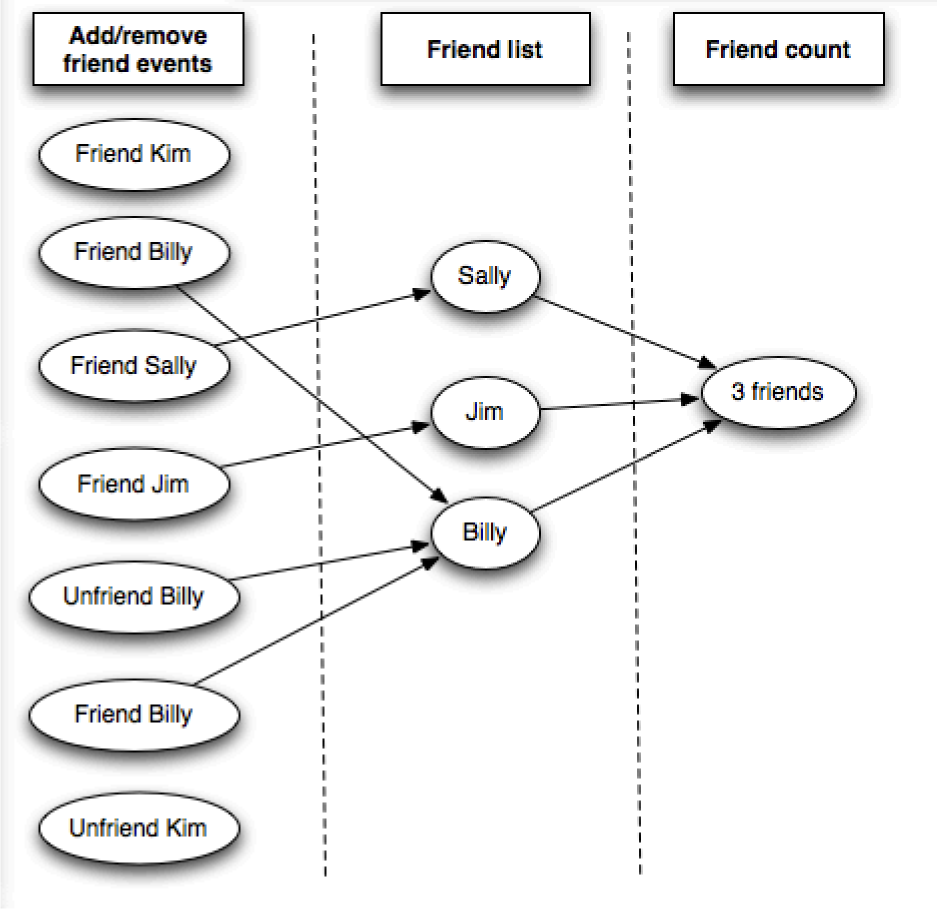


Figure 2.2 Information dependencies

Data is the information you have that can't be derived from any other information. It is the base data from which everything else derives. When you ask questions of your data, which we'll refer to as "queries", the answer is derived from many of the pieces of data you have. For example, your bank account balance is derived from your transaction history. We'll refer to any information you have that's derived from your base data as "views".

One person's data can be another person's view. Suppose you're writing a web crawler which extracts age information from user profiles on some imaginary social network, let's call it "FaceSpace". The age information that you extract narrows down that person's birthday to a 1 year period. For example, if you record that a person's age is 25 years old on February 1st, 2012, then that person was born sometime between February 1st, 1987 and January 31st, 1988. FaceSpace though has the actual birthday data and simply displays the derived information of the age on their web page. So your data is the age on a particular day, which from FaceSpace's perspective is just a view on their data, the actual birthday. FaceSpace's data is rawer than your data because your data is just one of many things that can be derived from their data.

2.1.1 Rawer data is more valuable

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.

Consider the difference between knowing how many people someone follows on Twitter and knowing the actual list of people that person follows. With the list of follows, you can ask questions about what the person is interested in, how far the person is from others on the social graph, what locations the people they follow are from, and so on. These are all questions you can't ask if you only have the friend count.

Now suppose you have an even rawer form of data than the list of follows. Suppose you have all the follow and unfollow events that went into constructing that list of follows. Now you can ask questions about what kind of people a person didn't like to follow, what are the attributes that distinguish between people a person follows and doesn't follow, the rate at which a person follows new people, and so on. As you can see, the rawer the data, the questions you can ask get deeper and more diverse.

Storing the rawest data possible is hugely valuable. The tradeoff is that rawer data typically entails more of it -- sometimes much more. However, one of the key benefits of using scalable Big Data technologies is that they are capable of storing and running computations on enormous amounts of data. So you have the flexibility to store raw data and take advantage of the benefits of raw data.

CONTAINERS ARE NOT A RAWER FORM OF DATA

Sometimes there is a grey area on whether one form of information is rawer than another. Consider again the example of extracting an age from someone's profile. Should the data you store be the age value or the full HTML of the page?

Suppose you stored the HTML as your data. We'd argue the HTML isn't a rawer form of information than the age value. Instead, it's just a container for the age value. There's no additional age-related information you can derive from the HTML that you can't derive from just the extracted age value. In contrast, a birthday **is** a rawer form of data because you can derive more information from it.

UNSTRUCTURED DATA IS RAWER THAN NORMALIZED DATA

Another grey area is the line between parsing and semantic normalization. Semantic normalization is the process of taking an unstructured form of data and turning it into a structured form of data. For example, an unstructured form of data your website might take in is a user's location. A user can input anything for that field, such as "San Francisco, CA", "SF", "Home", and so on. A semantic normalization algorithm would try to match the input with a known place. So your algorithm might semantically normalize ""San Francisco, CA", "SF", and "Home" to "San Francisco, California, United States", "San Francisco, California, United States", and null. "Home" turns into null because it doesn't specify a particular place.

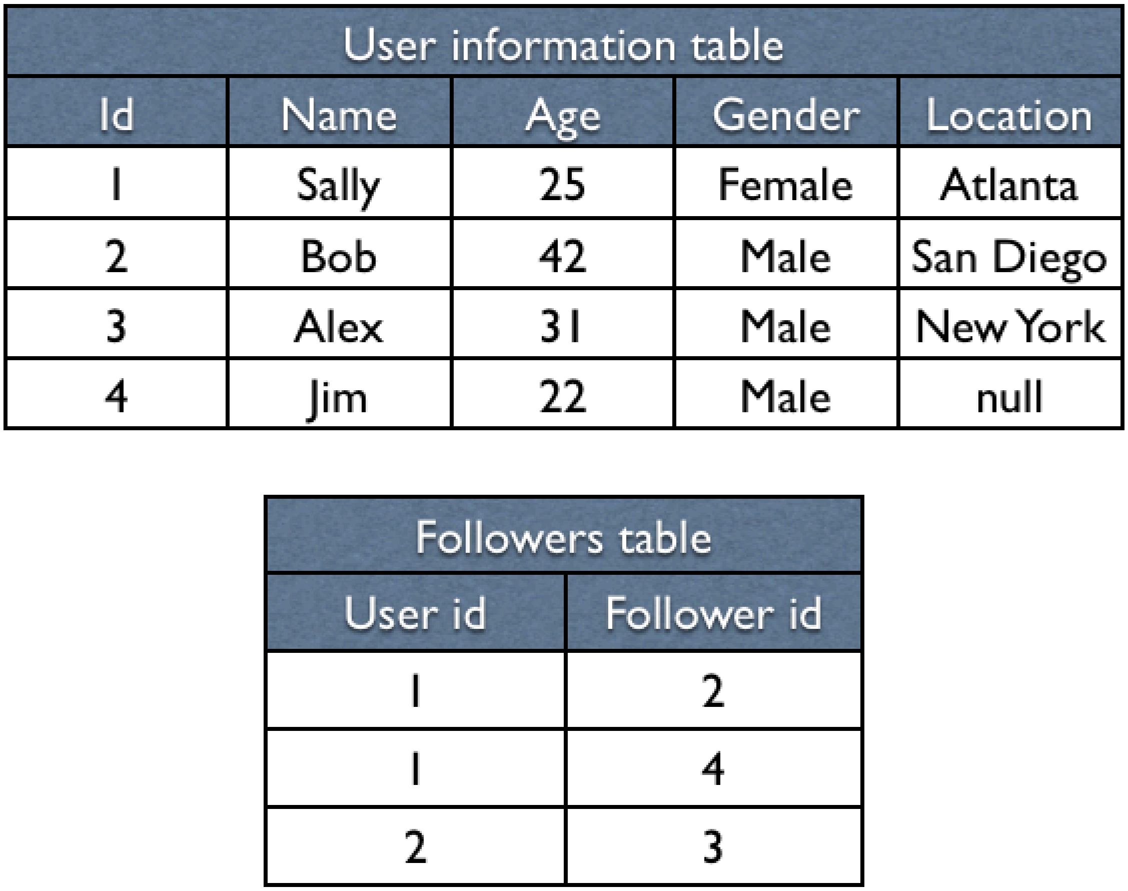
So if you come across a form of data such as an unstructured location string, should you store the unstructured string or store the semantically normalized form of the string? We argue that it's better to store the unstructured string because your semantic normalization algorithm may improve over time. If you store the unstructured string, you can re-normalize that data at a later time when your algorithms get better. If you only store the normalized location and discard the unstructured string, you won't be able to take advantage in improvements in your normalizer for past data.

As a rule of thumb, if your algorithm for extracting the data is simple, deterministic, and accurate, like extracting an age from an HTML page, you should go ahead and store the results of that algorithm. If the algorithm is probabilistic or non-deterministic, like extracting a normalized location from an unstructured location string, you should store the unstructured form of the data.

2.1.2 Data should be immutable

Another key question about data is whether it should be mutable or immutable. This may seem like a strange question to ask if you're used to using relational databases. After all, in the relational database world -- and the world of most databases -- "Update" is one of the four fundamental operations of databases, along with "Create", "Read", and "Delete". These operations are commonly known as CRUD. In the immutable world, you don't delete or update data. You only add data.

Before we get to the details of why an immutable data model is superior to a mutable one, let's look at an example of how you would reformulate a mutable data schema to be immutable. Consider the schema shown in Figure 2.3 for a simple Twitter-like social network.



User information table

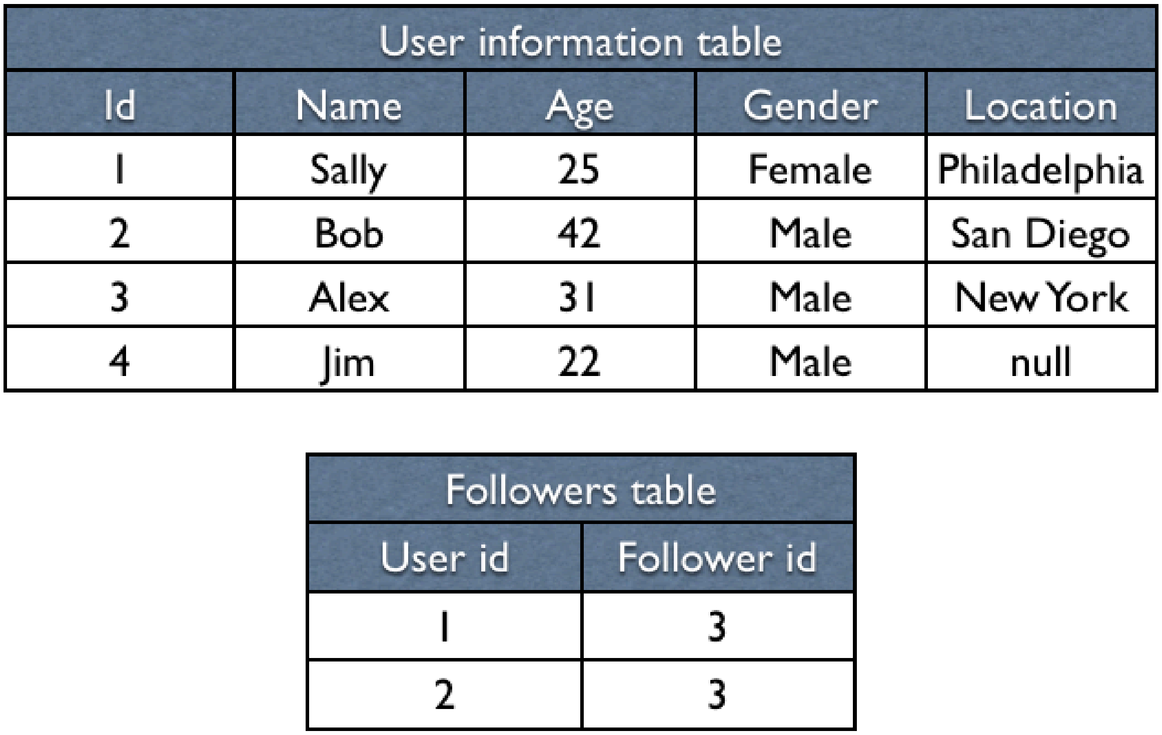
Id	Name	Age	Gender	Location
1	Sally	25	Female	Atlanta
2	Bob	42	Male	San Diego
3	Alex	31	Male	New York
4	Jim	22	Male	null

Followers table

User id	Follower id
1	2
1	4
2	3

Figure 2.3 Mutable schema for Twitter-like social network

Now suppose Sally changes her location to Philadelphia, loses Bob and Jim as followers, and gains Alex as a follower. Then the tables would mutate to look like Figure 2.4.

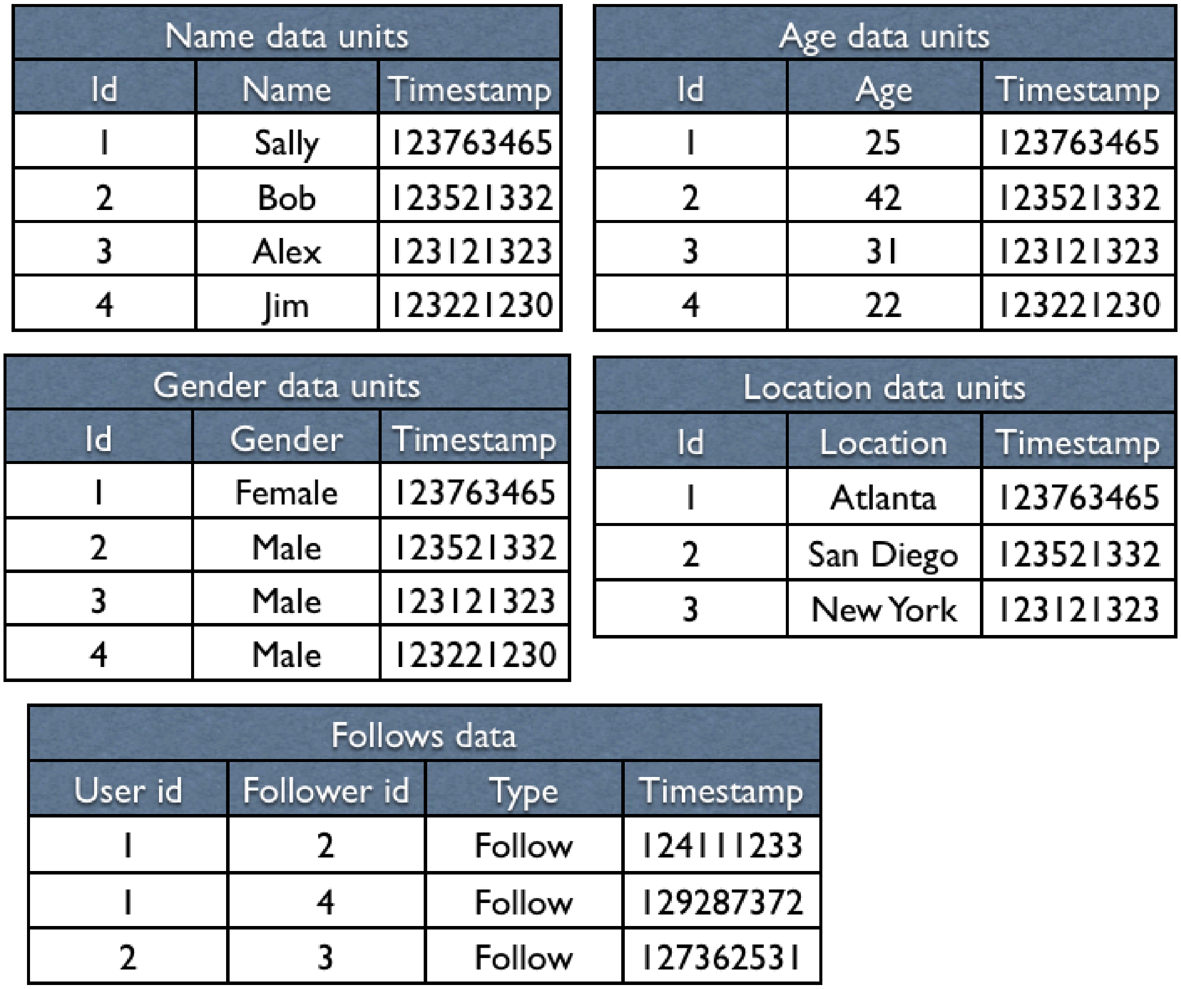


User information table				
Id	Name	Age	Gender	Location
1	Sally	25	Female	Philadelphia
2	Bob	42	Male	San Diego
3	Alex	31	Male	New York
4	Jim	22	Male	null

Followers table	
User id	Follower id
1	3
2	3

Figure 2.4 Capturing change with mutable schema

In the immutable world, things look different. Rather than store a current snapshot of the world as done by the mutable schema, you instead tie each unit of data to a moment in time. So instead of saying "Sally lives in Atlanta", you say "Sally lives in Atlanta as of March 4th, 2011 7:14:26". Then, when Sally changes her location, you would write something like "Sally lives in Philadelphia as of April 2nd, 2012 13:11:01". Since the data units are tied to particular times, they can both be true. Sally's *current location* is a simple query on the data: look at all the locations and pick the one with the highest timestamp. Figure 2.5 shows a translation of the mutable schema into an immutable schema.



