

Big Data

Principles and best practices of
scalable realtime data systems

Nathan Marz
James Warren



MANNING



**MEAP Edition
Manning Early Access Program
Big Data
Version 18**

Copyright 2014 Manning Publications

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

brief contents

- 1. A new paradigm for big data*
- 2. Data model for big data*
- 3. Data storage on the batch layer*
- 4. Batch layer: scalability*
- 5. Batch layer: abstraction and composition*
- 6. Batch layer: tying it all together*
- 7. Serving layer*
- 8. Speed layer: Realtime views*
- 9. Speed layer: queuing and stream processing*
- 10. Speed layer: micro-batch stream processing*
- 11. Lambda Architecture in-depth*

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
<code>id</code>	<code>integer</code>
<code>user_id</code>	<code>integer</code>
<code>url</code>	<code>varchar(255)</code>
<code>pageviews</code>	<code>bigint</code>

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.

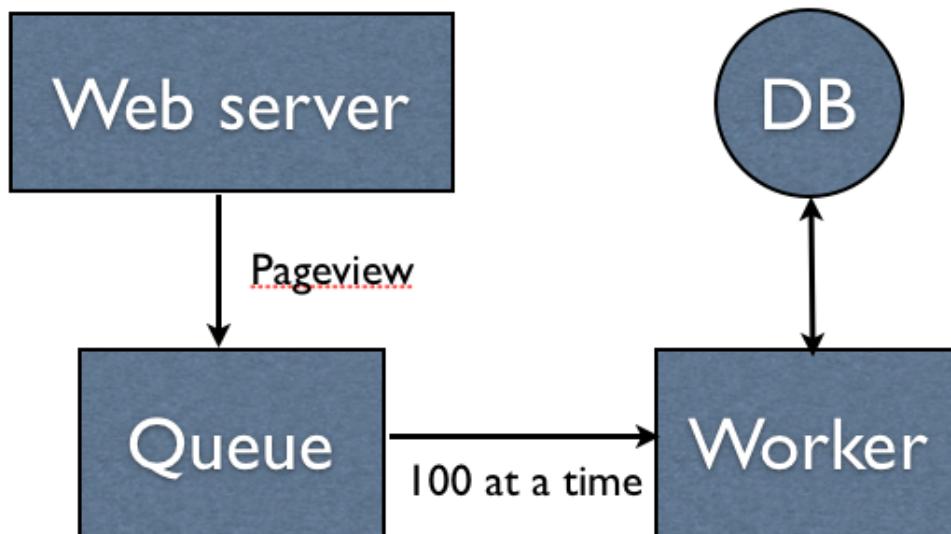


Figure 1.2 Batching updates with queue and worker

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 reshuffle 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 reshuffling, as that would be too slow. You have to do all the reshuffling 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 propagate 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 satisfies 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.

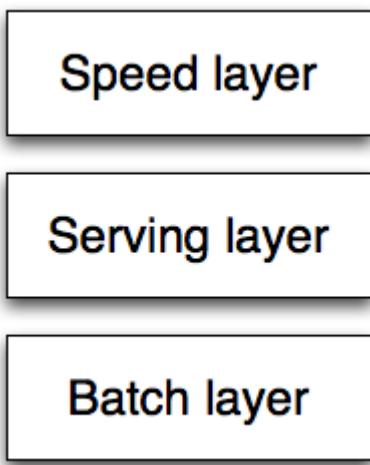


Figure 1.4 Lambda Architecture

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:

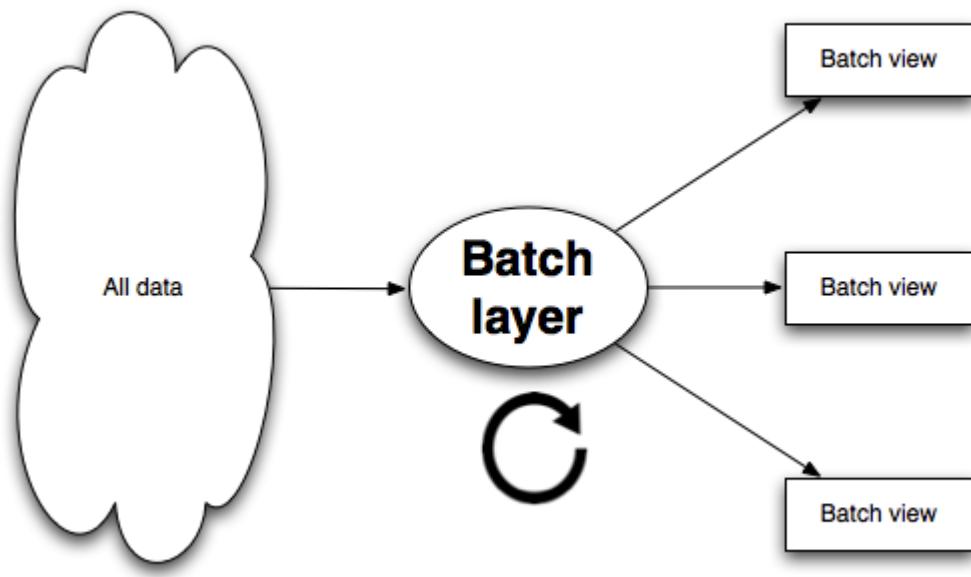


Figure 1.5 Batch layer

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 retrieve 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 been 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

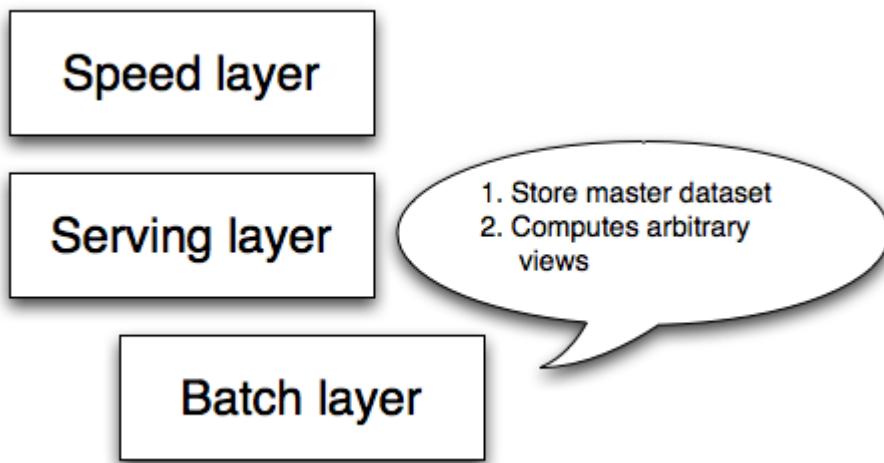


Figure 1.6 Batch layer

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 as 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.

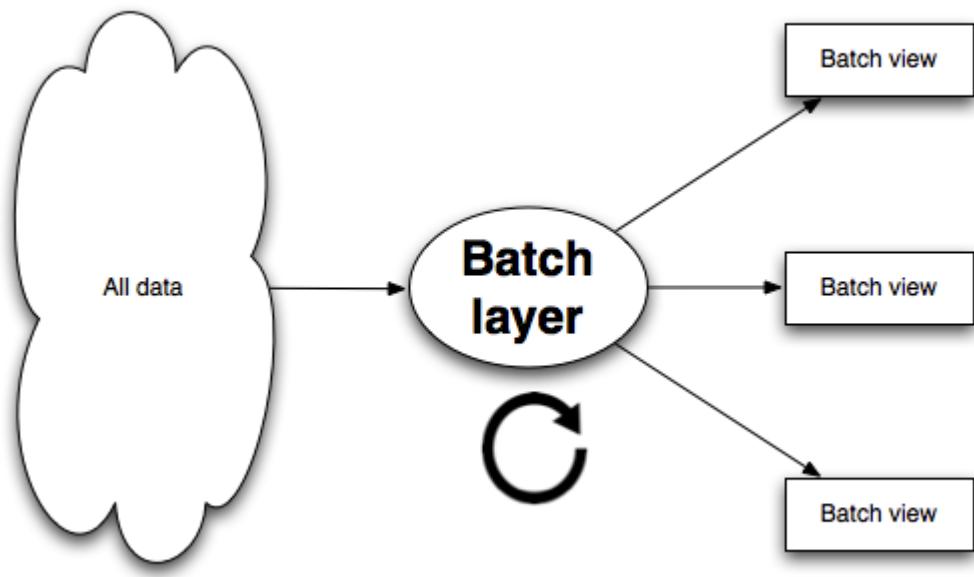


Figure 1.7 Batch layer

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

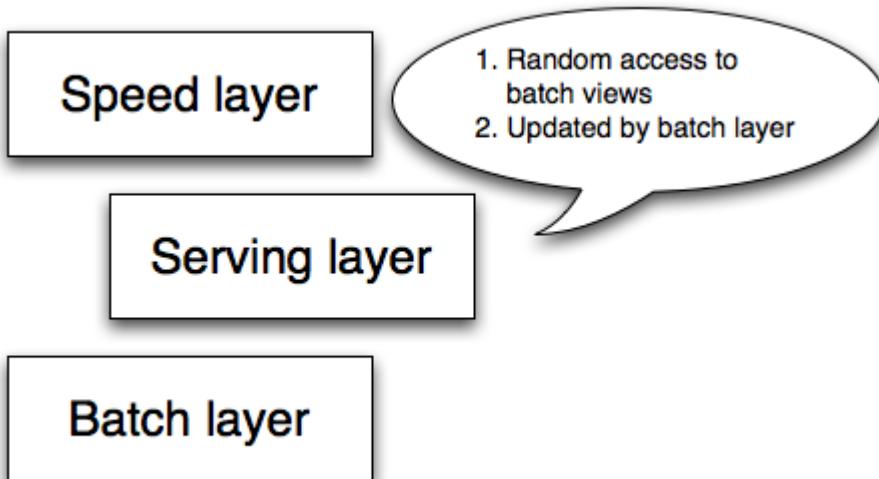


Figure 1.8 Serving layer

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 propagate through the batch layer into the serving layer where it can be queried. The important thing to notice is that other than low latency updates, the batch and serving layers satisfy every property desired in a Big Data system as outlined in Section 1.3. Let's go through them one by one:

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

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

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

1.6.4 Speed layer

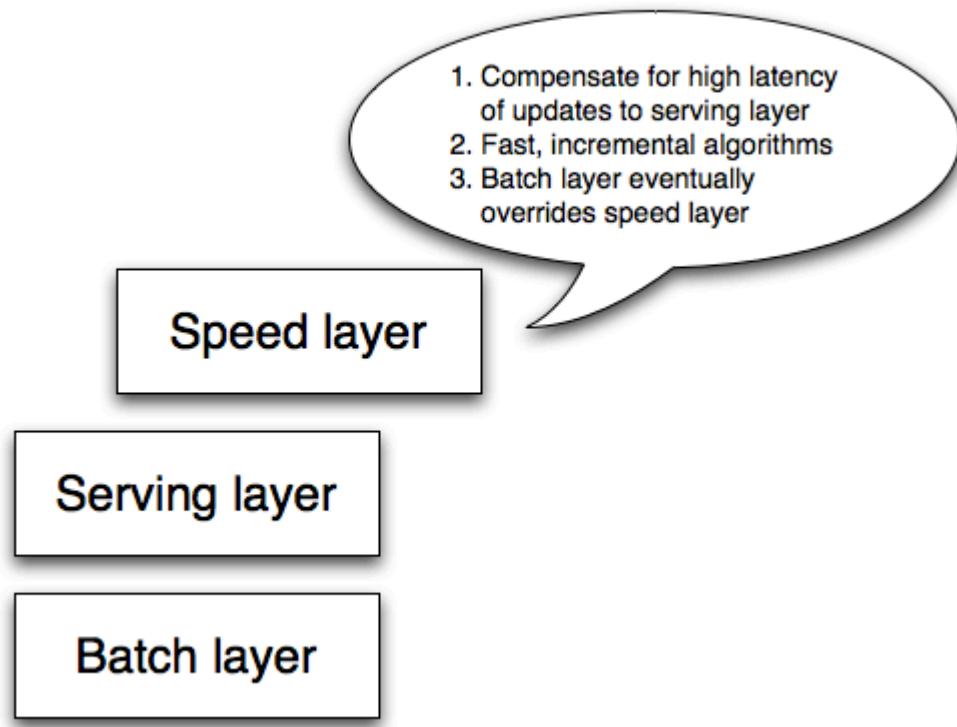


Figure 1.9 Speed layer

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

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

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

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

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

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

The last piece of the Lambda Architecture is merging the results 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.

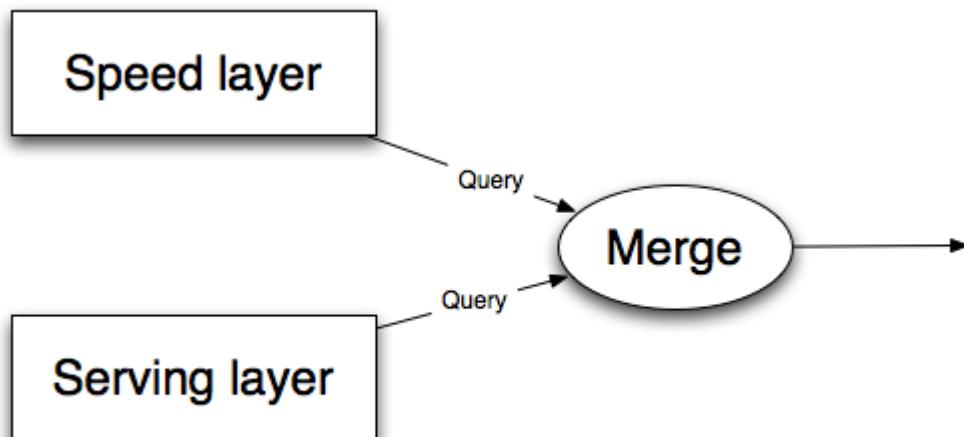


Figure 1.10 Satisfying application queries

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

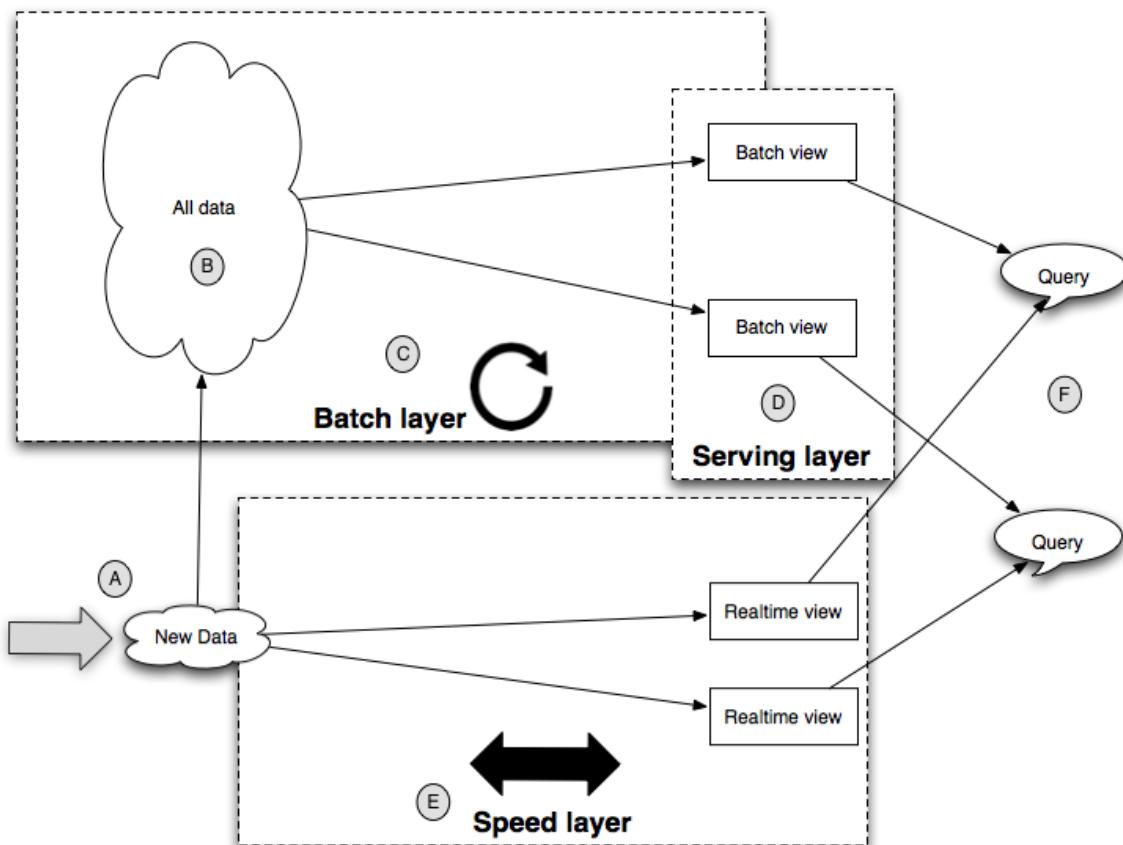


Figure 1.11 Lambda Architecture diagram

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.

Data model for Big Data

This chapter covers:

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

In the last chapter you saw what can go wrong when using traditional tools for building data systems and went back to first principles to derive a better design. You saw that every data system can be formulated as computing functions on data, and you learned the basics of the Lambda Architecture which provides a practical way to implement an arbitrary function on arbitrary data in real time.

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

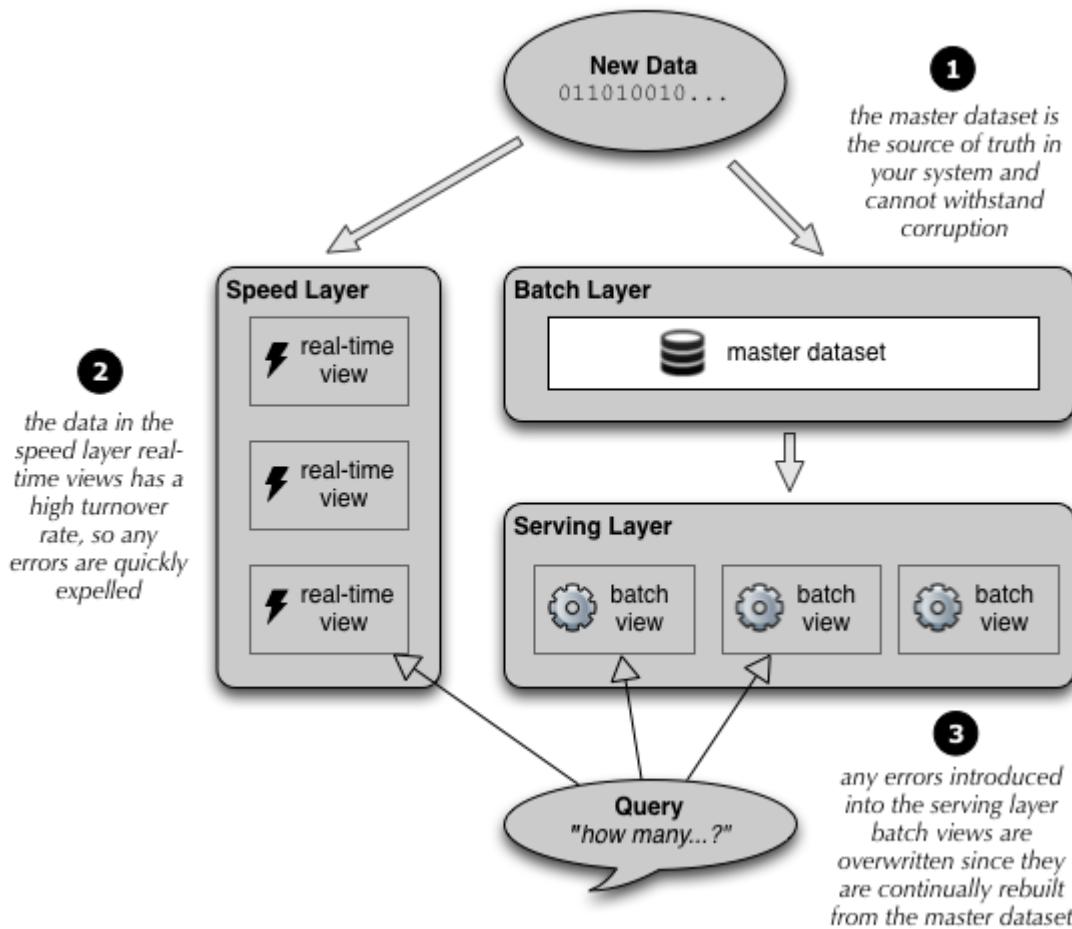


Figure 2.1 The master dataset in the Lambda Architecture serves as the source of truth of your Big Data system. Errors at the serving and speed layers can be corrected, but corruption at the master dataset is irreparable.

The master dataset is the only part of the Lambda Architecture that absolutely must be safeguarded from corruption. Overloaded machines, failing disks, and power outages all could cause errors, and human error with dynamic data systems is an intrinsic risk and inevitable eventuality. You must carefully engineer the master dataset to prevent corruption in all these cases, as fault tolerance is essential to the health of a long running data system.

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

To provide a roadmap for your undertaking, you will

- learn the key properties of data
- see how these properties are maintained in the fact-based model
- examine the advantages of the fact-based model for the master dataset

- express a fact-based model using graph schemas
- implement a graph schema using Apache Thrift

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

2.1 The properties of data

Keeping with the applied focus of the book, we will center our discussion around an example application. Suppose you are designing the next big social network - FaceSpace. When a new user - let's call him Tom - joins your site, he starts to invite his friends and family. So what information should you store regarding Tom's connections? You have a number of choices, ranging from potentially storing

- the sequence of Tom's friend and unfriend events
- Tom's current list of friends
- Tom's current number of friends

Figure 2.2 exhibits these options and their relationships.

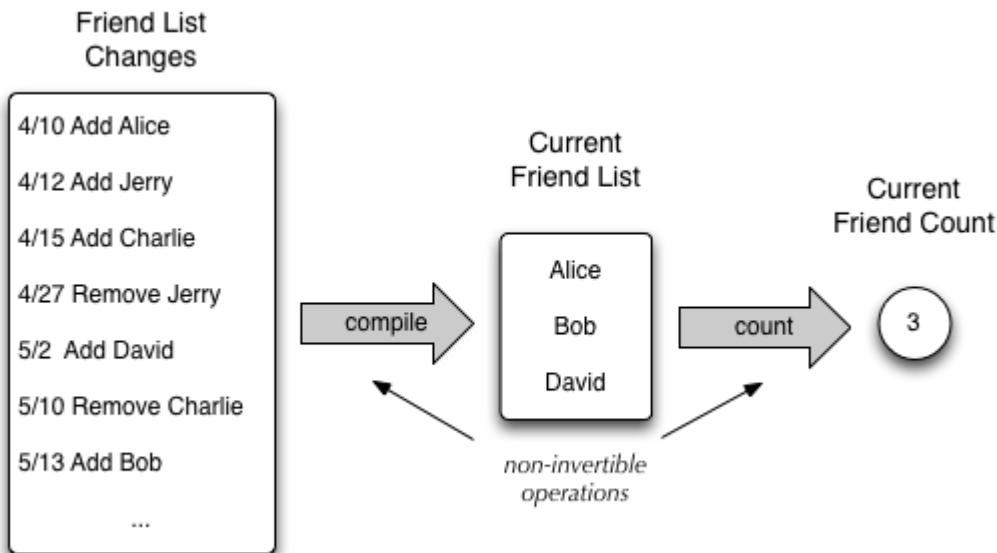


Figure 2.2 Three possible options for storing friendship information for FaceSpace. Each option can be derived from the one to its left, but it's a one way process.

This example illustrates information dependency. Note that each layer of information can be derived from the previous one, but it's a one way process. From the sequence of friend and unfriend events, we can determine the other quantities. However, if you only have the number of friends, it's impossible to determine

exactly who they are. Similarly, from the list of current friends, it's impossible to determine if Tom was previously a friend with Jerry, or whether Tom's network has been growing as of late.

The notion of dependency shapes the definitions of the terms we will use:

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

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

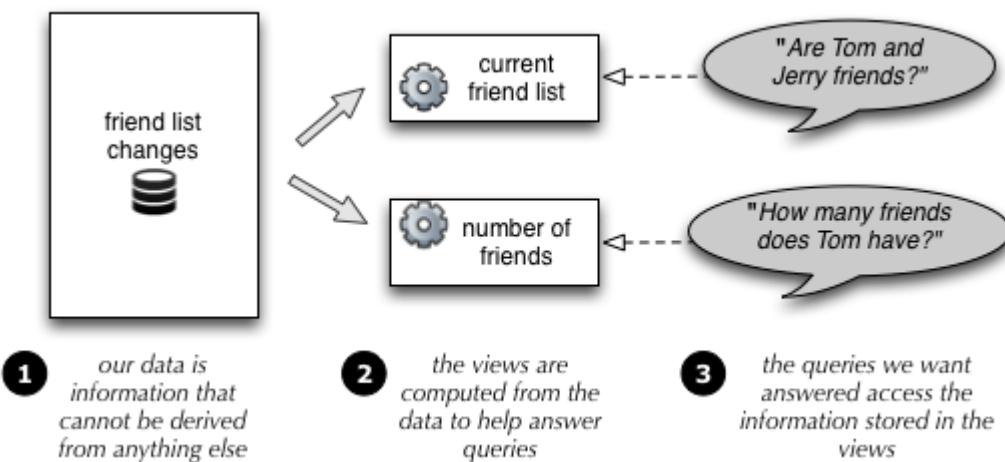


Figure 2.3 The relationships between data, views and queries.

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

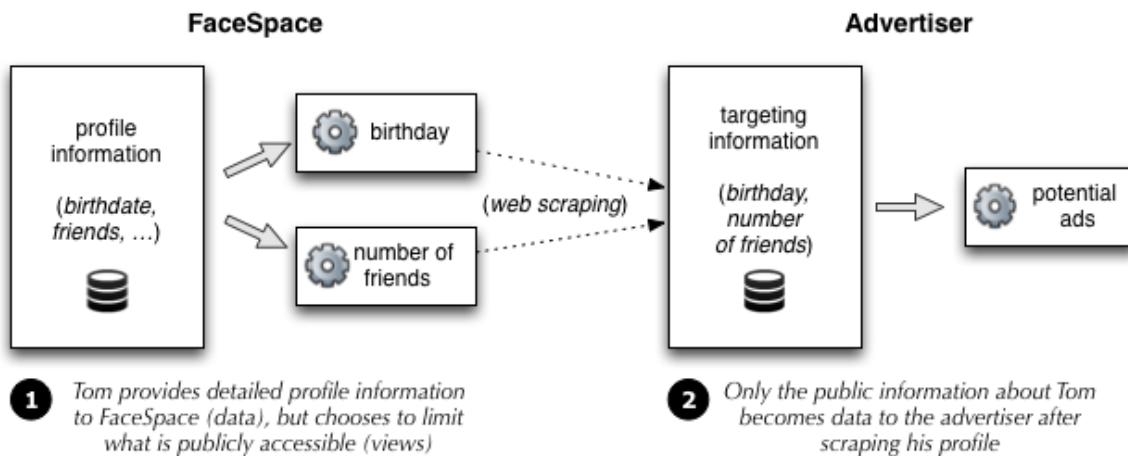


Figure 2.4 Classifying information as data or a view depends upon your perspective. As the owner of FaceSpace, Tom's birthday is a view since it is derived from the user's birthdate. However, this information is considered data to a third party advertiser.

Having established a shared vocabulary, we can introduce the key properties of data: *rawness*, *immutability*, and *perpetuity* - or the "eternal trueness of data". Foundational to your understanding of big data systems is your understanding of these three key concepts. If you're coming from a relational background, this could be confusing - typically you constantly update and summarize your information to reflect the current state of the world; you are not concerned with immutability or perpetuity. However, that approach limits the questions you can answer with your data, as well as fails to be robust to errors and corruption. It doesn't have to be so in the world of Big Data by enforcing these properties.

We will delve further into this topic as we discuss rawness of data.

2.1.1 Data is raw

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

While the FaceSpace example helps illustrate the value of rawness, we offer another to help drive the point home. Stock market trading is a fountain of information, with millions of shares and billions of dollars changing hands on a daily basis. With so many trades taking place, stock prices are historically recorded daily as an opening price, high price, low price and closing price. But those bits of

data often don't provide the big picture and can potentially skew your perception of what happened. For instance, look at Figure 2.5. It records the price data for Google, Apple and Amazon stock on a day when Google announced new products targeted at their competitors.

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

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

Figure 2.5 A summary of one day of trading for Google, Apple and Amazon stock: previous close, opening, high, low, close and net change.

If you have access to data stored at a finer time granularity, you can get a clearer picture of the events on that day and probe further into potential cause and effect relationships. Figure 2.6 depicts the minute-by-minute relative changes in the stock prices of all three companies, which suggests that both Amazon were indeed affected by the announcement, Amazon more so than Apple.



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

Also note that the additional data can suggest new ideas you may not have considered when examining the original daily stock price summary. For instance,

the more granular data makes us wonder if Amazon was more greatly affected because the new Google products compete with Amazon in both the table and cloud-computing markets.

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

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

UNSTRUCTURED DATA IS RAWER THAN NORMALIZED DATA

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



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

If you come across a form of data such as an unstructured location string, should you store the unstructured string or the semantically normalized form? We argue that it's better to store the unstructured string because your semantic normalization algorithm may improve over time. If you store the unstructured

string, you can renormalize that data at a later time when you have improved your algorithms. In the example above, you may later adapt the algorithm to recognize "North Beach" as a neighborhood in San Francisco, or you may want to use the neighborhood information for another purpose.

TIP**Store Unstructured Data When...**

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

MORE INFORMATION DOESN'T NECESSARILY MEAN RAWER DATA

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

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

At the other end of the spectrum, we could also store the full HTML of Tom's blog as your data. While it is considerably more information in terms of total bytes, the color scheme, stylesheets and JavaScript code of the site cannot be used to derive any additional information about Tom. They serve only as the container for the contents of the site and should not be part of your raw data.

2.1.2 Data is immutable

Immutable data may seem like a strange concept if you're well versed with relational databases. After all, in the relational database world - and most other databases as well - *update* is one of the fundamental operations. However, for immutability, you don't update or delete data, you only add more.¹ By using an immutable schema for Big Data systems, you gain two vital advantages:

Footnote 1 There are a few scenarios in which you can delete data, but these are special cases and not part of the day to day workflow of your system. We will discuss these scenarios in Section 2.1.4.

1. *Human fault tolerance.* This is the most important advantage of the immutable model. As we discussed in Chapter 1, human fault tolerance is an essential property of data systems. People will make mistakes, and you must limit the impact of such mistakes and have

mechanisms for recovering from them. With a mutable data model, a mistake can cause data to be lost because values are actually overridden in the database. With an immutable data model, **no data can be lost**. If bad data is written, earlier (good) data units still exist. Fixing the data system is just a matter of deleting the bad data units and recomputing the views built off the master dataset.

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

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

User Information					
id	name	age	gender	employer	location
1	Alice	25	female	Apple	Atlanta, GA
2	Bob	36	male	SAS	Chicago, IL
3	Tom	28	male	Google	San Francisco, CA
4	Charlie	25	male	Microsoft	Washington, DC
...

should Tom move
to a different city,
this value would
be overwritten

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

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

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

Name Data

user id	name	timestamp
1	Alice	2012/03/29 08:12:24
2	Bob	2012/04/12 14:47:51
3	Tom	2012/04/04 18:31:24
4	Charlie	2012/04/09 11:52:30
...

Age Data

user id	age	timestamp
1	25	2012/03/29 08:12:24
2	36	2012/04/12 14:47:51
3	28	2012/04/04 18:31:24
4	25	2012/04/09 11:52:30
...

Location Data

user id	location	timestamp
1	Atlanta, GA	2012/03/29 08:12:24
2	Chicago, IL	2012/04/12 14:47:51
3	San Francisco, CA	2012/04/04 18:31:24
4	Washington, DC	2012/04/09 11:52:30
...

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

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

Location Data

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

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

You now have two location records for Tom (user id #3), and since the data units are tied to particular times, they can both be true. Tom's *current location* is a

simple query on the data: look at all the locations and pick the one with the most recent timestamp. By keeping each field in a separate table, you only record the information that changed. This requires less space for storage as well as guarantees each record is new information and not simply carried over from the last record.

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

2.1.3 Data is eternally true

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

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

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

1. *Garbage collection*: When you perform garbage collection, you delete all data units that have "low value". You can use garbage collection to implement data retention policies that control the growth of the master dataset. For example, you may decide you to implement a policy that keeps only one location per person per year instead of the full history of each time a user changes locations.
2. *Regulations*: Government regulations may require you to purge data from your databases in certain conditions.

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

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

2.2 The fact-based model for representing data

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

In the discussion of immutability you saw a glimpse of the fact-based model, in that the master dataset continually grows with the addition of immutable, timestamped data. We'll now expand on what we already discussed to explain the fact-based model in full. We'll first introduce the fact-based model in the context of our FaceSpace example and discuss its basic properties. We'll then continue with discussing how and why you should make your facts identifiable. To wrap up, we'll explain the benefits of using the fact-based model and why it's an excellent choice for your master dataset.