Name data units		
Id	Name	Timestamp
1	Sally	123763465
2	Bob	123521332
3	Alex	123121323
4	Jim	123221230

Age data units		
Id	Age	Timestamp
1	25	123763465
2	42	123521332
3	31	123121323
4	22	123221230

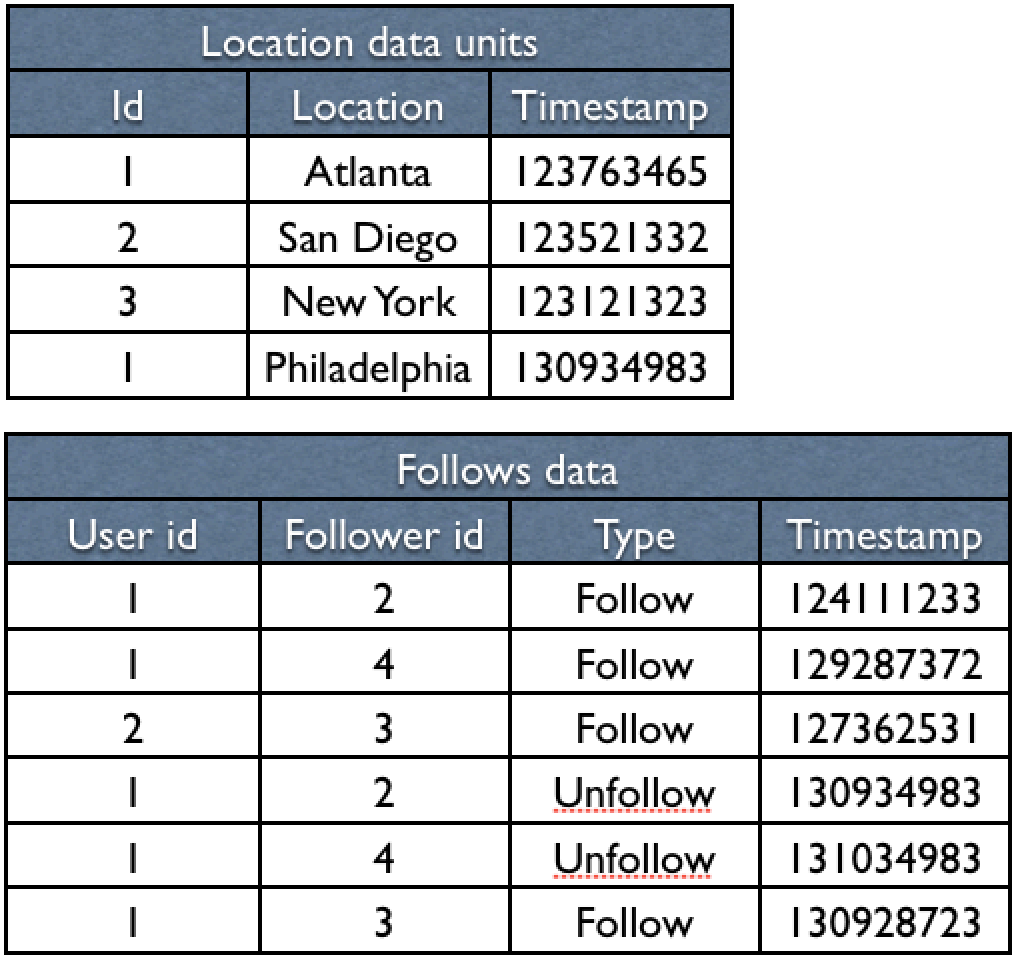
Gender data units		
Id	Gender	Timestamp
1	Female	123763465
2	Male	123521332
3	Male	123121323
4	Male	123221230

Location data units		
Id	Location	Timestamp
1	Atlanta	123763465
2	San Diego	123521332
3	New York	123121323

Follows data			
User id	Follower id	Type	Timestamp
1	2	Follow	124111233
1	4	Follow	129287372
2	3	Follow	127362531

Figure 2.5 Immutable schema for Twitter-like social network

Besides including a timestamp with each record, the biggest difference between the immutable schema and the mutable schema is storing the age, name, gender, and location as separate records. The reason for this becomes apparent when you look at how the data changes when you perform the same changes we performed on the mutable data to the immutable data. These changes are shown in Figure 2.6 (only showing the tables that have new data).



Location data units		
Id	Location	Timestamp
1	Atlanta	123763465
2	San Diego	123521332
3	New York	123121323
1	Philadelphia	130934983

Follows data			
User id	Follower id	Type	Timestamp
1	2	Follow	124111233
1	4	Follow	129287372
2	3	Follow	127362531
1	2	Unfollow	130934983
1	4	Unfollow	131034983
1	3	Follow	130928723

Figure 2.6 Capturing change with immutable schema

By storing the age, name, gender, and location information separately, a new fact about the location property can be specified independently of the other properties. If they were stored together in one record, it would be awkward to specify new facts about only one of the properties.

Also note that the immutable data model stores both follows and unfollows data. Like how the current location is a query on the immutable data, the current list of follows is a query on the follows and unfollows event data.

One of the tradeoffs of the immutable approach is that it uses more storage than the mutable approach. First of all, the "person id" is specified for every property, rather than just once per row as in the mutable approach. Second, rather than store just the current view of the world as done by the mutable approach, the entire history of events is stored. However, "Big Data" isn't called "Big Data" for nothing! You should take advantage of the ability to store large amounts of data with "Big Data" technologies to get the properties you need from your data model. The immutable model has enormous advantages over the mutable model.

The most important advantage of the immutable model is its human fault-tolerance. As we discussed in Chapter 1, human fault-tolerance is an essential property of data systems. You must assume that humans 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 the values are actually overridden in the database. With an immutable data model, **no data can be lost**. If bad data is written, the other 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. The benefits of having such strong human fault-tolerance in your master dataset cannot be overstated.

Another advantage of the immutable data model is that it's fundamentally simpler than a mutable data model. Whereas a mutable data model requires the CRUD operations, an immutable data model only requires the Create and Read operations. So CRUD is simplified to CR. Additionally, 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. Not requiring an index for all your data is a huge simplification. As you will see in the next chapter, storing a master dataset is as simple as using flat files.

2.1.3 Data should be eternal

Immutability implies certain properties of your data. One of these is that a piece of data, once true, must always be true. In other words, data is *eternal*. 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 eternal.

In general, your master dataset is consistently growing by adding new immutable and eternal pieces of data. There are some special cases though in which you do want to delete data, and these cases are not incompatible with data being eternal. Let's first consider the cases:

1. **Garbage collection**: Garbage collection deletes any data units that are of "low value". It can be used to implement data retention policies to control the growth of the master dataset. For example, you may decide you don't want to know the history of every time a user changed locations. So you might implement a policy that only keeps one location per person per year.
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.

2.2 Fact-based model

The fact-based model is a general-purpose data model to use for a master dataset. In the fact-based model, you represent your master dataset as an ever-growing list of immutable and eternal "facts". Each fact specifies a single concrete piece of information. In the discussion of immutability you saw a glimpse of the fact-based model. Now we'll expand on what we already discussed to explain the fact-based model in full.

Here are some example facts:

1. Alice is female
2. Bob is male
3. Alice and Charlie are friends

In contrast, the following examples would not fit into the data model because they capture too much data in each fact:

1. Alice is 25 years old and female
2. Charlie lives in San Francisco and is a programmer

Storing one fact per record makes it easy to write partial data about an entity without having to set fields to NULL. After all, properties like age, gender, and location are independent, so storing them separately lets you track changes to one property without needing to say anything about another property.

A crucial part of the fact-based data model is the notion of time. Each fact has associated with it a timestamp representing the earliest time that fact is believed to be true. The timestamp makes the facts eternal and makes them suitable for an immutable dataset. So in reality, the example facts above would really be something like:

1. Alice is female as of January 2nd, 2011 7:89:31
2. Bob is male as of February 6th, 2010 1:32:11
3. Alice and Charlie are friends as of September 21st, 2010 9:21:08

Time establishes an ordering the the truthfulness of the facts and allows more recent facts to override previous facts in your views. As you can see, concepts like "updates" and "deletes" are interpretations of all the facts to determine which individual record is true as of the current timestamp. With the fact based model, the full history of each entity is kept in the system.

2.2.1 Facts should be distinguishable

An additional constraint that we recommend placing on your facts is that each fact uniquely identify an event. Let's explain this idea through example. Suppose you want to store data about pageviews. Your first approach might look something like this (pseudocode):

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

This structure does 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. So if two pageview records are identical, there's no way to tell if they refer to the same event or two different pageview events. 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
```

This structure adds a 64 bit "nonce" field to the pageview structure. When a pageview data unit is created, a random 64 bit number is chosen for that nonce to distinguish this pageview from other pageviews that occur to the same URL at the same time and from the same ip address. The addition of the nonce field 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 distinguishable 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. It turns out that distinguishable facts makes implementing the rest of the Lambda Architecture much easier.

Fault-tolerant systems commonly deal with failed operations by retrying the operation. Ideally, the system is transactional, so that the operation succeeds only when the instigator of the operation receives a success notification. This is not always easy to do. As a relevant example, suppose you are appending new data units to your master dataset. Let's say that after you do the append, the network partitions and you are disconnected from the master datastore before receiving a success notification. Since you don't know if the append succeeded, you'll have to retry the append when you reconnect. However, if the other append did go through, you'll end up appending the same data units multiple times.

Now, there are ways to make these kinds of operations transactional. Doing so, however, can be fairly tricky and entail performance costs. And you'll have to ensure that your system is transactional up and down the stack. 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?

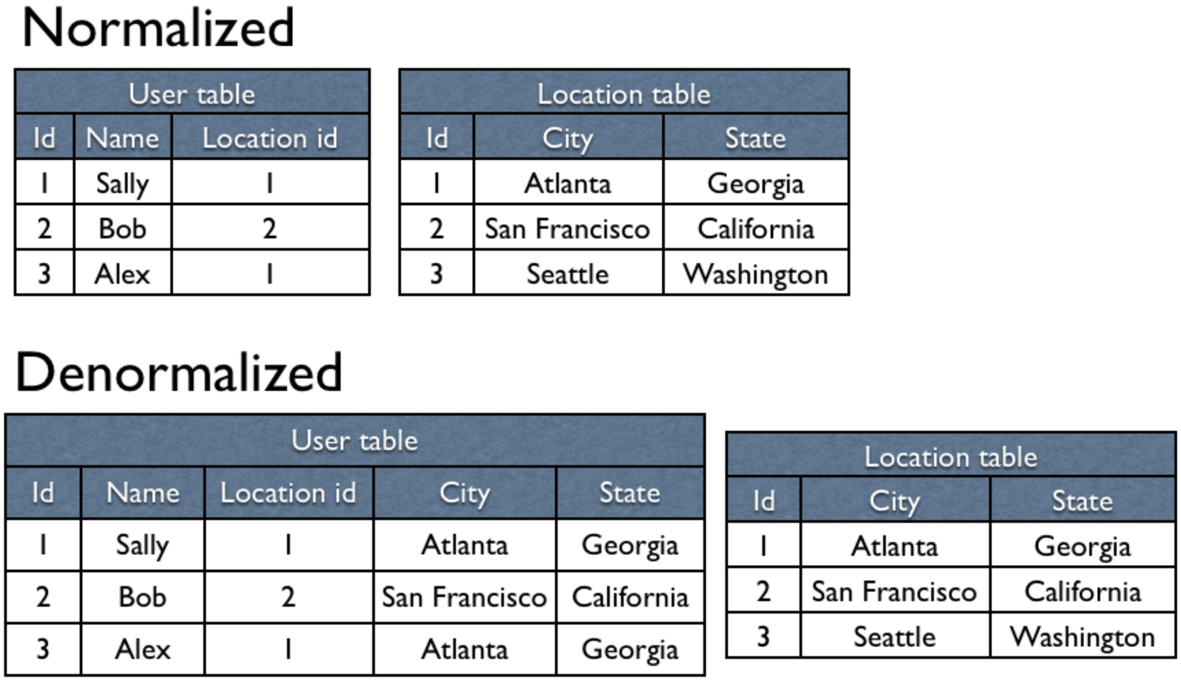
2.3 Benefits of fact-based model over a relational model

With a relational database, you don't store an ever-growing set of immutable data and maintain complex views on that data in realtime. That's not a pattern that relational databases were built to support.

There are a number of common issues you encounter when using a relational database. Let's look at those issues and see how they do and don't manifest themselves with the fact-based model and Lambda Architecture.

2.3.1 Normalization vs. denormalization

A common point of discussion in the relational database world is whether your data should be stored normalized or denormalized (this usage of the term "normalization" is completely different than "semantic normalization" we were discussing before). In a normalized schema, a piece of data is stored one time, and all queries on that data read from that one location. In a denormalized schema, a piece of data is stored redundantly in many different ways to optimize different queries. Denormalization is often done to avoid expensive joins.



Normalized

User table		
Id	Name	Location id
1	Sally	1
2	Bob	2
3	Alex	1

Location table		
Id	City	State
1	Atlanta	Georgia
2	San Francisco	California
3	Seattle	Washington

Denormalized

User table				
Id	Name	Location id	City	State
1	Sally	1	Atlanta	Georgia
2	Bob	2	San Francisco	California
3	Alex	1	Atlanta	Georgia

Location table		
Id	City	State
1	Atlanta	Georgia
2	San Francisco	California
3	Seattle	Washington

Figure 2.7 Normalized vs. denormalized schema

An example of denormalization is shown in Figure 2.7. When a person's location is updated, the city and state is copied from the locations table into the person table. The data is pre-joined so that reads can be faster. Denormalization of course has the serious drawback of making it really hard to update a piece of data, as you have to find every place it has been redundantly placed. It also makes it possible for the database to get corrupt if not all instances of a piece of data are kept in sync.

The need for denormalization in relational databases is an artifact of mixing data storage with the indexing required to perform application queries. In the Lambda Architecture, these are cleanly separated. Take a look again at the Lambda Architecture diagram in Figure 2.8.

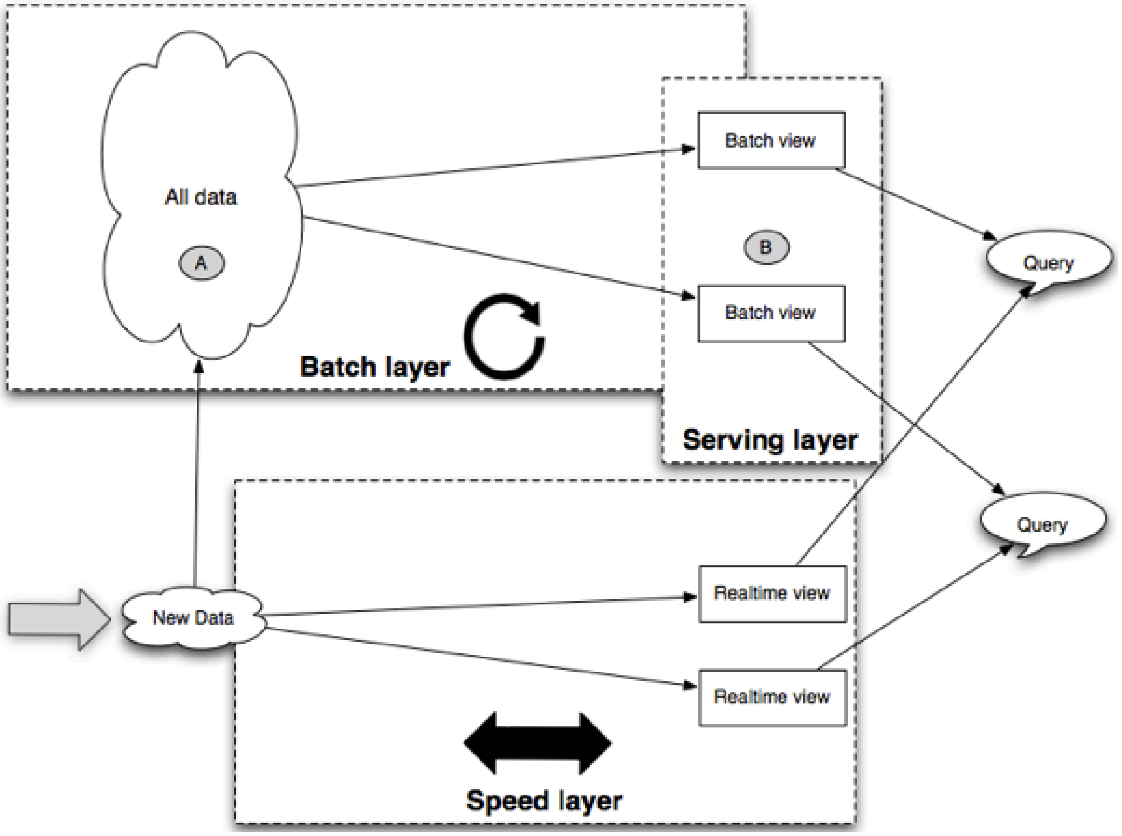


Figure 2.8 Lambda Architecture

The master dataset at (A) is fully normalized. As you saw in the discussion of the fact-based model, no data is stored redundantly. Queries are performed based on the batch views, shown at (B). The batch views index the data in a way to optimize the queries that they support. Where the batch views are similar to denormalization is that one piece of data from the master dataset may get indexed into many batch views. However, they key difference with denormalization is that the batch views are defined as functions on the master dataset. There's no concern as to how to perform an update, because the only source of truth is the master dataset. The batch views are derived by running a function on the master dataset, so there's no possibility of getting out of sync with the master dataset. The Lambda Architecture gives you the conceptual benefits of full normalization with the performance benefits of indexing data in different ways to optimize queries.

2.3.2 Updating current view of the world vs. storing an immutable history

In the relational database world, you tend to update your current view of the world over time by mutating the data you store. For example, you might store the location of each of your users in a table and update the location whenever it changes. In the fact-based model, you instead store the full history of the world by tagging each data unit with the timestamp it was known to be true. Instead of modifying the current location, you instead store the full history of locations over time. The "current location" then becomes a query on the history of locations.

There's a lot of benefits to storing a history rather than a current view of the world. Besides the human fault-tolerance properties that we already discussed, the historical data is really useful for doing analytics and debugging your applications.

So far we've been discussing the design of a data model for the master dataset at a high level. Now we're going to dive into the details of practically implementing a fact-based data model.

2.4 The value of an enforceable schema

You have two choices for modeling data in the master dataset. You can use an unstructured format like JSON which allows essentially anything to be written to the master dataset. Or you can create an enforceable schema that rigorously defines the structure of the data in the master dataset.

Using an enforceable schema has huge advantages over using an unstructured format. It requires a bit more up front work, but an enforceable schema will catch a ton of data-related errors, save huge amounts of time debugging why an expected field doesn't exist, and give confidence to developers of what data they can expect.

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. When the error appears close to the bug, it is easy to catch and fix. When it appears in a MapReduce job later on, it will take a lot longer to trace back the problem.

A schema documents the data so that developers can make use of it. Additionally, it provides mechanisms for a safe evolution of the data model over time.

Schemas are defined using a serialization framework. A serialization framework lets you define a schema in a language-neutral way and then generates code for serializing and deserializing objects of those types in any languages. This means that you can create an object in one language and read it in another language. Serialization frameworks provide mechanisms for evolving that schema as well.

In a relational database, the schema language is part of the database system and is integrated with how the database stores and processes that data. In the Big Data world, you use your own serialization framework that's separate from the storage and processing pieces. You get the flexibility to fine-tune this component to work exactly as needed to fit the fact-based model.

2.5 Introducing Apache Thrift

Thrift is a tool that can be used to define statically typed, enforceable schemas. There are other tools that can be used for this purpose, such as Protocol Buffers or Avro. We chose to demonstrate Thrift because it meets the requirements for demonstrating the principles of creating a schema for a fact-based data model. Remember, the purpose of this book is not to provide a survey of all the possible tools for every situation, but to use the tools to illustrate the fundamental concepts. That said, Thrift is practical and widely used for this purpose.

Thrift is a widely used project that originated at Facebook. It can also be used for making language neutral RPC servers, but we will only be using it for its schema creation capabilities. After explaining how to use Thrift, we will look at how to create a schema for a fact-based data model with Thrift.

The workhorses of Thrift are the "struct" and "union" type definitions, and Thrift has built-in mechanisms for evolving a schema over time.

2.5.1 Structs

The following code shows how to define a struct using the Thrift Interface Definition Language (IDL). Defining a struct is like defining a class in an object-oriented language: you specify all the data the object contains. The difference is that a Thrift struct only contains data and doesn't specify any extra behavior for the object. Fields in a struct can be:

- Primitive types like strings, ints, longs, and doubles. In the Thrift IDL, these are referred to as "string", "i32", "i64", and "double" respectively.
- Collections of other types. Thrift supports "list", "map", and "set".
- Another Thrift struct or union.

```
struct Person {
  1: string twitter_username;
  2: string full_name;
  3: list<string> interests;
}
```

The following code listing shows how to serialize a struct with Java. As you can see, we're using ArrayList, a native Java data structure, as part of the Person object.

```
List<String> interests = new ArrayList<String>() {{
    add("hadoop");
    add("nosql");
}};
Person person = new Person("joesmith", "Joe Smith", interests);
TSerializer serializer = new TSerializer();
byte[] serialized = serializer.serialize(person);
```

Here's how to deserialize a Person object in Python. When the object is deserialized, it will be using native Python data structures for any collection types.

```
person = Person()
deserialize(person, serialized_bytes)
```

Fields in structs can be defined as being either required or optional. If a field is defined as required, than a value for that field must be provided or else Thrift will give an error upon serialization or deserialization. If a field is optional, the value will be null if not provided. You should always declare fields as being either required or optional. The following code listing shows how to define a struct containing required and optional fields.

```
struct Tweet {
 1: required string text;
 2: required i64 id;
 3: required i64 timestamp;
 4: required Person person;
 5: optional i64 response_to_tweet_id;
}
```

2.5.2 Unions

You can also define unions in Thrift. A union is a struct that must have exactly one field set. Unions are useful for representing polymorphic data. The following listing shows how to define a "PersonID" using a Thrift union that can be one of many different kinds of identifiers.

```
union PersonID {
  1: string email;
  2: i64 facebook_id;
  3: i64 twitter_id;
}
```

2.5.3 Evolving a schema

Thrift is designed so that schemas can be evolved over time. The key to evolving Thrift schemas over time is the numeric identifiers we've been using for every 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 be sure never to reuse that field id. When deserializing, Thrift will skip over any fields that don't match an id it's expecting. So the data for that field will just be ignored in the existing data. If you were to reuse that field id, Thrift will try to deserialize that old data into your 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 will not be deserializable. Note that this point does not apply to unions since unions have no notion of required and optional fields.

Now that you've learned how to define schemas using Thrift, let look at the patterns you'll want to use when defining a fact-based data model for a master dataset.

2.6 Implementing fact-based data model for SuperWebAnalytics.com with Thrift

A fact-based data model captures one piece of data in each fact. That means a fact either says something about an entity in your system or about the relationship between two entities in your system. A good way to interpret a fact-based data model is as a graph. As you are about to see, graphs are a very flexible representation of data. We will be referring to the implementation of a fact-based model using Thrift as a "graph schema".

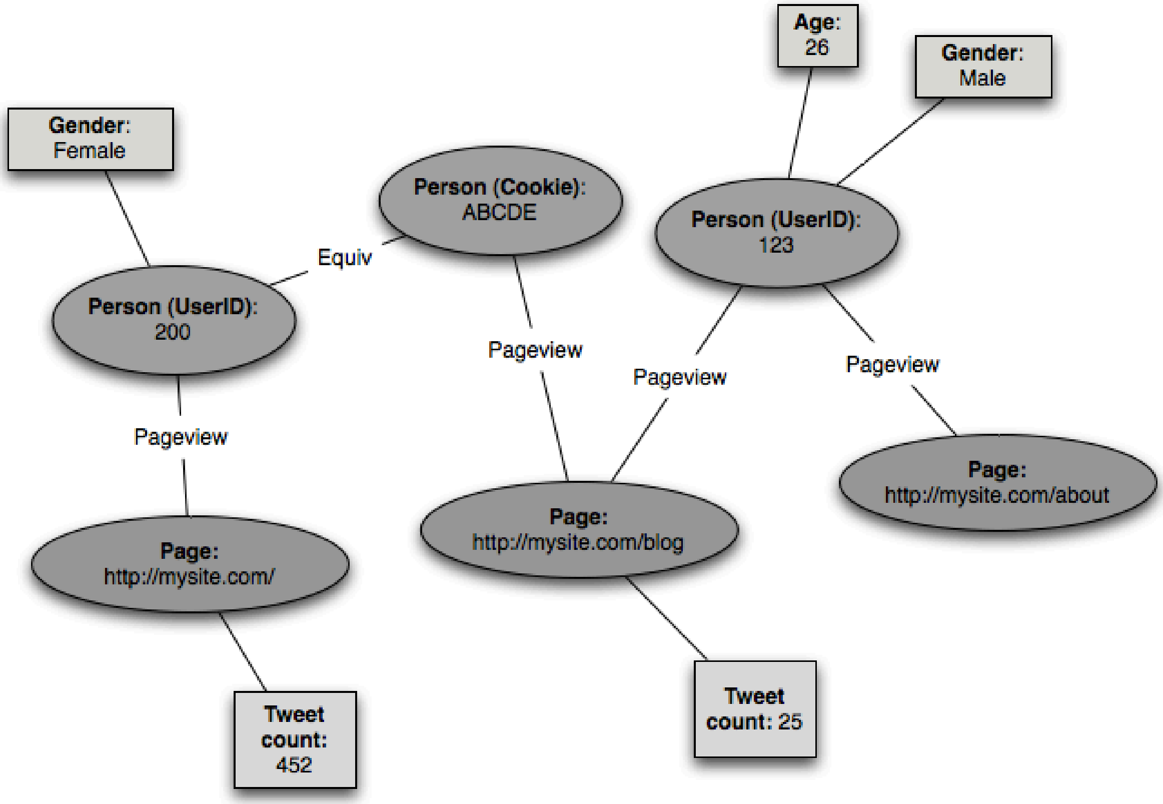


Figure 2.9 Visualization of graph schema

Figure 2.9 shows a visualization of what a graph schema looks like. This particular schema is the schema we will be using for SuperWebAnalytics.com, the Big Data application we will be building throughout the book.

Graphs contain nodes, edges, and properties. Nodes are entities in the system. In this example nodes can either be people or pages. As you can see there are two kinds of people: people who are logged in and are represented by their user id, and people who we can only identify by a cookie stored on their web browser.

Properties are information about entities. In this example, people can have ages or genders, while pages can have a tweet count indicating how many times the page was referenced on Twitter.

Edges are relationships between entities. In this example, there are two kinds of edges. A pageview edge occurs between a person and a page, while an equiv edge occurs between two person identifiers indicating they are the same person. An equiv edge may be created when a person first logs in and would be used to indicate that a cookie and a user id are actually the same person.

Now that you know what a graph schema is, let's use Thrift to codify the nodes, properties, and edges for the graph schema of SuperWebAnalytics.com.

2.6.1 Nodes

Let's start with the nodes. Nodes can have different types. In SuperWebAnalytics, there are two kinds of people (cookies and logged in users) and one kind of page. Union structures are ideal for representing nodes.

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

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

2.6.2 Properties

Next, 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 tweet_count;
}

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.

2.6.3 Edges

Each edge is 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;
}
```

2.6.4 Tying everything together into Data objects

So far, all the edges and properties are defined as separate types. Since you want to store all the data together, you should tie all the different data types into one object type. Tying everything together into one type gives a single interface for all your data and makes it easy to store the data together. This is accomplished by wrapping every type into a "DataUnit" union. Finally, each DataUnit is paired with metadata that you'll want stored with each piece of data. The following code listing completes the schema.