2.2.1 An example of the fact-based model

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

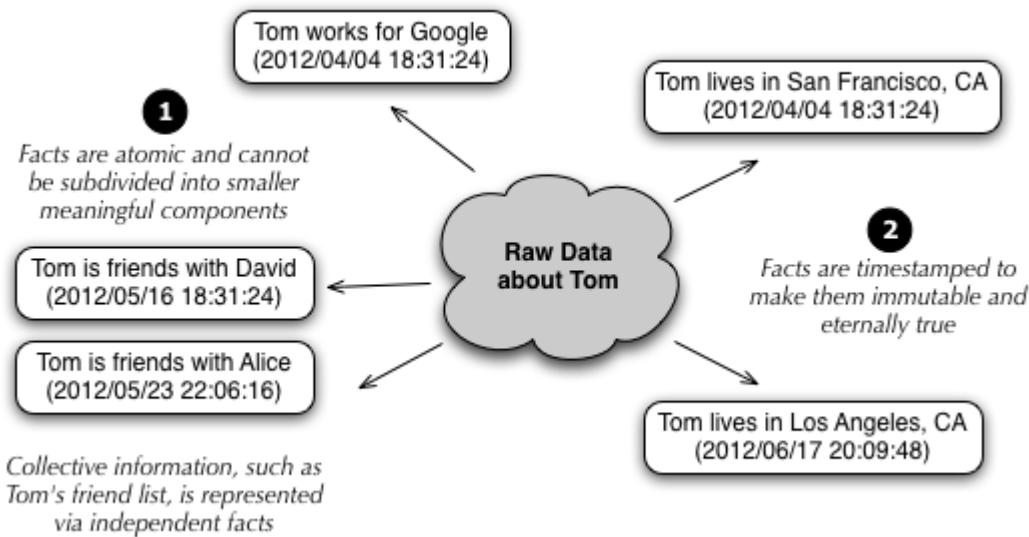


Figure 2.11 All of the raw data concerning Tom are deconstructed into time-stamped, atomic units we call facts.

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

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

Second, facts are atomic since they cannot be subdivided further into meaningful components. Collective data, such as Tom's friend list in the figure, are represented as multiple, independent facts. As a consequence of being atomic, there is no redundancy of information across distinct facts.

These properties make the fact-based model a simple and expressive model for your dataset, yet there is an additional property we recommend imposing on your facts: *identifiability*. We next discuss in depth how and why you make facts identifiable.

2.2.2 Making facts identifiable

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

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

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

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

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

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

When a pageview fact is created, a random 64 bit number is chosen as a nonce to distinguish this pageview from other pageviews that occur for the same URL at the same time and from the same ip address. The addition of the nonce makes it possible to distinguish pageview events from each other, and if two pageview data units are identical (all fields including the nonce), you know they refer to the exact same event.

Making facts identifiable means that you can write the same fact to the master dataset multiple times without changing the semantics of the master dataset. Your queries can filter out the duplicate facts when doing their computations. As it turns out, and as you will see later, having distinguishable facts makes implementing the rest of the Lambda Architecture much easier.

SIDE BAR Duplicates aren't as rare as you might think

At a first look, it may not be obvious why we care so much about identity and duplicates. After all, to avoid duplicates, the first inclination would be to ensure an event is recorded just once. Unfortunately life isn't always so simple when dealing with Big Data.

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

One common "fault" these systems must anticipate is a network partition where the destination datastore becomes available. For these situations, fault-tolerant systems commonly handle failed operations by retrying until success. Since the sender would not know which data was last received, a standard approach would be to resend all data yet to be acknowledged by the recipient. However, if part of the original attempt did make it to the metastore, you'll end up with duplicates in your dataset.

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

To quickly recap, the fact-based model

- stores your raw data as atomic facts,
- keeps the facts immutable and eternally true by using timestamps, and
- ensures each fact is identifiable so that query processing can identify duplicates.

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

2.2.3 Benefits of the fact-based model

With a fact-based model, the master dataset will be an ever-growing list of immutable, atomic facts. This isn't a pattern that relational databases were built to support - if you come from a relational background, your head may be spinning. The good news is that by changing your data model paradigm, you gain numerous advantages.

THE DATASET IS QUERYABLE AT ANY TIME IN ITS HISTORY

Instead of storing only the current state of the world as you would using a mutable, relational schema, you have the ability to query your data for any time covered by your dataset. This is a direct consequence of facts being timestamped and immutable. "Updates" and "deletes" are performed by adding new facts with more recent timestamps, but since no data is actually removed, you can reconstruct the state of the world at the specified time of your query.

THE DATA IS HUMAN FAULT-TOLERANT

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

Location Data		
user id	location	timestamp
1	Atlanta, GA	2012/03/29 08:12:24
2	Chicago, IL	2012/04/12 14:47:51
3	San Francisco, CA	2012/04/04 18:31:24
4	Washington, DC	2012/04/09 11:52:30
5	Los Angeles, CA	2012/06/17 20:00:48
...

Human faults can easily be corrected by simply deleting erroneous facts. The record is automatically reset by using earlier timestamps.

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

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

THE DATASET EASILY HANDLES PARTIAL INFORMATION

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

THE DATA STORAGE AND QUERY PROCESSING LAYERS ARE SEPARATE

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

TIP

Normalization is an overloaded term

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

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

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

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

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

Figure 2.13 A simple denormalized schema for storing employment information.

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

In comparison, consider the normalized schema in Figure 2.14.

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

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

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

Figure 2.14 Two normalized tables for storing the same employment information.

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

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

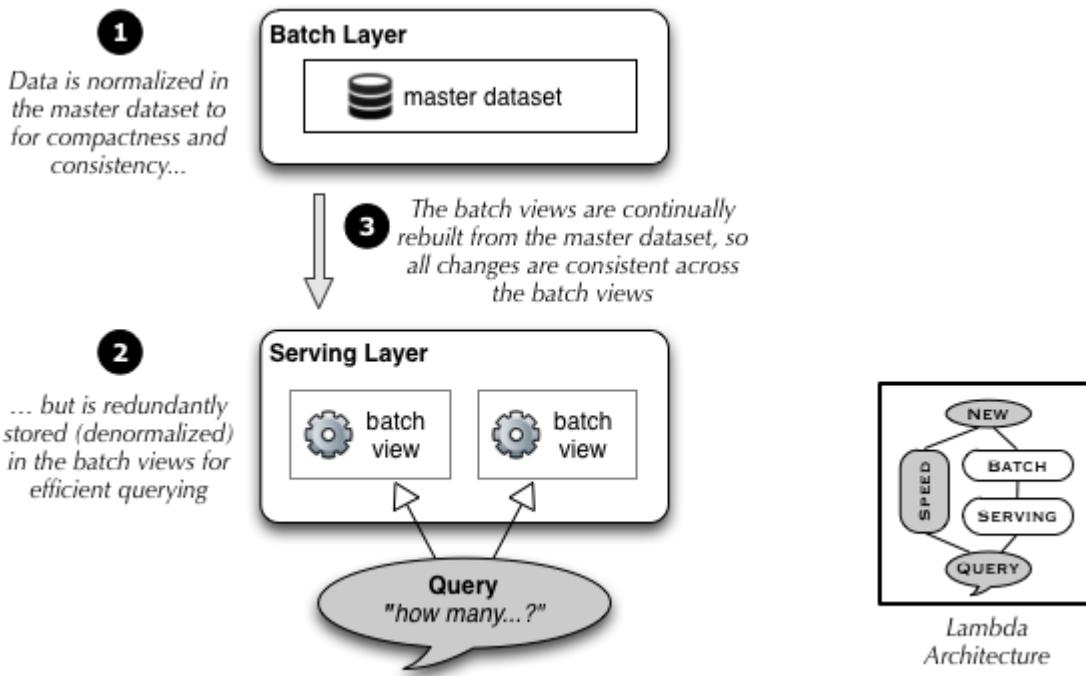


Figure 2.15 The Lambda Architecture has the benefits of both normalization and denormalization by separating objectives at different layers.

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

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

In summary, all of these benefits make the fact-based model an excellent choice

for your master dataset. But that's enough discussion at the theoretical level - let's dive into the details of practically implementing a fact-based data model.

2.3 Graph schemas and serialization frameworks

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

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

2.3.1 Elements of a graph schema

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

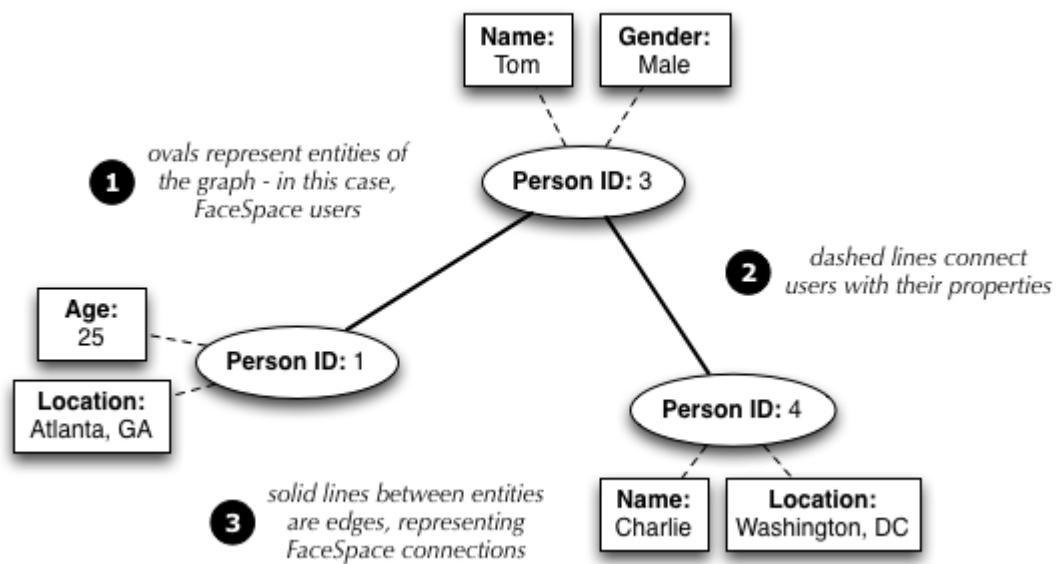


Figure 2.16 Visualizing the relationship between FaceSpace facts

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

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

represented by a user id. As another example, if FaceSpace allows users to identify themselves as part of a group, then the groups would also be represented by nodes.

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

WARNING
Edges are strictly between nodes

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

The graph schema is the full listing of all types of nodes, edges and properties, and it provides a complete description of the data contained within a dataset. We next discuss the need to ensure that all facts within a dataset rigidly adhere to the schema.

2.3.2 The need for an enforceable schema

At this point, information is stored as facts, and a graph schema describes the type of information contained in the dataset. You're all set, right? Well, not quite. You still need to decide in what format you will store your facts. A first idea might be to use a semi-structured text format like JSON. This would provide simplicity and flexibility, allowing essentially anything to be written to the master dataset. However, in this case it's too flexible for our needs.

To illustrate this problem, suppose we chose to represent Tom's age using JSON.

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

There are no issues with the representation of this single fact, but there is no way to ensure that all subsequent facts will follow the same format. From human error, the dataset could also possibly include facts like

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

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

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

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

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

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

Enforceable schemas are implemented using a *serialization framework*. A serialization framework provides a language-neutral way to define the nodes, edges and properties of your schema. It then generates code (potentially in many different languages) that serializes and deserializes the objects in your schema so they can be stored and retrieved from your master dataset.

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

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

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

2.4 A complete data model for SuperWebAnalytics.com

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

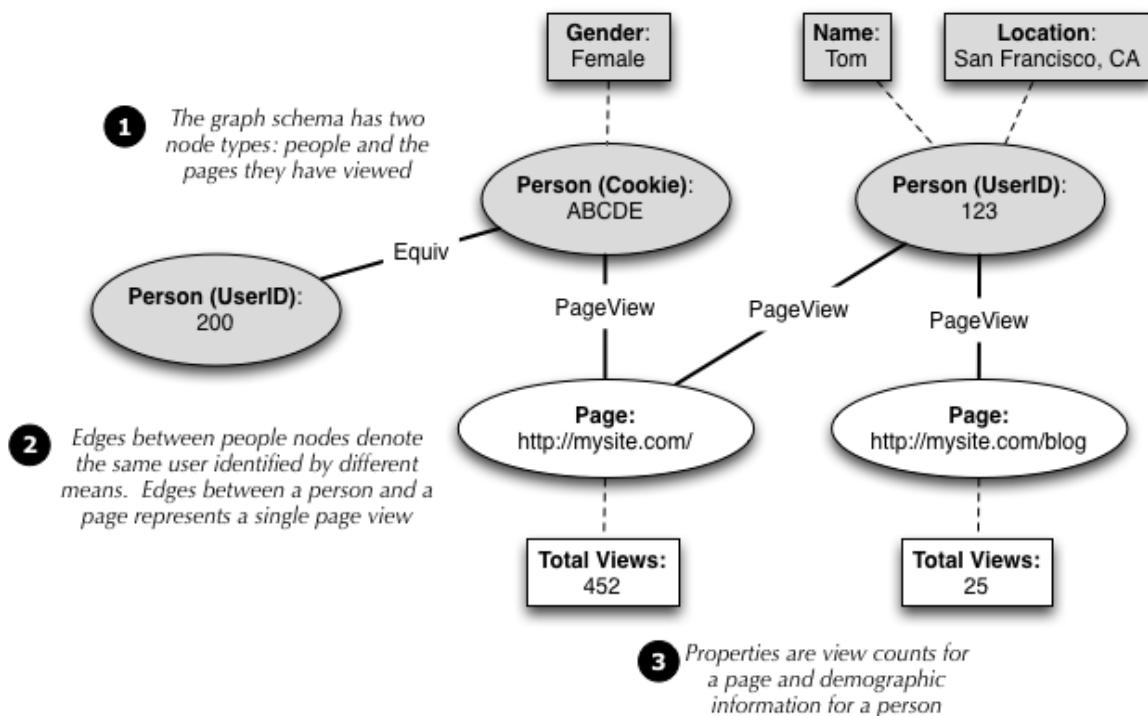


Figure 2.17 The graph schema for SuperWebAnalytics.com. There are two node types: people and edges. People nodes and their properties are slightly shaded to distinguish the two.

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

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

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

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

2.4.1 Using Thrift as a serialization framework.

Apache Thrift² is a tool that can be used to define statically typed, enforceable schemas. It provides an interface definition language to describe the schema in terms of generic data types, and this description can be later used to automatically generate the actual implementation in multiple programming languages.

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

SIDE BAR Other serialization frameworks

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

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

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

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

NODES

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

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

```

}

union PageID {
    1: string url;
}

```

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

EDGES

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

```

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

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

```

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

PROPERTIES

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

```

union PagePropertyValue {
    1: i32 page_views;
}

```

```

}

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

```

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

```

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

enum GenderType {
    MALE = 1,
    FEMALE = 2
}

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

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

```

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

2.4.2 Tying everything together into data objects

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

Listing 2.1 Completing the SuperWebAnalytics.com schema

```

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

struct Pedigree {
    1: required i32 true_as_of_secs;
}

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

```

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

2.4.3 Evolving your schema

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

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

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

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

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

Listing 2.2 Extending the SuperWebAnalytics.com schema

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

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

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

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

2.5 Summary

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

The fact-based model provides a simple yet expressive representation of your data by naturally keeping a full history of each entity over time. Its append-only nature makes it easy to implement in a distributed system, and it can easily evolve as your data and your needs change. You're not just implementing a relational

system in a more scalable way - you're adding whole new capabilities to your system as well.

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



Data storage on the batch layer

This chapter covers:

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

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

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

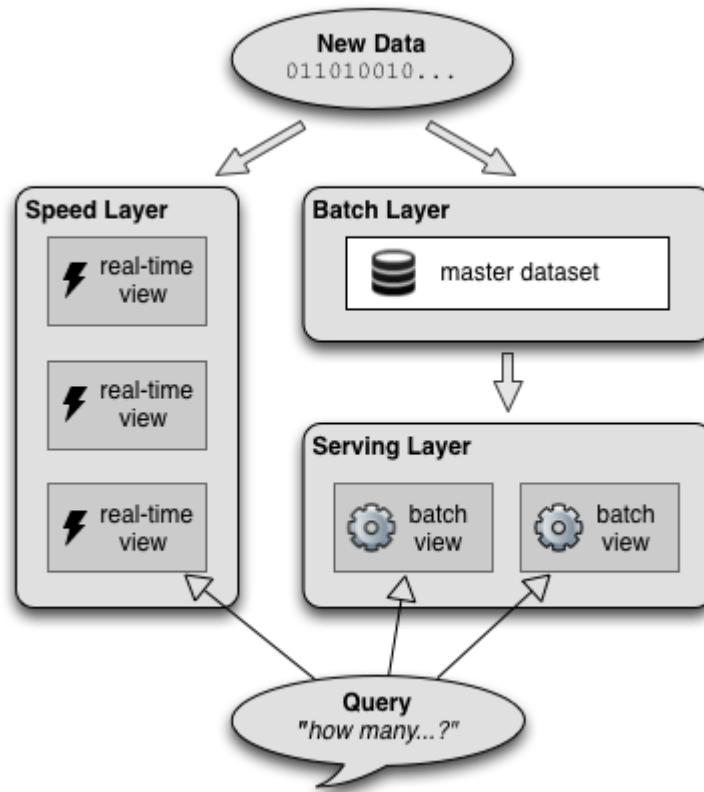


Figure 3.1 The batch layer must structure large, continually growing datasets in a manner that supports low maintenance as well as efficient creation of the batch views.

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

In this chapter, you will

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

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

3.1 Storage requirements for the master dataset

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

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

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

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

Table 3.1 A checklist of storage requirements for the master dataset

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

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

3.2 Implementing a storage solution for the batch layer

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

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

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

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

3.2.1 Introducing the Hadoop Distributed File System

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

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

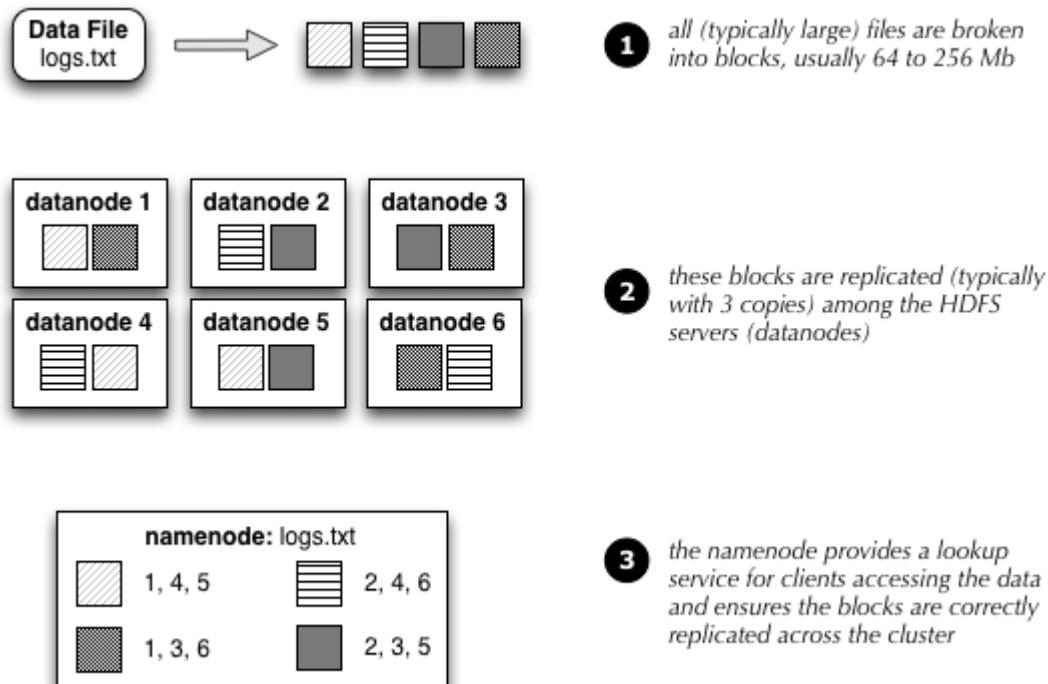


Figure 3.2 Files are chunked into blocks which are dispersed to datanodes in the cluster.

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

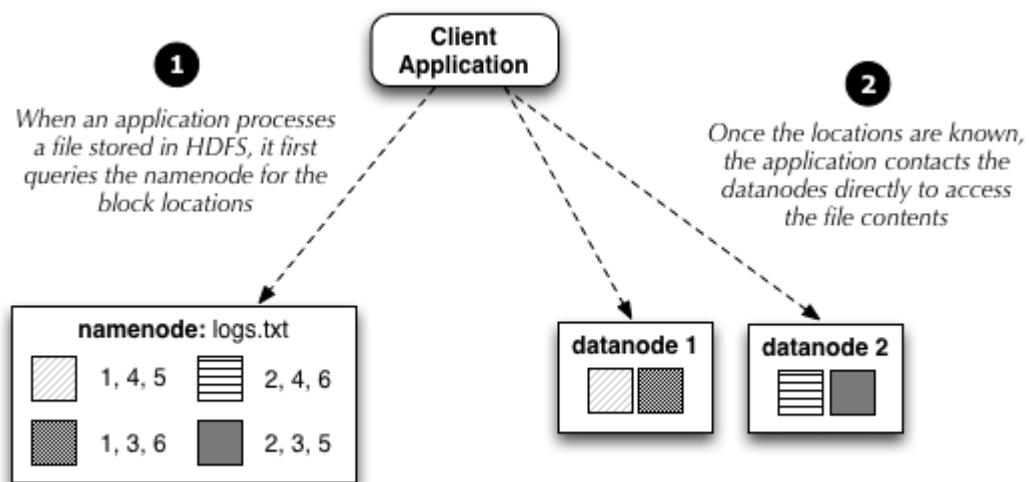


Figure 3.3 Files are chunked into blocks which are dispersed to datanodes in the cluster.

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

remains available even when individual nodes are offline.

SIDE BAR **Getting started with Hadoop**

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

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

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

3.2.2 Storing a master dataset with HDFS

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

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

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

```

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

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

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

You can list the directory contents:

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

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

And verify the contents of the file:

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

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

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

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

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

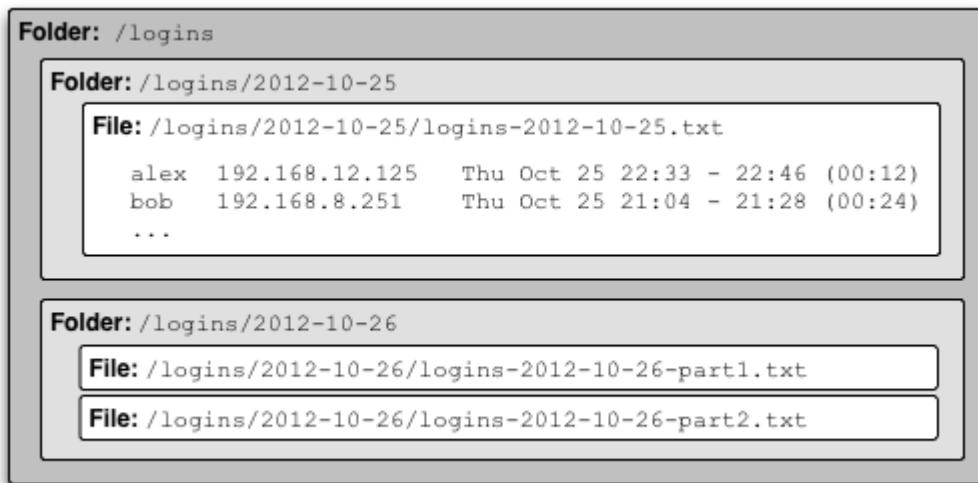


Figure 3.4 A vertical partitioning scheme for login data. By separating information for each date in a separate folder, a function can select only the folders containing data relevant to its computation.

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

3.2.3 HDFS: Meeting the storage requirement checklist

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

Table 3.2 How HDFS meets the storage requirement checklist

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

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

3.3 Maintaining the batch storage layer in HDFS

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

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

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

3.3.1 Appending to the master dataset

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

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

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

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

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

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

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

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

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

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

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

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

3.3.2 Consolidating the master dataset to eliminate small files

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

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

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

3.3.3 The need for a high level abstraction

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

Listing 3.2 Abstractions of HDFS maintenance tasks

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

public class PailMove {

    public static void mergeData(String masterDir, String updateDir)
        throws IOException
    {
        Pail target = new Pail(masterDir); ①
        Pail source = new Pail(updateDir);
        target.absorb(source); ②
        target.consolidate(); ③
    }
}
```

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

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

A QUICK RECAP

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

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

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

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

3.4 Data storage in the Batch Layer with Pail

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

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

SIDE BAR
Why the focus on Pail?

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

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

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

3.4.1 Basic Pail operations

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

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

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

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

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

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

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

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

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

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

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

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

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

3.4.2 Serializing objects into pails

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

Listing 3.3 A no-frills class for logins

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

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

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

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

Listing 3.4 Implementing the PailStructure interface

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

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

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

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

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

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

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

- ▀ Logins are later reconstructed when read from pailfiles
- ④ the `getTarget` method defines the vertical partitioning scheme but is not used in this example
- ⑤ `isValidTarget` determines whether the given path matches the vertical partitioning scheme but is also not used in this example

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

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

- ① create a pail with the new pail structure

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

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

- ① a pail supports the `Iterable` interface for its object type

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

3.4.3 Batch operations using Pail

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

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

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

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

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

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

3.4.4 Vertical partitioning with Pail

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

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

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

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

Listing 3.5 A vertical partitioning scheme for Login records

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

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

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

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

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

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

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

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

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

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

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

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

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

3.4.5 Pail file formats and compression

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

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

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

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

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

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

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

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

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

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

3.4.6 Summarizing the benefits of Pail

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

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

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

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

3.5 Putting it all together for SuperWebAnalytics.com

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

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

```

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

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

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

```

- ① all facts in the dataset are represented as a timestamp and a base unit of data
- ② the fundamental data unit describes the edges and properties of the dataset
- ③ property value can be of multiple types

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

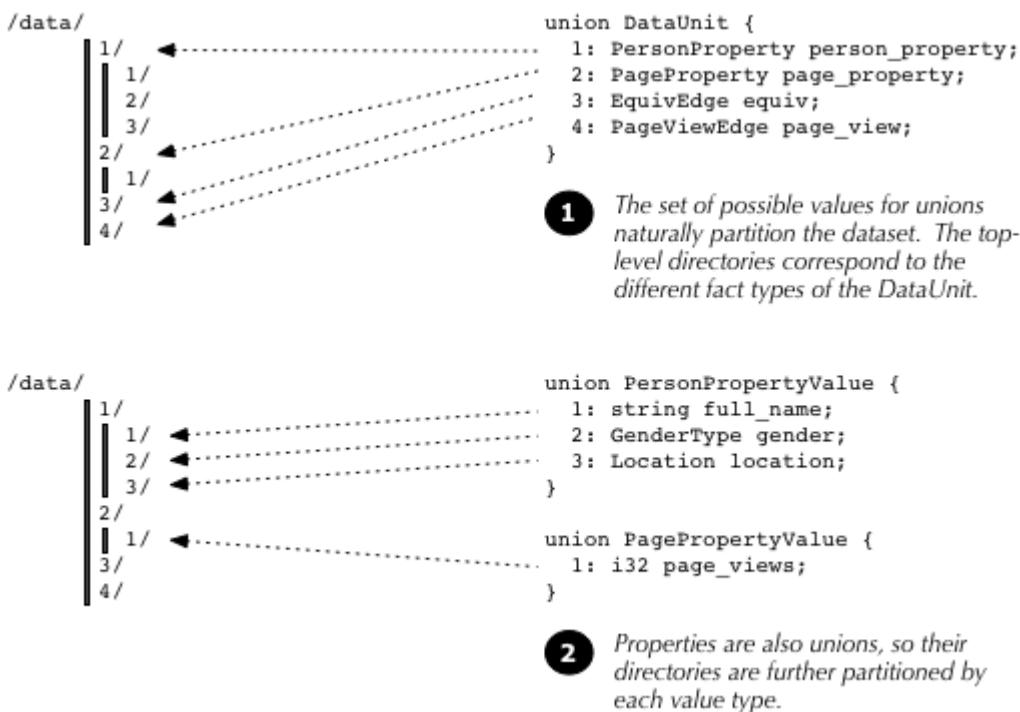


Figure 3.5 The unions within a graph schema provide a natural vertical partitioning scheme for a dataset.

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

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

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

3.5.1 A Structured Pail for Thrift Objects

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

Listing 3.6 A generic abstract pail structure for serializing Thrift objects

```

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

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

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

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

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

    protected abstract T createThriftObject(); 6
}

```

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

3.5.2 A Basic Pail for SuperWebAnalytics.com

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

Listing 3.7 A concrete implementation for Data objects

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

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

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

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

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

3.5.3 A Split Pail to Vertically Partition the DataSet.

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

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

```

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

```

```

Map<Short, FieldStructure>
{{1: PropertyStructure},
 {2: PropertyStructure},
 {3: EdgeStructure},
 {4: EdgeStructure}}}

```

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

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

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

```

public class SplitDataPailStructure extends DataPailStructure {

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

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

    // remainder of class elided
}

```

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

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

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

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

```
// methods are from SplitDataPailStructure

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

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

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

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

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

EdgeStructure class trivial.

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

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

Listing 3.9

```

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

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

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

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

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

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

```

```

    }

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

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

```

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

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

3.6 Conclusion

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

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

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

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

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

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

Batch layer: scalability



This chapter covers:

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

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

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

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

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

4.1 Motivating examples

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

4.1.1 Number of pageviews over time

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

```
function pageviewsOverTime(masterDataset, url,
                           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 using a function of the entire dataset, you simply iterate through every record and keep a counter of all the pageviews for that URL that fall within the specified range. After exhausting all the records, you then return the final value of the counter.

4.1.2 Gender inference

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

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

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

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

4.1.3 Influence score

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

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

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

- ❶ compute amount of influence between all pairs of people
- ❷ count how many people for which personId is the top influencer

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

4.2 Computing on the batch layer

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

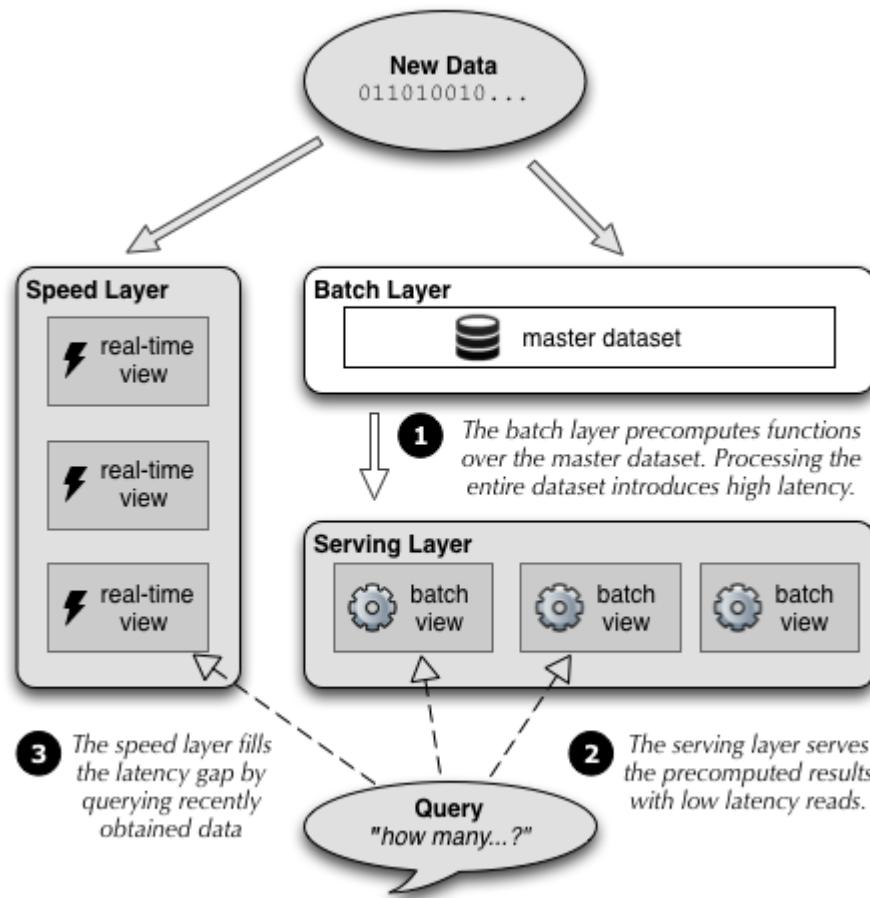


Figure 4.1 The roles of the Lambda Architecture layers in servicing queries on the dataset.