Listing 2.1 Completing the SuperWebAnalytics.com schema

```
enum Source {
  SELF = 1,
  BACKTYPE = 2
}

struct ExternalDataSystem {
}

struct PageViewSystem {
}

union OrigSystem {
  1: PageViewSystem page_view;
  2: ExternalDataSystem external_data;
}

struct Pedigree {
  1: required i32 true_as_of_secs;
  2: required Source source;
  3: required OrigSystem system;
}

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

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

Tying the data together is accomplished by listing every property and edge in the DataUnit union. The common metadata is kept in a "Pedigree" structure, and the DataUnit and Pedigree are stored in the "Data" struct. "Data" is the root structure of the schema.

The most important field in the pedigree is the "trueasofsecs" field. This indicates the earliest timestamp this piece of data is known to be true. As discussed earlier in the chapter, time provides an ordering to the facts and enables the facts to be eternal and thereby immutable. The other fields in the Pedigree are useful for debugging and tracing back problems. The "source" field indicates who gave you the data, while the "system" field indicates what part of your system collected the data. Other metadata specific to your organization that makes sense to collect on every piece of data should go into the Pedigree struct.

By representing the SuperWebAnalytics.com schema as a graph, it's very easy to evolve the schema over time as new kinds of data get added to the system.

2.7 Evolving a graph schema over time

The beauty of the graph schema is its consistent interface to arbitrarily diverse data. Since every piece of data contains a single fact you'll rarely have to modify the schema for existing types. Making schema additions is as simple as adding new properties, edges, and node types to the schema.

In the SuperWebAnalytics.com schema, adding an "age" property for people just requires adding the line "5: i16 age" to PersonPropertyValue. If you want to define a new kind of edge, you simply add that structure into the DataUnit union.

2.8 Summary

How you model your master dataset is extremely important. The decisions made surrounding the master dataset affect what kind of analytics you can do on your data and how you're going to consume that data. The structure of the master dataset must support extreme evolution of the kinds of data stored as the company's data types change over the years.

The fact based model frees the dataset by naturally keeping a full history of each entity over time. Additionally, its append-only nature makes it easy to implement in a distributed system. It has fundamental advantages over the relational model. You're not just implementing a relational system in a more scalable way, but you're adding whole new capabilities to your systems 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 to support recomputation
- Append-only storage
- Distributed filesystems
- Vertical partitioning
- Appends
- File formats and compression

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

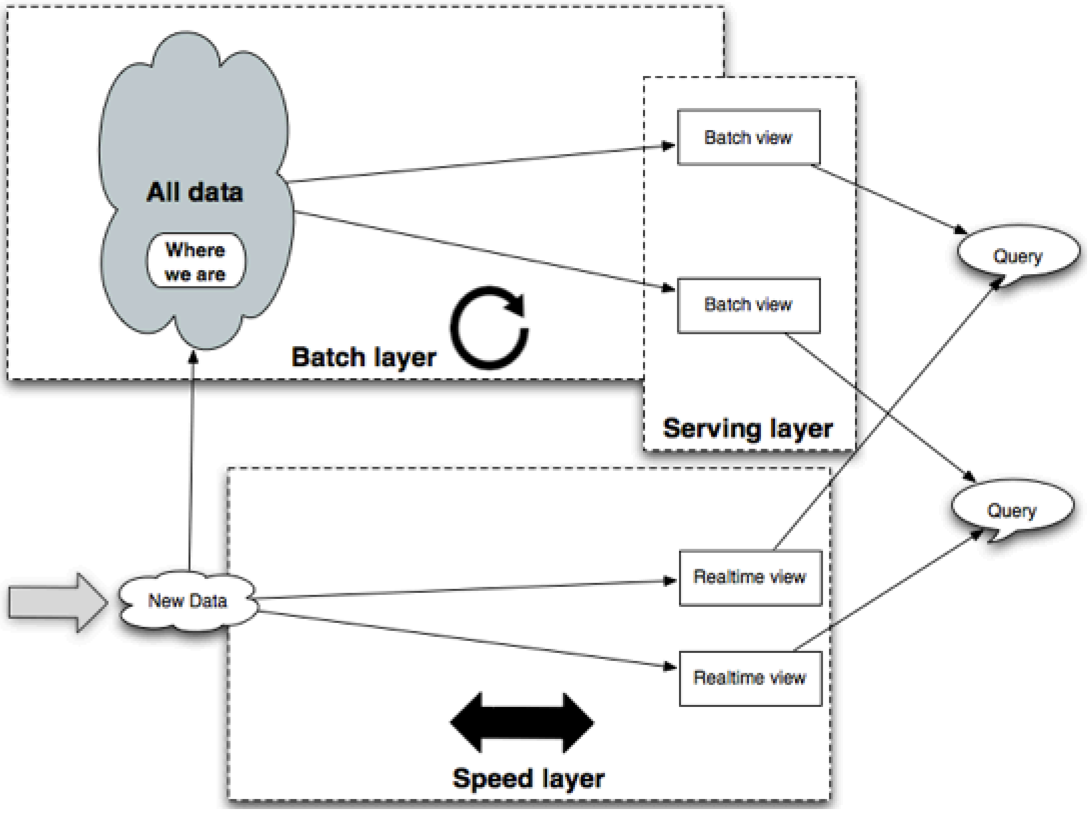


Figure 3.1 Where we are

Let's recap where we are in the Lambda Architecture. Like the last chapter, this chapter is all about the master dataset. How you store your data will affect how that data is consumed, so you have to take the consumption patterns of the batch layer into account when designing the storage for the master dataset.

The batch layer imposes some requirements on the storage of the master dataset. After looking at those requirements, you will learn how to implement the principles of storing a master dataset using the Hadoop Distributed Filesystem.

3.1 Requirements of data storage

To determine the requirements for data storage, you have to consider two things: how the data is going to be written, and how the data is going to be read. You learned in the last chapter why data should be immutable, so the only write operation that exists is adding a brand new piece of data to the system.

Next is how the data is going to be read. The purpose of the batch layer is computing functions of the complete dataset to produce its batch views, so this means that the batch layer storage system needs to be good at reading lots of data at once. Its nature of computing functions on the complete dataset means the batch layer does not need random access to individual pieces of data.

Given the read and write access patterns needed on the batch layer, let's generate a list of requirements for the data storage:

1. **Scalable storage:** The batch layer stores the complete dataset, so it must be easy to scale the storage to vast amounts of data, whether terabytes or petabytes.
2. **Scalable computation:** The batch layer computes functions of the complete dataset, so the storage must be amenable to running computation on the data in a scalable way. The only way to scale a computation on so much data is to parallelize it, so it must be possible to parallelize the computation of a function on the batch storage.
3. **Efficient appends of new data:** The basic write operation is adding new pieces of data to the system, which means it must be easy and efficient to append a new set of data objects to the master dataset.
4. **Ability to vertically partition data:** Although the batch layer is built to run functions on all your data at once, many computations don't require looking at all the data at once. For example, you may have a computation that only needs to look at your "birthday" data, or you may have a computation that only needs to look at data within a time range. Storing different kinds of data separately is called "vertical partitioning" and goes a long way towards making the batch layer efficient.
5. **Ability to tradeoff storage costs with processing performance:** Storage costs money. You should have the ability to compress your data (at the possible cost of performance) to meet your performance/cost requirements. This is especially important in the batch storage context because you're dealing with so much data.

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

3.2 Hadoop Distributed Filesystems

The Hadoop project consists of two subprojects: the Hadoop Distributed Filesystem (HDFS) and Hadoop MapReduce. HDFS exposes a filesystem-like interface for data storage, and MapReduce is a computation system for doing parallel batch computations at scale. The design of HDFS gives it the perfect properties necessary for meeting the requirements for storage in the batch layer. Let's briefly review that design and then look at how to use HDFS to store a master dataset.

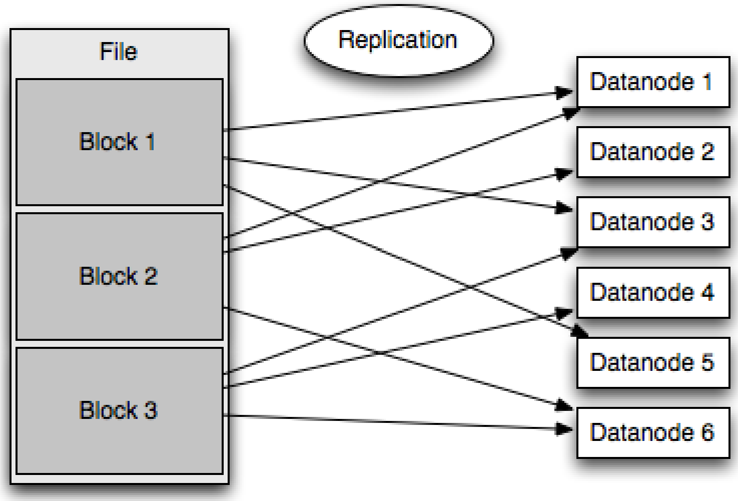


Figure 3.2 File storage on HDFS

Figure 3.2 shows how data is stored on HDFS. Like a regular filesystem, HDFS stores files and folders. A file in HDFS is chunked into fixed-size blocks. The blocks for a single file are spread across many nodes and is typically between 64MB and 256MB. Distributing a single file in this way across many nodes allows it to be easily processed in parallel. Additionally, each block is replicated across multiple nodes so that it is available for processing even when nodes are down (replicated 3 times by default). The use of replication makes HDFS fault-tolerant to the loss of machines in the cluster.

One of the intricacies of HDFS is that it's bad with small files. There can be an order of magnitude performance difference between a job that consumes 10GB stored in many files vs that same data stored in a few files. 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 affects how you'll want to store and process data as you'll want to make sure to avoid small files.

That's all you really need to know about distributed filesystems to begin making use of them. Appendix A contains more detailed information on HDFS if you're interested. Now that you have a general sense of how distributed filesystems work, let's explore how to store a master dataset using HDFS.

3.3 Storing a master dataset with HDFS

Storing a master dataset on HDFS is really quite straightforward. You store data units sequentially in files, with each file containing megabytes or gigabytes of data. You store all the files for a dataset together in a single folder on the filesystem. To add new data to the dataset, you simply add a new file containing the new data units. Storing a master dataset as files within a folder is illustrated in Figure 3.3.

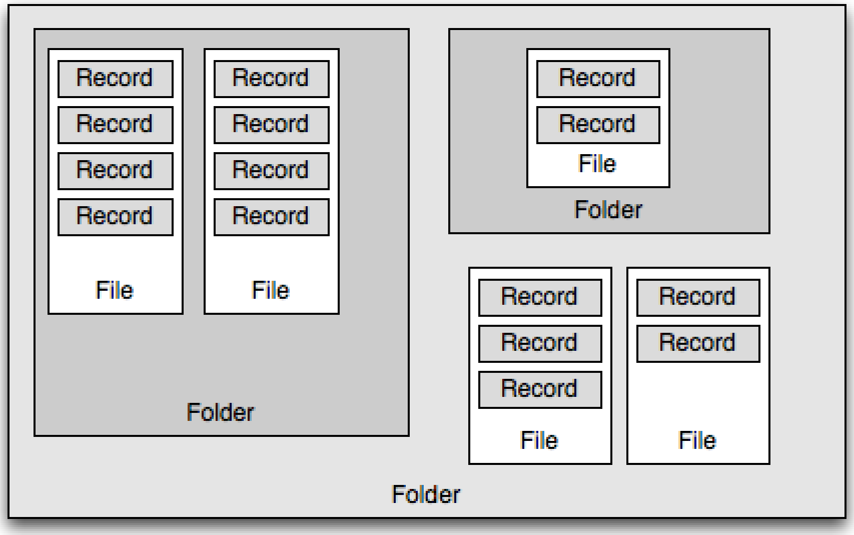


Figure 3.3 Storing master dataset on HDFS

Vertical partitioning is similarly easy to implement using the files and folders abstractions. To store different types of data within a master dataset separately, you just store the different types within different subfolders. For example, you might store your "age" data in one subfolder and your "gender" data in another folder.

Let's take a look at how HDFS satisfies the requirements for storage on the batch layer.

1. **Scalable storage:** HDFS evenly distributes the storage across a cluster of machines. You increase storage space, write throughput, and read throughput by adding more machines.
2. **Scalable computation:** HDFS integrates with Hadoop MapReduce, which can compute nearly arbitrary functions on the data stored in HDFS.
3. **Efficient appends of new data:** Appending new data is as simple as adding a new file to the folder containing the master dataset.
4. **Ability to vertically partition data:** Vertical partitioning is done grouping data into subfolders. Then a reader can look only at a select set of subfolders to read the subset of data it needs.
5. **Ability to tradeoff storage costs with processing performance:** You have full control over how you store your data units within the HDFS files. You can choose the file format you use as well as whether or not you use compression.

HDFS clearly meet the requirements for the batch layer. However, there are a lot of details to work out, such as:

- How do you know what file format data is stored in when you read it?
- How do you append two sets of data together?
- How do you deal with small files if they occur?
- How do you ensure the structural integrity of a master dataset? For example, how do you ensure that the vertical partitioning is enforced (data units of one type never get written to the wrong subfolder)?

While you can use the HDFS API directly to implement the batch layer, it turns out this can lead to many problems.

3.4 Problems with using the HDFS API directly

Let's do a thought experiment to see the problems you can run into when using the HDFS API directly. As you're about to see, using the filesystem API is cumbersome and you'll benefit from having a higher level abstraction to work with.

Suppose you have two folders containing files of data. Let's call them "F1" and "F2". You want to append the contents of F2 into F1 in a way that's scalable and robust. Appending one dataset into another dataset is a common operation that you might use to add new data to your master dataset.

The easiest approach is to use Hadoop's built-in "distributed copy" (distcp) command to copy the files from F2 into F1. Unfortunately, this won't work out of the box. If F1 and F2 both contain files of the same name, doing a straight distcp will override files in F1 with the files from F2. This will cause you to lose data. So before you do the distcp, you have to rename the files in one of the two locations to avoid this data loss.

There's still more issues to consider. If F1 and F2 have different file formats, you can't just copy the files from F2 into F1. You need to change the format of the files so that the whole folder is consistent. The way to change the format of the files is read the records individually in the input file and write them out individually in the new format into the output file. So you'll need to implement a distcp-like task on top of MapReduce that parallelizes this operation for you. Let's call this operation the coercer.

It turns out doing a distcp is much faster than doing a coerce. So if you want your code to be as performant as possible, it should use distcp when F1 and F2 have the same format, and use the coercer otherwise.

It gets worse. Suppose F1 is vertically partitioned by the type of data it has (each type of data is in a different subfolder). You need to make sure F2 is vertically partitioned in the same way before you copy files over, or else you'll corrupt the vertical partitioning in F1. But F1 and F2 only contain files and folders -- they don't tell you how they're supposed to be vertically partitioned. It's up to you as a programmer to never make the mistake of appending a differently partitioned set of data to F1. Programmers, of course, make mistakes, so it would be much better if you were prevented from making a big mistake like that in the first place.

It's clear that there's a lot of "stuff" that needs to happen to do a simple append operation properly. What you really want to do is something like this:

```
Pail source = new Pail("/tmp/source");
Pail target = new Pail("/tmp/target");
target.copyAppend(source);
```

This code does the append in one line of code, and it throws an exception if the append is invalid for any reason. This code does a distributed copy when the datasets are of the same format and coerces the data otherwise. This code does using 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.

3.5 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). Pail makes it significantly easier to manage a collection of records in a batch processing context. The Pail abstraction frees you from having to think about file formats and greatly reduces the complexity of the storage code. It makes it easy to vertically partition a dataset, and it provides a dead-simple API for common operations like appends, compression, and consolidation. Pail is just a Java library and underneath uses the standard file APIs provided by Hadoop. Pail makes it easy to satisfy all the requirements for storage on the batch layer.

You treat a pail like an unordered collection of records. Internally, those records are stored across files that can be nested into arbitrarily deep subdirectories, just like we described in the beginning of the chapter. This is illustrated again in Figure 3.4.

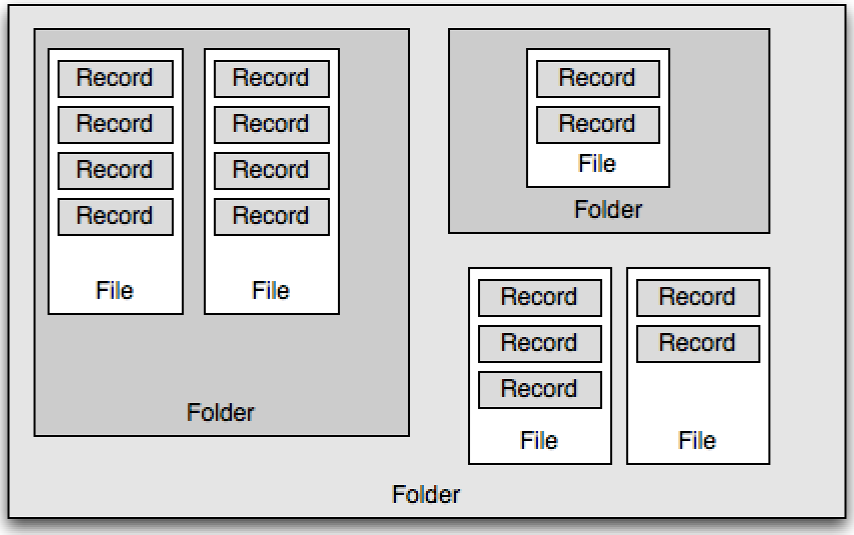


Figure 3.4 How records are stored in a Pail

Let's dive right into the code to see how Pail works. You'll start off looking at how to do basic operations with Pail, like add records and specify how to serialize and deserialize objects of any type. Then you'll look at how Pail makes doing mass record operations like appends and consolidations dead-simple. Then, you'll see how to use Pail to automatically vertically partition your dataset, and finally you'll see how you can tune the file format Pail uses to enable things like compression.

3.5.1 Basic Pail operations

The best way to understand how Pail works is to follow along and run these commands on your local computer. Your local filesystem will be treated as HDFS in the examples, so you'll 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 at "/tmp/mypail":

```
Pail pail = Pail.create("/tmp/mypail");
```

If you look on your filesystem, you'll see that a folder for "/tmp/mypail" was created and a file called "pail.meta" exists inside. Ignore that file for now as we'll come back to it later.

As you can see, Pail.create returned a Pail object that you can manipulate. Let's use that object to add some records to the pail. This pail stores raw binary data. The following code shows how to write some byte arrays to it.

```
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 look inside /tmp/mypail, you'll see a new file inside that contains the records. 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. Since files are named using globally unique names, a pail can be written to concurrently by multiple writers with no conflicts. Additionally, a reader can read from a pail while it's being written to without having to worry about half-written files.

If a pail already exists and you want to open it, you just use Pail's constructor to get an instance of it. Remember, Pail is backed by a distributed filesystem so multiple processes can be reading/writing to the same pail. Here's an example of opening an existing pail:

```
Pail pail_same = new Pail("/tmp/mypail");
```

If you ran this code on a directory that isn't part of a pail, you would get an exception.

3.5.2 Typed pails

You don't have to work with binary records when using Pail. Pail lets you work with real objects rather than binary records. At the file level, data is stored as a sequence of bytes. To work with real objects, you provide Pail with information about what type your records will be and how to serialize and deserialize objects of that type to and from binary data. You provide this information by implementing the PailStructure interface. Here's how to define the PailStructure for a pail that stores integers:

```
public class IntegerPailStructure implements PailStructure<Integer> {
    public Integer deserialize(byte[] bytes) {
        try {
            return new DataInputStream(
                new ByteArrayInputStream(bytes)).readInt();
        } catch (IOException e) {
           throw new RuntimeException(e);
        }
    }

    public byte[] serialize(Integer t) {
        ByteArrayOutputStream os = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(os);
        try {
            dos.writeInt(t);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        return os.toByteArray();
    }

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

    public List<String> getTarget(Integer t) {
        return Collections.EMPTY_LIST;
    }

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

To create an integer pail, you simply pass that PailStructure in as an extra argument on Pail.create:

```
Pail<Integer> intpail = Pail.create("/tmp/intpail",
  new IntegerPailStructure());
```

Now when writing records to the pail, you can give it integer objects directly and the Pail will handle the serialization for you. This is shown in the following code:

```
TypedRecordOutputStream int_os = intpail.openWrite();
int_os.writeObject(1);
int_os.writeObject(2);
int_os.writeObject(3);
int_os.close();
```

Likewise, when you read from the pail the pail will deserialize records for you. Here's how you can iterate through all the objects in the integer pail you just wrote to:

```
for(Integer record: intpail) {
    System.out.println(record);
}
```

This code will print out integers, just as you expect.

3.5.3 Appends

Pail has a number of common operations built into it as methods. These operations are where you really start to see the benefits of managing your records with Pail rather than managing files yourself. These operations are all implemented using MapReduce so they scale to however much data is in your pail, whether gigabytes or terabytes. We'll be talking about MapReduce a lot more in later chapters, but the gist of it is that the operations are automatically distributed across a cluster of worker machines.

One of Pail's operations is the append operation. Using the append operation, you can add all the records from one pail into another pail. Here's an example of appending a pail called "source" into a pail called "target":

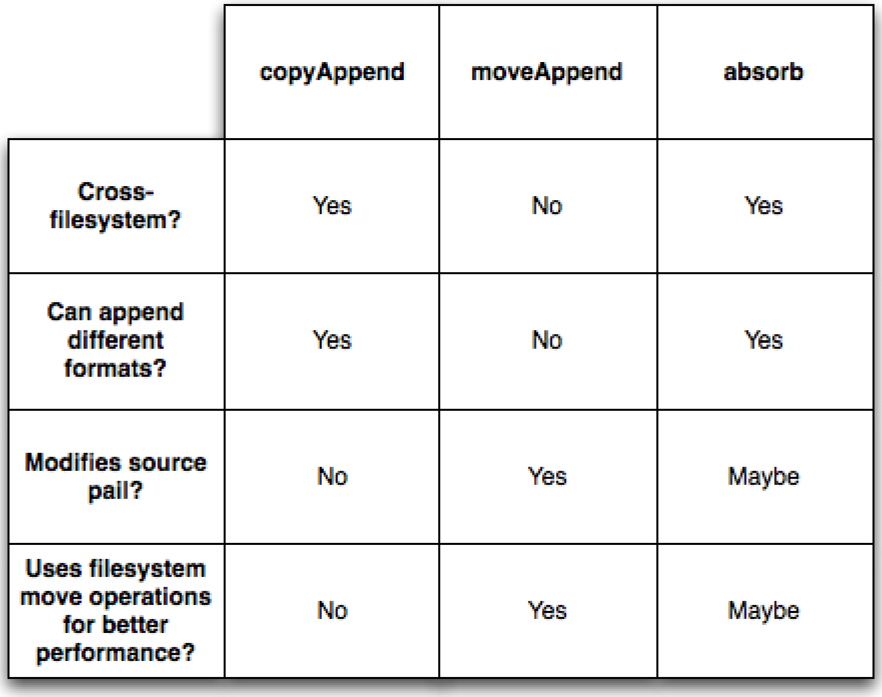
```
Pail source = new Pail("/tmp/source");
Pail target = new Pail("/tmp/target");
target.copyAppend(source);
```

As you can see, the code is super simple. It automates all the stuff we saw you had to do when using files and folders directly. Pail lets you focus on what you want to do with your data rather than worry about how to manipulate files appropriately.

Unlike the examples so far, the append operation won't complete instantly because it's launching an entire MapReduce job. It will block until the operation is complete, and it will throw an exception if for some reason the job failed.

The append operation is smart. It checks the pails to make sure it's valid to append the pails together. So for example, it won't let you append a pail that stores strings into a pail that stores integers.

There's a few kinds of appends you can use. XREF append_table compares these different appends. The append we used in our example was the "copyAppend" which performed the append by copying all the records from the source pail into the target pail. The copyAppend does not modify the source pail and can even be used to copy a pail between two separate distributed filesystems.



	copyAppend	moveAppend	absorb
Cross-filesystem?	Yes	No	Yes
Can append different formats?	Yes	No	Yes
Modifies source pail?	No	Yes	Maybe
Uses filesystem move operations for better performance?	No	Yes	Maybe

Figure 3.5 Comparing different kinds of appends

Another append is the "moveAppend". The moveAppend works by doing filesystem move operations to move the source pail's files into the target pail. The moveAppend operation is much faster than a copyAppend, but it has some restrictions. You can't moveAppend between two pails if they're stored on different filesystems or are using different file formats to store the data (more on file formats in a few sections). If you just want to append the contents of a pail into another pail in the most efficient way possible, and don't care if the source pail is modified, use the "absorb" operation. "Absorb" will do a moveAppend if possible; otherwise it will fall back on a copyAppend.

3.5.4 Consolidation

Sometimes your records end up spread across lots of small files. As discussed earlier in this chapter, this has a major performance cost associated with it when you want to process that data in a MapReduce job since MapReduce will need to launch a lot more tasks.

The solution is to combine those small files into larger files so that more data can be processed in a single task. Pail supports this directly by exposing a "consolidate" method. This method launches a MapReduce job that combines files into larger files in a scalable and efficient way. Here's how you would consolidate a pail:

```
pail.consolidate();
```

By default, Pail combines files to create new files that are as close to 128MB as possible. You can configure the target file size by passing the file size in as an argument to consolidate. 128MB is a good default because it is a typical block size used in HDFS installations.

3.5.5 Structured Pail

So far we've treated Pail as an unordered collection of records. Pail lets you provide structure to your records by organizing them into subdirectories as illustrated in Figure 3.6. For example, if you're storing data about people, you can store ages, genders, and locations in separate subdirectories. As another example, you could organize your data by time and store each hour of data in a separate subdirectory. This is called "vertical partitioning."

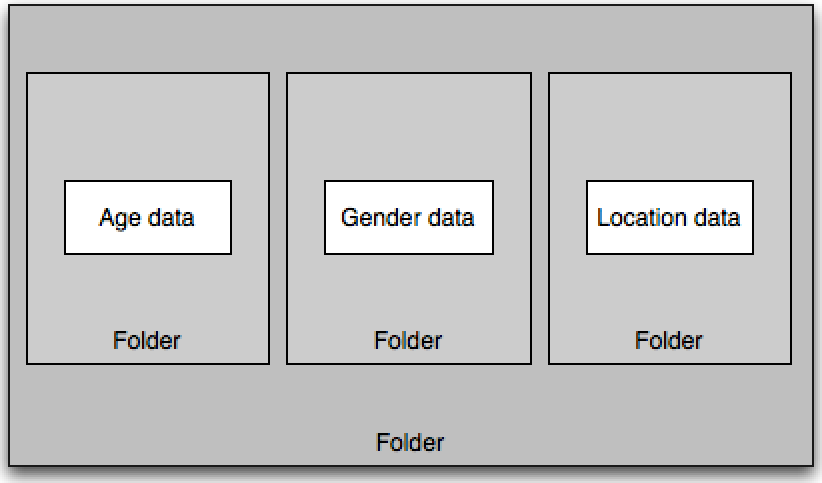


Figure 3.6 Vertical partitioning

By structuring the data, you can make batch processing computations much more efficient. You're able to avoid reading data you don't care about in your jobs, greatly lessening the cost of the "scan and filter" approach to batch processing. If you're only interested in consuming location data, you can ignore the data in the other subdirectories.

To create a structure for a pail, you have to implement two more methods in PailStructure. 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.

Let's look at an example that creates a PailStructure that stores integers and stores odd and even numbers separately. Odd numbers will go into the "odd" subdirectory, and even numbers will go into the "even" subdirectory. Here's how to define the PailStructure:

```
public class OddEvenIntegerPailStructure extends IntegerPailStructure {
    @Override
    public List<String> getTarget(Integer t) {
        List<String> ret = new ArrayList<String>();
        String classifier;
        if(t % 2 == 0) {
            classifier = "even";
         } else {
            classifier = "odd";
         }
        ret.add(classifier);
        return ret;
    }

    @Override
    public boolean isValidTarget(String... strings) {
        if(strings.length==0) return false;
        return strings[0].equals("even") || strings[0].equals("odd");
    }
}
```

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. For example, it is used during consolidation so that Pail doesn't combine files across directories that contain records that should be partitioned separately.

Let's look at the effect of the odd/even pail structure by writing some records to a pail of this structure:

```
Pail<Integer> odd_even_pail =
  Pail.create("/tmp/oddeven",
              new OddEvenIntegerPailStructure());
TypedRecordOutputStream oe_os = odd_even_pail.openWrite();
oe_os.writeObject(1);
oe_os.writeObject(2);
oe_os.writeObject(3);
oe_os.writeObject(4);
oe_os.writeObject(5);
oe_os.close();
```

The odd and even numbers get written to separate files in separate subdirectories. You can verify this by printing out the records in the odd subpail and the even subpail separately:

```
System.out.println("Odd numbers:");
for(Integer record: odd_even_pail.getSubPail("odd")) {
    System.out.println(record);
}
System.out.println("Even numbers:");
for(Integer record: odd_even_pail.getSubPail("even")) {
    System.out.println(record);
}
```

If you run this code, you'll see that the odd and even numbers print separately and the PailStructure works as advertised.

Pail is smart about enforcing the structure of a pail. Whenever you write data to it or do operations, it makes sure that the structure is never violated. For example, when you append two pails together, Pail makes sure they have the same structure. Imagine trying to manage the vertical partitioning without the pail abstraction. It's all too easy to forget that two datasets are partitioned differently and accidentally do an append. If you did this, you'd ruin one of the datasets. Pail protects you from making these kinds of mistakes besides providing a very simple interface to doing these operations.

3.5.6 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. It will automatically coerce any data added to that Pail to that file format.

A file is just a sequence of bytes. A file format is a protocol for storing a sequence of records in a file. Figure 3.7 shows a simple format that prepends a record's size before the record.

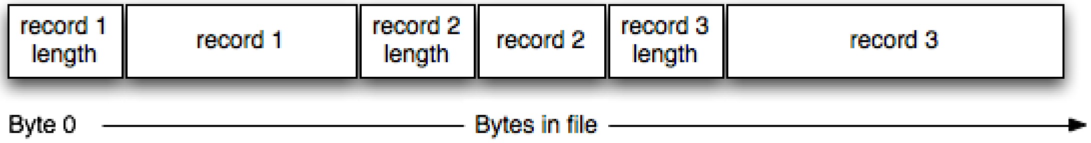


Figure 3.7 A basic file format for storing records

If you want to create your own file format, you would implement the PailFormat class. You could then create a pail of that format using an alternate Pail.create method in which you give it the fully qualified classname of a PailFormat to use:

```
Pail customPail =
  Pail.create("/tmp/custompail",
              new PailSpec("manning.example.SimpleFormat"));
```

By default, Pail uses the Hadoop SequenceFile format. 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:

```
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);
Pail compressedPail =
  Pail.create("/tmp/compressed",
              new PailSpec("SequenceFile", options));
```

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.

If you have an existing pail and want to create an equivalent pail of a different file format, you can use Pail's coerce method. Like the other operations, coerce runs as a MapReduce job and is scalable. Here's an example of compressing the data in one pail into a new pail:

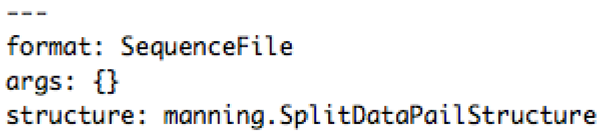
```
Map<String, Object> compressOptions =
  new HashMap<String, Object>();
compressOptions.put(SequenceFileFormat.TYPE_ARG,
                    SequenceFileFormat.TYPE_ARG_BLOCK);
```

```
compressOptions.put(SequenceFileFormat.CODEC_ARG,
                    SequenceFileFormat.CODEC_ARG_GZIP);
pail.coerce("/tmp/compressed_copy",
            "SequenceFile",
            compressOptions);
```

Pail is smart about dealing with pails with different types. When you append pails together, Pail will automatically coerce the file types if necessary, and this all happens transparently to you. If you were just working with files directly, dealing with files of different formats would be painful. Your application logic would have to know which files are of which types and manually do coercions when necessary. Since Pail knows its own format, this is all automated. Most importantly, Pail protects you from making a mistakenly using the wrong file format code.

3.5.7 Pail implementation

It's helpful to know a few of the details of how Pail is implemented. The location where you create a new pail is called the root, and a special file is stored there called "pail.meta". "pail.meta" stores information about what file format the pail is using and what PailStructure the pail is using. An example of the "pail.meta" file is shown in Figure XREF pail_meta_pic.



```
---
format: SequenceFile
args: {}
structure: manning.SplitDataPailStructure
```

Figure 3.8 pail.meta

When you create a pail, it checks that no pail.meta exists in any parent directory. This ensures that you don't have a pail root within a pail, which wouldn't make any sense. Pail will throw an exception if you try to create a new pail within a pail.

Pail ensures that files always appear atomically within the pail. You never have to worry about reading a half-finished file. This combined with pail's use of globally unique identifiers for filenames lets Pail be used in a multi-reader, multi-writer environment.

3.6 Pail Structure for SuperWebAnalytics.com

In Chapter 2 you developed a graph schema for the SuperWebAnalytics.com data. Pail can be used to store those data objects. In particular, every property and edge can be stored in a different subpail. This lets you do jobs on the data much more efficiently as you can ignore attributes we don't care about reading. To accomplish this, let's implement a PailStructure that inspects the Thrift schema and maps a Data object to the appropriate pail subdirectory.

Figure 3.9 shows what the pail structure looks like on the distributed filesystem. It uses the Thrift field ids specified in the schema 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.

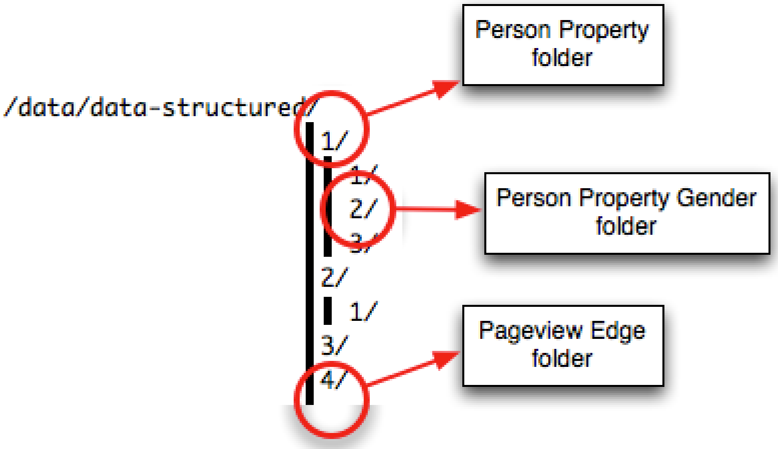


Figure 3.9 Physical structure of SplitDataPail on Distributed Filesystem

The code implementing this PailStructure is shown in Listing 3.1. At a high level, all it does is use the Thrift schema to determine where data objects should go. Don't worry so much about the details of this code. Most of what it's doing is inspecting the Thrift schema in an efficient way. Accomplishing this is a little bit verbose. What matters is that this code works for any graph schema, and it continues to work even as the schema is evolved over time.

Listing 3.1 Pail structure for SuperWebAnalytics.com

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

    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);
        }
    }

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

        }
    }

    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());
        }
    }

    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;
    }
}
```

Now you can use these "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 those records.

3.7 Conclusion

The high level requirements dictated by the Lambda Architecture are straightforward. You saw that as we mapped those requirements to a practical implementation using HDFS, additional requirements popped up. Some of these requirements had to do with the intricacies of HDFS, such as the small files problem, and other requirements had to do with automating common tasks, minimizing complexity, and preventing human error.

It's important to be able to think about and manipulate your data at the record level and not at the file level. By abstracting away file formats and directory structure into the Pail abstraction, you're able to do exactly that. The Pail abstraction frees you from having to think about the details of the data storage while making it easy to do robust, enforced vertical partitioning as well as common operations like appends and consolidation. Without the Pail abstraction, these basic tasks are manual and difficult.

The Pail abstraction plays an important role in making robust batch workflows. However, it ultimately takes up very few lines of code in the code for a workflow. Vertical partitioning happens automatically, and tasks like appends and consolidation are just 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 see 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

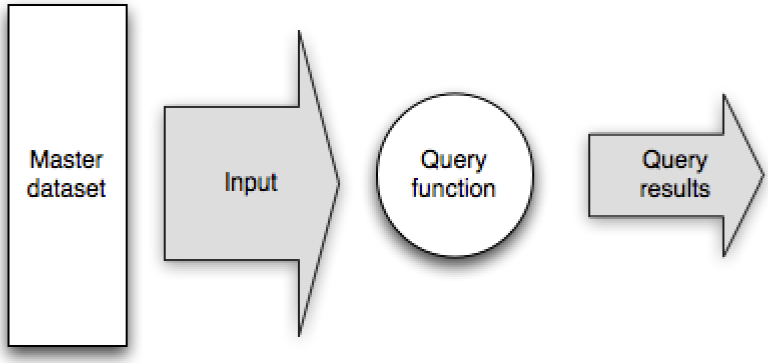


Figure 4.1 Computing queries by running function on the master dataset directly

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.