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

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

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

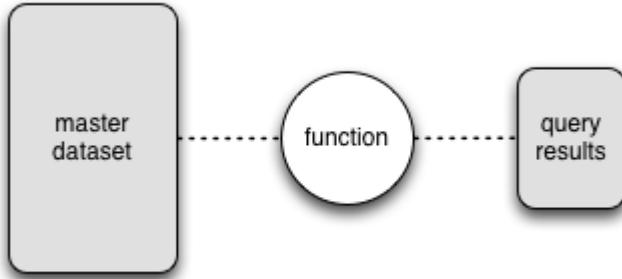


Figure 4.2 Precomputing a query by running a function on the master dataset directly

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

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

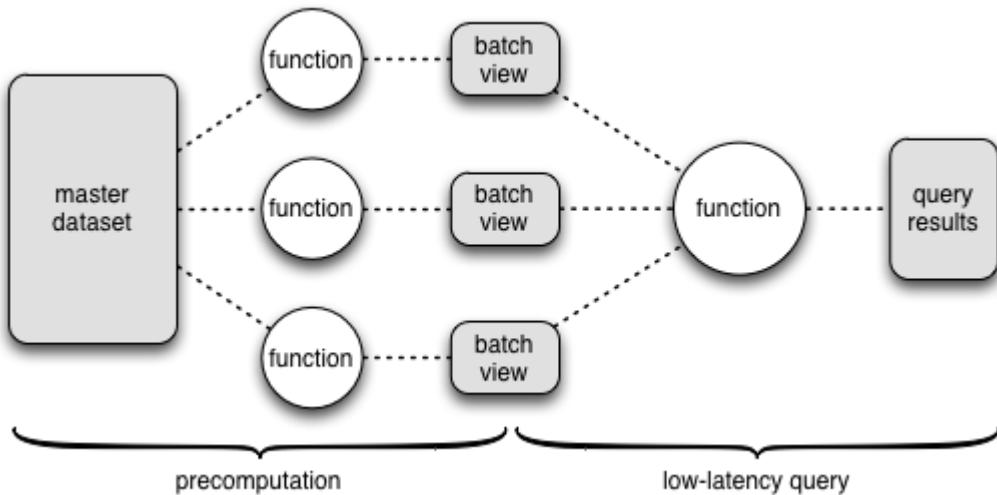


Figure 4.3 Splitting a query into precomputation and on the fly components

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

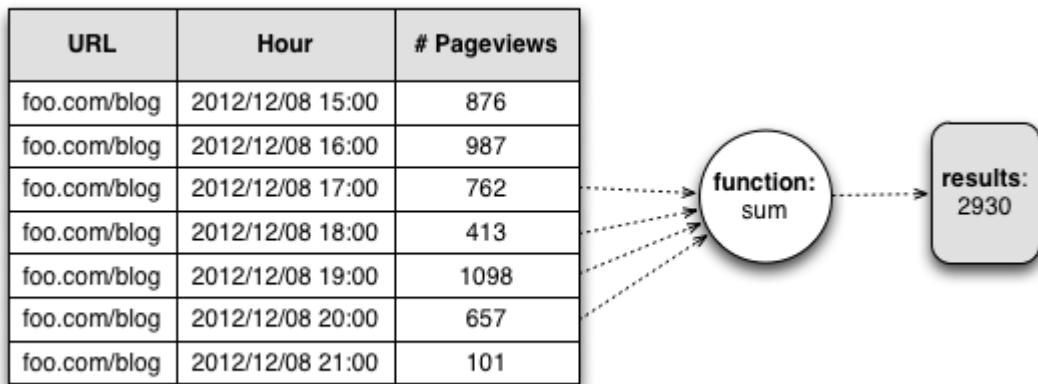


Figure 4.4 Computing the number of pageviews by querying an indexed batch view

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

4.3 Recomputation algorithms vs. incremental algorithms

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

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

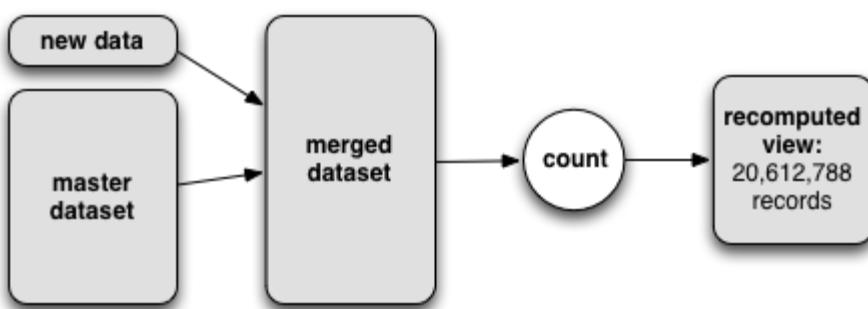


Figure 4.5 A recomputing algorithm to update the number of records in the master dataset. New data is appended to the master dataset, then all records are counted.

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

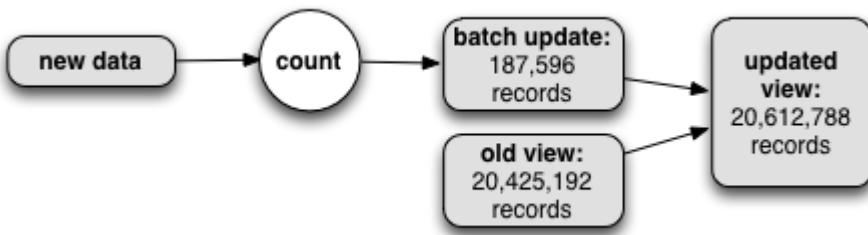


Figure 4.6 An incremental algorithm to update the number of records in the master dataset. Only the new data set is counted, with the total used to update the batch view directly.

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

4.3.1 Performance

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

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

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

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

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

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

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

Recomputation Batch View

Incremental Batch View

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

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

4.3.2 Human fault-tolerance

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

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

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

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

4.3.3 Generality of algorithm

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

Footnote 1 We will discuss HyperLogLog further in subsequent chapters.

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

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

4.3.4 Choosing a style of algorithm

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

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

Figure 4.8 Comparing recomputation and incremental algorithms.

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

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

4.4 Scalability in the batch layer

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

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

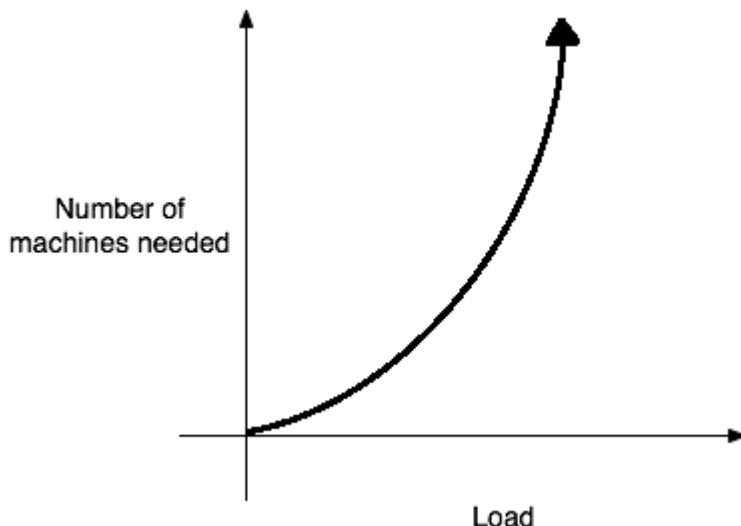


Figure 4.9 Non-linear scalability

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

SIDE BAR **What scalability doesn't mean...**

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

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

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

4.5 MapReduce: a paradigm for Big Data computing

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

The canonical MapReduce example is *word count*. Word count takes a dataset of text and determines the number of times each word appears throughout the text. The map function in MapReduce executes once per line of text and emits any

number of key-value pairs. For word count, the map function emits a key-value pair for every word in the text, setting the key to the word and the value to the number one.

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

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

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

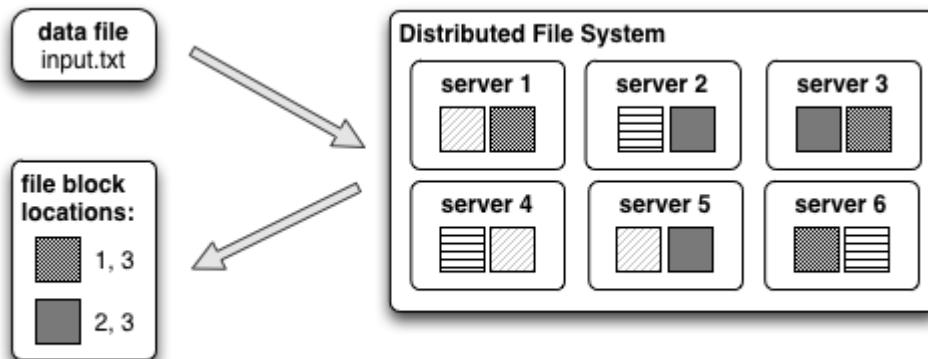
There's a lot happening under the hood to run a program like word count across a cluster of machines, but the MapReduce framework handles most of the details for you. The intent is for you to focus on *what* needs to be computed without worrying about the details for *how* it is computed.

4.5.1 Scalability

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

Let's walk through how a program like word count executes on a MapReduce

cluster. The input to your MapReduce program is stored within a distributed filesystem such as the Hadoop Distributed Filesystem (HDFS) you encountered in the last chapter. Before processing the data, the program first determines which machines in your cluster host the blocks containing the input - see Figure 4.10.



Before a MapReduce program begins processing data, it first determines the block locations within the distributed filesystem.

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

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

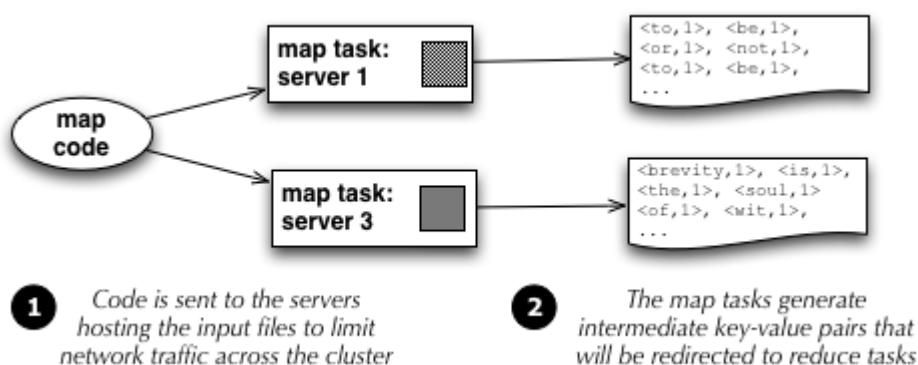
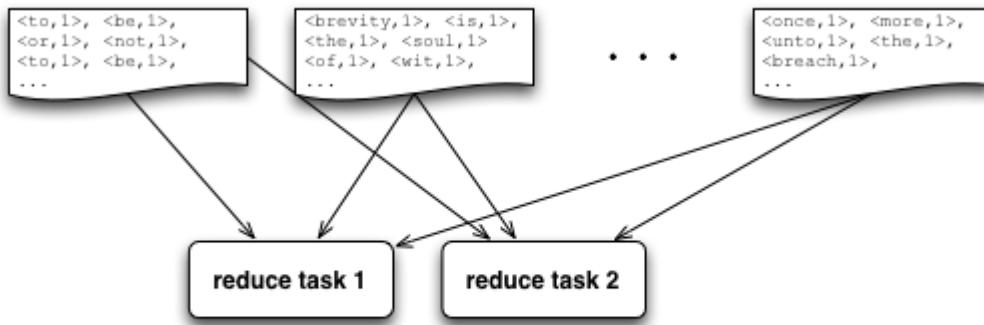


Figure 4.11 MapReduce promotes data locality, running tasks on the servers that host the input data

Like maps, there are also reduce tasks spread across the cluster. Each of these tasks is responsible for computing the reduce function for a subset of keys

generated by the map tasks. Because the reduce function requires all values associated with a given key, a reduce task cannot begin until all map tasks are complete.

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



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

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

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

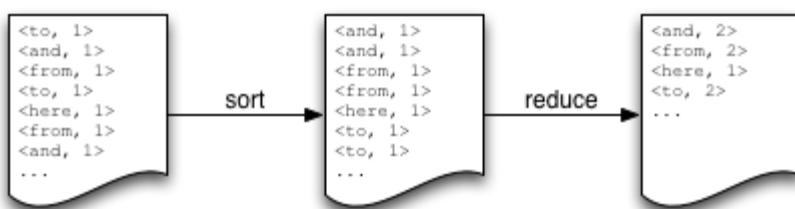


Figure 4.13 A reduce task sorts the incoming data by key, then performs the reduce function on the resulting groups of values

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

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

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

4.5.2 Fault-tolerance

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

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

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

4.5.3 Generality of MapReduce

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

IMPLEMENTING NUMBER OF PAGEVIEWS OVER TIME

```

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

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

```

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

IMPLEMENTING GENDER INFERENCE

```

function map(record) {
    emit(record.userid, normalizeName(record.name)) ①
}

function reduce(userid, vals) {
    allNames = new Set()
    for(normalizedName in vals) {
        allNames.add(normalizedName) ②
    }
    maleProbSum = 0.0
    for(name in allNames) {
        maleProbSum += maleProbabilityOfName(name) ③
    }
    maleProb = maleProbSum / allNames.size()
    if(maleProb > 0.5) { ④
        gender = "male"
    } else {
        gender = "female"
    }
    emit(new InferredGender(userid, gender))
}

```

- ① semantic normalization occurs during the mapping stage
- ② a set is used to remove any potential duplicates
- ③ average the probabilities of being male
- ④ return the most likely gender

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

IMPLEMENTING INFLUENCE SCORE

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

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

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

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

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

- ❶ the first job determines the top influencer for each user
- ❷ the top influencer data is then used to determine the number of people each user influences

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

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

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

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

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

4.6 Word Count on Hadoop

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

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

4.6.1 Mapper

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

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

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

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

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

SIDE BAR Hadoop Writables

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

4.6.2 Reducer

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

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

①

- Hadoop provides an iterator over all values for the current key

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

4.6.3 Application

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

```

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

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

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

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

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

        JobClient.runJob(conf); 5
    }
}

```

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

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

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

4.7 The low level nature of MapReduce

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

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

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

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

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

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

```

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

```

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

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

```

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

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

```

- ① use the word length as the key

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

the reducer.

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

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

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

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

```
public static class AverageReducer extends MapReduceBase
    implements Reducer<IntWritable, IntWritable,
                       IntWritable, DoubleWritable> {

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

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

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

```

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

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

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

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

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

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

    JobClient.runJob(conf); ⑤
}

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

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

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

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

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

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

    JobClient.runJob(conf);
}

```

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

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

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

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

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

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

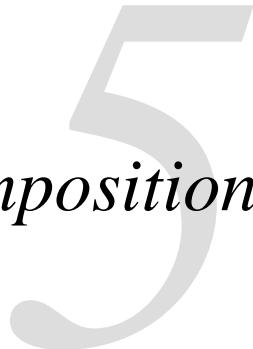
You have seen this pattern before: HDFS is a well-suited platform for storing the master dataset, but the low level APIs complicate the execution of simple tasks. Instead, Pail serves as a high level abstraction to provide an easy API for common tasks. Analogously, in the next chapter we will introduce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

4.8 Conclusion

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

However, it can be hard to think in MapReduce. Although MapReduce provides the essential primitives of fault-tolerance, parallelization, and task scheduling, it's clear that working with the raw MapReduce API is tedious and limiting.

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



Batch layer: abstraction and composition

This chapter covers:

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

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

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

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

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

5.1 An illustrative example

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

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

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

① truncated for brevity

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2 Sources of complexity in data processing code

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

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

5.2.1 Inability to modularize due to performance cost

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

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

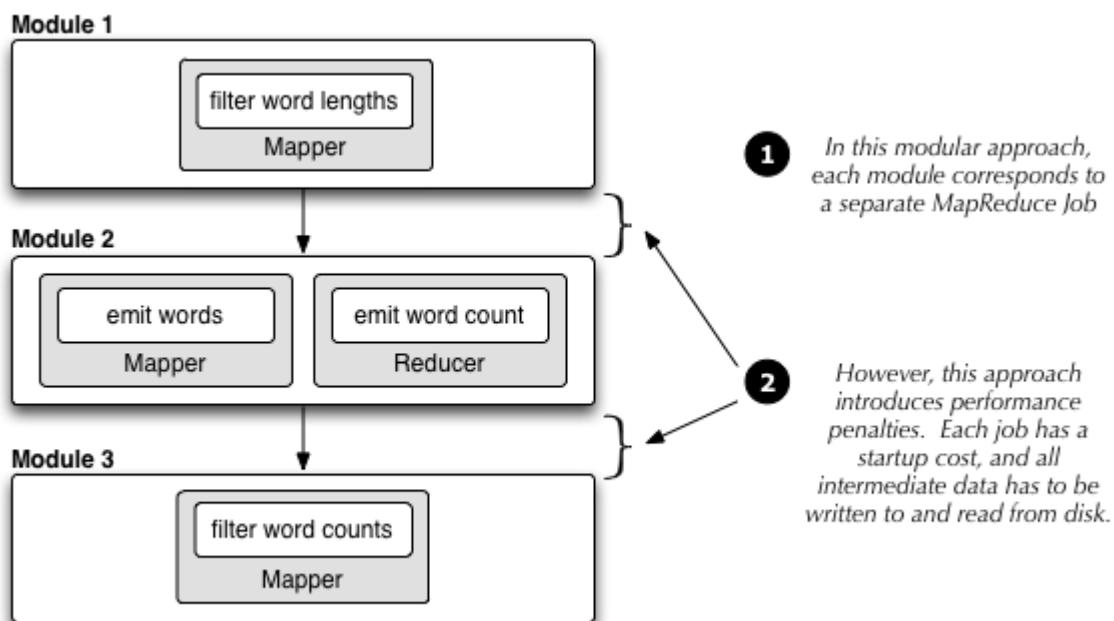


Figure 5.1 Using separate MapReduce jobs as modules imposes latency due to job startup costs and serialization of results between jobs.

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

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

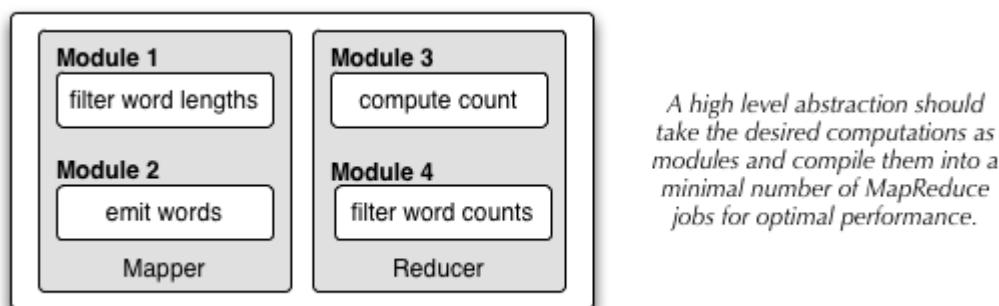


Figure 5.2 A high level abstraction should compiling modularized functionality into as few MapReduce jobs as possible

5.2.2 Custom languages

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

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

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

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

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

5.2.3 Poorly composable abstractions

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

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

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

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

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

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

5.3 An Introduction to JCascalog

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

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

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

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

5.3.1 The JCascalog data model

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

?name	?age	?gender
"alice"	28	"f"
"jim"	48	"m"
"emily"	21	"f"
"david"	25	"m"

}

1 The shared schema defines names for each field contained in a tuple

2 Each tuple corresponds to a separate record and can contain different types of data.

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

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

SIDE BAR An abundance of punctuation?!

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

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

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

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

AGE		GENDER		FOLLOWS		INTEGER	
?person	?age	?person	?gender	?person	?follows	?num	
"alice"	28	"alice"	"f"	"alice"	"david"	-1	
"bob"	33	"bob"	"m"	"alice"	"bob"	0	
"chris"	40	"chris"	"m"	"bob"	"david"	1	
"david"	25	"emily"	"f"	"emily"	"gary"	2	

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

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

5.3.2 The structure of a JCascalog query

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

```
Api.execute(new StdoutTap(), ①
           new Subquery("?person") ②
               .predicate(AGE, "?person", "?age") ③
               .predicate(new LT(), "?age", 30));
```

- ① the destination tap
- ② the output fields
- ③ predicates that define the desired output

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

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

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

Additional example predicates are shown below in Figure 5.5:

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

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

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

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

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

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

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

5.3.3 Querying multiple datasets

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

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

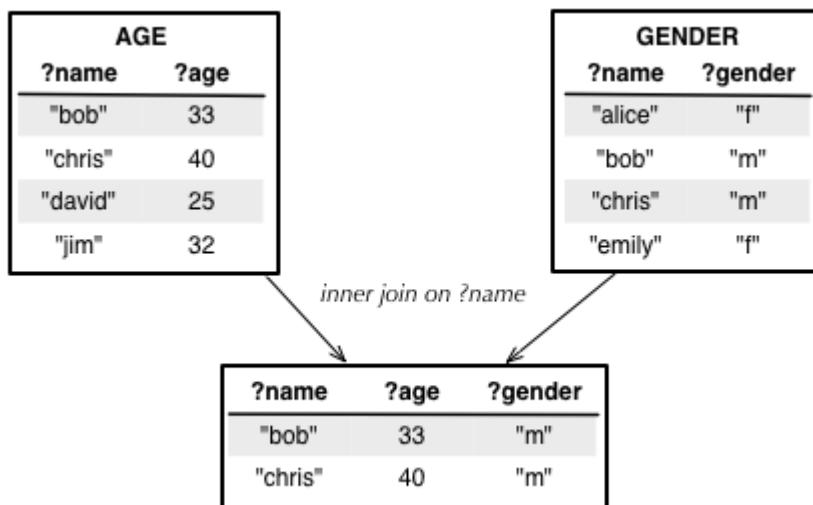


Figure 5.7 This inner join of the AGE and GENDER datasets merges the data for tuples for values of ?person that are present in both datasets.

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

Langauge	Query	Description
SQL	<pre>SELECT AGE.person, AGE.age, GENDER.gender FROM AGE INNER JOIN GENDER ON AGE.person = GENDER.person</pre>	<i>This clause explicitly defines the join condition</i>
JCascalog	<pre>new Subquery("?person", "?age", "?gender") .predicate(AGE, "?person", "?age") .predicate(GENDER, "?person", "?gender);</pre>	<i>By specifying "?person" as a field name for both datasets, JCascalog does an implicit join using the shared name.</i>

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

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

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

Join Type	Query	Results																					
Left Outer Join	<pre>new Subquery("?person", "?age", "!!gender") .predicate(AGE, "?person", "?age") .predicate(GENDER, "?person", "!!gender");</pre>	<table border="1"> <thead> <tr> <th><code>?name</code></th><th><code>?age</code></th><th><code>?gender</code></th></tr> </thead> <tbody> <tr> <td>"bob"</td><td>33</td><td>"m"</td></tr> <tr> <td>"chris"</td><td>40</td><td>"m"</td></tr> <tr> <td>"david"</td><td>25</td><td>null</td></tr> <tr> <td>"jim"</td><td>32</td><td>null</td></tr> </tbody> </table>	<code>?name</code>	<code>?age</code>	<code>?gender</code>	"bob"	33	"m"	"chris"	40	"m"	"david"	25	null	"jim"	32	null						
<code>?name</code>	<code>?age</code>	<code>?gender</code>																					
"bob"	33	"m"																					
"chris"	40	"m"																					
"david"	25	null																					
"jim"	32	null																					
Full Outer Join	<pre>new Subquery("?person", "!!age", "!!gender") .predicate(AGE, "?person", "!!age") .predicate(GENDER, "?person", "!!gender");</pre>	<table border="1"> <thead> <tr> <th><code>?name</code></th><th><code>?age</code></th><th><code>?gender</code></th></tr> </thead> <tbody> <tr> <td>"alice"</td><td>null</td><td>"f"</td></tr> <tr> <td>"bob"</td><td>33</td><td>"m"</td></tr> <tr> <td>"chris"</td><td>40</td><td>"m"</td></tr> <tr> <td>"david"</td><td>25</td><td>null</td></tr> <tr> <td>"emily"</td><td>null</td><td>"f"</td></tr> <tr> <td>"jim"</td><td>32</td><td>null</td></tr> </tbody> </table>	<code>?name</code>	<code>?age</code>	<code>?gender</code>	"alice"	null	"f"	"bob"	33	"m"	"chris"	40	"m"	"david"	25	null	"emily"	null	"f"	"jim"	32	null
<code>?name</code>	<code>?age</code>	<code>?gender</code>																					
"alice"	null	"f"																					
"bob"	33	"m"																					
"chris"	40	"m"																					
"david"	25	null																					
"emily"	null	"f"																					
"jim"	32	null																					

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

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

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

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

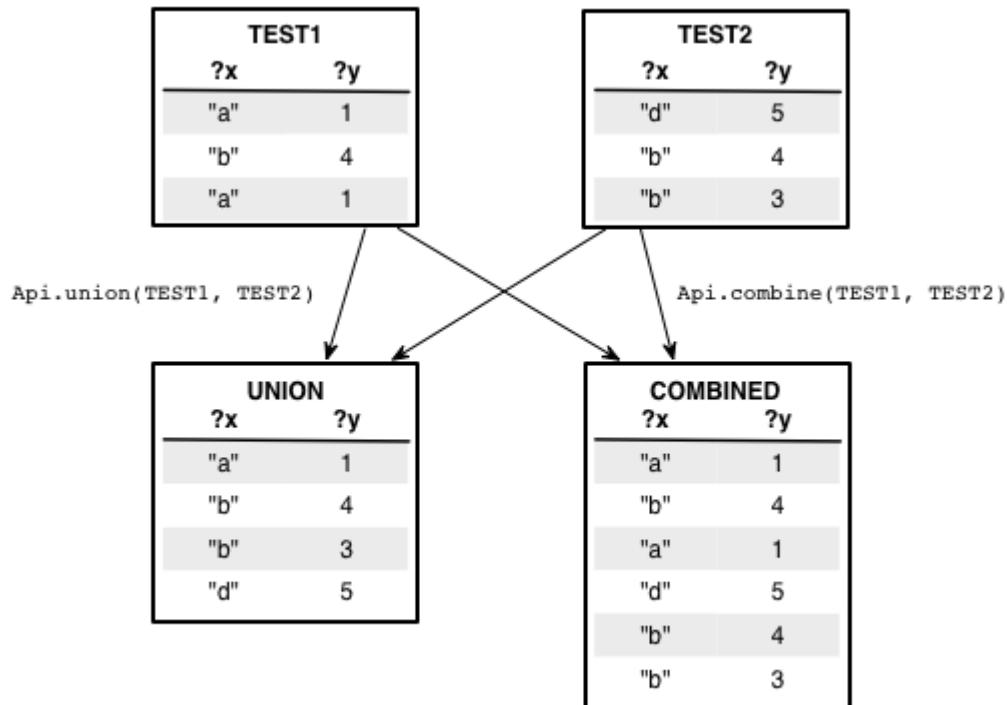


Figure 5.10 JCascalog provides two different means to merge compatible datasets: combine and union. Combine does a simple aggregation of the two sets, whereas union removes any duplicate tuples.

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

5.3.4 Grouping and aggregators

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

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

```

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

```

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

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

```

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

```

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

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

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

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

5.3.5 Stepping though an example query

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

VAL1		VAL2	
?a	?b	?a	?c
"a"	1	"b"	4
"b"	2	"b"	6
"c"	5	"c"	3
"d"	12	"d"	15
"d"	1		

Figure 5.11 Test data for our query execution walkthrough

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