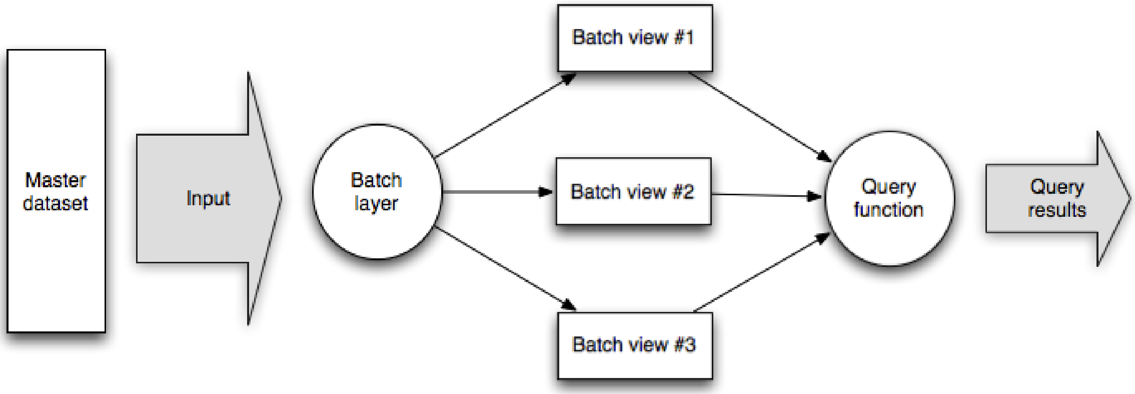
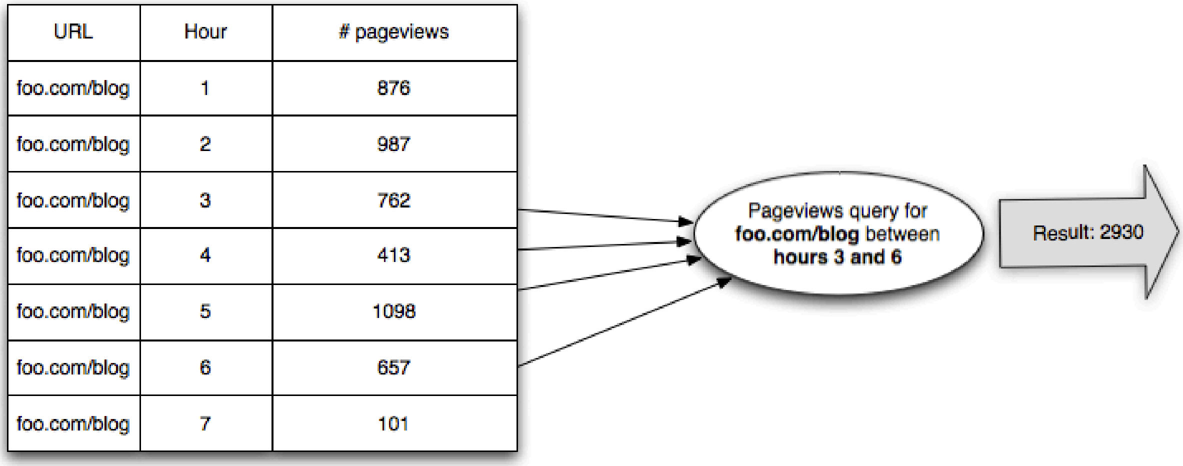


Figure 4.2 Computing queries by running function on precomputed batch views

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.



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

Pageviews query for **foo.com/blog** between **hours 3 and 6**
Result: 2930

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.

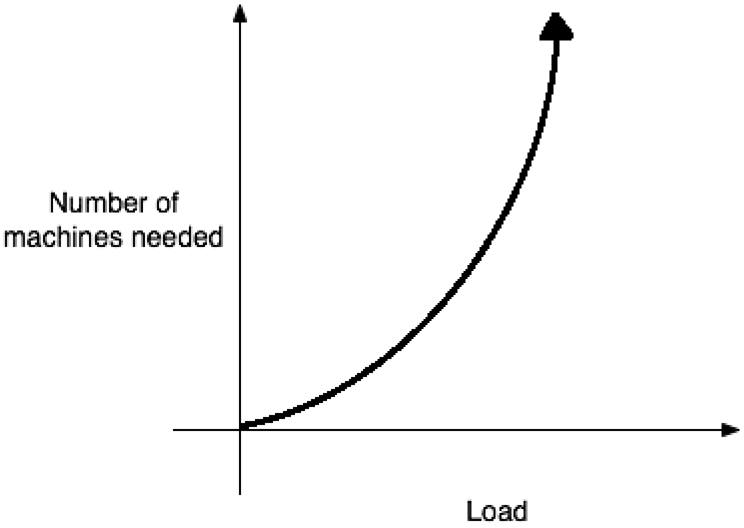


Figure 4.4 Non-linear scalability

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.

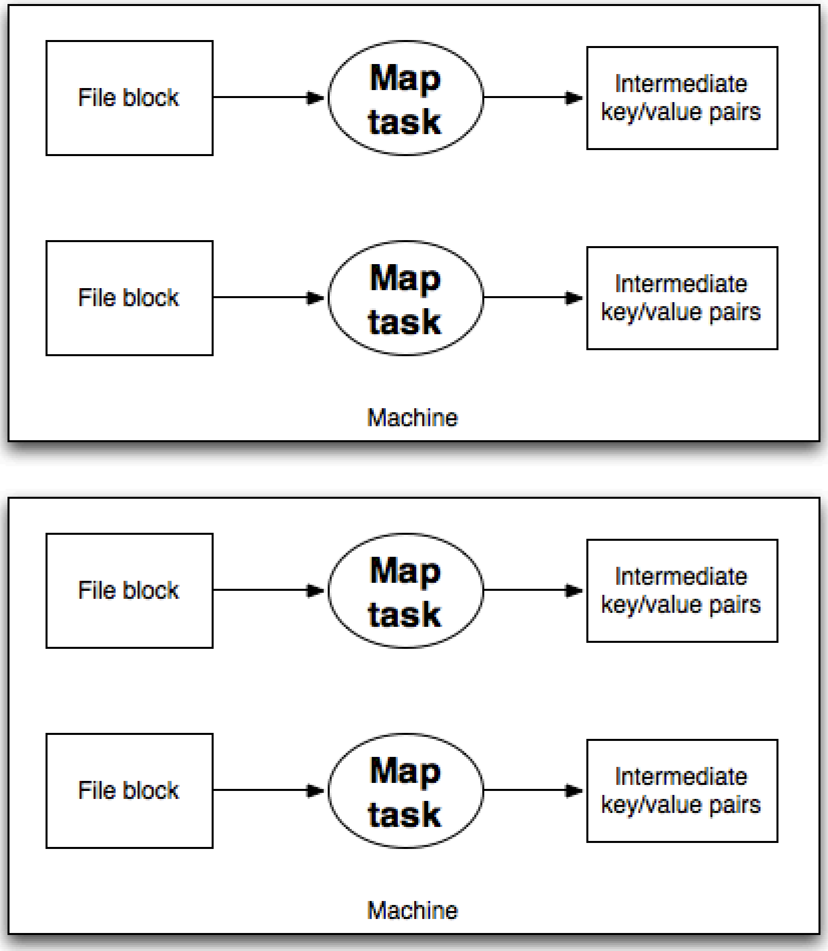

File block
Map task
Intermediate key/value pairs
File block
Map task
Intermediate key/value pairs
Machine
File block
Map task
Intermediate key/value pairs
File block
Map task
Intermediate key/value pairs
Machine

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.

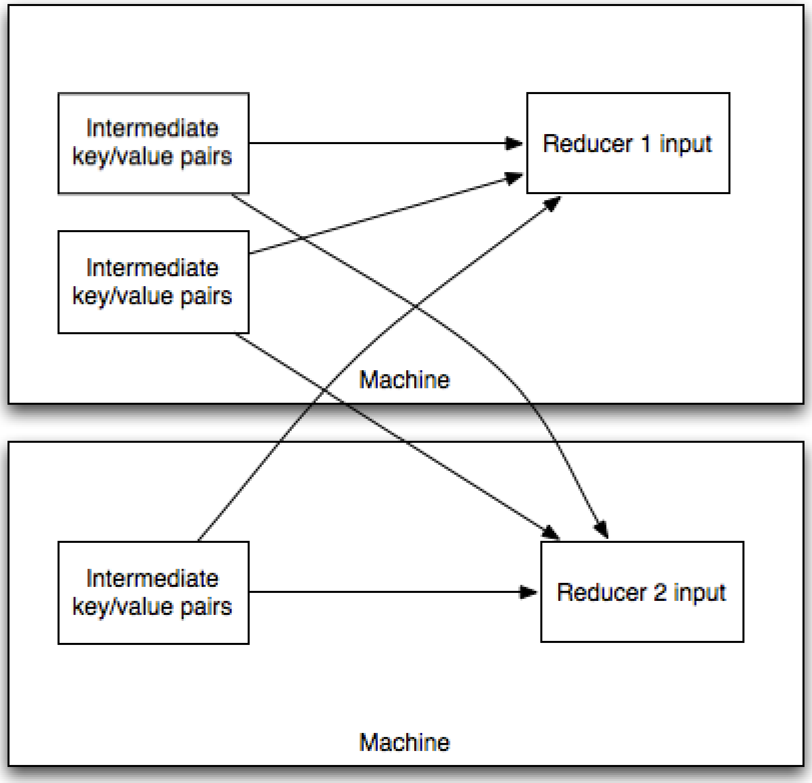

Intermediate key/value pairs
Reducer 1 input
Intermediate key/value pairs
Machine
Intermediate key/value pairs
Reducer 2 input
Machine

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.

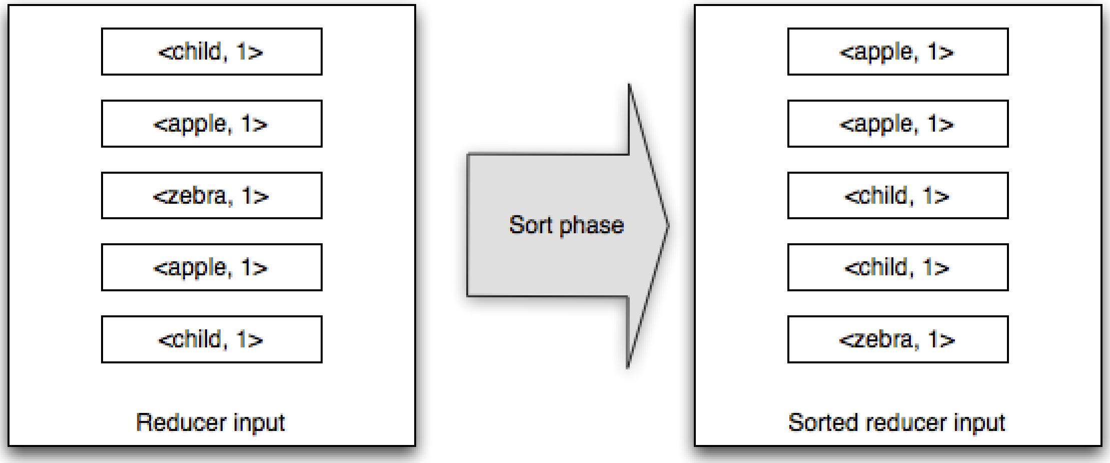


Figure 4.7 Sort phase

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.

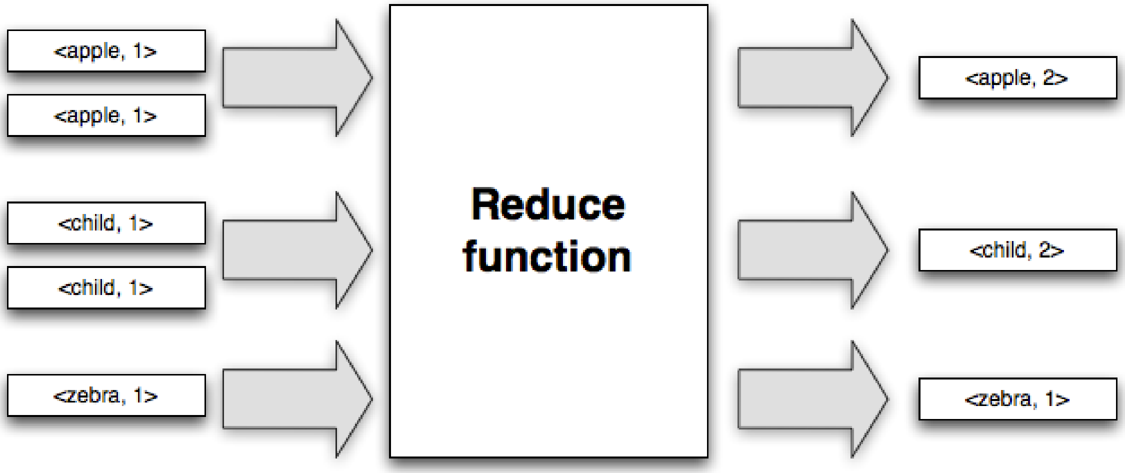


Figure 4.8 Reduce phase

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.

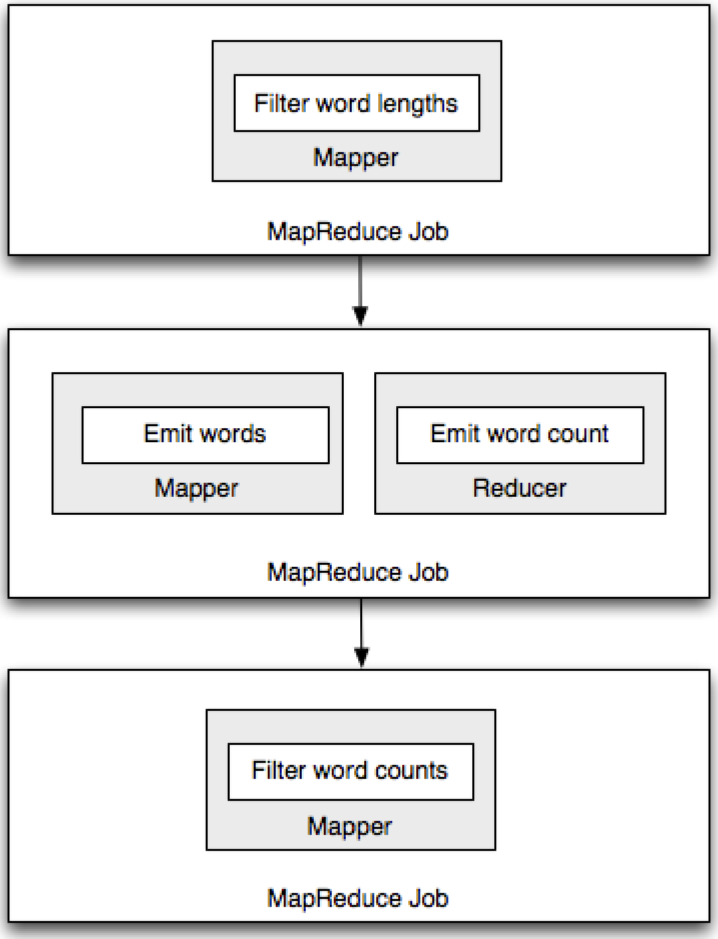


Figure 5.1 Modularized MapReduce jobs

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.

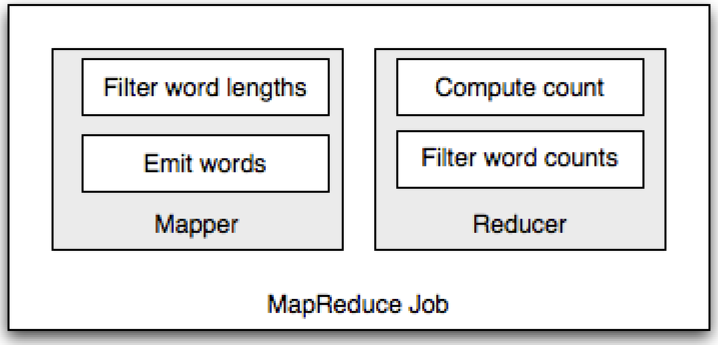


Figure 5.2 Compiling modularized functionality into as few MapReduce jobs as possible

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.

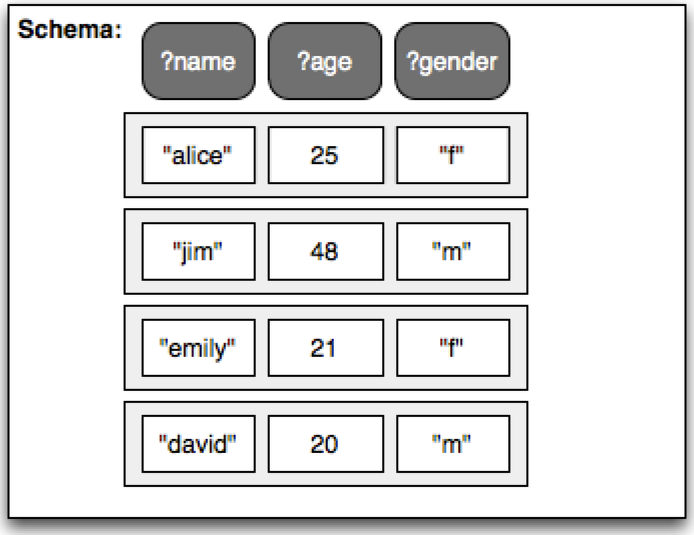


Figure 5.3 Set of tuples with schema

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.