```

new Subquery( "?a", "?avg" )
    .predicate(VAL1, "?a", "?b") ①
    .predicate(VAL2, "?a", "?c")
    .predicate(new Multiply(), 2, "?b").out( "?double-b" ) ②
    .predicate(new LT(), "?b", "?c")
    .predicate(new Count(), "?count") ③
    .predicate(new Sum(), "?double-b").out( "?sum" )
    .predicate(new Div(), "?sum", "?count").out( "?avg" )
    .predicate(new Multiply(), 2, "?avg").out( "?double-avg" ) ④
    .predicate(new LT(), "?double-avg", 50);

```

- ① generators for the test datasets
- ② pre-aggregator function and filter
- ③ multiple aggregators
- ④ post-aggregator predicates

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

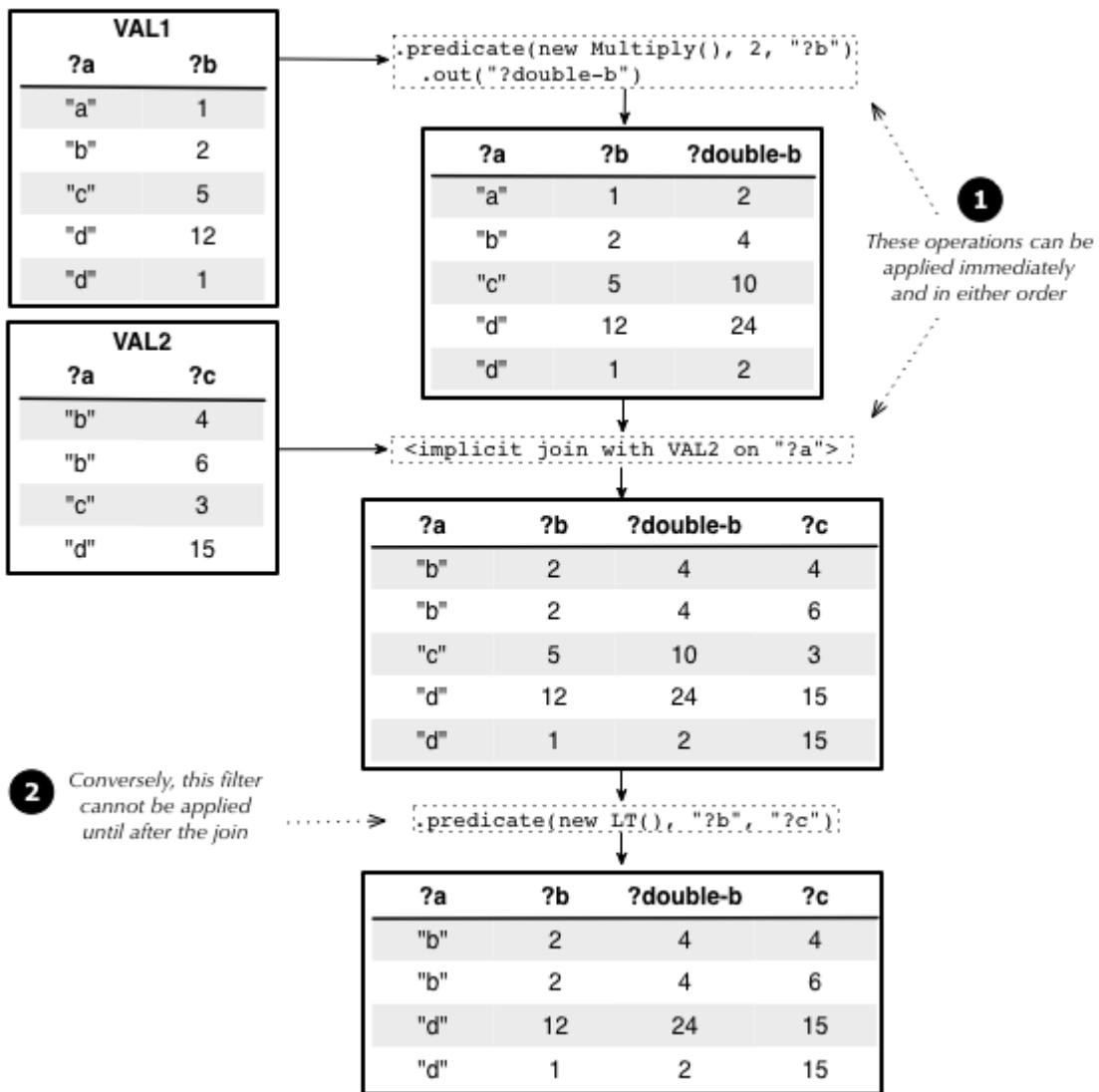


Figure 5.12 The first stage of execution entails of applying all functions, filters and joins where the input variables are available.

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

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

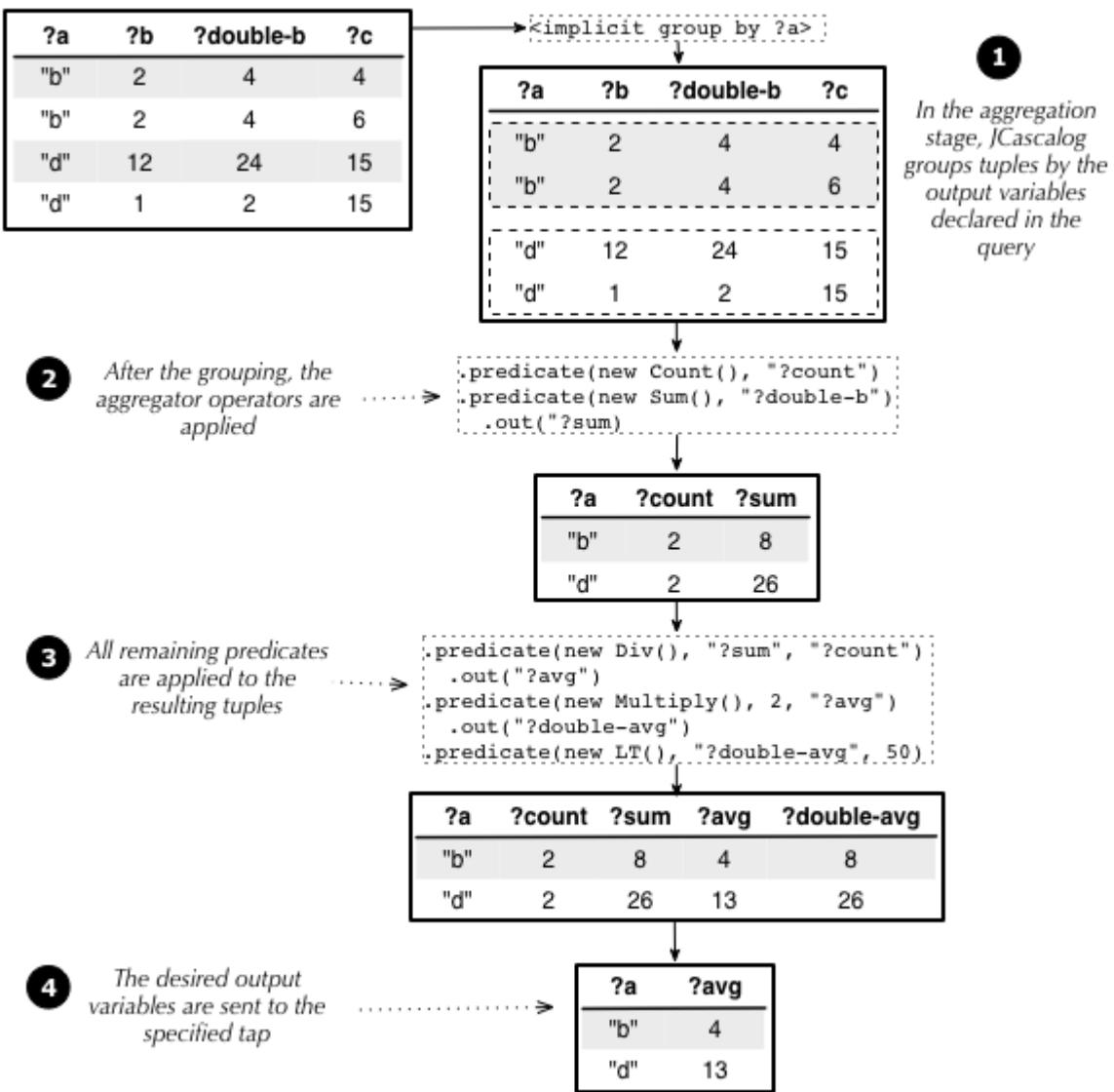


Figure 5.13 The aggregation and post-aggregation stages for the query. The tuples are grouped based on the desired output variables, then all aggregators are applied. All remaining predicates are then executed and the desired output is returned.

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

SIDE BAR**A verbose explanation**

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

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

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

5.3.6 Custom predicate operations

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

FILTERS

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

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

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

FUNCTIONS

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

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

- ① obtain the value from the input tuple
- ② emit a new tuple with the incremented value

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

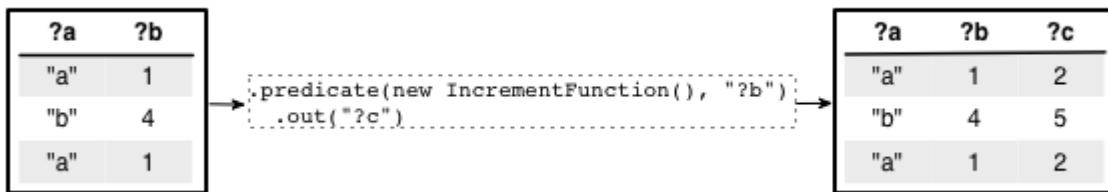


Figure 5.14 The `IncrementFunction` predicate applied to some sample tuples

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

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

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

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

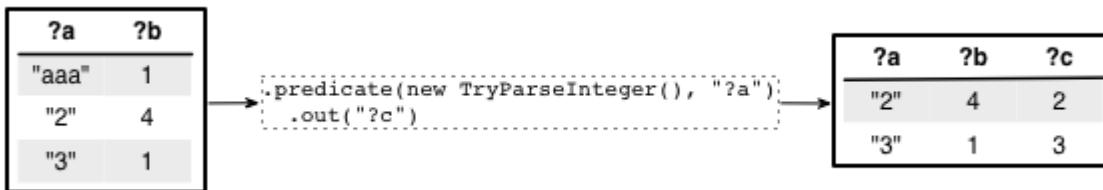


Figure 5.15 The TryParseInteger function filters rows that where ?a cannot be converted to an integer value

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

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

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

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

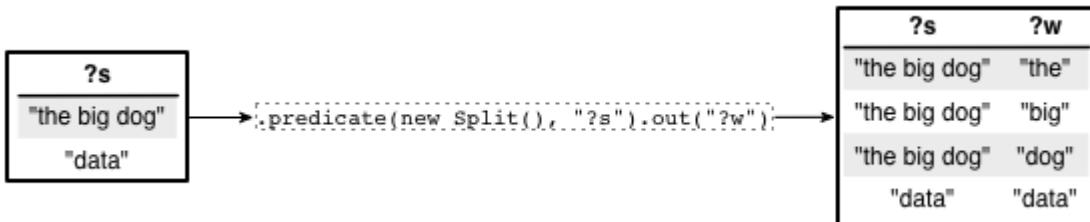


Figure 5.16 The Split function can emit multiple tuples from a single input tuple

AGGREGATORS

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

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

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

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

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

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

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

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

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

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

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

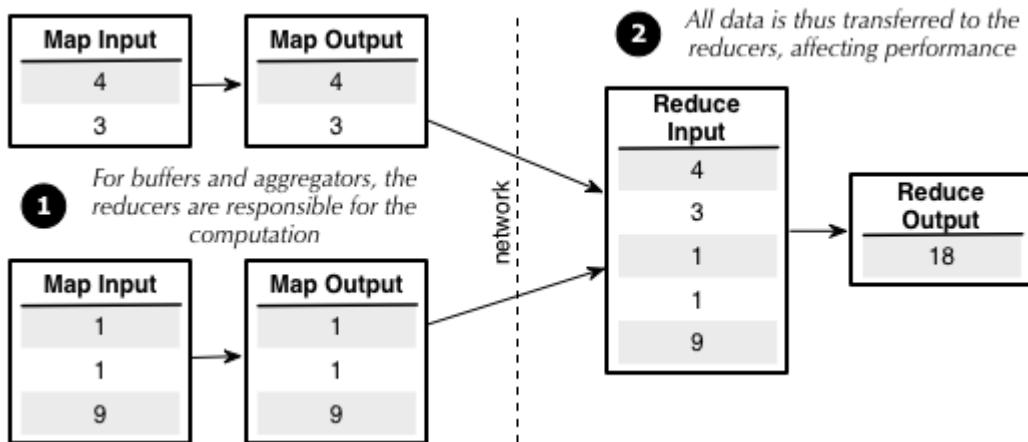


Figure 5.17 Execution of sum Aggregator and sum Buffer at the MapReduce level

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

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

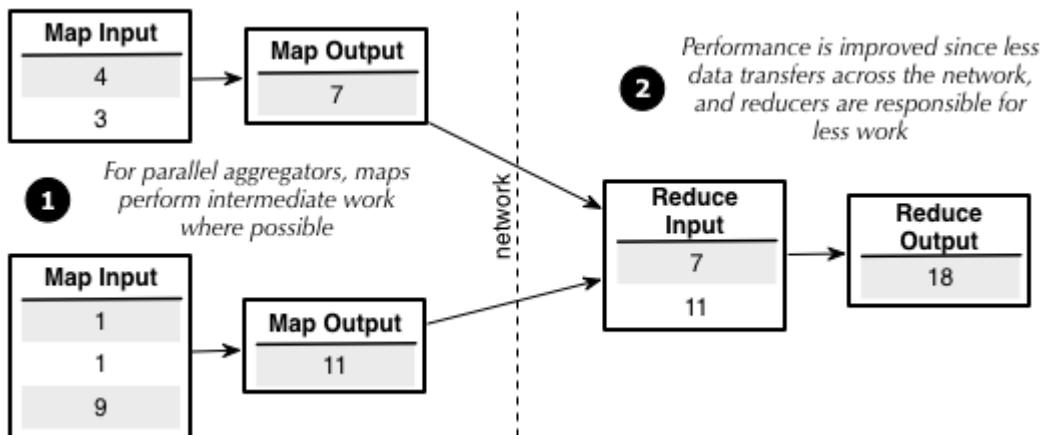


Figure 5.18 Execution of a sum Parallel Aggregator at the MapReduce level

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

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

The following listing implements sum as a Parallel Aggregator.

```
public static class SumParallel implements ParallelAgg {
    public void prepare(FlowProcess process, OperationCall call) {}

    public List<Object> init(List<Object> input) {
        return input; ①
    }

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

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

Parallel Aggregators can be chained with other Parallel Aggregators or regular Aggregators. However, when chained with regular Aggregators, Parallel

Aggregators are unable to do partial aggregations in the map tasks and will act like regular Aggregators.

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

5.4 Composition

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

5.4.1 Combining subqueries

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

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

```
Subquery manyFollows = new Subquery( "?person" ) ①
    .predicate(FOLLOWERS, "?person", "_")
    .predicate(new Count(), "?count")
    .predicate(new GT(), "?count", 2); ③

Api.execute(new StdoutTap(),
    new Subquery( "?person1", "?person2" )
        .predicate(manyFollows, "?person1") ④
        .predicate(manyFollows, "?person2")
        .predicate(FOLLOWERS, "?person1", "?person2")); ⑤
```

- ① the first subquery determines all people that follow more than 2 others
- ② consider only the follower, not the source

- 3 count the number of people each user follows and keep those with count greater than 2
- 4 use the results of the first subquery as the source for this subquery
- 5 only keep records where follower and source are both present in the result of the first subquery

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

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

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

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

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

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

5.4.2 Dynamically created subqueries

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

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

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

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

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

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

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

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

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

① disregard all fields but the buyer

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

Suppose you have a set of retweet data with each record denoting a retweet of some other tweet, and you want find all chains of retweets of a certain length. That is, for a chain of length 4, you want to know all retweets of retweets of retweets of tweets. The original dataset consists of pairs of tweet identifiers. Notice that you can transform these pairs into chains of length 3 by joining the dataset with itself. Similarly, you can then find chains of length 4 by joining the length 3 chains with the original pairs. To illustrate, here's a query that returns chains of length 3 given an input generator of pairs:

```
public static Subquery chainsLength3(Object pairs) {
    return new Subquery("?a", "?b", "?c")
        .predicate(pairs, "?a", "?b")
        .predicate(pairs, "?b", "?c");
}
```

An additional join finds all chains of length 4:

```
public static Subquery chainsLength4(Object pairs) {
    return new Subquery("?a", "?b", "?c", "?d")
        .predicate(pairs, "?a", "?b")
        .predicate(pairs, "?b", "?c")
        .predicate(pairs, "?c", "?d");
}
```

To generalize this process to find chains of any length, you need a function that generates a subquery with the correct number of predicates and variables. This can be accomplished by writing some fairly simple Java code:

```
public static Subquery chainsLengthN(Object pairs, int n) {
    List<String> genVars = new ArrayList<String>();
    for(int i=0; i<n; i++) {
        genVars.add(Api.genNullableVar());
```

①

```

    }

Subquery ret = new Subquery(genVars);
for(int i=0; i<n-1; i++) {
    ret = ret.predicate(pairs, genVars.get(i), genVars.get(i+1)); ②
}
return ret;
}

```

- ① generate unique nullable output variables
- ② loop to define the required number of joins

An interesting note about this function is that it's not specific to retweet data: in fact, it can take any subquery or source of data containing pairs and return a subquery that computes chains.

Let's look at one more example of a dynamically created subquery. Suppose you want to draw a random sample of N elements from a dataset of unknown size. The simplest strategy to accomplish this in a distributed and scalable way is with the following algorithm:

1. Generate a random number for every element
2. Find the N elements with the smallest random numbers

JCascalog has a built-in aggregator named "Limit" for performing the second step. Limit uses a strategy similar to parallel aggregators where it finds the smallest N elements on each map task, then combines the results from all the map tasks to find the N smallest elements overall. The following code implements this strategy to draw a random sample:

```

public static Subquery fixedRandomSample(Object data, int n) {
    List<String> inputVars = new ArrayList<String>();
    List<String> outputVars = new ArrayList<String>();
    for(int i=0; i < Api.numOutFields(data); i++) { ①
        inputVars.add(Api.genNullableVar()); ②
        outputVars.add(Api.genNullableVar());
    }

    String randVar = Api.genNullableVar(); ③
    return new Subquery(outputVars)
        .predicate(data, inputVars)
        .predicate(new RandLong(), randVar) ④
        .predicate(Option.SORT, randVar) ⑤
}

```

```

    .predicate(new Limit(n), inputVars).out(outputVars); ⑥
}

```

- ① introspect the input dataset to determine the correct number of input and output fields
- ② generate separate fields for input and output variables
- ③ create a separate field to hold the random values
- ④ use the JCascalog RandLong function to append each input tuples with a random value
- ⑤ perform secondary sorting on the random values
- ⑥ use the Limit aggregator to find N random tuples from the dataset

The brilliant aspect of this algorithm is its scalability: it parallelizes the computation of the fixed sample without ever needing to centralize all the records in one place.

When writing JCascalog queries, you will notice that certain combinations of predicates are frequently used together. In these situations it is simpler and more efficient to express the collective functionality with a single operation. We next delve into how JCascalog supports this ability by use of predicate macros.

5.4.3 Predicate macros

A predicate macro is an operation that JCascalog expands to another set of predicates. Since JCascalog represents all operations as predicates, macros can create powerful abstractions by composing predicates together, whether they are aggregators, filters or functions.

You've already seen one example of a predicate macro with the definition of Average from the beginning of this chapter. Let's look at that definition once more:

```

PredicateMacroTemplate Average =
  PredicateMacroTemplate.build("?val") ①
    .out("?avg") ②
    .predicate(new Count(), "?count") ③
    .predicate(new Sum(), "?val").out("?sum") ④
    .predicate(new Div(), "?sum", "?count").out("?avg"); ⑤

```

- ① use a template to define a macro with one input variable
- ② the macro returns a single output
- ③ average expands to three predicates: count, sum and div
- ④ temp variables store the results of the aggregation
- ⑤ divide aggregate results to compute final output

Average consists of three predicates composed together: a count aggregation, a sum aggregation, and a division function. Figure 5.19 demonstrates how Average is called and its resulting expansion:

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

```



```
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")
  .out("?result");
```

Example source code using the Average predicate macro

Behind the scenes, JCascalog expands the macro into its constituent predicates using unique field names so not to conflict with the surrounding subquery

Figure 5.19 Predicate macros provide powerful abstractions to write simple queries that JCascalog automatically expands into the constituent predicates

The definition of Average uses a JCascalog template to specify simple compositions of predicates when the number of input and output fields are fixed. However, not everything can be specified with a template. For example, suppose you wanted to create a predicate macro that computes the number of distinct values for a given set of variables, like so:

```
new Subquery("?distinct-followers-count")
  .predicate(FOLLOWS, "?person", "_")
  .predicate(new DistinctCount(), "?person") ①
  .out("?distinct-followers-count");
```

- ① the desired macro counts the number of distinct followers

This subquery determines the number of distinct users that follow at least one other person. Unlike calculating the average of a single variable, you could potentially calculate distinct counts for variable sets of any size. Predicate macros can support this generality of arbitrary sets, but you must define the macro manually instead of using a template.

Towards this end, you first need to define an aggregator that performs the actual computation. This aggregator must work even if the number of tuples for a group is so large that it could not be contained in memory. To solve this problem, you can make use of a feature called *secondary sorting* that sorts the group of tuples before being processed by the aggregator. Once sorted, the aggregator only

increments the distinct count if the current tuple is different from its predecessor. The code to perform the aggregation is provided below:

```
public static class DistinctCountAgg extends CascalogAggregator {
    static class State { ①
        int count = 0;
        Tuple last = null;
    }

    public void start(FlowProcess process, AggregatorCall call) {
        call.setContext(new State()); ②
    }

    public void aggregate(FlowProcess process, AggregatorCall call) {
        State s = (State) call.getContext(); ③
        Tuple t = call.getArguments().getTupleCopy();
        if(s.last==null || !s.last.equals(t)) {
            s.count++; ④
        }
        s.last = t; ⑤
    }

    public void complete(FlowProcess process, AggregatorCall call) {
        State s = (State) call.getContext();
        call.getOutputCollector().add(new Tuple(s.count)); ⑥
    }
}
```

- ① internal state to track the current count and the previously seen tuple
- ② for each group, initialize the tracking state
- ③ when processing a tuple, retrieve the current state
- ④ increase the distinct count only if the current tuple differs from the previous one
- ⑤ always update the last seen tuple in the state
- ⑥ when all tuples of the group have been processed, emit the distinct count

`DistinctCountAgg` contains the logic to compute the unique count given a sorted input; unsurprisingly, JCascalog has an `Option.SORT` predicate to specify how to sort the tuples for each group. The following listing demonstrates how you define the sort and compute the distinct count by hand:

```
public static Subquery distinctCountManual() {
    return new Subquery("?distinct-followers-count")
        .predicate(FOLLOWS, "?person", "_")
        .predicate(Option.SORT, "?person") ①
```

```
.predicate(new DistinctCountAgg(), "?person")
.out("?distinct-followers-count");
```

- ① sorts the tuple by

Of course, you would much prefer a macro so that you don't have to specify the sort and aggregator each time you want to do a distinct count. The most general form of a predicate macro is a function that takes a list of input fields, a list of output fields, and returns a set of predicates. The following is the definition of DistinctCount as a regular PredicateMacro:

```
public static class DistinctCount implements PredicateMacro {
    public List<Predicate> getPredicates(Fields inFields,
                                           Fields outFields) { ①
        List<Predicate> ret = new ArrayList<Predicate>();
        ret.add(new Predicate(Option.SORT, inFields)); ②
        ret.add(new Predicate(new DistinctCountAgg(),
                             inFields,
                             outFields)); ③
        return ret;
    }
}
```

- ① the input and output fields are determined when the macro is used within a subquery
- ② groups are sorted by the provided input fields
- ③ for this macro distinct count emits a single field, but the general macro form supports multiple outputs

5.4.4 Dynamically created predicate macros

You previously saw how regular Java functions can dynamically create subqueries, so it is no great surprise that you can do the same with predicate macros. This is an extremely powerful technique that showcases the advantages of having your query tool just be a library for your general purpose programming language.

Consider the following query:

```
new Subquery("?x", "?y", "?z")
.predicate(TRIPLETS, "?a", "?b", "?c") ①
.predicate(new IncrementFunction(), "?a").out("?x") ②
.predicate(new IncrementFunction(), "?b").out("?y")
.predicate(new IncrementFunction(), "?c").out("?z");
```

- ① read a dataset containing triples of numbers
- ② return a new triplet where each field is incremented

Although a simple query, there is considerable repetition since it must explicitly apply the IncrementFunction to each field from the input data. It would be nice to have a macro that eliminates this repetition, like so:

```
new Subquery(?x", "?y", "?z")
  .predicate(TRIPLETS, "?a", "?b", "?c")
  .predicate(new Each(new IncrementFunction()), "?a", "?b", "?c")
  .out("?x", "?y", "?z");
```

Rather than repeatedly using the IncrementFunction, the Each macro applies the function to the specified input fields and generates the desired output. The expansion of the macro matches the three separate predicates in the original query. The Each macro...

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

    public Each(Object op) {
        _op = op; ①
    }

    public List<Predicate> getPredicates(Fields inFields,
                                          Fields outFields) {
        List<Predicate> ret = new ArrayList<Predicate>();
        for(int i=0; i<inFields.size(); i++) {
            Object in = inFields.get(i);
            Object out = outFields.get(i);
            ret.add(new Predicate(_op,
                Arrays.asList(in),
                Arrays.asList(out))); ②
        }
        return ret;
    }
}
```

- ① Each is parameterized with the predicate operation to use
- ② the macro creates a predicate for each given input/output field pair

Let's look at another example of a dynamic predicate macro. We earlier defined the IncrementFunction as its own function that increments its argument, but in

reality it is simply the Plus function with one argument set to "1". A useful macro would generate new predicates by abstracting away the partial application of a predicate operation. We could then define the Increment operation like this:

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

As you can see, Partial is a predicate macro that fills in some of the input fields. It allows you to rewrite the query that increments the triplets as so:

```
new Subquery( "?x", "?y", "?z")
  .predicate(TRIPLETS, "?a", "?b", "?c")
  .predicate(new Each(new Partial(new Plus(), 1)), "?a", "?b", "?c")
  .out(" ?x", " ?y", " ?z");
```

After expanding all the predicate macros, this query translates to:

```
new Subquery( "?x", "?y", "?z")
  .predicate(TRIPLETS, "?a", "?b", "?c")
  .predicate(new Plus(), 1, "?a").out(" ?x")
  .predicate(new Plus(), 1, "?b").out(" ?y")
  .predicate(new Plus(), 1, "?c").out(" ?z");
```

The definition of Partial is straightforward:

```
public static class Partial implements PredicateMacro {
  Object _op;
  List<Object> _args;

  public Partial(Object op, Object... args) {
    _op = op;
    _args = Arrays.asList(args);
  }

  public List<Predicate> getPredicates(Fields inFields,
                                         Fields outFields) {
    List<Predicate> ret = new ArrayList<Predicate>();
    List<Object> input = new ArrayList<Object>();
    input.addAll(_args);
    input.addAll(inFields);
    ret.add(new Predicate(_op, input, outFields));
    return ret;
  }
}
```

```

    }
}
```

The predicate macro simply prepends any provided input fields to the input fields specified when the subquery is created. As you can see, dynamic predicate macros gives you great power to manipulate the construction of your subqueries.

5.5 Conclusion

The way you express your computations is crucially important in order to avoid complexity, prevent bugs, and increase productivity. The main techniques for fighting complexity are abstraction and composition, and it's important that your data processing tool encourage these techniques rather than make them difficult.

In the next chapter, we will tie things together by showing how to use JCascalog along with the graph schema from Chapter 2, and the Pail from Chapter 3, to build out the batch layer for SuperWebAnalytics.com. These examples will be more sophisticated than what you saw in this chapter and show how the abstraction and composition techniques you saw in this chapter apply towards a realistic use case.



Batch layer: Tying it all together

This chapter covers:

- Building a batch layer from end to end
- Ingesting new data into the master dataset
- Practical examples of precomputation
- Integrating Thrift-based graph schemas, Pail and JCascalog

In the last few chapters you have learned all the pieces of the batch layer: formulating a schema for your data, storing a master dataset, and running computations at scale with a minimum of complexity. In this chapter you will tie these pieces together into a coherent batch layer. No new theory is introduced in this chapter - our goal is to reinforce the concepts of the previous chapters by going through a batch layer implementation from start to finish. There is great value in understanding how the theory maps to the nuts and bolts implementation of a non-trivial example.

Specifically, you will learn how to create the batch layer for our running example of SuperWebAnalytics.com. SuperWebAnalytics.com is complex enough to require a fairly sophisticated batch layer, but not so complex as to lose you in the details. You will see that the various batch layer abstractions fit together nicely and that the resulting batch layer for SuperWebAnalytics.com is quite elegant.

After reviewing the product requirements for SuperWebAnalytics.com, we will give a broad overview of what the batch layer must accomplish and what should be precomputed for each batch view. Then you will implement each portion of the batch layer using Thrift, Pail and JCascalog. Throughout the chapter, keep in mind the flexibility of the batch layer. We will cover the implementation of three example batch views, but it is very easy to extend the batch layer to compute new

views. This means the batch layer is inherently prepared to adapt to changing customer and application requirements.

6.1 Design of the SuperWebAnalytics.com batch layer

You will be building the batch layer for SuperWebAnalytics.com to support the computation of three types of queries. Recall from Chapter 4 that the goal of the batch layer is to precompute views so that the specified queries can be satisfied with low latency. After introducing the queries SuperWebAnalytics.com will support, we will discuss the batch views needed to answer them.

6.1.1 Supported queries

SuperWebAnalytics.com will support three distinct types of queries:

1. **Page view counts by URL sliced by time.** "What were the pageviews for each day over the past year?" and "How many pageviews have there been in the past 12 hours?"
2. **Unique visitors by URL sliced by time.** "How many unique users frequented this domain in 2010?" and "How many unique people visited this domain each hour for the past three days?"
3. **Bounce rate analysis.** "What percentage of people visit the page without visiting any other pages on this website?"

The way people are modeled makes the second query type more challenging. Recall that the SuperWebAnalytics.com schema represents a person as either the user ID of a logged-in user or via a cookie identifier from the browser. A single person could therefore visit the same site under different identifiers - their cookie may change if they clear the cookie, or the user could register with multiple user IDs. The schema handles this multiplicity by defining *equiv* edges that indicate when two different user representations are actually the same person. The equiv graph for a person can be arbitrarily complex, as shown in Figure 6.1. Accurately computing the second query type requires that you must analyze the data to determine which pageviews belong to the same person but using different identifiers.

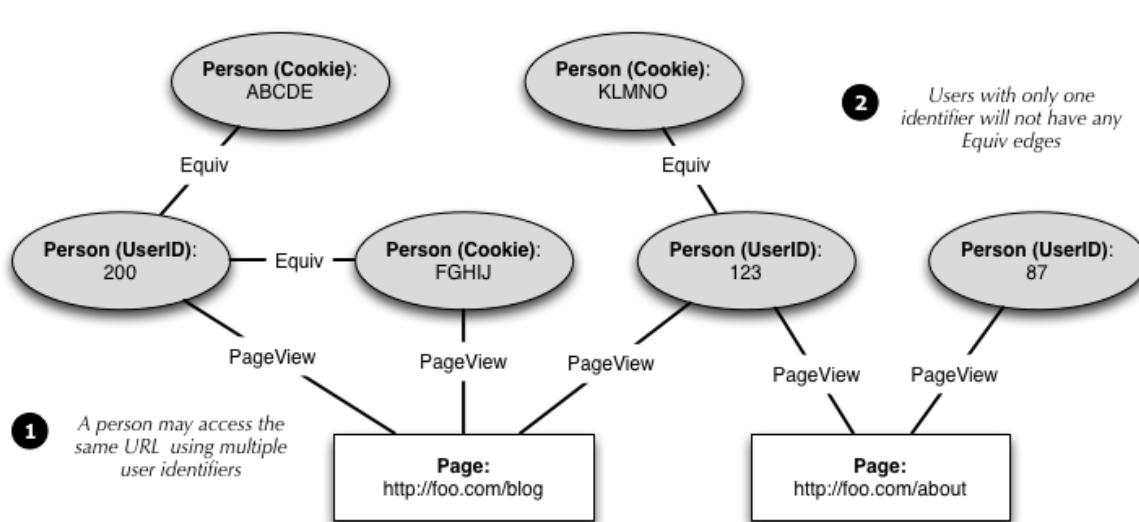


Figure 6.1 Examples of different pageviews for same person being captured using different identifiers

6.1.2 Batch views

Next we will review the batch views needed to satisfy each query. The key to each batch view is striking a balance between the size of the precomputed views and the amount of on-the-fly computation required at query time.

PAGEVIEWS OVER TIME

We want to be able to retrieve the number of pageviews for a URL for any time range down to the granularity of an hour. As previously mentioned in Chapter 4, precomputing the pageview counts for every possible time range is infeasible, as that would require an unmanageable 380 million precomputed values for every URL for each year covered by the dataset. Instead, you can precompute a smaller amount and require more computation to be done at query time.

The simplest approach is to precompute the number of pageviews of each URL for every hour bucket. This would result with a batch view that looks like Figure 6.2. To resolve a query, you retrieve the value for every hour bucket in the time range and sum the values together.

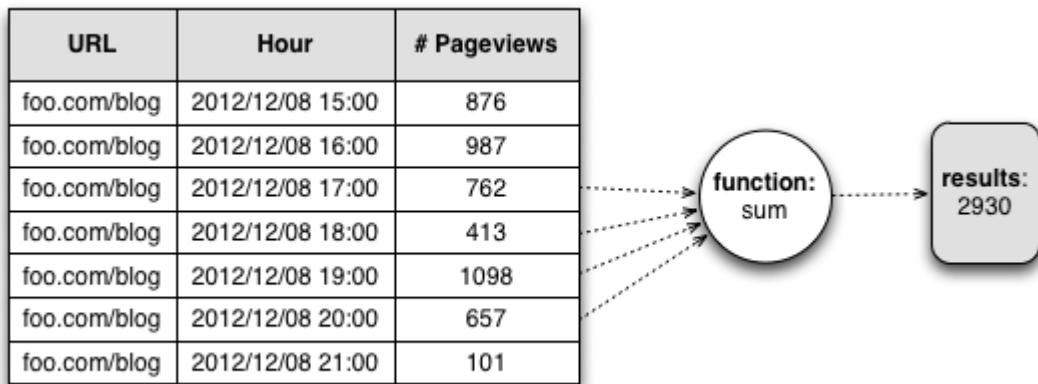


Figure 6.2 Precomputing pageviews with an hourly granularity

However there is a problem with this approach - the query becomes slower as you increase the size of the time range. Finding the number of pageviews for a one year time period requires approximately 8760 values to be retrieved from the batch view and added together. As many of those values are going to be served from disk, this can cause the latency of queries with large ranges to be substantially higher than queries with small ranges.

Fortunately the solution is simple. Instead of precomputing values only using an hourly granularity, you can also precompute at coarser granularities such as day, 7 day (week) and 28 day (month) intervals. An example best demonstrates how this improves latency.

Suppose you want to compute the number of pageviews from March 3rd at 3am through September 17th at 8am. If you only used hourly values, this query would require retrieving and summing the values for 4805 hour buckets. Alternatively, using coarser granularities can substantially reduce the number of retrieved values. The idea is to retrieve values for each month between March 3rd and September 17th, then add or subtract values for more refined intervals to get the desired range. This idea is illustrated in Figure 6.3:

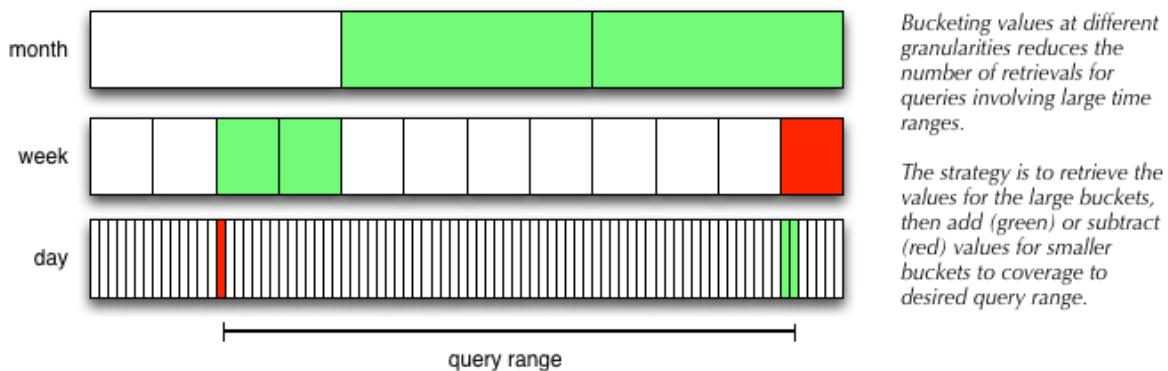


Figure 6.3 Optimizing pageviews over large query ranges using coarser granularities

For this query, only 26 values need to be retrieved - almost a 200x improvement! You may wonder how expensive it is to precompute values for the day, 7 day and 28 day intervals in addition to the hourly buckets. Astonishingly there is hardly any additional cost. Figure 6.4 presents how many time buckets are needed for each granularity for a one year period.

Granularity	Number of Buckets in 1 Year
hourly	8760
daily	~ 365
weekly	~ 52
monthly	~ 13

Figure 6.4 Number of buckets in a one year period for each granularity

Adding up the numbers, the day, 7 day and 28 day buckets require an additional 430 values to be precomputed for every URL for a one year period. That is only a 5% increase in precomputation for a 200x reduction in the query time work for large ranges - a more than acceptable tradeoff.

UNIQUE VISITORS OVER TIME

The next query type determines the number of unique visitors for a specified time interval. This seems like it should be similar to pageviews over time, but there is one key difference: unique counts are not additive. Whereas you can get the total number of pageviews for a two hour period by adding the values for each individual hour together, you cannot do the same for this query type. This is because a unique count represents the size of a *set* of elements, and there may be overlap between the sets for each hour. If you simply added the counts for the two hours together, you would double-count the people that visited the URL in both time intervals.

The only way to compute the number of uniques with perfect accuracy over any time range is to compute the unique count on the fly. This requires random access to the set of visitors for each URL for each hour time bucket. This is doable but expensive, as essentially your entire master dataset must be indexed. Alternatively, you can use an approximation algorithm that sacrifices some accuracy to vastly decrease the amount of data to be indexed in the batch view. An example of an approximation algorithm for distinct counting is the HyperLogLog algorithm. For every URL and hour bucket, HyperLogLog only requires information on the order of 1KB to estimate set cardinalities of up to one billion with a maximum 2% error rate.¹

Footnote 1 The HyperLogLog algorithm is described in the research paper available at <http://algo.inria.fr/flajolet/Publications/FIFlGaMe07.pdf>

Although an intriguing algorithm, we want to avoid becoming sidetracked with the details of HyperLogLog. Instead we will be using a library that implements the HyperLogLog algorithm for us. The library provides the following interface:

```
interface HyperLogLog {
    long cardinality();
    boolean offer(Object o);
    HyperLogLog merge(HyperLogLog... otherSets);
}
```

Each HyperLogLog object represents a set of elements and supports adding new elements to the set, merging with other HyperLogLog sets, and retrieving the size of the set. Using HyperLogLog makes the uniques over time query very similar to the pageviews over time query. The key differences are that a relatively

larger value is computed for each URL and time bucket, and the HyperLogLog merge function is used to combine time buckets instead of summing counts together. As with pageviews over time, HyperLogLog sets for 1 day, 7 day and 28 day granularities are also created to reduce the amount of work to be done at query time.

BOUNCE RATE ANALYSIS

The final query type is to determine the bounce rate for every domain. The batch view for this query is simple: a map from each domain to the number of bounced visits and the total number of visits. The bounce rate is simply the ratio of these two values.

The key to precomputing these values is defining what exactly constitutes a visit. We will define two pageviews as being part of the same visit if they are from the same user to the same domain and are separated by less than half an hour. A visit is considered a bounce if it only contains one pageview.

6.2 Workflow overview

Now that the specific requirements for the batch views are understood, we can define the batch layer workflow at a high level. The basis of the workflow is illustrated in Figure 6.5:

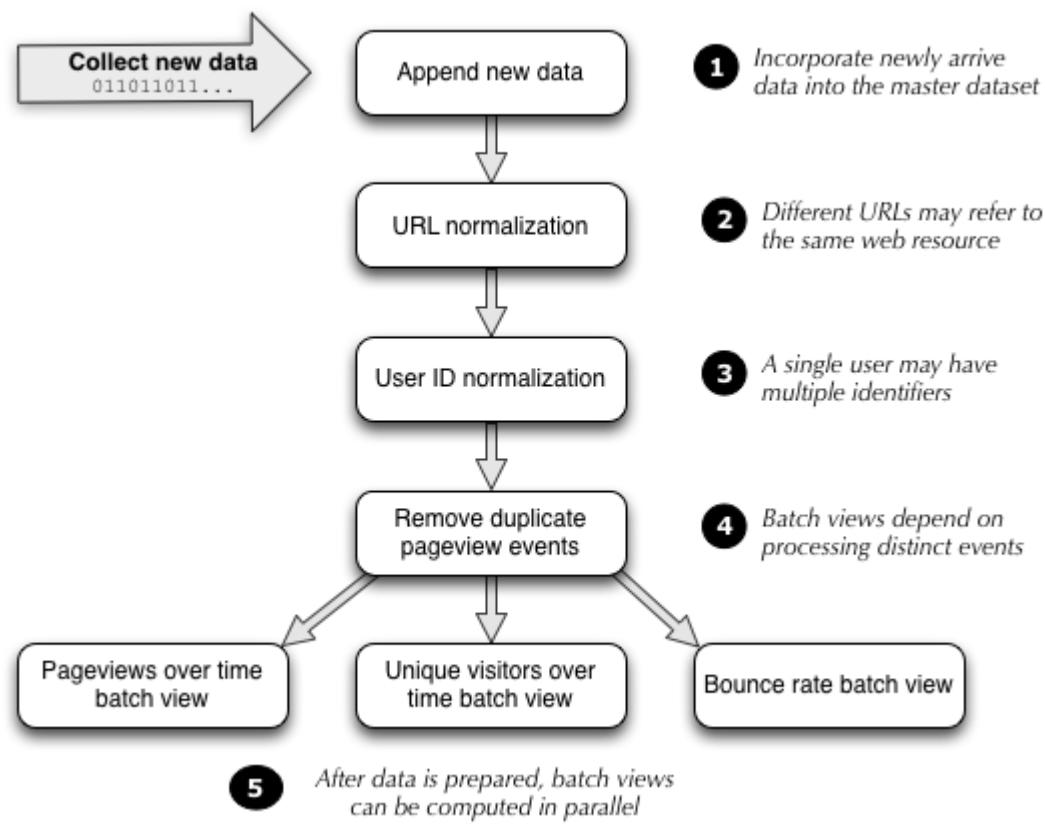


Figure 6.5 Batch workflow for SuperWebAnalytics.com

At the start of the batch layer workflow is a single folder on the distributed filesystem that contains the master dataset. Since it is very common for computations to use only a small subset of the different properties and edges, the master dataset is a pail where each property and edge type is stored in a different subfolder. The first step of the workflow occurs when someone wants to add more data to the system. To do so, they first create a separate pail to store new data units, then append the contents of this new pail into the master dataset. Afterwards, all of the merged data is removed from the new data pail since it has been incorporated into the batch layer.

The next two steps normalize the data in preparation to compute the batch views. The first normalization step accounts for the fact that different URLs can refer to the same resource. For example, the distinct URLs "www.mysite.com/blog/1?utmcontent=1" and "http://mysite.com/blog/1" refer to the same location. This first normalization step transforms all URLs to a standard format so that future computations can correctly aggregate the data.

The second normalization step is needed since data for the same person may exist under different user identifiers. In order to support queries about visits and

visitors, you must select a single identifier for each person. This latter normalization step processes the equiv graph to accomplish this task. As the batch views only make use of the pageviews data, only the pageview edges will be converted to use these selected user IDs.

The next step de-duplicates the pageview events. Recall from Chapter 2 the advantages of having your data units contain enough information to make them uniquely identifiable. In problematic scenarios (e.g., network partitioning), it is common to register the same pageview multiple times to ensure the event is recorded. De-duplicating the pageviews is needed to compute the batch views, as they depend upon the distinct events in the dataset.

The final step is to use the normalized data to compute the batch views described in the previous section. Note that this workflow is a pure recomputation workflow - every time new data is added, the batch views are recomputed from scratch. In a later chapter, you will learn that in many cases you can incrementalize the batch layer so recomputing using the entire master dataset is not always required. However, it is absolutely essential to have the pure recomputation workflow defined because you need to recompute from scratch in case the views become corrupted.

6.3 Preparing the workflow

A quick preparation step is required before you begin implementing the workflow itself. Many parts of the workflow manipulate objects defined in the Thrift schema, such as Data, PageViewEdge, PageID and PersonID objects. Hadoop needs to know how to serialize and deserialize these objects so they can be transferred between machines during MapReduce jobs. To do so, you must register a serializer for your objects - the cascading-thrift project² provides an implementation suitable for this purpose. The following code snippet demonstrates how to register it for the batch workflow.

Footnote 2 <https://github.com/cascading/cascading-thrift>

```
public static void setApplicationConf() {
    Map conf = new HashMap();
    String sers = "backtype.hadoop.ThriftSerialization," + ①
        "org.apache.hadoop.io.serializer.WritableSerialization"; ②
    conf.put("io.serializations", sers);
    Api.setApplicationConf(conf);
}
```

- ① the Thrift serializer for SuperWebAnalytics.com objects
- ② the default serializer for Hadoop Writable objects

This code informs Hadoop to use both the Thrift serializer as well as the default Hadoop serializer. When registering multiple serializers, Hadoop will automatically determine the appropriate serializer when it needs to serialize an object. This code sets the configuration globally and will be used by every job in the batch workflow.

6.4 Ingesting new data

The first step of the workflow is to add new data to the master dataset pail. Although straightforward in concept, there may be synchronization issues. Omitting the details of the actual append for a moment, suppose you tried the following:

```
// do not use!
public static void badNewDataAppend(Pail masterPail, Pail newDataPail)
    throws IOException {
    appendNewDataToMasterDataPail(masterPail, newDataPail);
    newDataPail.clear();
}
```

This seems simple enough, but there is a hidden race condition in this code. While the append is running, more data may be written into the new data pail. If you clear the new data pail after the append finishes, you will also delete any new data that was written while the append job was running.

Fortunately there is an easy solution. Pail provides *snapshot* and *deleteSnapshot* methods to solve this problem. The *snapshot* method stores a snapshot of the pail in a new location, while *deleteSnapshot* removes from the original pail only the data that exists in the snapshot. With these methods, the following code ensures that the only removed data is data that was successfully appended to the master dataset pail:

```
public static void ingest(Pail masterPail, Pail newDataPail)
    throws IOException {
    FileSystem fs = FileSystem.get(new Configuration());
    fs.delete(new Path("/tmp/swa"), true);
    fs.mkdirs(new Path("/tmp/swa")); ①
```

```

Pail snapshotPail = newDataPail.snapshot( "/tmp/swa/newDataSnapshot" ); ②
appendNewDataToMasterDataPail(masterPail, snapshotPail); ③
newDataPail.deleteSnapshot(snapshotPail); ④
}

```

- ① /tmp/swa is used as a temporary workspace throughout the batch workflow
- ② take a snapshot of the new data pail
- ③ append data from the snapshot to the master dataset
- ④ after the append, delete only the data that exists in the snapshot

Note that this code also creates a temporary working space at /tmp/swa. Many stages of the workflow will require a space for intermediate data, and it is opportune to initialize this staging area before the first step executes.

The next problem is to structure the new data. Each file within the new data pail may contain data units of all property types and edges. Before this data can be appended into the master dataset, it must first be reorganized to be consistent with the structure used for the master dataset pail. This process of reorganizing a pail to have a new structure is called *shredding*.

To shred a pail, you must be able to write to and read from pails via JCascalog queries. Recall that in JCascalog, the abstraction for sinking and sourcing data is called a *tap*. The dfs-datastores project³ provides a PailTap implementation so that pails can be used as input and output for JCascalog queries. When used as a source, a PailTap inspects the pail and automatically deserializes the records it contains. The following code creates a tap to read all the data from a pail as a source for a query:

Footnote 3 <https://github.com/nathanmarz/dfs-datastores>

```

public static void pailTapUsage() {
    Tap source = new PailTap( "/tmp/swa/snapshot" ); ①
    new Subquery( "?data" ).predicate( source, "_", "?data" ); ②
}

```

- ① the snapshot is a SuperWebAnalytics.com pail, so the tap will emit Thrift Data objects
- ② the tap emits the file containing the record and the record itself; the filename is not needed in the workflow so can be ignored

A PailTap also supports reading a subset of the data within the pail. For pails using the SplitDataPailStructure from Chapter 3, you can construct a PailTap that

reads only the equiv edges contained in the pail.

```
public static void pailTapSubset() {
    PailTapOptions opts = new PailTapOptions(); ①
    opts.attrs = new List[] { ②
        new ArrayList<String>() {{
            add(" " + DataUnit._Fields.EQUIV.getThriftFieldId()); ③
        }}
    };
    Tap equivils = new PailTap("/tmp/swa/snapshot", opts); ④
}
```

- ① relays custom configurations to the PailTap
- ② the attributes are an array of lists; each list contains the directory path of a subfolder to be used as input
- ③ create a list containing the relative path of the equiv edges
- ④ create the tap with the specified options

This functionality is needed quite often so should be wrapped into a function for future use:

```
public static PailTap attributeTap(String path,
                                    final DataUnit._Fields... fields) {
    PailTapOptions opts = new PailTapOptions();
    opts.attrs = new List[] {
        new ArrayList<String>() {
            for(DataUnit._Fields field: fields) {
                add(" " + field.getThriftFieldId()); ①
            }
        }
    };
    return new PailTap(path, opts);
}
```

- ① multiple subfolders can be specified as input to the tap

When sinking data from queries into brand new pails, you must declare the type of records you will be writing to the PailTap. You do this by setting the *spec* option to contain the appropriate PailStructure. To create a pail that shreds the data units by attribute, you can use the SplitDataPailStructure from Chapter 3.

```
public static PailTap splitDataTap(String path) {
```

```

PailTapOptions opts = new PailTapOptions();
opts.spec =
    new PailSpec((PailStructure) new SplitDataPailStructure());
return new PailTap(path, opts);
}

```

Now you can use PailTap and JCascalog to implement the shredding part of the workflow. Your first attempt to shred might look something like this:

```

// do not use!
public static void badShred() {
    PailTap source = new PailTap("/tmp/swa/snapshot");
    PailTap sink = splitDataTap("/tmp/swa/shredded");

    Api.execute(sink,
                new Subquery("?data").predicate(source, "_", "?data"));
}

```

Logically this query is correct. However, when you attempt to run this query on a massive input dataset on HDFS, you will encounter strange issues like NameNode errors and file handle limits. What you have run into are limitations within Hadoop itself. The problem with the query is that it creates countless small files, and as discussed in Chapter 4, Hadoop does not play well with an enormous number of small files.

To understand why this happens, you have to understand how the query executes. This query does not involve aggregations or joins, so it can execute as a map-only job and skip the reduce stage. Normally this is highly desirable, as the reduce step is the far more expensive step. However, suppose your schema has 100 different edge and property types. A single map task could therefore create 100 separate output files, one for each record type. If processing your input data requires 10,000 mappers (roughly 1.5TB of data stored in 128MB blocks), then the output will consist of approximately one million files - too much for Hadoop to handle.

You can solve this problem by artificially introducing a reduce step into the computation. Unlike mappers, you can explicitly control the number of reducers via the job configuration. If you ran our hypothetical job on 1.5TB of data with 100 reducers, you would generate a much more manageable 10,000 files. The following code includes an "identity aggregator" to force the query to perform a reduce step.

```

public static Pail shred() throws IOException {
    PailTap source = new PailTap("/tmp/swa/snapshot");
    PailTap sink = splitDataTap("/tmp/swa/shredded");

    Subquery reduced = new Subquery("?rand", "?data")
        .predicate(source, "_", "?data-in")
        .predicate(new RandLong())
        .out("?rand") ①
        .predicate(new IdentityBuffer(), "?data-in") ②
        .out("?data"); ③

    Api.execute(sink,
        new Subquery("?data").predicate(reduced, "_", "?data"));

    Pail shreddedPail = new Pail("/tmp/swa/shredded");
    shreddedPail.consolidate(); ④
    return shreddedPail;
}

```

- ① assign a random number to each record
- ② use an identity aggregator to get each data record to the reducer
- ③ after the reduce stage, project out the random number
- ④ consolidate the shredded pail to further reduce the number of files

Now that the data is shredded and the number of files has been minimized, you can finally append it into the master dataset pail:

```

public static void appendNewData(Pail masterPail,
                                 Pail snapshotPail) throws IOException {
    Pail shreddedPail = shred();
    masterPail.absorb(shreddedPail);
}

```

Once the new data is ingested into the master dataset, you can begin normalizing the data.

6.5 URL normalization

The next step of the workflow is to normalize all of the URLs in the master dataset. The query to accomplish this task requires a custom function that implements the normalization logic. For demonstration purposes, a rudimentary implementation is provided below:

```

public static class NormalizeURL extends CascalogFunction { ①

    public void operate(FlowProcess process, FunctionCall call) {
        Data data = ((Data) call.getArguments().getObject(0)).deepCopy(); ②
        DataUnit du = data.get_dataunit();

        if(du.getSetField() == DataUnit._Fields.PAGE_VIEW) { ③
            normalize(du.get_page_view().get_page());
        }
        call.getOutputCollector().add(new Tuple(data));
    }

    private void normalize(PageID page) {
        if(page.getSetField() == PageID._Fields.URL) {
            String urlStr = page.get_url();
            try {
                URL url = new URL(urlStr);
                page.set_url(url.getProtocol() + "://" + url.getHost()
                    + url.getPath()); ④
            } catch(MalformedURLException e) {}
        }
    }
}

```

- ① the function takes a Data object and emits a normalized Data object
- ② the input object is cloned so it can be safely modified
- ③ for the supported batch views, only pageview edges need to be normalized
- ④ pageviews are normalized by extracting standard components from the URL

You can then use this function to create a normalized version of the master dataset as shown in the following query:

```

public static void normalizeURLs() {
    Tap masterDataset = new PailTap("/data/master");
    Tap outTap = splitDataTap("/tmp/swa/normalized_urls");
    Api.execute(outTap,
        new Subquery("?normalized")
            .predicate(masterDataset, "_", "?raw")
            .predicate(new NormalizeURL(), "?raw")
            .out("?normalized"));
}

```

6.6 User identifier normalization

The next step is to select a single user identifier for each person. This is the most sophisticated portion of the workflow as it involves a fully distributed iterative graph algorithm. Despite its complexity, it will only require approximately one hundred lines of code to accomplish this task.

User IDs are marked as belonging to the same person via equiv edges. If you were to visualize these edges from a dataset, you would see numerous independent subgraphs as shown in Figure 6.6:

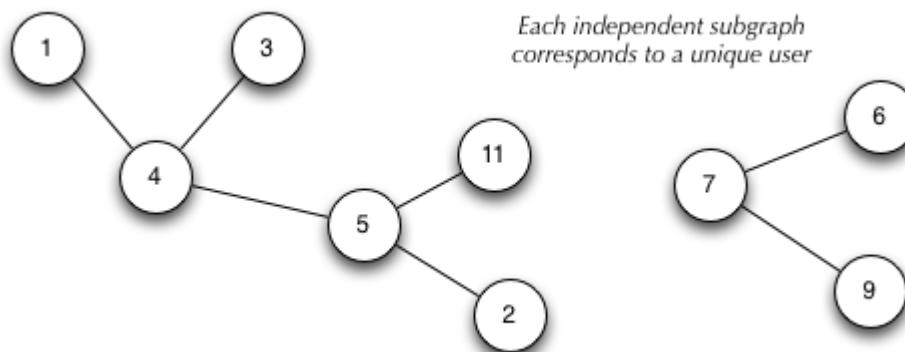


Figure 6.6 Example equiv graph

Each subgraph represents a unique user. For each person, you need to select a single identifier and create a mapping from the other IDs to this identifier, as shown in Figure 6.7:

User ID	Mapped User ID
2	1
3	1
4	1
5	1
11	1
7	6
9	6

1 If a user has multiple IDs, map the extraneous IDs to the minimum value for the user

2 Only need to store remapped IDs, so selected IDs (1, 6) are omitted

Figure 6.7 Mapping from user IDs to a single identifier for each set

You will accomplish this by transforming the original equiv graph to the form depicted in Figure 6.7. For the example, this transformation is shown in Figure

xref:transform2, where every user ID associated with a person maps to a single ID uniquely chosen for that person.

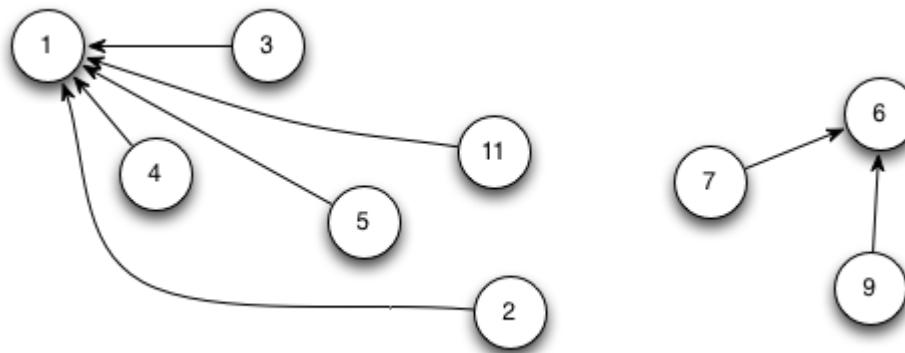


Figure 6.8 Original equiv graph transformed so that all nodes in a set point to a single node

This idea must be translated into a concrete algorithm that runs scalably using MapReduce. All of our previous MapReduce computations involved executing a single query to generate the desired output. For this algorithm, however, it is impossible to get the desired results in a single step. Instead, you can take an iterative approach where each query modifies the graph to a state closer to the desired structure shown in Figure 6.8. Once you have defined the iterative step, you execute it repeatedly until no further progress is made. This is known as reaching a *fixed point* where the resulting output is the same as the input. When this point is reached, the graph has attained the desired state.

At each iteration, the algorithm will examine the neighbors for each node in the graph. It will determine the smallest ID among the connected nodes, then ensure each edge points to the node with this minimum value. This process is illustrated for a single node in Figure 6.9:

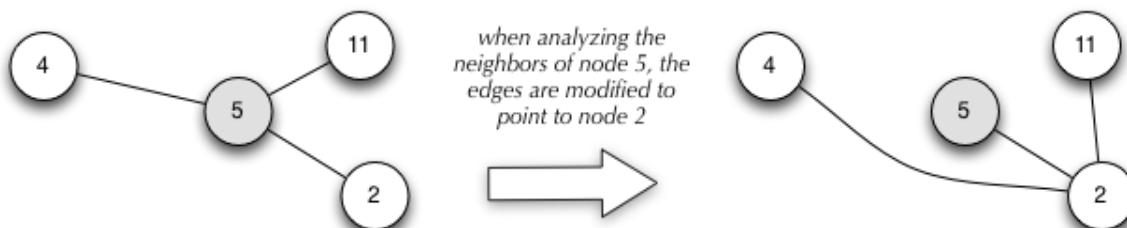


Figure 6.9 Example of modifying the edges around a single node move in a single iteration

You can see how this algorithm works on the equiv graph from Figure 6.6.

Figure xref:equivsteps shows the transformations of the graph until it reaches the fixed point.

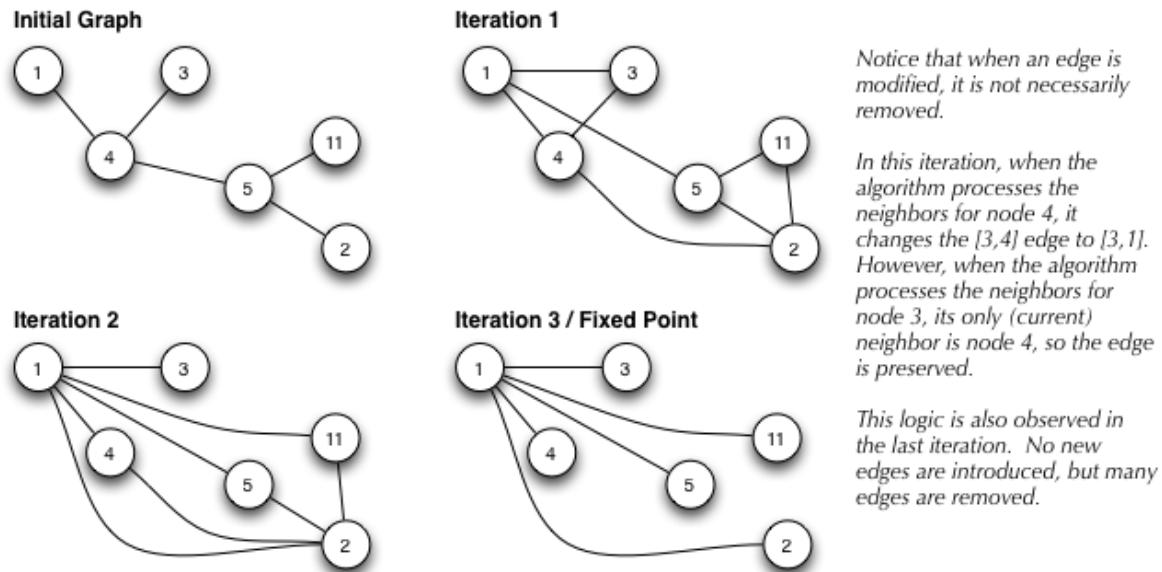


Figure 6.10 Iterating the algorithm until a fixed point is reached

SIDE BAR Ordering Thrift data types

You may recall that instead of integers, PersonIDs are actually modeled as Thrift unions:

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

Fortunately, Thrift provides a natural ordering for all Thrift structures which can be used to determine the "minimum" identifier. Integers are useful for explanatory purposes, but the following code will appropriately handle the Thrift data structures.

You can now begin implementing the iterative algorithm. The output of each iteration will be stored in a new folder on the distributed filesystem, using the template "/tmp/swa/eqv{iteration number}" for the path. These outputs will consist of 2-tuples of user IDs. The following code creates the initial dataset by transforming the equiv edges objects stored in the master dataset:

```

public static class EdgifyEquiv extends CascalogFunction { ①
    public void operate(FlowProcess process, FunctionCall call) {
        Data data = (Data) call getArguments().get Object(0);
        EquivEdge equiv = data.get_dataunit().get_equiv();
        call.getOutputCollector()
            .add(new Tuple(equiv.get_id1(), equiv.get_id2()));
    }
}

public static void initializeUserIdNormalization() throws IOException {
    Tap equivils = attributeTap("/tmp/swa/normalized_urls",
                                DataUnit._Fields.EQUIV);
    Api.execute(Api.hfsSeqfile("/tmp/swa/equivils0"),
                new Subquery("?node1", "?node2")
                    .predicate(equivils, "_", "?data")
                    .predicate(new EdgifyEquiv(), "?node1", "?node2"));
}

```

- ① a custom function to extract the identifiers from the equiv edges
- ② the initialized data is stored as iteration zero

The first requirement of the iterative step is to determine the immediate neighbors for each node. You could potentially group the tuple stream using the first element of the tuple, but this would exclude edges where the given node is stored in the last element. The following custom function emits edges in both orientations to account for this possibility:

```

public static class BidirectionalEdge extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        Object node1 = call getArguments().get Object(0);
        Object node2 = call getArguments().get Object(1);
        if (!node1.equals(node2)) { ①
            call.getOutputCollector().add(new Tuple(node1, node2));
            call.getOutputCollector().add(new Tuple(node2, node1));
        }
    }
}

```

- ① filter any edges that connect a node to itself
- ② emit edges using both [a, b] and [b, a] orderings

This dual representation ensures that when the grouping occurs, all edges containing the node are collected regardless of whether the node is located in the first or last position. Once grouped, you need a custom aggregator to implement the algorithm logic and denote new edges:

```

public static class IterateEdges extends CascalogBuffer {
    public void operate(FlowProcess process, BufferCall call) {
        PersonID grouped = (PersonID) call.getGroup().getObject(0); ①
        TreeSet<PersonID> allIds = new TreeSet<PersonID>(); ②
        allIds.add(grouped);

        Iterator<TupleEntry> it = call getArgumentsIterator();
        while(it.hasNext()) {
            allIds.add((PersonID) it.next().getObject(0));
        }

        Iterator<PersonID> allIdsIt = allIds.iterator();
        PersonID smallest = allIdsIt.next(); ③
        boolean progress =
            allIds.size() > 2 && !grouped.equals(smallest); ④

        while(allIdsIt.hasNext()) {
            PersonID id = allIdsIt.next();
            call.getOutputCollector().add(new Tuple(smallest, id, progress)); ⑤
        }
    }
}

```

- ① get the node used for grouping tuples
- ② the TreeSet contains the node and all of its neighbors
- ③ a TreeSet is sorted, so the first element is the smallest
- ④ if the grouped node is not the smallest and is connected to at least two other nodes, then a new edge will be created
- ⑤ emit the edges generated during this iteration

When the grouped node is the smallest amongst its neighbors, then the emitted edges are unchanged. Otherwise, if the node has more than one neighbor, then some edges will be modified to point to the smallest identifier. With these two functions, the query for the iterative step is fairly simple:

```

public static Subquery iterationQuery(Tap source) {
    Subquery iterate = new Subquery("?b1", "?node1", "?node2", "?is-new")
        .predicate(source, "?n1", "?n2") ①
        .predicate(new BidirectionalEdge(), "?n1", "?n2")
        .out("?b1", "?b2")
        .predicate(new IterateEdges(), "?b2") ②
        .out("node1", "node2", "is-new");

    iterate = Api.selectFields(iterate,
        new Fields("node1", "node2", "is-new")); ③
}

```

```

    return (Subquery) iterate;
}

```

- ➊ the source tap emits the tuples of user IDs from the previous iteration
- ➋ from the declared output of the query, JCascalog groups tuples using ?b1
- ➌ remove the grouping identifier as it is no longer needed

This subquery addresses the logic of the algorithm; completing the iterative step requires adding the appropriate source and sink taps and executing the query:

```

public static Tap userIdNormalizationIteration(int i) {
    Tap source = (Tap) Api.hfsSeqfile("/tmp/swa/equivs" + (i - 1));
    Tap sink = (Tap) Api.hfsSeqfile("/tmp/swa/equivs" + i); ➊
    Tap progressSink = (Tap) Api.hfsSeqfile("/tmp/swa/equivs" + "-new"); ➋

    Subquery iteration = iterationQuery(source);
    Subquery newEdgeSet = new Subquery("?node1", "?node2")
        .predicate(iteration, "?node1", "?node2", "_")
        .predicate(Option.DISTINCT, true); ➌
    Subquery progressEdges = new Subquery(?node1", "?node2")
        .predicate(iteration, "?node1", "?node2", true); ➍

    Api.execute(Arrays.asList(sink, progressSink),
               Arrays.asList(newEdgeSet, progressEdges)); ➎

    return progressEdgesSink;
}

```

- ➊ all edges are emitted to the output of the iterative step
- ➋ new edges are additionally stored in a separate path
- ➌ avoid writing duplicate edges to the sink
- ➍ only the new edges in this iteration are written to the progress sink
- ➎ execute both newEdgeSet and progressEdges queries in parallel

In addition to storing all the edges as input for the next iteration, the iterative step also stores the new edges in a separate folder. This provides an easy means to determine if the current iteration generated any new edges; if not, the fixed point has been reached. The following code implements the iterative loop and this termination logic.

```

public static int userIdNormalizationIterationLoop() {
    int iter = 1; ➊

```

```

while(true) {
    Tap progressEdgesSink = userIdNormalizationIteration(iter);
    FlowProcess flowProcess = new HadoopFlowProcess(new JobConf());
    if(! flowProcess.openTapForRead(progressEdgesSink).hasNext()) { ②
        return iter; ③
    }
    iter++; ④
}

```

- ① tracks the current iteration
- ② terminate if no new edges were generated during this iteration
- ③ the last iteration determines the path of the final output
- ④ if new edges were generated, increase the counter and loop

The last requirement to complete this workflow step is to change the PersonIDs in the pageview data to use the selected user identifiers. This transformation can be achieved by performing a join of the pageview data with the final iteration of the equiv graph. A couple of custom functions are needed before you can execute this join. First, you must unravel the Thrift pageview objects to extract the necessary fields:

```

public static class ExtractPageViewFields extends CascalogFunction { ①
    public void operate(FlowProcess process, FunctionCall call) {
        Data data = (Data) call.getArguments().getObject(0);
        PageViewEdge pageview = data.get_dataunit().get_page_view();
        if(pageview.get_page().getSetField() == PageID._Fields.URL) {
            call.getOutputCollector().add(
                new Tuple(pageview.get_page().get_url(), ②
                          pageview.get_person(),
                          data.get_pedigree().get_true_as_of_secs())); ③
        }
    }
}

```

- ① extract the relevant parameters from PageViewEdge objects
- ② emit the URL, PersonID and timestamp of the page view
- ③ although the timestamp is not immediately required, this function will be reused in other parts of the workflow

The second required function takes a pageview Data object and the new PersonID, and it returns a new pageview Data object with an updated PersonID.

```

public static class MakeNormalizedPageview extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        PersonID newId = (PersonID) call.getArguments().getObject(0); ①
        Data data = ((Data) call.getArguments().getObject(1)).deepCopy(); ②
        if(newId != null) {
            data.get_dataunit().get_page_view().set_person(newId);
        }
        call.getOutputCollector().add(new Tuple(data)); ③
    }
}

```

- ① newId may be null if the user ID in the pageview is not part of the equiv graph
- ② clone the Data object so it can be modified safely
- ③ emit the potentially modified pageview Data object

Note that it is perfectly valid for a PersonID to exist in the pageview data but not in any equiv edges. This situation occurs when only one identifier was ever recorded for the user. In these cases, the PersonID is not be affected.

With these two functions, you can now perform the join to modify the pageviews to use the normalized user IDs. Note that an outer join is required for the pageviews with user IDs that are not contained in the equiv graph.

```

public static void modifyPageViews(int iter) throws IOException {
    Tap pageviews = attributeTap("/tmp/swa/normalized_urls",
                                 DataUnit._Fields.PAGE_VIEW);
    Tap newIds = (Tap) Api.hfsSeqfile("/tmp/swa/equivs" + iter); ①
    Tap result = splitDataTap("/tmp/swa/normalized_pageview_users");

    Api.execute(result,
                new Subquery("normalized-pageview")
                    .predicate(newIds, "!!newId", "?person") ②
                    .predicate(pageviews, "_", "?data")
                    .predicate(new ExtractPageViewFields("?data")
                        .out("_", "?person", "_") ③
                        .predicate(new MakeNormalizedPageview(), "!!newId", "?data")
                        .out("normalized-pageview")); ④
}

```

- ① use the final output from the iterative loop
- ② perform a join on ?person; the prefix of !!newId indicates this is an outer join
- ③ join on the user identifier in the page view
- ④ create and emit new normalized pageview

The last task is to define a wrapper function to execute the distinct phases of

this workflow step:

```
public static void normalizeUserIds() throws IOException {
    initializeUserIdNormalization();
    int numIterations = userIdNormalizationIterationLoop();
    modifyPageViews(numIterations);
}
```

That concludes the user ID normalization portion of the workflow. This is a great example of the benefit of specifying the MapReduce computations using a library of your general purpose programming language. A significant part of the logic, such as the iteration and fixed point checking, were written as normal Java code.