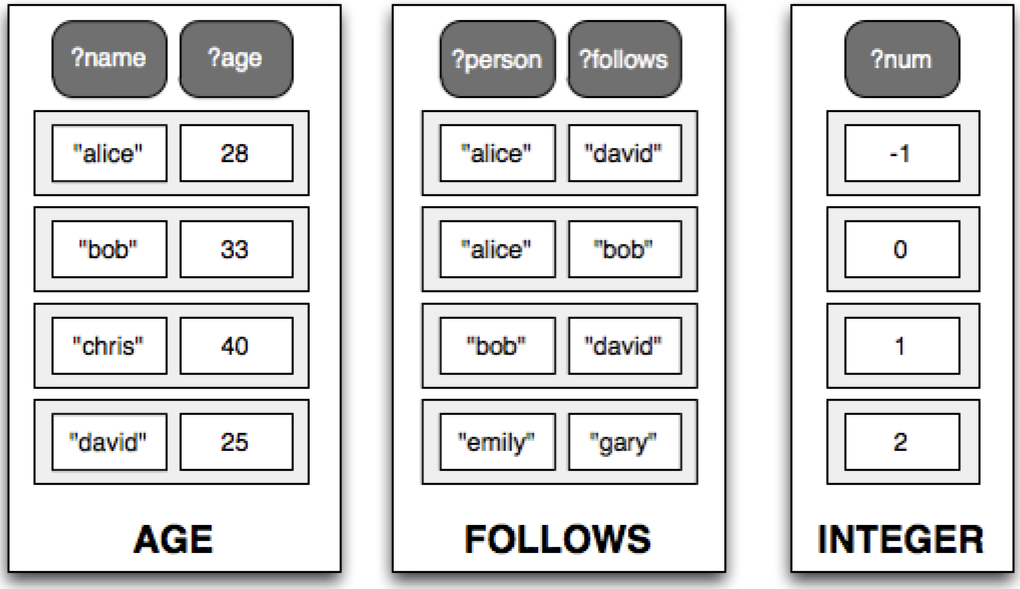
A filter specifies a constraint on a set of inputs fields. The "less than" and "greater than" operations are examples of this kind of predicate.

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.



AGE

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

FOLLOWS

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

INTEGER

?num
-1
0
1
2

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:

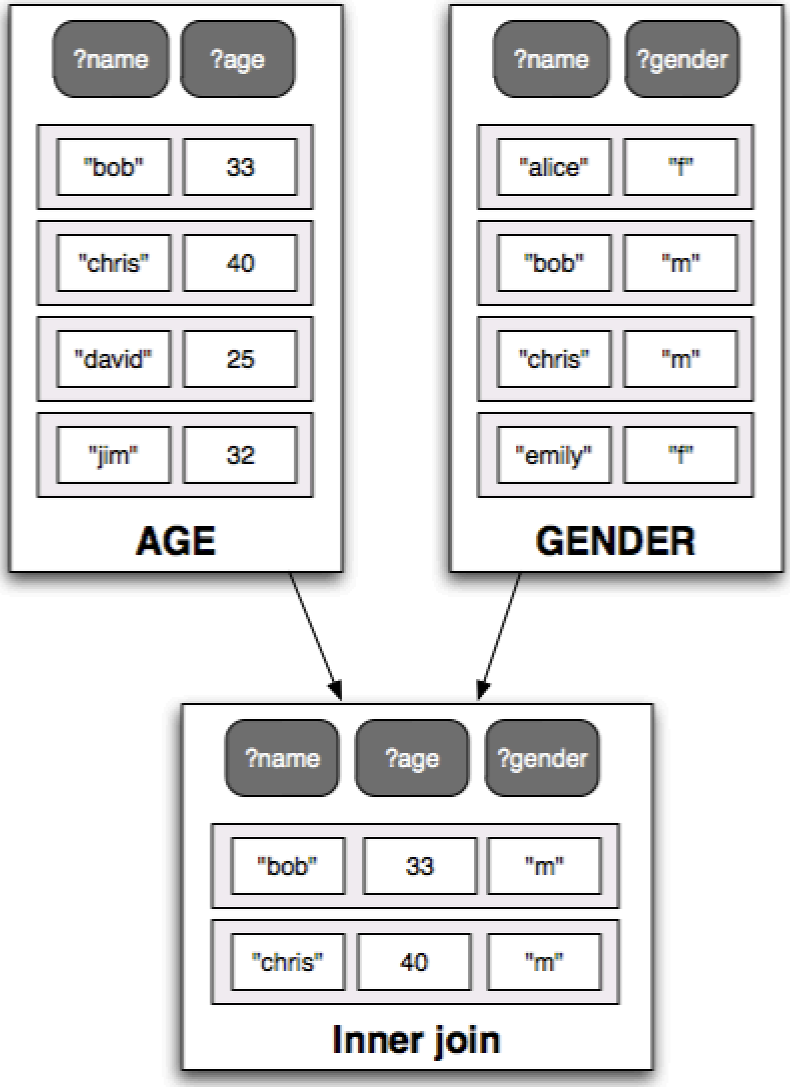
```
new Subquery("?n")
    .predicate(INTEGER, "?n")
    .predicate(new Multiply(), "?n", "?n", "?n").out("?n");
```

?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.



AGE

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

GENDER

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

Inner join

?name	?age	?gender
"bob"	33	"m"
"chris"	40	"m"

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.

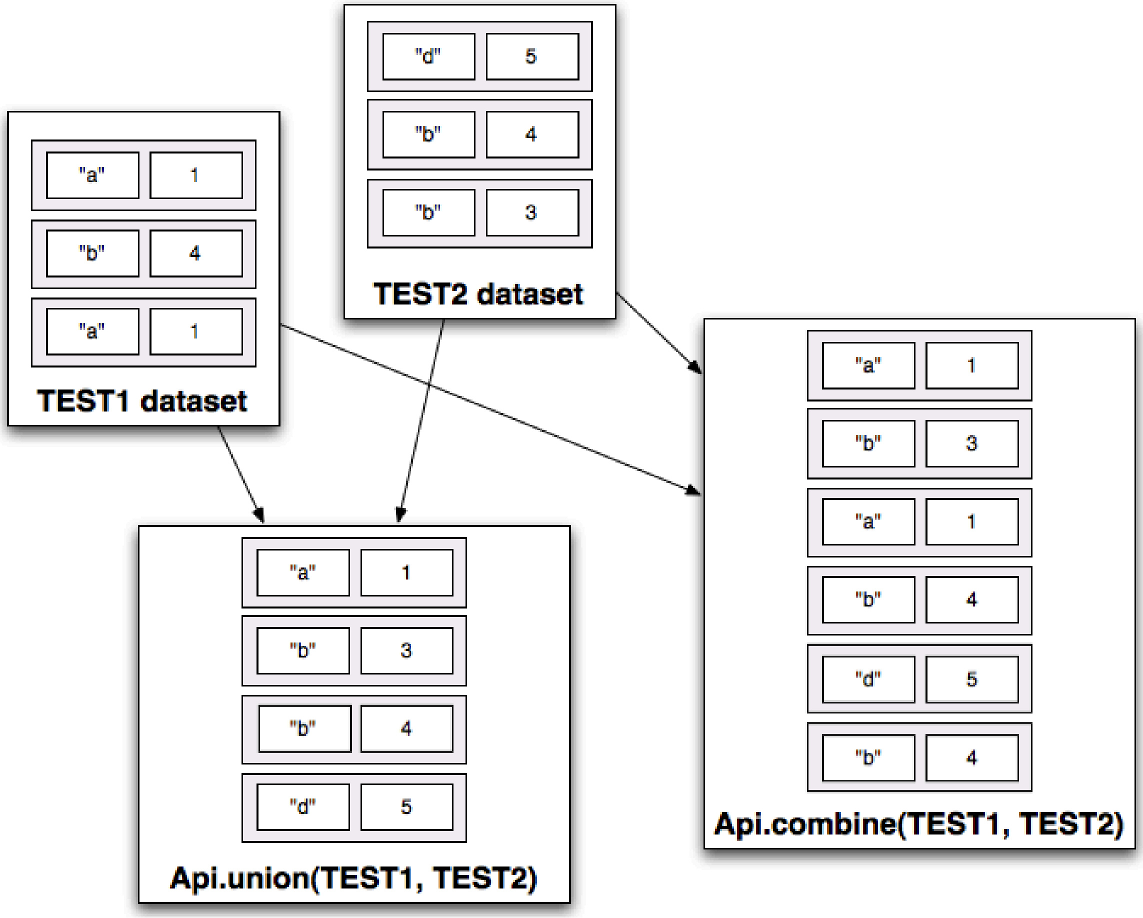


Figure 5.6 Union and combine functions

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.

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

VAL1

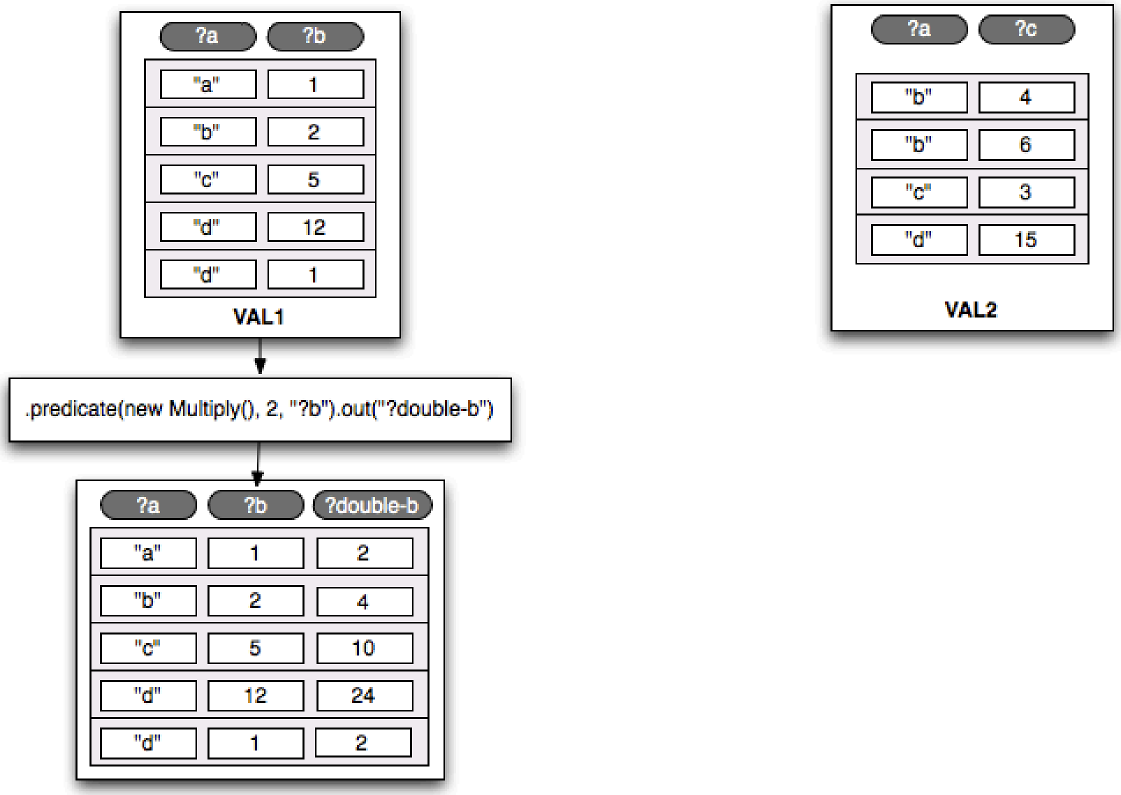
?a	?c
"b"	4
"b"	6
"c"	3
"d"	15

VAL2

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.



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

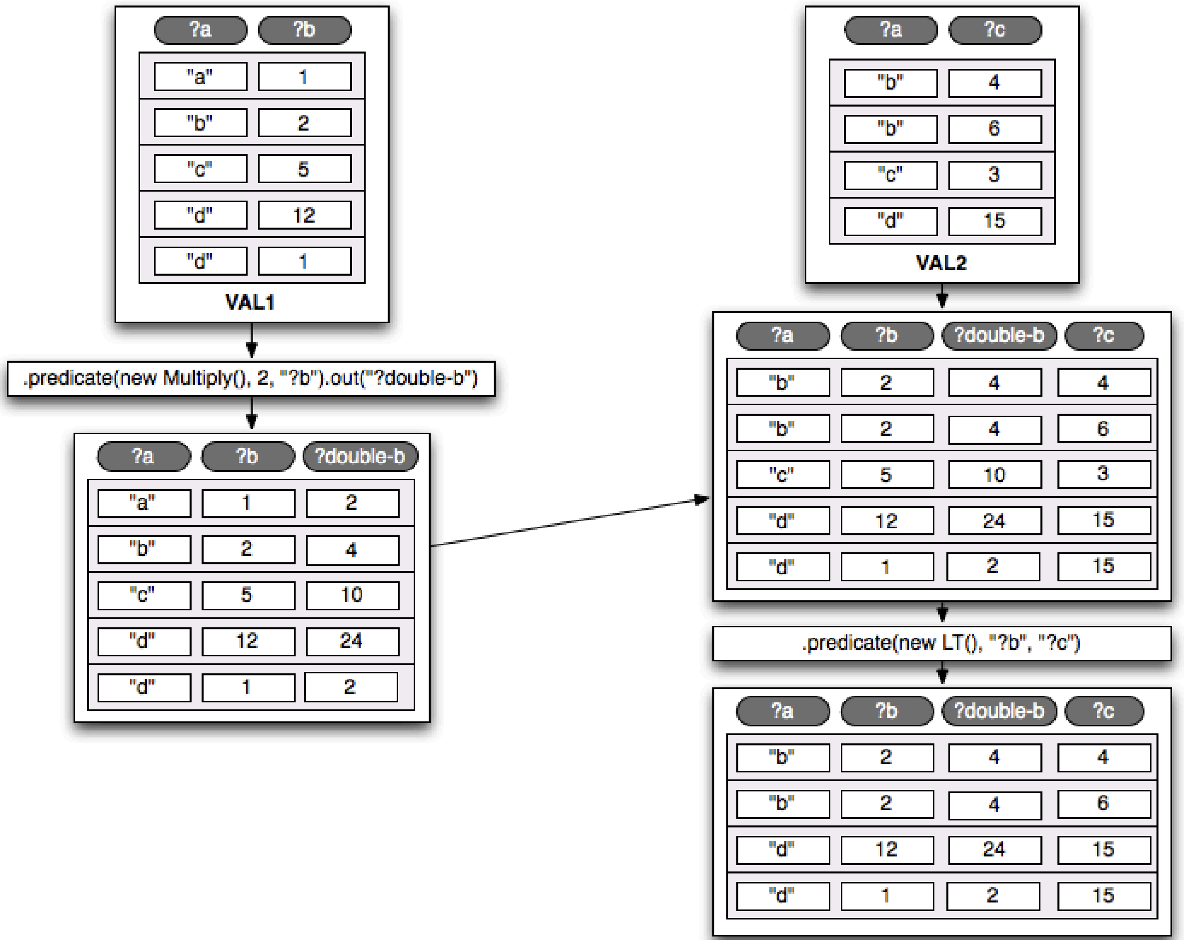
?a	?c
"b"	4
"b"	6
"c"	3
"d"	15

```
.predicate(new Multiply(), 2, "?b").out("?double-b")
```

?a	?b	?double-b
"a"	1	2
"b"	2	4
"c"	5	10
"d"	12	24
"d"	1	2

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.



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

.predicate(new Multiply(), 2, "?b").out("?double-b")

?a	?b	?double-b
"a"	1	2
"b"	2	4
"c"	5	10
"d"	12	24
"d"	1	2

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

?a	?b	?double-b	?c
"b"	2	4	4
"b"	2	4	6
"c"	5	10	3
"d"	12	24	15
"d"	1	2	15

.predicate(new LT(), "?b", "?c")

?a	?b	?double-b	?c
"b"	2	4	4
"b"	2	4	6
"d"	12	24	15
"d"	1	2	15

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.

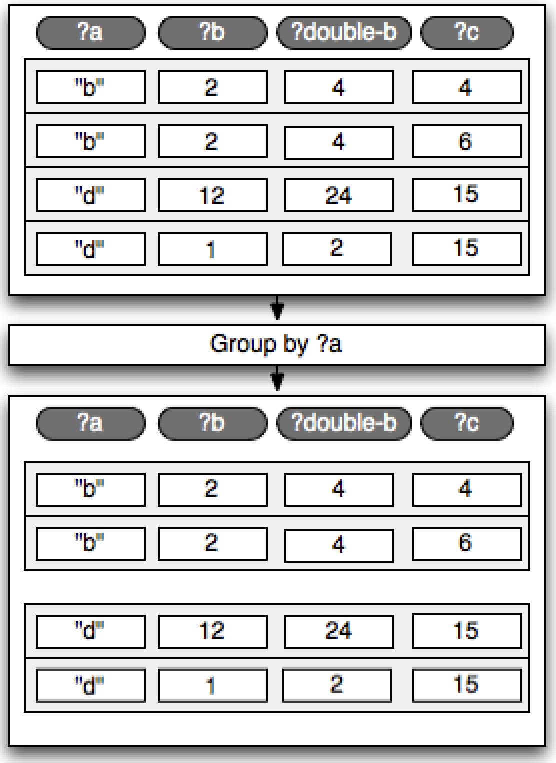
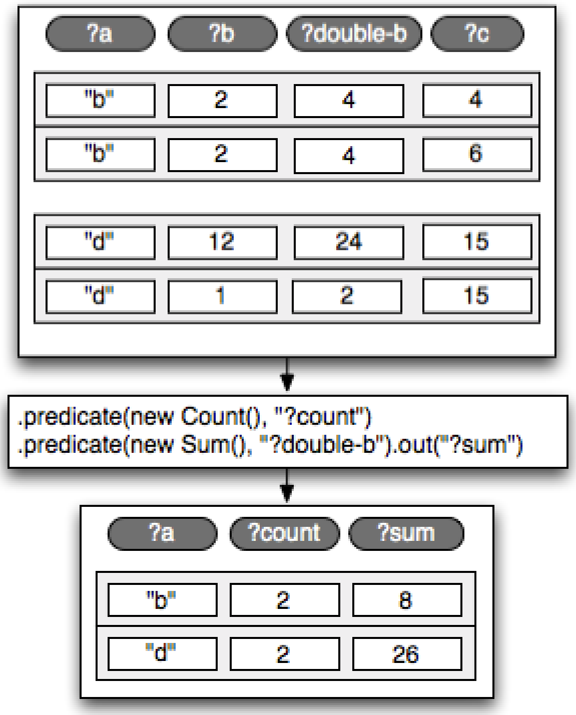


Figure 5.10 Grouping of tuples during aggregation phase

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.