6.7 De-duplicate pageviews

The final preparation step prior to computing the batch views is deduplicating the pageview events. This is a trivial query to write:

```
public static void deduplicatePageviews() {
    Tap source = attributeTap("/tmp/swa/normalized_pageview_users",
        DataUnit._Fields.PAGE_VIEW); ①
    Tap outTap = splitDataTap("/tmp/swa/unique_pageviews");

    Api.execute(outTap,
        new Subquery("?data")
            .predicate(source, "?data")
②            .predicate(Option.DISTINCT, true));
}
```

- ① restrict source tap to only read pageviews from the pail
- ② the distinct predicate removes all duplicate pageview objects

6.8 Computing batch views

The data is now ready to compute the batch views. This computation step will generate flat files; in the next chapter, you will learn how to index the batch views so they can be queried in a random access manner.

6.8.1 Pageviews over time

As outlined earlier, the pageviews over time batch view should aggregate the pageviews for each URL at hourly, daily, 7 day and 28 day granularities. The approach you will take is to first aggregate the pageviews at an hourly granularity. This will reduce the size of the data by many orders of magnitude. Afterwards, you will roll up the hourly values to obtain the counts for the larger buckets. The latter operations will be much faster due to the smaller size of the input.

To roll up the pageviews to an hourly granularity, you will first need a function that determines the hour bucket for a timestamp:

```
public static class ToHourBucket extends CascalogFunction {
    private static final int HOUR_IN_SECS = 60 * 60;

    public void operate(FlowProcess process, FunctionCall call) {
        int timestamp = call getArguments().getInteger(0);
        int hourBucket = timestamp / HOUR_IN_SECS;
        call.getOutputCollector().add(new Tuple(hourBucket));
    }
}
```

With this function, it is a very standard JCascalog query to determine the hourly counts:

```
public static Subquery hourlyRollup() {
    Tap source = new PailTap("/tmp/swa/unique_pageviews");
    return new Subquery("?url", "?hour-bucket", "?count")
        .predicate(source, "?pageview")
        .predicate(new ExtractPageViewFields(), "?pageview") ①
            .out("?url", "_", "?timestamp")
        .predicate(new ToHourBucket(), "?timestamp")
            .out("?hour-bucket")
        .predicate(new Count(), "?count"); ②
}
```

- ① reuse the pageview extraction code from earlier
- ② group by ?url and ?hour-bucket

The next subquery rolls up the hourly values into all the granularities needed for the batch view. This requires another custom function to generate all the granularities for a given hour bucket.

```

public static class EmitGranularities extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        int hourBucket = call.getArguments().getInteger(0);
        int dayBucket = hourBucket / 24;
        int weekBucket = dayBucket / 7;
        int monthBucket = dayBucket / 28;

        call.getOutputCollector().add(new Tuple("h", hourBucket)); ①
        call.getOutputCollector().add(new Tuple("d", dayBucket));
        call.getOutputCollector().add(new Tuple("w", weekBucket));
        call.getOutputCollector().add(new Tuple("m", monthBucket));
    }
}

```

- ① the function emits four 2-tuples for each input
- ② the first element is either h, d, w or m to indicate the hour, day, week or month granularity; the second element is the numerical value of the time bucket

With this function, computing the rollups for all the granularities is just a simple sum:

```

public static Subquery pageviewBatchView() {
    Subquery pageviews =
        new Subquery("?url", "?granularity", "?bucket", "?total-pageviews")
            .predicate(hourlyRollup(), "?url", "?hour-bucket", "?count") ①
            .predicate(new EmitGranularities(), "?hour-bucket")
                .out("?granularity", "?bucket") ②
            .predicate(new Sum(), "?count").out("?total-pageviews"); ③
    return pageviews;
}

```

- ① execute the hourly counts subquery
- ② emit the buckets for all granularities
- ③ sum the pageview counts by url, granularity and bucket

6.8.2 Unique visitors over time

The batch view for unique visitors over time contains a HyperLogLog set for every time granularity tracked for every URL. It is essentially the same computation as done to compute pageviews over time, except instead of aggregating counts you aggregate HyperLogLog sets.

You will need two new custom operations to do this query. The first is an aggregator that constructs a HyperLogLog set from a sequence of user identifiers:

```

public static class ConstructHyperLogLog extends CascalogBuffer { ①
    public void operate(FlowProcess process, BufferCall call) {
        HyperLogLog hll = new HyperLogLog(8192); ②
        Iterator<TupleEntry> it = call.getArgumentsIterator();
        while(it.hasNext()) {
            TupleEntry tuple = it.next();
            hll.offer(tuple.getObject(0)); ③
        }
        try {
            call.getOutputCollector().add(new Tuple(hll.getBytes())); ④
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}

```

- ① the function generates a HyperLogLog set for a set of visitors
- ② construct a HyperLogLog set using 1KB of storage
- ③ add all objects to the set
- ④ emit the storage bytes of the HyperLogLog object

The next function is another custom aggregator that combines the HyperLogLog sets for hourly granularities into HyperLogLog sets for coarser intervals:

```

public static class MergeHyperLogLog extends CascalogBuffer {
    public void operate(FlowProcess process, BufferCall call) {
        Iterator<TupleEntry> it = call.getArgumentsIterator();
        HyperLogLog merged = null; ①
        try {
            while(it.hasNext()) {
                TupleEntry tuple = it.next();
                byte[] serialized = (byte[]) tuple.getObject(0);
                HyperLogLog hll = HyperLogLog.Builder.build(serialized); ②
                if(merged == null) ③
                    merged = hll;
                else {
                    merged = (HyperLogLog) merged.merge(hll);
                }
            }
            call.getOutputCollector().add(new Tuple(merged.getBytes())); ④
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}

```

- ① create a new HyperLogLog set to contain the merged results
- ② reconstruct a HyperLogLog set from storage bytes
- ③ merge the current set into the results
- ④ emit the storage bytes for merged set

The following listing uses these operations to compute the batch view. Note the similarity to the pageviews over time query.

```
public static void uniquesView() {
    Tap source = new PailTap("/tmp/swa/unique_pageviews");

    Subquery hourlyUniques =
        new Subquery("?url", "?hour-bucket", "?hyper-log-log")
            .predicate(source, "?pageview")
            .predicate(new ExtractPageViewFields(), "?pageview")
                .out("?url", "?user", "?timestamp")
            .predicate(new ToHourBucket(), "?timestamp")
                .out("?hour-bucket")
            .predicate(new ConstructHyperLogLog(), "?user")
                .out("?hyper-log-log"); ①

    Subquery uniques =
        new Subquery("?url", "?granularity", "?bucket", "?aggregate-hll")
            .predicate(hourlyUniques, "?url", "?hour-bucket", "?hourly-hll")
            .predicate(new EmitGranularities(), "?hour-bucket")
                .out("?granularity", "?bucket")
            .predicate(new MergeHyperLogLog(), "?hourly-hll")
                .out("?aggregate-hll");
    return uniques;
}
```

- ① the first subquery determines hourly HyperLogLog sets for each URL
- ② the second subquery determines the HyperLogLog sets for all granularities

It is possible to create a function that abstracts away the parts common to the pageviews query and the unique visitors query. We will leave that as an exercise for the reader.

SIDE BAR**Further optimizing the HyperLogLog batch view**

The implementation we have shown uses the same size for every HyperLogLog set: 1000 bytes. The HyperLogLog set needs to be that large in order to get a reasonably accurate answer for URLs which may receive millions or hundreds of millions of visits. However, most websites using SuperWebAnalytics.com won't get nearly that many pageviews, so it is wasteful to use such a large HyperLogLog set size for them.

For further optimization, you could look at the total pageview count for URLs on that domain and tune the size of the HyperLogLog set accordingly. Using this approach can vastly decrease the space needed for the batch view, at the cost of adding some complexity to the view generation code.

6.8.3 Bounce rate analysis

The final batch view computes the bounce rate for each URL. As outlined in the beginning of the chapter, you will compute two values for each domain: the total number of visits and the number of bounced visits.

The key part of this query is tracing each visit a person made as they browsed the internet. An easy way to accomplish this is to examine all the pageviews a person made for a particular domain, sorted by chronological order. You can then use the time difference between successive pageviews to determine whether they belong to the same visit. If a visit contains only one pageview, it counts as a bounced visit.

To do this in a JCascalog query, you need two custom operations. The first function extracts a domain from a URL:

```
public static class ExtractDomain extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        String urlStr = call.getArguments().getString(0);
        try {
            URL url = new URL(urlStr); ①
            call.getOutputCollector().add(new Tuple(url.getAuthority()));
        } catch(MalformedURLException e) {}
    }
}
```

- ① use Java native libraries to extract the domain

The next function is a custom aggregator that iterates through a sorted list of pageviews and counts the number of visits and bounces:

```

public static class AnalyzeVisits extends CascalogBuffer {
    private static final int VISIT_LENGTH_SECS = 60 * 30; ①

    public void operate(FlowProcess process, BufferCall call) {
        Iterator<TupleEntry> it = call.getArgumentsIterator(); ②
        int bounces = 0;
        int visits = 0;
        Integer lastTime = null; ③
        int numInCurrVisit = 0;
        while(it.hasNext()) {
            TupleEntry tuple = it.next();
            int timeSecs = tuple.getInteger(0);
            if(lastTime == null || (timeSecs - lastTime) > VISIT_LENGTH_SECS) {
                visits++; ④
                if(numInCurrVisit == 1) { ⑤
                    bounces++;
                }
                numInCurrVisit = 0;
            }
            numInCurrVisit++;
        }
        if(numInCurrVisit==1) { ⑥
            bounces++;
        }
        call.getOutputCollector().add(new Tuple(visits, bounces)); ⑦
    }
}

```

- ① two successive pageviews belong to the same visit if they are separated by less than 30 minutes
- ② assumes that the pageviews are sorted chronologically
- ③ tracks the time of the previous pageview
- ④ register the beginning of a new visit
- ⑤ determine if previous visit was a bounce
- ⑥ determine whether last pageview was a bounce
- ⑦ emit visit and bounce counts

By combining these functions you can then compute the number of visits and bounces for each user on each domain. Finally, to compute the number of visits and bounces in aggregate, you simply sum together the user visit information:

```
public static Subquery bouncesView() {
```

```

Tap source = new PailTap( "/tmp/swa/unique_pageviews" );

Subquery userVisits =
    new Subquery( "?domain", "?user", "?num-user-visits",
                  "?num-user-bounces")
        .predicate(source, "?pageview")
        .predicate(new ExtractPageViewFields(), "?pageview")
            .out( "?url", "?user", "?timestamp" )
        .predicate(new ExtractDomain(), "?url")
            .out( "?domain" )
        .predicate(Option.SORT, "?timestamp") ①
        .predicate(new AnalyzeVisits(), "?timestamp")
            .out( "?num-user-visits", "?num-user-bounces" ); ②

Subquery bounces =
    new Subquery( "?domain", "?num-visits", "?num-bounces")
        .predicate(userVisits, "?domain", "_",
                  "?num-user-visits", "?num-user-bounces" )
        .predicate(new Sum(), "?num-user-visits")
            .out( "?num-visits" )
        .predicate(new Sum(), "?num-user-bounces")
            .out( "?num-bounces" ); ③

return bounces;
}

```

- ① sort pageviews chronologically to analyze visits
- ② bounces and visits are determined per user
- ③ sum bounces and visits for all users to calculate the batch view

Relax. Take a deep breath. After much time and effort, you have successfully completed the recomputation-based layer for SuperWebAnalytics.com!

6.9 Conclusion

The batch layer for SuperWebAnalytics.com is just a few hundred lines of code, yet the business logic involved is quite sophisticated. The various abstractions fit together well – there was a fairly direct mapping from what we wanted to accomplish at each step and how we accomplished it. Here and there, hairy details arose due to the nature of the toolset – notably Hadoop's small files issue – but these were not difficult to overcome.

As we have indicated a few times, what you developed in this chapter is a recomputation-based workflow where the batch views are always recomputed from scratch. There is a large class of problems for which you can incrementalize the batch layer and make it much more resource-efficient: you will see how to do this in a later chapter.

You should now have a good feel for the flexibility of the batch layer. It is

really easy to extend the batch layer to compute new views: each stage of the workflow is free to run an arbitrary function on all the data.

We next proceed to the serving layer so that the batch views can be quickly read in a random-access manner.



Serving Layer

This chapter covers:

- Tailoring batch views to the queries they serve
- A new answer to the data normalization versus denormalization debate
- Advantages of batch-writable, random-read, no random-write databases
- ElephantDB as an example of a serving layer database

At this point you have learned how to precompute arbitrary views of any dataset by making use of batch computation. For the views to be useful, you must be able to access their contents with low latency. As shown in Figure 7.1, this is the role of the serving layer. The serving layer indexes the views and provides interfaces so that the precomputed data can be quickly queried.

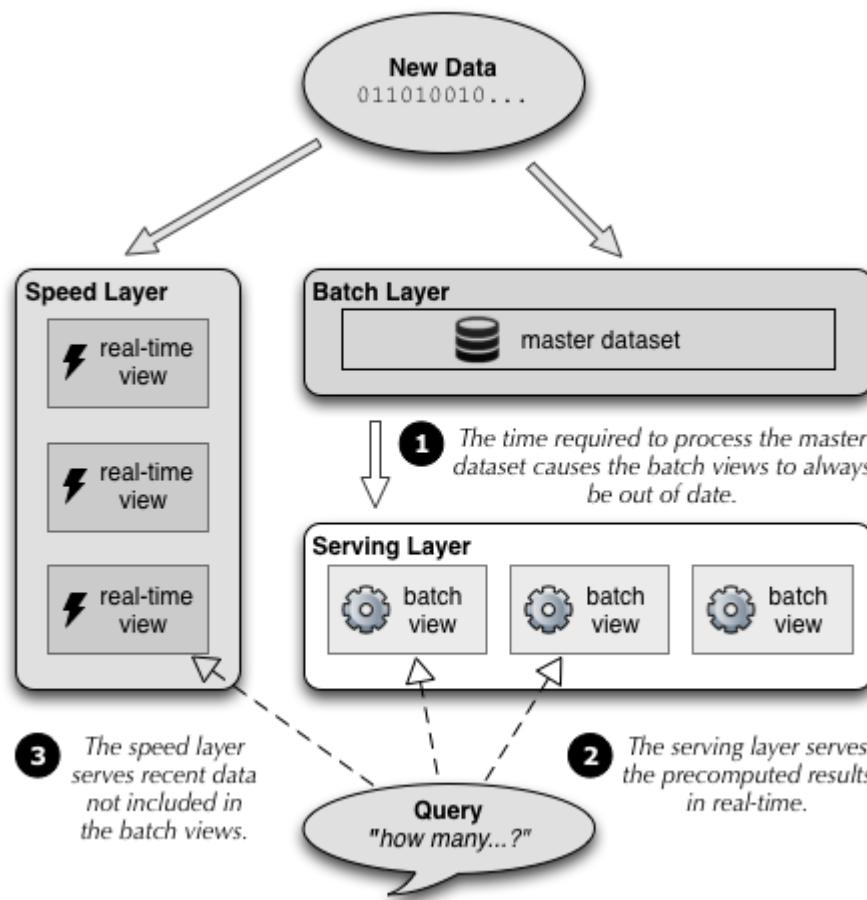


Figure 7.1 In the Lambda Architecture, the serving layer provides low latency access to the results of calculations performed on the master dataset. The serving layer views are slightly out-of-date due to the time required for batch computation.

The serving layer is the last component of the batch section of the Lambda Architecture. It is tightly tied to the batch layer since the batch layer is responsible for continually updating the server layer views. These views will always be out of date due to the high latency nature of batch computation. This is not a concern since the speed layer will be responsible for any data not yet available in the serving layer.

Unfortunately, the serving layer is an area where the tooling lags behind the theory. It wouldn't be hard to build a general purpose serving layer implementation – in fact, it would be significantly easier than building any of the currently existing NoSQL databases. We will present the full theory behind creating a simple, scalable, fault-tolerant and general purpose serving layer, then we will use the best tooling available to demonstrate the underlying concepts.

While investigating the serving layer you will learn

- indexing strategies to minimize latency, resource usage and variance
- the requirements for the serving layer in the Lambda Architecture
- how the serving layer solves the long-debated normalization versus denormalization problem
- a simple architecture for a serving layer database
- a practical implementation of the serving layer using ElephantDB

We begin by examining the key issues you face when structuring a serving layer view.

7.1 Performance metrics for the serving layer

As with the batch layer, the serving layer is distributed among many machines for scalability. The indexes of the serving layer are created, loaded and served in a fully distributed manner. When designing these indexes, you must consider two main performance metrics: throughput and latency. In this context, *latency* is the time required to answer a single query, whereas *throughput* is the number of queries that can be served within a given amount of time. The relationship between the structure of the serving layer indexes and these metrics are best explained via an example.

We briefly return to our long-running SuperWebAnalytics.com example - specifically the pageviews over time query. The objective is to serve the number of pageviews for each hour given a specific URL and a particular range of hours. To further simplify the discussion, suppose the pageview counts are only produced using an hourly granularity. The resulting view would look similar to Figure 7.2.

URL	Bucket	Pageviews
foo.com/blog/1	0	10
foo.com/blog/1	1	21
foo.com/blog/1	2	7
foo.com/blog/1	3	38
foo.com/blog/1	4	29
bar.com/post/a	0	178
bar.com/post/a	1	91
bar.com/post/a	2	568

Figure 7.2 The pageviews over time batch view with hourly granularity.

A straightforward way to index this view would use a key/value strategy with

[URL, hour] pairs as keys and pageviews as values. The index would be partitioned using the key, so pageview counts for the same URL would reside on different partitions. Different partitions would exist on separate servers, so retrieving a range of hours for a single URL would involve fetching values from multiple servers in your serving layer.

While this design works in principle, it faces serious issues with both latency and throughput. To start, the latency would be consistently high. Since the values for a particular URL are spread throughout your cluster, you will need to query numerous servers to get the pageview counts for a large range of hours. The key observation is that the response times of servers vary. For instance, one server may be slightly more loaded than the others; another may be performing garbage collection at the time. Even if you parallelize the fetch requests, the overall query response time is limited by the speed of the slowest server.

To illustrate this point, suppose a query requires fetching data from three servers. A representative sample of the distribution of response times is shown in Figure 7.3.

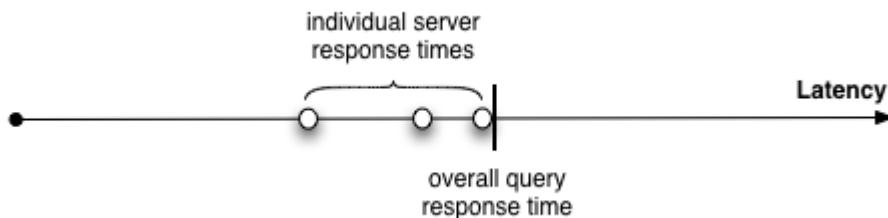


Figure 7.3 When distributing a task over multiple servers, the overall latency is determined by the slowest response time.

For comparison, suppose the query hits 20 servers. A typical distribution of latencies would look like Figure 7.4.

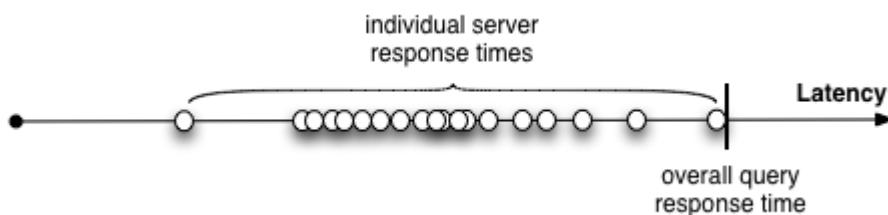


Figure 7.4 If you increase the number of servers involved in a distributed task, you also increase the likelihood that at least one will respond slowly.

In general, the more servers a query touches, the higher overall latency of the

query. This is due to the simple fact that involving more servers increases the likelihood that at least one will respond slowly. Consequently, the variance of server response times turns the worst case performance of one server into the common case performance of queries. This is a serious problem for achieving good latency for the pageviews over time query.

Another problem with this key/value strategy is poor throughput, particularly if your servers use disks and not solid state drives. Retrieving a value for a single key requires a disk seek, and a single query may fetch values for dozens or more keys. Disk seeks are expensive operations for traditional hard drives. Since there are a finite number of disks in your cluster, there is a hard limit to the number of disk seeks that can be achieved per second. Suppose that on average a query fetches 20 keys per query, the cluster has 100 disks, and each disk can perform 500 seeks per second. In this case, your cluster can only serve 2500 queries per second - a surprisingly small amount given the number of disks.

But all is not lost - a different indexing strategy has much better latency and throughput characteristics. The idea is to collocate the pageview information for a single URL on the same partition and store it sequentially. Fetching the pageviews will then only require a single seek and scan rather than numerous seeks. Scans are extremely cheap relative to seeks, so this is far more resource efficient. Additionally, only a single server needs to be contacted per query, so you are no longer subject to the variance issues of the previous strategy. The layout of the index for this strategy is shown in Figure 7.5.

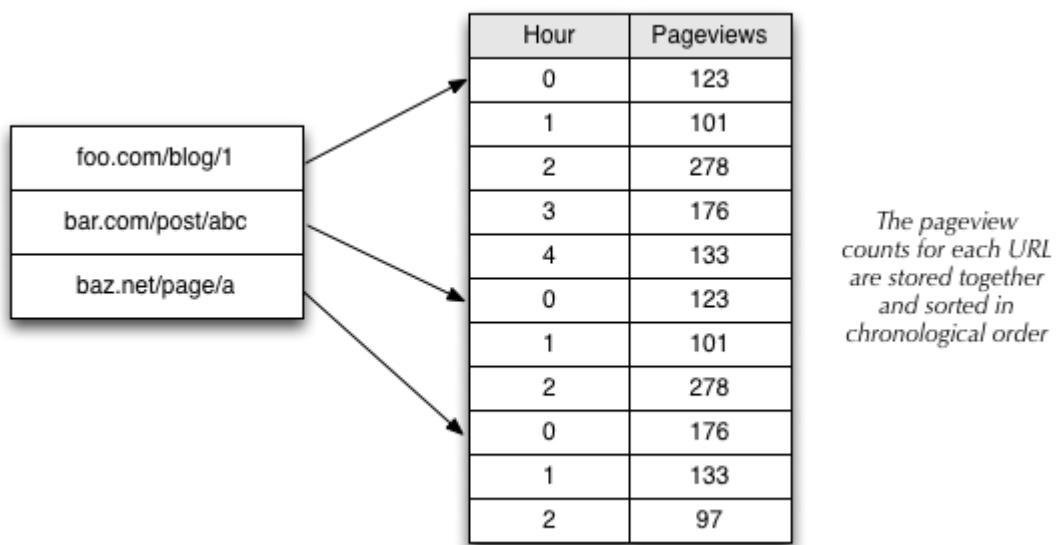


Figure 7.5 A sorted index promotes scans and limits disk seeks to improve both latency and throughput.

These two examples demonstrate that the way you structure your serving layer indexes has drastic effects on the performance of your queries. A vital advantage of the Lambda Architecture is that it allows you to tailor the serving layer for the queries they serve to optimize efficiency.

7.2 The serving layer solution to the normalization / denormalization problem

The serving layer solves one of the long-standing problems in the relational database world: the normalization versus denormalization dilemma. To grasp the solution and its implications, you first need to understand the underlying issues.

The normalization versus denormalization decision is ultimately a choice between unacceptable tradeoffs. In the relational world, you want to store your data fully normalized; this involves defining relationships between independent datasets to minimize redundancy. Unfortunately querying normalized data can be slow, so you may need to store some information redundantly to improve response times. This denormalization process increases performance, but it comes with the huge complexity of keeping the redundant data consistent.

To illustrate this tension, suppose you are storing user location information in relational tables, such as shown in Figure 7.6. Each location has an identifier, and each person uses one of those identifiers to indicate their location. A query to retrieve the location for a specific individual requires a join between the two tables. This is an example of a fully normalized schema as no information is stored redundantly.

User ID	Name	Location ID	Location ID	City	State	Population
1	Sally	3	1	New York	NY	8.2M
2	George	1	2	San Diego	CA	1.3M
3	Bob	3	3	Chicago	IL	2.7M

Figure 7.6 A normalized schema uses multiple independent datasets with little or no redundant data.

Now suppose you observe that retrieving the city and state for a user is an extremely common operation in your application. Joins are expensive, and you decide that you need better performance from this operation. The only means to avoid the join would be to redundantly store the city and state information in the user table. This technique of redundantly storing information to avoid joins is

called denormalization, and the resulting schema in this case would resemble Figure 7.7.

User ID	Name	Location ID	City	State
1	Sally	3	Chicago	IL
2	George	1	New York	NY
3	Bob	3	Chicago	IL

Location ID	City	State	Population	
1	New York	NY	8.2M	
2	San Diego	CA	1.3M	
3	Chicago	IL	2.7M	

Figure 7.7 Denormalized tables store data redundantly to improve query performance.

Denormalization is not an ideal solution - as the application developer, it is your responsibility to ensure all redundant data is consistent. This raises uncomfortable questions such as: What happens if the different copies of a field become inconsistent? What are the semantics of the data in this case? Remember that mistakes are inevitable in long-lived systems, so given enough time inconsistencies will occur.

Fortunately the split between the master dataset and the serving layer in the Lambda Architecture solves the normalization versus denormalization problem. Within the batch layer you can normalize your master dataset to your heart's content. The computation on the batch layer reads the master dataset in bulk, so there is no need to design the schema to optimize for random-access reads. Complementarily, the serving layer is completely tailored to the queries they serve, so you can optimize as needed to attain maximal performance. These optimizations in the serving layer can go far beyond denormalization. In addition to pre-joining data, you can also perform additional aggregation and transformation to further improve efficiency.

As to the question of consistency in the Lambda Architecture, it is absolutely true that information will be redundantly stored between the batch and serving layers. The key distinction is that the serving layer is defined to be a function of the master dataset. If an error introduces inconsistencies, you can easily correct them by recomputing the serving layer from scratch.

7.3 Requirements of serving layer database

The Lambda Architecture dictates a certain set of requirements on a serving layer database. However, what is *not* required of a serving layer database is far more interesting than the requirements. First, the requirements:

- **Batch writable.** The batch views for a serving layer are produced from scratch. When a new version of a view becomes available, it must be possible to completely swap out the older version with the updated view.
- **Scalable.** A serving layer database must be capable of handling views of arbitrary size. As with the distributed filesystems and batch computation framework previously discussed, this requires it to be distributed across multiple machines.
- **Random reads.** A serving layer database must support random reads where indexes provide direct access to small portions of the view. This requirement is necessary to have low latency on queries.
- **Fault-tolerant.** Since a serving layer database is distributed, it must be tolerant to machine failures.

Hopefully nothing on this list is a surprise. However, a customary requirement missing from this list - one that is standard on all familiar databases - is *random writes*. Such functionality is completely irrelevant to the serving layer because the views are only produced in bulk. To be clear, random writes do exist in the Lambda Architecture, but they are isolated within the speed layer to achieve low latency updates. Updates to the serving layer generate new views in their entirety, so a serving layer database does not need the ability to modify small portions of the current view.

This is an amazing result since random writes are responsible for the majority of complexity in databases – and even more complexity in distributed databases. To understand this complexity, consider one particularly nasty detail of how random-write databases work: the use of write-ahead logs and compaction. Modifying a disk index for every operation would be too expensive for the operation of a database, so pending operations are written to a sequential write-ahead log. These changes are then periodically applied to the disk index in bulk. This process of applying events from the write-ahead log to the on-disk index is called *compaction*.

Compaction is a fairly intensive operation. The server places substantially higher demand on the CPU and disks during compaction, which can lead to highly variable performance. Databases such as HBase and Cassandra are well-known for requiring careful configuration to avoid problems or server lockups during compaction. This is just one of the many complexities taken on by a database when

it must support random writes. Another source is the need to synchronize reads and writes so that half-written values are never read. When a database doesn't have random writes, it can optimize the read path and get better performance than a random read/write database.

A rough but good indicator of the complexity can be seen in the size of the code base. ElephantDB, a database built specifically to be a serving layer database, is only a few thousand lines of code. HBase and Cassandra, two popular distributed read/write databases, are hundreds of thousands of lines long. Lines of code isn't normally a good complexity metric, but in this case the staggering difference should be telling.

A simpler database is more predictable since it fundamentally does fewer things. It accordingly is less likely to have bugs and will be easier to operate. Since the serving layer views contain the overwhelming majority of your queryable data, the simplicity of serving layer databases is a great property to have for an architecture.

7.4 Example of serving layer database: ElephantDB

Having covered the requirements, you can now consider an example of a database built specifically to be used as a serving layer database. ElephantDB is a key/value database where both keys and values are stored as byte arrays. You will examine the basic architecture of ElephantDB to understand how it meets the requirements of the serving layer, then you will review its API to retrieve the contents of a batch view.

ElephantDB partitions the batch views over a fixed number of shards. Each view uses a pluggable sharding scheme that assigns keys to the shards that will store the corresponding values. One common scheme determines the target shard by dividing the hash of a key by the total number of shards, then returning the remainder. Informally we will call this technique *hash modding*. It distributes the keys evenly among the shards and provides an easy means to determine which shard holds a given key, but there are scenarios where customized schemes are better tailored to the view. Once assigned to a shard, the key/value is stored in a local indexing engine. By default this is BerkeleyDB, but the engine is configurable and could be any key/value indexing engine that runs on a single machine.

There are two aspects to ElephantDB: view creation and view serving. View creation occurs in a MapReduce job at the end of the batch layer workflow where the generated partitions are stored in the distributed filesystem. Views are then

served by a dedicated ElephantDB cluster that loads the shards from the distributed filesystem and interacts with clients that support random read requests. We will briefly discuss these two roles before finally diving into using ElephantDB.

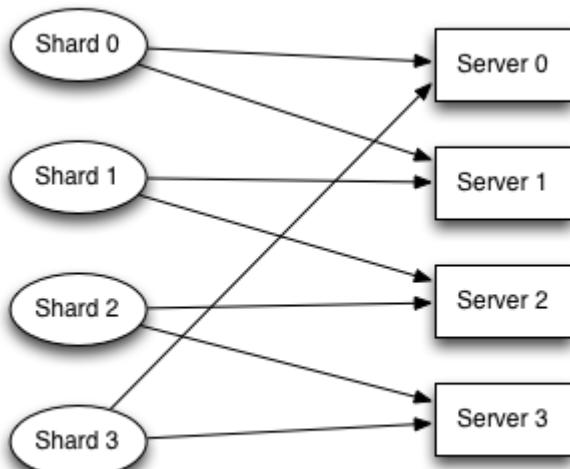
7.4.1 View creation in ElephantDB

The ElephantDB shards are created by a MapReduce job whose input is a set of key/value pairs. The number of reducers is configured to be the number of ElephantDB shards, and the keys are partitioned to the reducers using the specified sharding scheme. Consequently each reducer is responsible for producing exactly one shard of an ElephantDB view. Each shard is then indexed (e.g. into a BerkeleyDB index) and uploaded to the distributed filesystem.

Note that the view creation process does not directly send the shards to the ElephantDB servers. Such a design would be poor because the client-facing machines would not control their own load and query performance could suffer. Instead, the ElephantDB servers pull the shards from the filesystem at a throttled rate that allows them to maintain their performance guarantees to clients.

7.4.2 View serving in ElephantDB

An ElephantDB cluster is comprised of a number of machines that divide the work of serving the shards. To fairly share the load, the shards are evenly distributed among the servers. ElephantDB also supports replication where each shard is redundantly hosted across a predetermined number of servers. For example, with 40 shards, 8 servers and a replication factor of 3, each server hosts 15 shards and each shard exists on 3 different servers. This makes the cluster tolerant to machine failures, allowing full access to the entire view even when machines are lost. Of course, only so many machines can be lost before portions of the view become unavailable, but replication makes this possibility far less likely. Replication is illustrated in Figure 7.8.



With replication, each shard exists on multiple servers

Figure 7.8 Replication stores shards in multiple locations to increase tolerance to individual machine failures.

ElephantDB servers are responsible for retrieving their assigned shards from the distributed filesystem. When a server detects a new version of a shard is available, it does a throttled download of the new partition. The download is controlled so not to saturate the I/O of the machine and affect live reads. Upon completing the download, it switches to new partition and deletes the old one.

After an ElephantDB server has downloaded its shards, the contents of the batch views are accessible via a basic API. We earlier mentioned that there is no general purpose serving layer database - this is where the limitations of ElephantDB become apparent. Since ElephantDB uses a key/indexing model, the API only allows for the retrieval of values for specified keys. A general serving layer database would provide a richer API, such as the ability to scan over key ranges. It is important to recognize this limitation, but ElephantDB is a viable solution and demonstrates the desired qualities of a serving layer database.

7.4.3 Using ElephantDB

The simplicity of ElephantDB makes it straightforward to use. There are three separate aspects to using ElephantDB: creating shards, setting up a cluster to serve requests, and using the client API to query the batch views. We will step through each of these components.

CREATING ELEPHANTDB SHARDS

The tap abstraction makes it simple to create a set ElephantDB shards using JCascalog. ElephantDB provides a tap to automate the shard creation process. If you have a subquery that generates key/value pairs, creating the ElephantDB view is as simple as executing that subquery into the tap:

```
public static elephantDbTapExample (Subquery subquery) {
    DomainSpec spec = new DomainSpec(new JavaBerkDB(), ①
        new HashModScheme()); ②
    Object tap = EDB.makeKeyValTap("/output/path/on/dfs", spec, 32); ③
    Api.execute(tap, subquery); ④
}
```

- ① use BerkeleyDB as the local storage engine
- ② apply hash mod partitioning as the sharding scheme
- ③ create 32 shards at the given distributed filesystem path
- ④ direct the output of the subquery to the constructed tap

Under the hood, the configured tap automatically configures the MapReduce job to correctly partition the keys, creates each index, and uploads each index to the distributed filesystem.

SETTING UP AN ELEPHANTDB CLUSTER

There are two required configurations for establishing a ElephantDB cluster: a local configuration and a global configuration. The local configuration contains server-specific properties as well as the addresses where the global configuration and the actual shards reside. A basic local configuration resides on the each individual server and resembles the following:

```
{:local-root "/data/elephantdb" ①
:hdfs-conf {"fs.default.name" "hdfs://namenode.domain.com:8020"} ②
:blob-conf {"fs.default.name" "hdfs://namenode.domain.com:8020"} } ③
```

- ① the local directory to store downloaded shards
- ② the address of the distributed filesystem that stores the shards
- ③ the address of the distributed filesystem hosting the global configuration
- ④ The global configuration contains information needed by every server in the

cluster. This includes the replication factor, the TCP port that servers should use to accept requests and the views served by this cluster. A single cluster can serve multiple domains, so the configuration contains a map from the domain name to their HDFS locations. A basic global configuration would appear like the following listing:

```

{:replication 1 ①
:hosts [ "edb1.domain.com" "edb2.domain.com" "edb3.domain.com" ] ②
:port 3578 ③
:domains { "tweet-counts" "/data/output/tweet-counts-edb"
           "influenced-by"  "/data/output/influenced-by-edb"
           "influencer-of"   "/data/output/influencer-of-edb" } } ④

```

- ① the replication factor of all views for all servers
- ② hostnames of all servers in the cluster
- ③ the TCP port the server will use to accept requests
- ④ identifiers of all views and their locations on the distributed filesystem

These configurations are so simple they almost appear incomplete. For example, there is no explicit assignment from the servers to the specific shards they will host. In this particular case, the servers use their position in the hosts list as input to a deterministic function to calculate the shards they should download. The simplicity of the configurations reflect the ease of using ElephantDB.

SIDE BAR How do you actually start an ElephantDB server?

The process of launching an ElephantDB follows standard Java practices, such as building a project uberjar and passing the configuration locations via a command line statement. Rather than provide details which could quickly become out-of-date, we refer you to the project website (<http://github.com/nathanmarz/elephantdb>) for specifics.

QUERYING AN ELEPHANTDB CLUSTER

ElephantDB exposes a simple Thrift API for issuing queries. After connecting to any ElephantDB server, you can issue queries like so:

```

public static void clientQuery(ElephantDB.Client client,
                               String domain,
                               byte[] key) {

```

```

    client.get(domain, key);
}

```

If the connected server does not store the requested key locally, it will communicate with the other servers in the cluster to retrieve the desired values.

7.5 Building the serving layer for SuperWebAnalytics.com

Having covered the basics, you can now create the optimized ElephantDB views for each query in SuperWebAnalytics.com. First up is the pageviews over time view. At the end of Chapter 6 you had produced a view as shown in Figure 7.9.

URL	Granularity	Bucket	Pageviews
foo.com/blog/1	h	0	10
foo.com/blog/1	h	1	21
foo.com/blog/1	h	2	7
foo.com/blog/1	w	0	38
foo.com/blog/1	m	0	38
bar.com/post/a	h	0	213
bar.com/post/a	h	1	178
bar.com/post/a	h	2	568

Figure 7.9 Pageviews over time batch view

Recall that the both the keys and values in ElephantDB are stored as byte arrays. For the pageviews over time view, you need to encode the URL, granularity and time bucket into the key. The following JCascalog functions implement the required serializations for composite keys and the pageview values:

```

public static class ToUrlBucketedKey extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        String url = call getArguments().getString(0);
        String gran = call getArguments().getString(1);
        Integer bucket = call getArguments().getInteger(2);

        String keyStr = url + "/" + gran + "-" + bucket; ①
        try {
            call.getOutputCollector()
                .add(new Tuple(keyStr.getBytes("UTF-8")));
        } catch(UnsupportedEncodingException e) {
            throw new RuntimeException(e);
        }
    }
}

```

```

    }

public static class ToSerializedLong extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        long val = call.getArguments().getLong(0);
        ByteBuffer buffer = ByteBuffer.allocate(8); 3
        buffer.putLong(val);
        call.getOutputCollector().add(new Tuple(buffer.array())); 4
    }
}

```

- 1** concatenate the key components
- 2** convert to bytes using UTF-8 encoding
- 3** configure ByteBuffer to hold a single long value
- 4** extract the byte array from the buffer

The next step is to create the ElephantDB tap. To avoid the variance problem discussed at the beginning of the chapter, you need a custom ShardingScheme to ensure that all key/value pairs for a single URL exist on a single shard. The following snippet accomplishes this by hash modding only the URL portion of the composite key:

```

private static String getUrlFromSerializedKey(byte[] ser) {
    try {
        String key = new String(ser, "UTF-8");
        return key.substring(0, key.lastIndexOf("/")); 1
    } catch(UnsupportedEncodingException e) {
        throw new RuntimeException(e);
    }
}

public static class UrlOnlyScheme implements ShardingScheme {
    public int shardIndex(byte[] shardKey, int shardCount) {
        String url = getUrlFromSerializedKey(shardKey);
        return url.hashCode() % shardCount; 2
    }
}

```

- 1** extract the URL from the composite key
- 2** return the hash mod of the URL

The following JCascalog subquery puts the pieces together to transform the batch layer view into key/value pairs appropriate for ElephantDB:

```

public static void pageviewElephantDB(Subquery batchView) {
    Subquery toEdb =
        new Subquery("?key", "?value") ①
            .predicate(batchView, "?url", "?gran", "?bucket", "?total-views")
            .predicate(new ToUrlBucketedKey(), "?url", "?gran", "?bucket")
            .out("?key")
            .predicate(new ToSerializedLong(), "?total-views")
            .out("?value");

    DomainSpec spec = new DomainSpec(new JavaBerkDB(),
                                     new UrlOnlyScheme(),
                                     32); ②

    Tap tap = EDB.makeKeyValTap("/outputs/edb/pageviews", spec); ③
    Api.execute(tap, toEdb);
}

```

- ① the subquery must return only two fields corresponding to keys and values
- ② define the local storage engine, sharding scheme and total number of shards
- ③ specify the HDFS location of the shards
- ④ execute the transformation

The pageviews over time view would benefit from a more general serving layer database that could store the time buckets for each URL sequentially and in chronological order. As discussed in the beginning of the chapter, such a database would utilize disk scans and minimize expensive disk seeks. Such a serving layer database does not exist at the time of this writing, though creating one would be much simpler than most currently available NoSQL databases.

The next query is the unique pageviews over time query. This is almost identical to the pageviews over time query, with the exception that the HyperLogLog sets are already byte arrays so do not require serialization. This view uses the same sharding scheme as the pageviews over time view in order to avoid the variance problem.

```

public static void uniquesElephantDB(Subquery uniquesView) {
    Subquery toEdb =
        new Subquery("?key", "?value")
            .predicate(uniquesView, "?url", "?gran", "?bucket", "?value")
            .predicate(new ToUrlBucketedKey(), "?url", "?gran", "?bucket") ①
            .out("?key");

    DomainSpec spec = new DomainSpec(new JavaBerkDB(),
                                     new UrlOnlyScheme(),
                                     32);

```

```

    Tap tap = EDB.makeKeyValTap("/outputs/edb/uniques", spec);
    Api.execute(tap, toEdb);
}

```

- ① only the composite key needs to be serialized
- ② change the output directory for the unique pageviews shards

The last view is a map from each domain to the number of visits and the number of bounces. You can reuse the framework from the previous queries, but you still need custom serialization code for the string keys and compound value:

```

public static class ToSerializedString extends CascalogFunction { ①
  public void operate(FlowProcess process, FunctionCall call) {
    String str = call.getArguments().getString(0);

    try {
      call.getOutputCollector().add(new Tuple(str.getBytes("UTF-8")));
    } catch(UnsupportedEncodingException e) {
      throw new RuntimeException(e);
    }
  }
}

public static class ToSerializedLongPair extends CascalogFunction { ②
  public void operate(FlowProcess process, FunctionCall call) {
    long l1 = call.getArguments().getLong(0);
    long l2 = call.getArguments().getLong(1);
    ByteBuffer buffer = ByteBuffer.allocate(16);
    buffer.putLong(l1);
    buffer.putLong(l2);
    call.getOutputCollector().add(new Tuple(buffer.array()));
  }
}

```

- ① this serialization function is essentially identical to the one for the composite keys
- ② allocate space for two long values

Queries against this view will fetch only one domain at a time, so there are no concerns about variance in server response times. Normal hash mod sharding is suitable for this case.

```

public static void bounceRateElephantDB(Subquery bounceView) {
  Subquery toEdb =
    new Subquery("?key", "?value")
      .predicate(bounceView, "?domain", "?bounces", "?total")

```

```

.predicate(new ToSerializedString(), "?domain")
.out("?key")
.predicate(new ToSerializedLongPair(),"?bounces", "?total")
.out("?value");

DomainSpec spec = new DomainSpec(new JavaBerkDB(),
    new HashModScheme( ), ①
    32);
Tap tap = EDB.makeKeyValTap("/outputs/edb/bounces", spec);
Api.execute(tap, toEdb);
}

```

- ① use hash mod sharding scheme provided by ElephantDB

As you can see, integrating the batch views into the serving layer is almost no work at all.

7.6 Conclusion

You saw in this chapter the fundamental concepts of the serving layer and an example of a serving layer database in ElephantDB. Although there is no completely general serving layer database currently available, the key/value indexing model of ElephantDB demonstrates the value of the serving layer in the Lambda Architecture:

- the ability to tailor views to optimize latency and throughput,
- the simplicity from not supporting random writes,
- the capacity to store normalized data in the batch layer and denormalized data in the serving layer, and
- the inherent error-tolerance and correction of the serving layer since it is can be recomputed from the master dataset.

Now that you understand the batch and the serving layers, next up is learning the final piece of the Lambda Architecture: the speed layer. The speed layer will compensate for the high latency updates of the serving layer and allow queries to access up to date data.



Speed layer: Real-time views

This chapter covers:

- The theoretical model of the speed layer
- How the batch layer eases the responsibilities of the speed layer
- Using random-write databases for real-time views
- The CAP theorem and its implications
- The challenges of incremental computation
- Expiring data from the speed layer
- Cassandra as an example of a real-time view database

To this point our discussion of the Lambda Architecture has revolved around the batch and serving layers - components that involve computing functions over every piece of data you have. These layers satisfy all the desirable properties of a data system save one: low-latency updates. The sole job of the speed layer is to satisfy this final requirement.

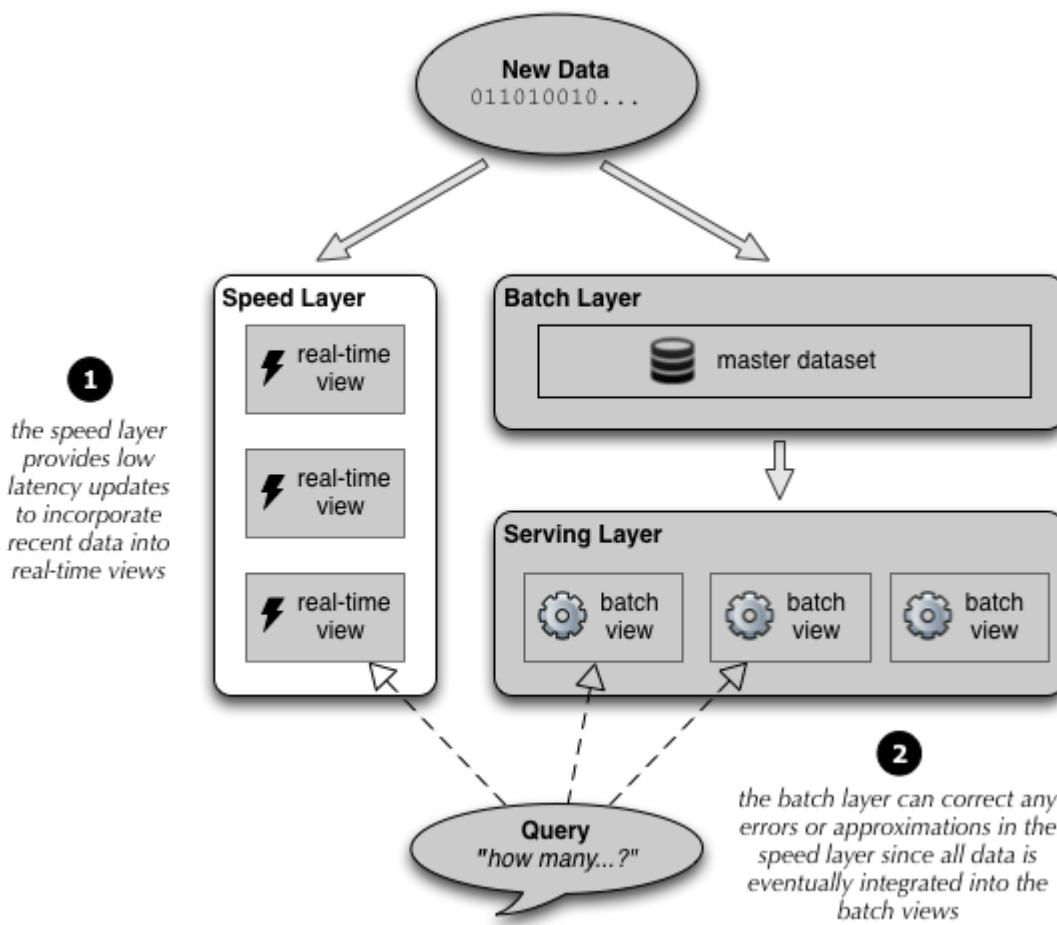


Figure 8.1 The speed layer allows the Lambda Architecture to serve low-latency queries over up-to-date data.

Running functions over the entire master dataset – potentially petabytes of data – is a resource intensive operation. To lower the latency of updates as much as possible, the speed layer must take a fundamentally different approach than the batch and serving layers. As such, the speed layer is based on *incremental computation* instead of batch computation.

Incremental computation introduces many new challenges and is significantly more complex than batch computation. Fortunately, the narrow requirements of the speed layer provide two advantages. First, the speed layer is only responsible for data yet to be included in the serving layer views. This data is at most a few hours old and is vastly smaller than the master dataset. Processing data on a smaller scale allows for greater design flexibility. Second, the speed layer views are transient. Once data is absorbed into the serving layer views, it can be discarded from the

speed layer. Even though the speed layer is more complex and thus more prone to error, any errors are short-lived and will be automatically corrected through the simpler batch and serving layers.

As we have repeatedly stated, the power of the Lambda Architecture lies in the separation of roles in the different layers. In traditional data architectures such as those based on relational databases, *all that exists is a speed layer*. These systems have limited options to battling the complexities of incremental computation.

There are two major facets of the speed layer: storing the real-time views and processing the incoming data stream so as to update those views. This chapter will focus on the structure and storage of real-time views. We will begin with an overview of the theoretical foundation of the speed layer, then continue onto the various challenges you encounter with incremental computation. Next we will demonstrate how to expire data from the speed layer, and we will end the chapter with a look at Cassandra, a NoSQL database that can be used for persistence in the speed layer.

8.1 Computing real-time views

The basic objective of the speed layer is the same as the batch and serving layers: to produce views that can be efficiently queried. The key differences are that the views only represent recent data and that they must be updated very shortly once new data arrives. What "very shortly" means varies per application, but it typically ranges from a few milliseconds to a few seconds. This requirement has far-reaching consequences on the computational approach to generate the speed layer views.

To understand the implications, consider one simple approach to the speed layer. Similar to the batch/serving layers producing views by computing a function on the entire master dataset, the speed layer could produce its views by running a function over all of the recent data (that is, data yet to be absorbed into the serving layer). This is attractive for both its simplicity and the consistency with how the batch layer works, as shown in Figure 8.2.

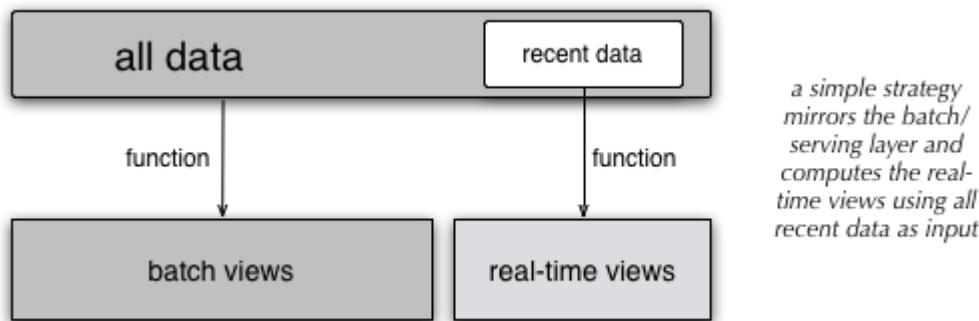


Figure 8.2 "real-time view = function(recent data)" strategy

Unfortunately this scheme proves impractical for many applications once you consider its latency and resource usage characteristics. Suppose your data system receives 32GB of new data per day and that the batch/serving layer requires 6 hours to update the views. The speed layer would be responsible for at most 6 hours of data - about 8GB. While not a huge amount, 8GB is substantial when attempting to achieve sub-second latencies. Additionally, running a function on 8GB of data each time you receive a new piece of data will be extremely resource intensive. If the average size of a data unit is 100 bytes, the 8GB of recent data equates to approximately 86 million data units. Keeping the real-time views up to date would thus require an unreasonable amount of $86M * 8GB$ worth of processing every 6 hours. You could reduce the resource usage by batching the updates, but this greatly increases the update latency.

If your application can accept latency on the order of a few minutes, this simple strategy is a fine approach. However, in general you will need to produce real-time views in a resource efficient manner with millisecond-level latencies. For the remainder of the chapter we will confine our discussion to this scenario. In general, any workable solution relies on using incremental algorithms as depicted in Figure 8.3.

Figure 8.3 incremental "real-time view = function(new data, previous real-time view)" strategy

The idea is to update the real-time views as data comes in, thereby reusing the work that previously went into producing the views. This requires the use of random read / random write databases so that updates can be performed on existing views. In the next section we will further discuss these databases while delving into the details of storing the speed layer views.

8.2 Storing real-time views

The obligations of the speed layer views are quite demanding - the Lambda Architecture requires low-latency random reads, and using incremental algorithms necessitates low-latency random updates. The underlying storage layer must therefore meet the following requirements:

1. **Random reads.** A real-time view should support fast random reads to answer queries quickly. This means the data it contains must be indexed.
2. **Random writes.** To support incremental algorithms, it must also be possible to modify a real-time view with low latency.
3. **Scalability.** As with the serving layer views, the real-time views should scale with the amount of data they store and the read/write rates required by the application. Typically this implies that real-time views can be distributed across many machines.
4. **Fault tolerance.** If a disk or a machine crashes, a real-time view should continue to function normally. Fault-tolerance is accomplished by replicating data across machines so there are backups should a single machine fail.

These properties are common to a class of databases that have been dubbed *NoSQL databases*. NoSQL databases support diverse data models and index types, so you can select one or more real-time view databases to meet your indexing requirements. For example, you may choose Cassandra to store indexes with a key/value format, then use ElasticSearch for indexes that support search queries. In this chapter we will focus on Cassandra because it supports many interesting index types, but ultimately you have great power and flexibility in choosing a combination of databases to fit your exact speed layer needs.

8.2.1 Eventual accuracy

Your selection of databases determines *how* the real-time views are stored, but you have great flexibility in *what* you store to answer queries. In many cases the contents of your real-time views will exactly mirror the contents of your batch views. For example, a query that returns pageviews will store exact counts in both the batch layer and in the speed layer. However, this not need be the case since frequently it is difficult to incrementally compute functions which can be easily computed in batch. You have previously encountered this scenario before when determining unique counts. In batch, this is easy to compute since you process the entire dataset at once, but it is much harder in real-time since you need to store the entire set to correctly update the counts.

In these cases you can take a different approach in the speed layer and approximate the correct answer. Since all data is eventually represented in the

batch/serving layer views, any approximations you make in the speed layer are continually corrected. This means any approximations are only temporary and your queries exhibit eventual accuracy. This is a really powerful technique that gives you the best of all worlds: performance, accuracy, and timeliness. Eventually accurate approaches are common with sophisticated queries such as those that require real-time machine learning. Such views tend to correlate many kinds of data together and cannot incrementally produce an exact answer in any reasonable way.

8.2.2 Amount of state stored in the speed layer

Speed layers store relatively small amounts of state because they only represent views on recent data. This is actually a benefit because real-time views are much more complex than serving layer views. We briefly revisit the complexities of real-time views that are bypassed by the serving layer:

1. **Incremental compaction.** When a read/write database receives an update, it logs the update to an append-only file and periodically applies the changes in bulk. The bulk modification process (compaction) is a resource intensive process and could potentially starve the machine of resources needed to rapidly serve queries.
2. **Concurrency.** A read/write database can potentially receive many reads or writes for the same value at the same time. It therefore needs to coordinate these reads and writes to prevent returning stale or inconsistent values. Sharing mutable state across threads is a notoriously complex problem, and control strategies such as locking are notoriously bug-prone.

It is important to note that the speed layer is under less pressure since it stores considerably less data than the serving layer. The separation of roles and responsibilities within the Lambda Architecture limits complexity to the speed layer and compensates for complexity errors in the batch / serving layer views.

Now that you understand the basics of the speed layer, let's take a deeper look at the challenges you face when doing incremental computation as opposed to batch computation.

8.3 Challenges of incremental computation

In Chapter 4 we discussed the differences between recomputation and incremental algorithms. To summarize, incremental algorithms are less general and less human fault-tolerant than recomputation algorithms, but they provide much higher performance. It is this higher performance that we are leveraging in the speed layer. However, an additional challenge arises when using incremental algorithms in a real-time context: the interaction between incremental algorithms and a theorem called the CAP theorem. This challenge can be particularly hairy and is important to understand.

The CAP theorem is about fundamental tradeoffs between consistency, where reads are guaranteed to incorporate all previous writes, and availability, where every query returns an answer instead of erroring. Unfortunately the theorem is often explained in a very misleading and inaccurate way. We would normally avoid presenting an inaccurate explanation of anything, but this interpretation is so widespread it is necessary to discuss it to clarify the misconceptions. CAP is typically stated as "you can have at most two of consistency, availability, and partition-tolerance". The problem with this explanation is that the CAP theorem is entirely about what happens to data systems when not all machines can communicate with each other. Saying a data system is consistent and available but not partition-tolerant makes no sense, because the theorem is entirely about what happens under partitions.

The proper way to present the CAP theorem is that when a distributed data system is partitioned, it can be consistent or available but not both. Should you choose consistency, sometimes a query will receive an error instead of an answer. When you choose availability, at best you will have *eventual consistency* where reads will eventually represent all previous writes. If an update occurs while the system is partitioned, not all servers will be notified of the change. Any queries to the separated machines will therefore use out-of-date information when providing an answer. However, once the partition is resolved, the updates will eventually become accessible to every machine in the cluster.

8.3.1 Validity of the CAP theorem

It is fairly easy to understand why the CAP theorem is true. Say you have a simple distributed key/value database where every node in your cluster is responsible for a separate set of keys, e.g. there is no replication. When you want to read or write data for a particular key, there is a single machine responsible for handling the request.

Now suppose you suddenly cannot communicate with some of the machines in the distributed system. Obviously you cannot read or write data to nodes that are inaccessible to you. You can increase the fault tolerance of distributed data system by replicating data across multiple servers. With a replication factor of three, every key would be replicated to three nodes. Now if you become partitioned from one replica, you can still retrieve the value from another location. However it is still possible to be partitioned from all replicas, just much less likely.

The real question is how to handle writes under partitions. There are a few options - for example, you could refuse to perform an update unless all replicas can be updated at once. With this policy, every read request is guaranteed to return the latest value. This is a consistent strategy since clients partitioned from each other will always receive the same value (or get an error) when doing reads.

Alternatively, you can choose to update whatever replicas are available and synchronize with other replicas when the partition is resolved. This gets tricky when a scenario like Figure 8.4 occurs. In this situations clients are partitioned differently and communicate with different subgroups of machines. The replicas will diverge if you are willing to update a subset of the replicas, and merging is more complicated than simply picking the latest update.

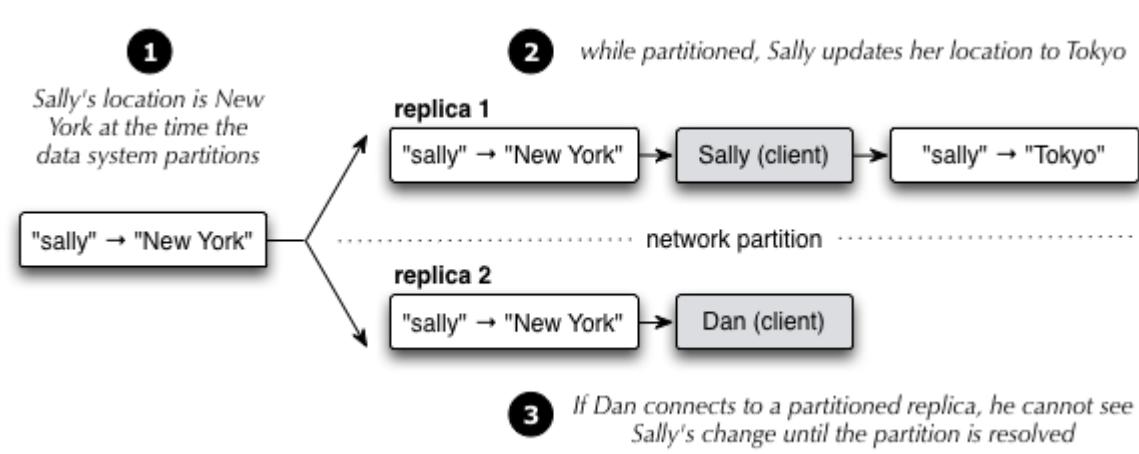


Figure 8.4 Replicas can diverge if updates are allowed under partitions

With a partial update strategy, your data system is available but not consistent. In Figure 8.4, Sally may update her location to Tokyo, but Dan's replica cannot be updated so he would read an out of date value. These examples demonstrate it is impossible to have both availability and consistency during a partition for there is no means to communicate with the replica that has the most recent information.

SIDE BAR Extreme Availability - Sloppy Quorums

Some distributed databases have an option called sloppy quorums which provides availability in the extreme – writes are accepted even if replicas for that data are not available. Instead, a temporary replica will be created and then merged into the official replicas once they become available. With sloppy quorums, the potential number of replicas for a piece of data is equal to the number of nodes in the cluster if every node is partitioned from every other node. While useful, keep in mind such an approach increases the incidental complexity of your system.

As additional examples, the batch and serving layers are distributed systems and are subject to the CAP theorem like any other system. The only writes in the batch layer are new pieces of immutable data. These writes do not require coordination between machines because every piece of data is independent. If data cannot be written to the incoming data store in the batch layer, it is buffered locally and retried later. As for the serving layer, reads are always stale due to the high latency of the batch layer. Not only is the serving layer not consistent, it is not even eventually consistent because it is always out-of-date. Accordingly, both the batch and serving layers choose availability over consistency.

Note that nothing complex was needed to determine these properties – the logic in the batch/serving layers is simple and easy to understand. Unfortunately this is *not* the case when striving for eventual consistency using incremental algorithms in a real-time context.

8.3.2 The complex interaction between the CAP theorem and incremental algorithms

As we just discussed, if you choose high availability for a distributed system, a partition will create multiple replicas of the same value that are updated independently of one another. When the partition resolves, the values must be merged together so that the new value incorporates every update during the partition - no more or no less. The problem is there is no simply way to accomplish this for every use case, so it falls on you as a developer to identify a working strategy. As an example, consider how you would implement eventually consistent counting.

In this scenario we will assume you are only storing the count as the value. Suppose the network partitions, the replicas evolve independently, then the partition is corrected. When it comes time to merge the replica values, you find that one replica has a count of 110 and another has a count of 105. What should the new value be? The cause for confusion is that you are unaware at what point they began to diverge. If they diverged at the value 105, the updated count should be 110. If they diverged at 0, the right answer would be 215. All you know for certain is the right answer is somewhere between these two bounds.

To implement eventually consistent counting correctly, you need to make use of structures called conflict-free replicated datatypes (commonly referred to as CRDTs). There are a number of CRDTs for a variety of values and operations: sets that support only addition, sets that support addition and removal, numbers that support increments, numbers that support increments and decrements and so forth. The G-Counter is a CRDT that supports only incrementing, which is exactly what you need to solve the current counter issue. An example G-Counter is shown in Figure 8.5.



Figure 8.5 A G-counter is a grow only counter where a replica only increments its assigned counter. The overall value of the counter is the sum of the replica counts.

A G-Counter stores a different value per replica rather than just a single value. The actual count is then the sum of the replica counts. If a conflict between

replicas is detected, the new G-counter takes the max value for each replica. Since counts only increase and only one server in the system will be updating the count for a given replica, the max value is guaranteed to be the correct one. This is illustrated in Figure 8.6.

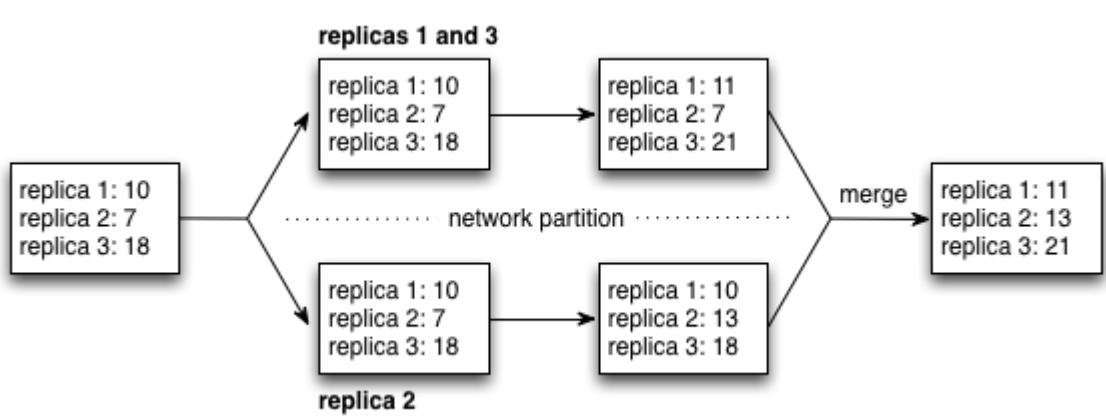


Figure 8.6 Merging G-Counters

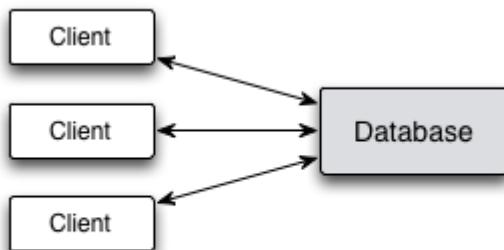
As you can see, implementing counting is far more complex in a real-time eventually consistent context. It is not sufficient to keep a simple count, you also need a strategy to repair values that diverge in a partition. The algorithms can also introduce further complexity. If you allow for decrements as well as increments, the data structure and merge algorithm become even more complicated. These merge algorithms – commonly called *read repair algorithms* – are huge sources of human error. This is not surprising given the complexity.

Unfortunately there is no escape from this complexity if you want eventual consistency in the speed layer. However, you do have one thing going for you: the Lambda Architecture gives you inherent protection from making mistakes. If the real-time view becomes corrupted because you forgot an edge case or messed up the merge algorithm, the batch/serving layers will later automatically correct the mistake in the serving layer views. The worst a mistake can lead to is temporary corruption. Architectures without a batch layer backing up the real-time, incremental portions would just have permanent corruption.

8.4 Asynchronous versus synchronous updates

The architecture for the speed layer differs depending on whether the real-time views are updated synchronously or asynchronously. A synchronous update is something you've likely done a million times: the application issues a request directly to the database and blocks until the update is processed. For example, if a user registers an e-mail address, you may choose to issue a synchronous update to the database to record the new information. Synchronous updates are fast because they communicate directly with the database, and they facilitate coordinating the update with other aspects of the application (such as displaying a spinning cursor while waiting for the update to complete).