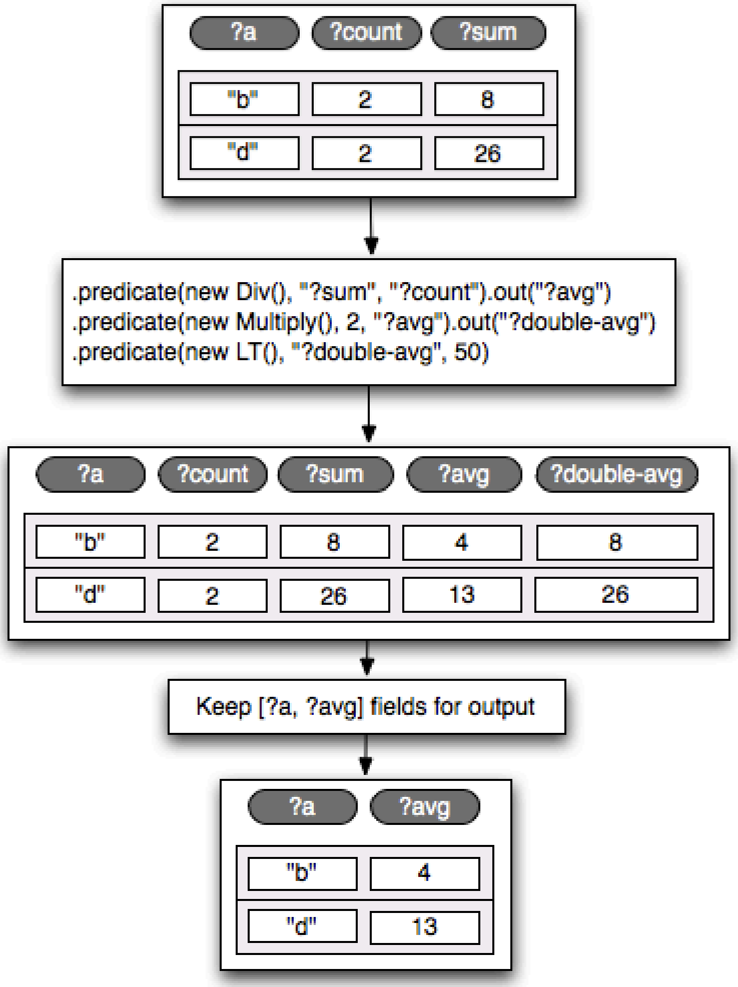
?a	?b	?double-b	?c
"b"	2	4	4
"b"	2	4	6
"d"	12	24	15
"d"	1	2	15

```
.predicate(new Count(), "?count")
.predicate(new Sum(), "?double-b").out("?sum")
```

?a	?count	?sum
"b"	2	8
"d"	2	26

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.



?a	?count	?sum
"b"	2	8
"d"	2	26

```
.predicate(new Div(), "?sum", "?count").out("?avg")
.predicate(new Multiply(), 2, "?avg").out("?double-avg")
.predicate(new LT(), "?double-avg", 50)
```

?a	?count	?sum	?avg	?double-avg
"b"	2	8	4	8
"d"	2	26	13	26

Keep [?a, ?avg] fields for output

?a	?avg
"b"	4
"d"	13

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;
    }
}
```

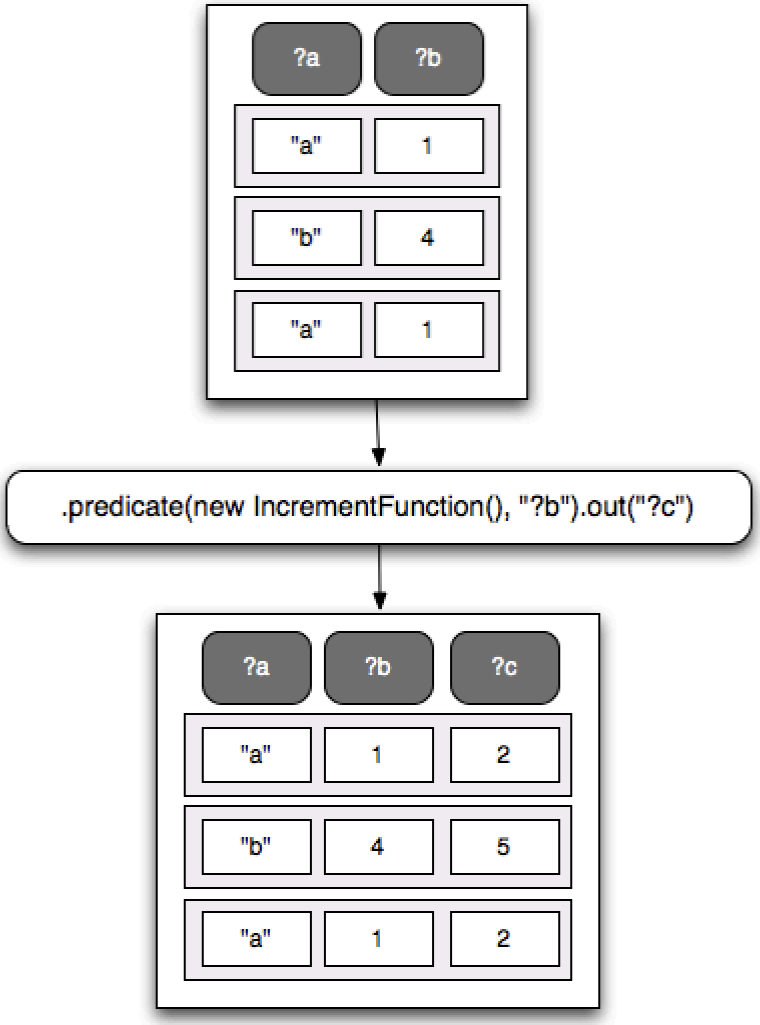
The input arguments are obtained via the *FilterCall#getArguments* method and contains the input fields specified when the predicate is used.

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.



?a	?b
"a"	1
"b"	4
"a"	1

```
.predicate(new IncrementFunction(), "?b").out("?c")
```

?a	?b	?c
"a"	1	2
"b"	4	5
"a"	1	2

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.

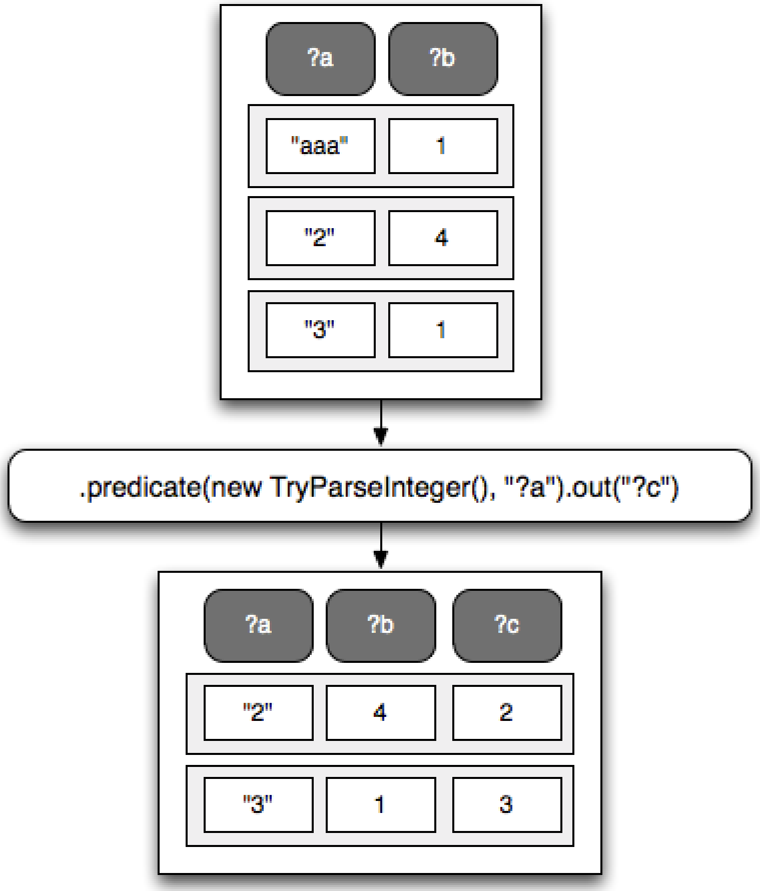


Figure 5.14 A function that also acts as a filter

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.

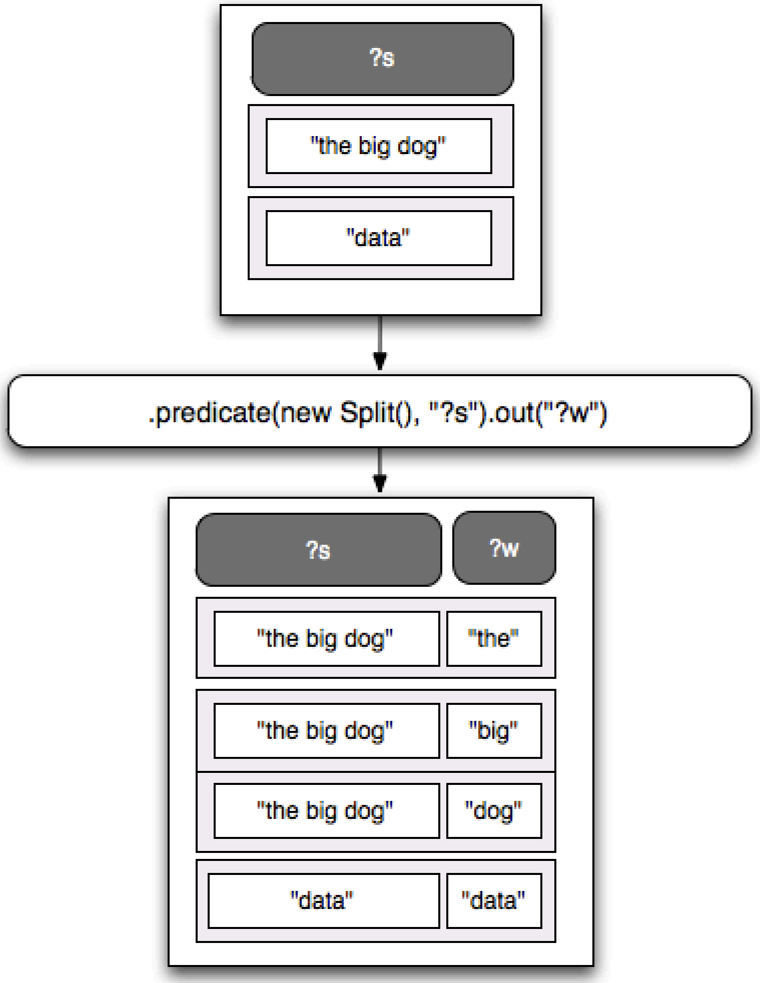


Figure 5.15 A function that that emits multiple tuples

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.

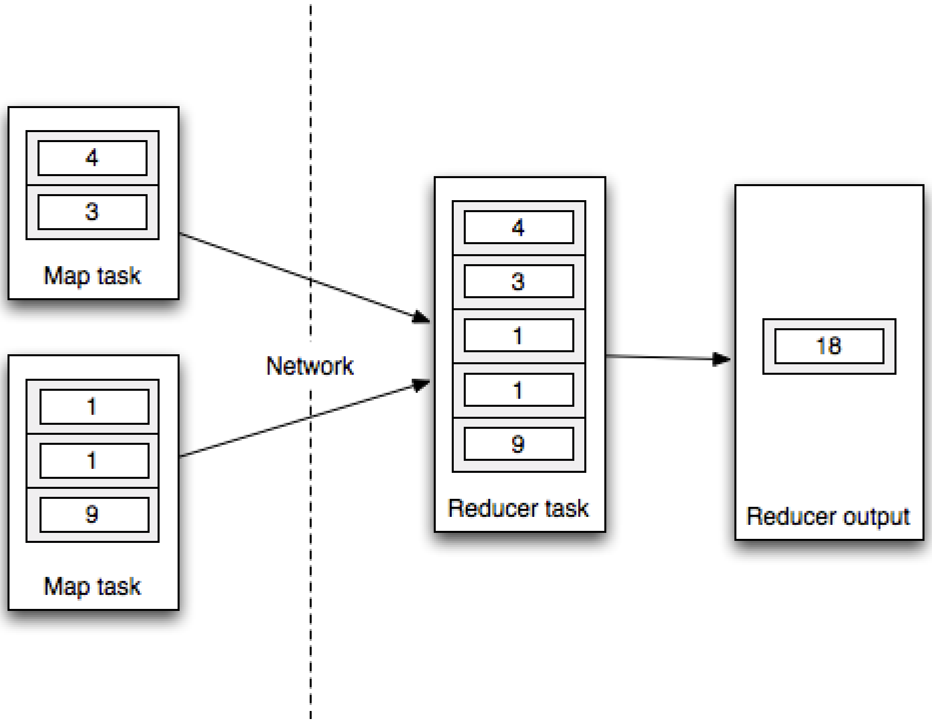


Figure 5.16 Execution of aggregators and buffers at MapReduce level

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.

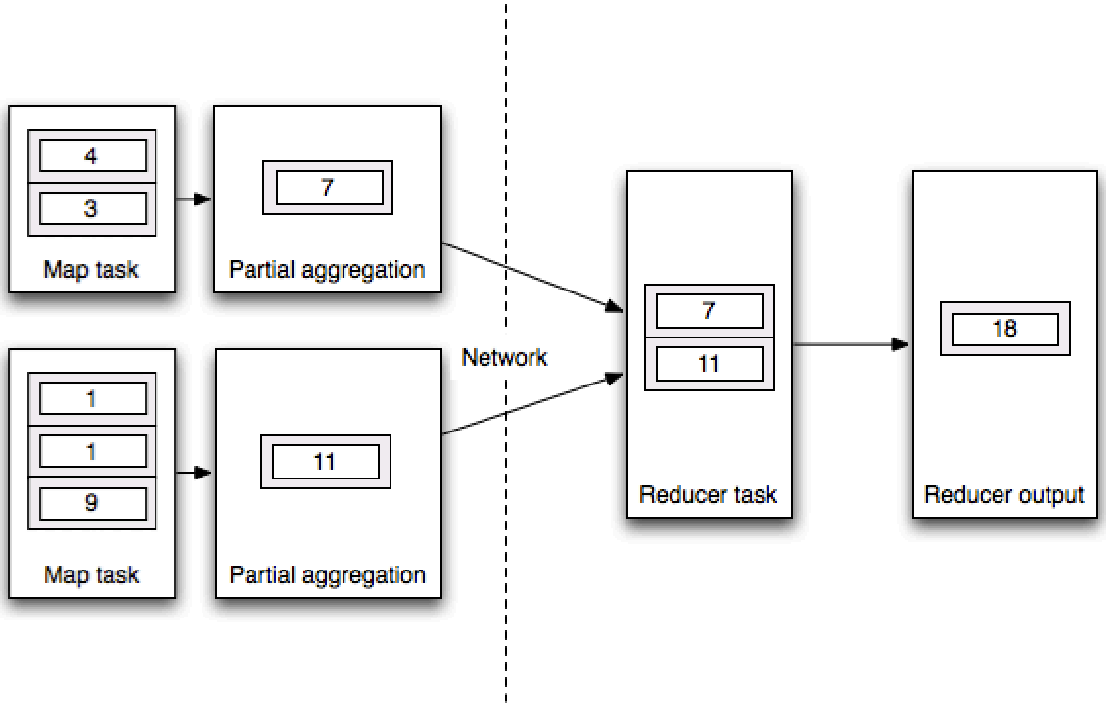


Figure 5.17 Execution of parallel aggregation at MapReduce level

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.