The speed layer architecture for synchronous updates is illustrated in Figure 8.7. Not surprisingly, the application simply issues updates directly to the database.



With synchronous updates, clients communicate directly with the database and block until the update is completed

Figure 8.7 A simple speed layer architecture using synchronous updates.

In contrast, asynchronous update requests are placed in a queue with the updates occurring at a later time. In the speed layer, this delay could range from a few milliseconds to a few seconds, but it could take even longer if there is an excess of requests. Asynchronous updates are slower than synchronous updates because they require additional steps before the database is modified, and it is impossible to coordinate them with other actions since you cannot control when they are executed. However, asynchronous updates provide many advantages. First, you can read multiple messages from the queue and perform batch updates to the database, greatly increasing throughput. They also readily handle varying load: if the number of update request spikes, the queue buffers the additional requests until all updates are executed. Conversely, a traffic spike with synchronous updates could overload the database, leading to dropped requests, timeouts and other errors that disrupt your application.

The speed layer architecture for asynchronous updates is illustrated in Figure 8.8. In the next chapter you will learn about queues and stream processing in much more detail.

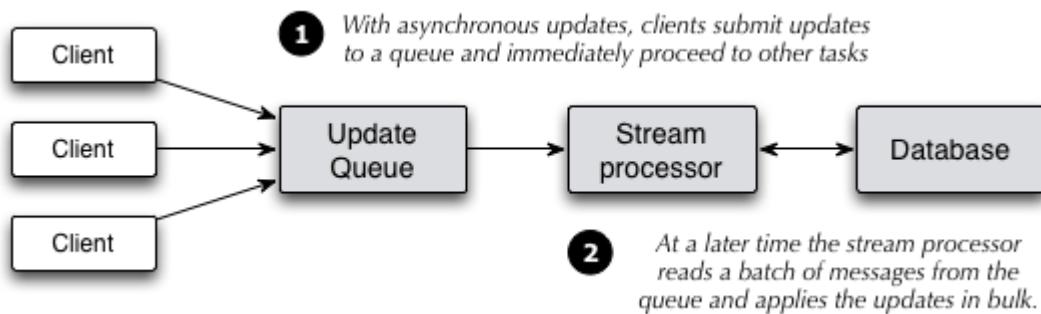


Figure 8.8 Asynchronous updates provide higher throughput and readily handle variable loads

There are uses for both synchronous and asynchronous updates. Synchronous updates are typical among transactional systems that interact with users and require coordination with the user interface. Asynchronous updates are common for analytics-oriented workloads or workloads not requiring coordination. The architectural advantages of asynchronous updates – better throughput and better management of load spikes – suggest implementing asynchronous updates unless you have a good reason not to do so.

8.5 Expiring real-time views

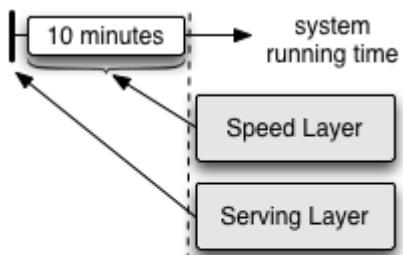
Incremental algorithms and random write databases make the speed layer far more complex than the batch and serving layers, but one of the key benefits of the Lambda Architecture is the transient nature of the speed layer. Since the simpler batch and serving layers continuously override the speed layer, the speed layer views only need to represent data yet to be processed by the batch computation workflow. Once a batch computation run finishes, you can then discard a portion of the speed layer views - the parts now absorbed into the serving layer - but obviously you must keep everything else.

Ideally a speed layer database would provide support to directly expire entries, but this is typically not an option with current databases. Tools like Memcached provide a similar behavior to set time-delayed expirations on key/value pairs, but they are not well suited for this problem. For example, you could set the expiration of an entry to be the expected time before it is integrated into the serving layer

views (with perhaps extra time to serve as a buffer.) However, if for unexpected reasons the batch layer processing requires additional time, those speed layer entries would prematurely expire.

Instead, we present a generic approach for expiring speed layer views that works regardless of the speed layer databases being used. To understand this approach, let's first get an understanding of what exactly needs to be expired each time the serving layer is updated. Suppose you have a complete Lambda Architecture implementation and you turn on your application for the first time. The system has yet to have any data, so initially both the speed layer views and serving layer views are empty.

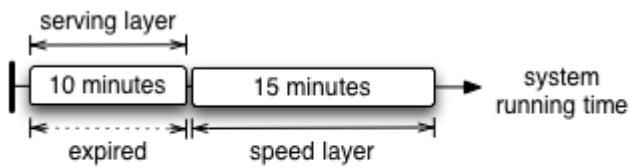
When the batch layer first runs, it will operate on no data. Say the batch layer computation takes 10 minutes due to the overhead of running jobs, creating empty indexes and so forth. At the end of those 10 minutes, the serving layer views remain empty but the speed layer views now represent 10 minutes of data. This situation is illustrated in Figure 8.9.



After the first batch computation run, the serving layer remains empty but the speed layer has processed recent data.

Figure 8.9 The state of the serving layer views and speed layer views at the end of the first batch computation run

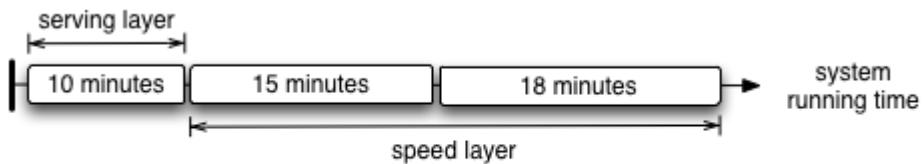
The second run of the batch layer immediately commences to process the 10 minutes of data that accumulated during the first run. For illustrative purposes, say the second run takes 15 minutes. When it finishes, the serving layer views will represent *the first 10 minutes* of data while the speed layer views will represent all 25 minutes of data. The first 10 minutes can now be expired from the speed layer views, as shown in Figure 8.10.



When the second batch computation run finishes, the first segment of data has been absorbed into the serving layer and can be expired from the speed layer views.

Figure 8.10 A portion of the real-time views can be expired after the second run completes.

Finally, suppose the third run of the batch layer takes 18 minutes. Consider the instant *before* the third run completes, as depicted in Figure 8.11.



Just prior to the completion of the batch layer computation, the speed layer is responsible for data that accumulated for the prior two runs.

Figure 8.11 The serving and speed layer views immediately before the completion of the third batch computation run

At this point the serving layer still only represents 10 minutes of data, leaving the speed layer to handle the remaining 33 minutes of data. This figure demonstrates that the speed layer views compensate for between one and two runs of the batch layer, depending on how far the batch layer has progressed through its workflow. When a batch layer run finishes, data from three runs ago can safely be discarded from the speed layer.

The simplest way to accomplish this task is to maintain *two sets* of real-time views and alternate clearing them after each batch layer run, as shown in Figure 8.12. By doing so, one of those sets of real-time views will exactly represent the data necessary to compensate for the serving layer views. After each batch layer run, the application should switch to reading from the real-time view with more data (and away from the one that was just cleared).

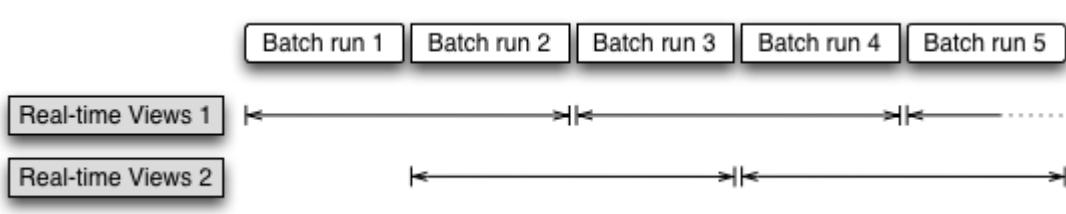


Figure 8.12 Alternating clearing between two different sets of real-time views guarantees one set always contains the appropriate data for the speed layer

At a first glance it may appear expensive to maintain two real-time views, for it essentially doubles the storage cost of the speed layer. The key is that the speed layer views only represent *a minuscule portion of your data* – at most a few hours worth. Compared to potentially years of data represented by your serving layer, this can be less than 0.1% of all the data in your system. This scheme does introduce redundancy, but it is an acceptable price for a general solution to expire real-time views.

8.6 Cassandra: an example speed layer view

We now take a look at Cassandra, a NoSQL database that you might choose to store a real-time view. There are many publicly available resources to understand in-depth the inner workings of the database, so we focus on the properties of Cassandra from a user perspective.

8.6.1 Cassandra's data model

While many tout Cassandra as a column-oriented database, we find that terminology to be somewhat confusing. Instead, it is easier to consider the data model as a map with sorted maps as values (or optionally, a sorted map with sorted maps as values). Cassandra allows standard operations on the nested maps, such as adding key/value pairs, looking up by key and getting ranges of keys. To introduce the terminology, Figure 8.13 illustrates the different aspects of the Cassandra data model:

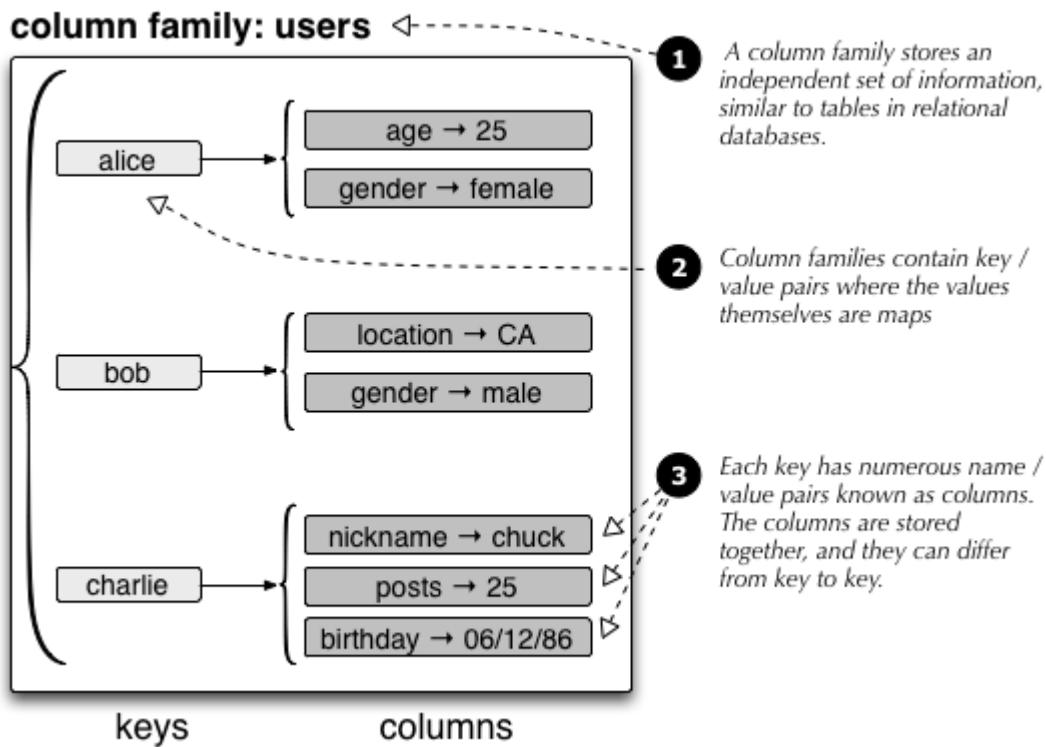


Figure 8.13 The Cassandra data model consists of column families, keys and columns.

To expand upon the terms:

- *Column families* are analogous to tables in relational databases, and each column family stores a completely independent set of information.
- If you consider a column family as a giant map, *keys* are the top level entries into that map. Cassandra uses the keys to partition a column family across a cluster.
- Each key points to another map of name/value pairs called *columns*. All columns for a key are physically stored together, making it inexpensive to access ranges of columns. Different keys can have different sets of columns, and it is possible to have thousands - or even millions - of columns for a given key.

To understand this model fully, we return to our SuperWebAnalytics.com example. Specifically, let's see how to model the pageviews over time view using Cassandra. For pageviews over time, you want to retrieve the pageviews for a particular URL and a particular granularity (hour, day, week and four weeks) over a specific range of time buckets. To store this information in Cassandra, the key will be a [URL, granularity] pair and the columns will be the name/value pairs of time buckets and pageviews. Figure 8.14 demonstrates some sample data for this view.

column family: pageviews

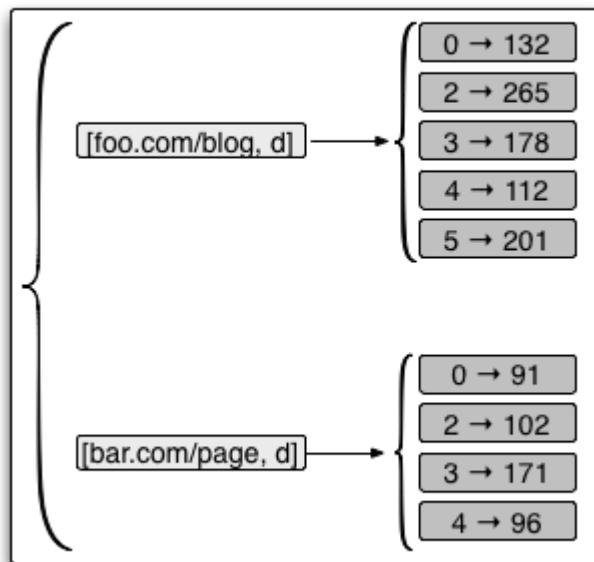


Figure 8.14 Pageviews over time represented in Cassandra

This is an efficient way to store pageviews since columns are sorted - in this case, by the time bucket - and stored physically together. Contrast this with an alternative scheme where the Cassandra key is a triplet of URL / granularity / time bucket and there is a single pageviews column. Querying a range of pageviews in this scheme requires lookups for multiple Cassandra keys. Since each key could potentially reside on a different server, these queries have high latency due to variance in server response times as discussed in Chapter 7.

8.6.2 Using Cassandra

Having covered the Cassandra data model, we are ready to implement the pageviews over time scheme with keys as a URL/granularity pair and a column per time bucket. This implementation will be demonstrated using the Hector Java client for Cassandra.

First you need a client that can issue operations to a particular column family. Since schemas only have to be created once per column family, we have omitted schema definition code to avoid cluttering the code. We refer you to the Hector or Cassandra documentation if you need further details.

```
Cluster cluster = HFactory.getOrCreateCluster("mycluster", "127.0.0.1"); ①
Keyspace keyspace = ②
    HFactory.createKeyspace("superwebanalytics", cluster);
```

```
ColumnFamilyTemplate<String, Long> template = ③
    new ThriftColumnFamilyTemplate<String, Long> (keyspace,
                                                "pageviews",
                                                StringSerializer.get(),
                                                LongSerializer.get());
```

- ① create a cluster object to connect to a distributed Cassandra cluster
- ② a keyspace is a container for all the column families of an application
- ③

Once you have a client to communicate with the cluster, you can retrieve the pageviews for a given URL and a given range of time buckets. The following code calculates total pageviews for "foo.com/blog" at a daily granularity for time buckets 20 to 55:

```
SliceQuery<String, Long, Long> slice = ①
    HFactory.createSliceQuery(keyspace,
                               StringSerializer.get(),
                               LongSerializer.get(),
                               LongSerializer.get());
slice.setColumnFamily("pageviews");
slice.setKey("foo.com/blog-d"); ③

ColumnSliceIterator<String, Long, Long> it =
    new ColumnSliceIterator<String, Long, Long>(slice,
        20L, 55L, false); ④
long total = 0;
while(it.hasNext()) {
    total += it.next().getValue(); ⑤
}
```

- ① create a range template for a Cassandra query
- ② the serializers are for the keys (url/granularity), names (buckets) and values (pageviews), respectively
- ③ assign the column family and key for the query
- ④ obtain a column iterator for the given range
- ⑤ traverse the iterator and sum the pageviews to obtain the total

The ColumnSliceIterator traverses all columns in the given range. If the range is enormous (such as containing tens of thousands of columns), it will automatically buffer/batch those columns from the server so not to run out of memory. Remember, because the columns are ordered and stored together, slice operations over a range of columns are very efficient.

Updates are equally easy. The following code adds 25 to the pageviews of "foo.com/blog" for time bucket 7 at "daily" granularity.

```

long currVal;
HColumn<Long, Long> col =
    template.querySingleColumn("foo.com/blog-d", ①
        7L,
        LongSerializer.get()); ②
if (col==null) ③
    currVal = 0;
else
    currVal = col.getValue();

ColumnFamilyUpdater<String, Long> updater =
    template.createUpdater("foo.com/blog-d");
updater.setLong(7L, currVal + 25L);
template.update(updater); ⑤ ④

```

- ① retrieve the current value for the specified URL, granularity and time bucket
- ② serializer for pageview value
- ③ if no value is recorded, there have been no pageviews
- ④ create an update template for the given key
- ⑤ increment the pageviews and perform the update

For an analytics use case like pageviews over time, you would normally batch updates for multiple time buckets to increase efficiency. We'll discuss this further in the next chapter.

8.6.3 Advanced Cassandra

It is worth mentioning a few advanced features of Cassandra that make it suitable for a broader range of real-time view types. The first addresses how Cassandra partitions keys among nodes in the cluster. You can choose between two partitioner types: the RandomPartitioner and the OrderPreservingPartitioner.

The RandomPartitioner makes Cassandra act like a hashmap and assigns keys to partitions using the hash of the key. This results in the keys being evenly distributed across the nodes in the cluster. In contrast, the OrderPreservingPartitioner stores the keys in order, causing Cassandra to behave like a sorted map. Keeping the keys sorted enables you to do efficient queries on ranges of keys.

Though there are advantages to keeping the keys sorted, there is a cost for using the OrderPreservingPartitioner. When preserving the order of the keys, Cassandra

attempts to split the keys so that each partition contains approximately the same number of keys. Unfortunately there is no good algorithm to determine balanced key ranges on the fly. In an incremental setting, clusters can become unbalanced with some servers being overloaded while others have virtually no data. This is another example of complexity you face with real-time, incremental computation that you avoid with batch computation. In a batch context you know all the keys beforehand, so you can evenly split keys among partitions as part of the computation.

Cassandra has another feature called *composite columns* that extends the sorted map idea even further. They let you nest the maps arbitrarily deep – for example, you could model your index as a map of sorted maps of sorted maps of sorted maps.

8.7 Conclusion

The speed layer is very different than the batch layer. Rather than compute functions of your entire dataset, you instead compute using much more complex incremental algorithms on much more complex forms of storage. The Lambda Architecture allows you to keep the speed layer small and therefore much more manageable.

You've learned the basic concepts of the speed layer and the details around managing real-time views. The next step to cover is connecting the stream of data being generated with the real-time views. In next chapter, you will see how to combine a database like Cassandra with a stream processing engine to accomplish this feat.

9

Speed layer: Queueing and stream processing

This chapter covers:

- Single-consumer vs. multi-consumer queues
- One-at-a-time stream processing
- Problems with "queues and workers" approach to stream processing
- Implementing stream processing with Apache Storm

In the last chapter you saw two kinds of architectures for the speed layer: synchronous, where the application requires coordination and thus writes directly to the database, and asynchronous, where updates to the database happen independently from the application that created the data. There's not much more to cover from an architecture standpoint on synchronous speed layers – applications just write directly to the database, but there's a lot to cover with asynchronous architectures.

This chapter will cover the basics of queuing and stream processing, the two foundations of asynchronous architectures. Like how in batch processing the key was the ability to withstand failures and retry computations when necessary, in stream processing systems how fault-tolerance and retries are handled are of the utmost importance. As usual, the story is more complex in the speed layer, and there's many more tradeoffs that you'll have to keep in mind as you design applications.

After a discussion of the need for persistent queuing, we'll launch into an overview of the simplest kind of stream processing – one at a time processing. You'll see how the fault-tolerance and retry story plays out for that kind of processing. The open source projects Apache Storm and Apache Kafka will be used to illustrate these concepts.

9.1 Queuing

It would be possible to asynchronously process an event without a queue. The event would simply get handed off to some worker who would process the event independently, possibly updating a database. This is illustrated in Figure 9.1. However, a scheme like this has no guarantees on whether data is processed or not. If a worker dies then some set of events will fail to process. By writing the event into a persistent queue first, and then having a worker process events off of that queue, you can retry events off the queue when a worker fails.

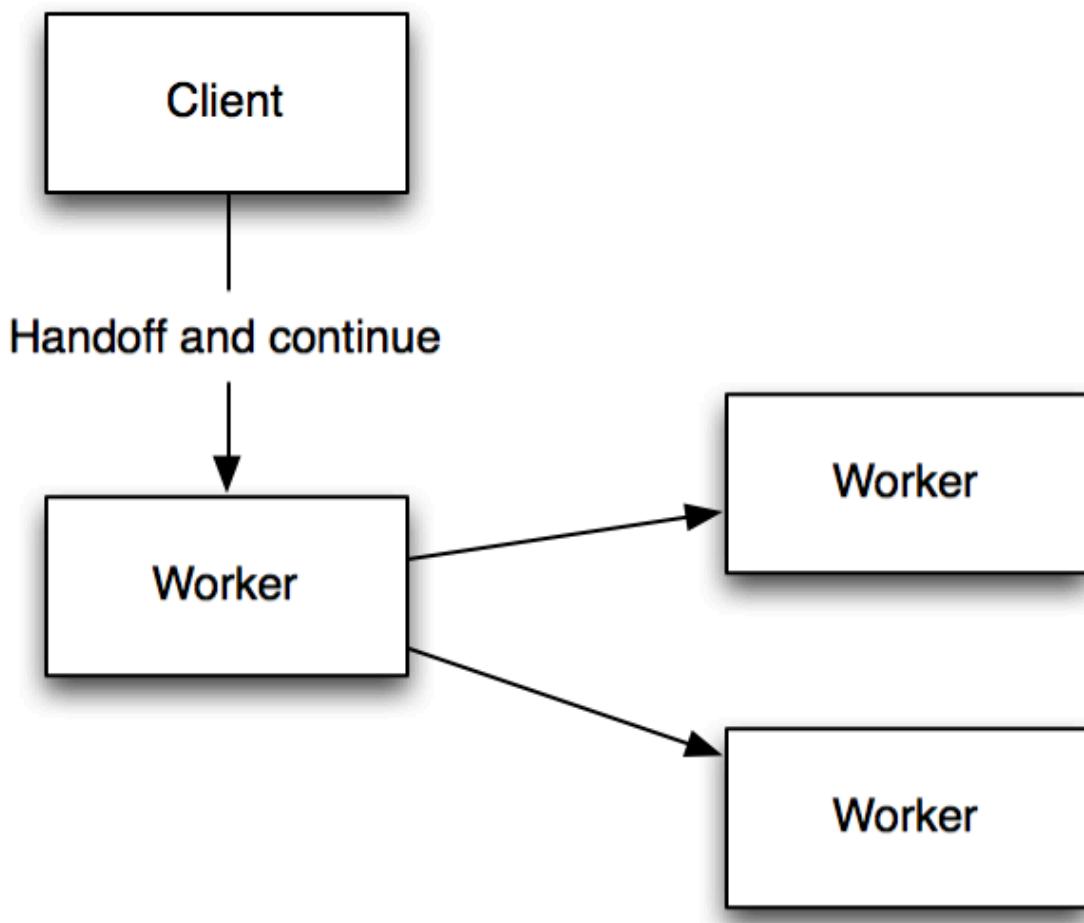


Figure 9.1 Asynchronous processing without queues

Likewise, suppose you're processing events asynchronously with no queue and there's a sudden burst of traffic that far exceeds the resources in your processing cluster. Your cluster will get overwhelmed and will fail to process many of the messages. A queue solves this problem as well by providing a place for events to buffer when downstream processors hit their limit on what they can process.

So far you know that a queue should provide a way to persist an event stream and replay events if downstream processing fails – but the specific semantics of a good queue design is so far vague. A good place to start is with queue interfaces you're already familiar with, such as the Java "Queue" interface. Here are the main methods on that interface (there are other methods on the Queue interface, but these are basically variants of these methods and aren't important for the purposes of this discussion):

```
interface Queue {
    void add(Object item);
    Object poll();
    Object peek();
}
```

The basic methods on this queue interface are to add new items, look at what's on the head of the queue without removing it (peek), and taking the last item on the queue off and returning it (poll). An implementation of queue such as LinkedBlockingQueue or ConcurrentLinkedQueue lets you use this queue interface among many threads, where some number of threads can be adding to the queue and some number of other threads can be reading from it.

9.1.1 Single-consumer queue servers

The interface just shown is a natural starting point when designing a persistent queue to be used in the speed layer. In fact, many such queues with this kind of interface exist, such as Kestrel and RabbitMQ. These queues have interfaces essentially like this (simplified):

```
struct Item {
    long id;
    byte[] item;
}

interface Queue {
    Item get();
    void ack(long id);
    void fail(long id);
}
```

The idea is that when you get an item off the queue, the item is not taken off immediately (like peek). "get" also returns an identifier which you can use to "ack" or "fail" the item. If you "ack" the item, it will then be removed from the queue. If you "fail" the item (or your client disconnects or a timeout occurs), then the queue server will allow another client to receive that item via another "get" call. This design ensures that items are only taken off the queue after it has been successfully processed. It's possible that it could be processed multiple times, such as if a client processes it and dies right before acking, but the important thing is there's a guarantee that it's processed at least once.

This all sounds well and good, but there is actually a very deep flaw in this design for queues: what if multiple applications want to consume the same stream? This is incredibly common. For example, you may have a stream of pageviews, and one consumer wants to build a view of pageviews over time while another wants to build a view of unique visitors over time. One way to do this would be to put all the applications in the same consumer, as shown in Figure 9.2.

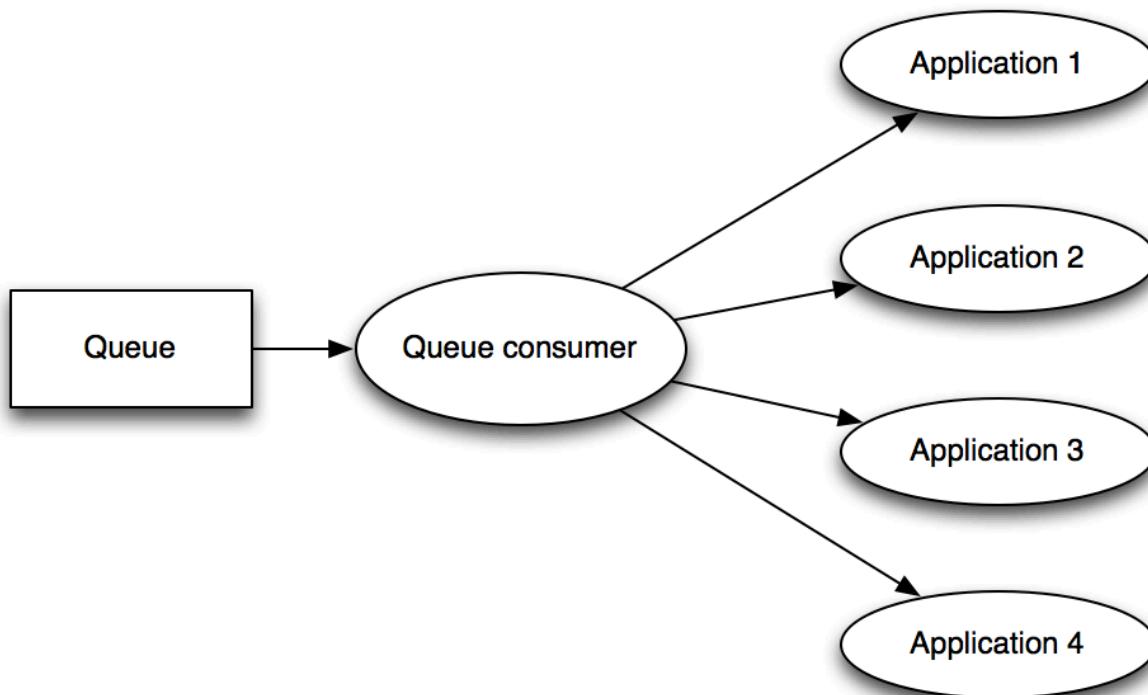


Figure 9.2 Many applications sharing a queue consumer

However, when done this way, all the applications have to live in the same codebase and run in the same workers. This is simply a bad idea, as it eliminates isolation between independent applications. Now if one application has a bug it

will affect all the other applications it's coexisting with. This problem is especially exacerbated in larger organizations where multiple teams want to share the same stream. Now if one team messes up their code, it can break the code of other teams sharing that stream. It's much better, and much saner, to have independence between independent applications, so that when one application breaks or has a bug, other applications are unaffected.

The only way to achieve independence with the queue design you've seen so far is to duplicate the queue per consumer. So if you have three consumers, keep three copies of the queue on the queue server. Kestrel, for example, has a built-in feature that will do exactly this. The problem with this is now you're massively increasing the load on the queue server, its load is now proportional to the number of consumers times the amount of incoming events rather than just the amount of incoming events. It's perfectly feasible to want to build dozens of views off of the same stream, so this implies more than an order of magnitude load increase on your queues.

What you really want is a queue that can be used by many consumers, where adding a consumer is cheap and does not cause such an increase in load. When you think about it, the fundamental issue with single consumer queues is that the queue is responsible for keeping track of what is consumed. Because of this restrictive condition that an item is either "consumed" or "not consumed", it is unable to handle the idea of multiple clients wanting to consume the same item.

9.1.2 Multi-consumer queues

There's an alternative design for queues that does not suffer from the problems with single consumer queues. The idea is rather than have the queue server keep track of what is consumed, instead have the application keep track of what it has consumed. The application can then replay from any point in the past. In this case the queue server is unable to know when an item has been consumed by all consumers, so instead it offers a service-level agreement (SLA) on the stream: it only keeps up to a certain amount of the stream available, such as the last 12 hours or the last 50GB.

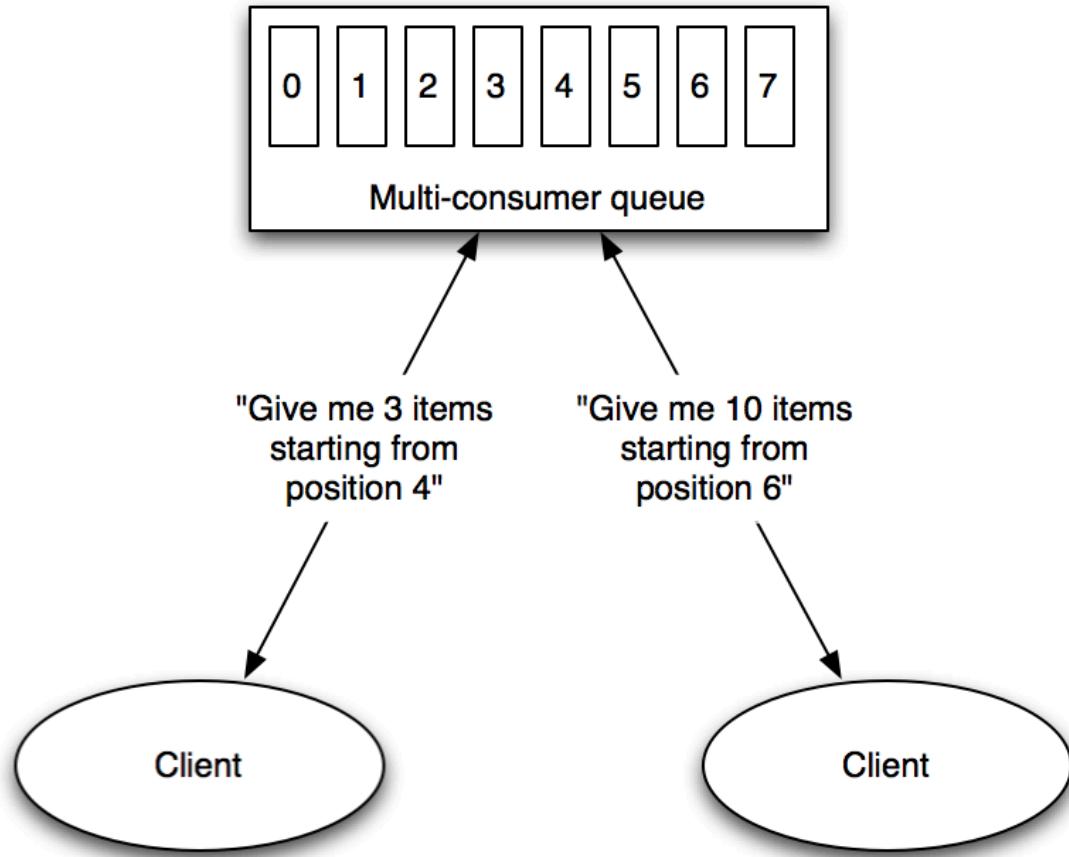


Figure 9.3 Multi-consumer queue

Apache Kafka is a good example of a queue implementation that implements this model of queuing, and it exposes an interface similar to what has been described.

There are other notable differences between single-consumer and multi-consumer queues. With a single consumer queue, once a message has been acked, it is deleted and can no longer be replayed. So if events are being processed in parallel, and an event fails even though later events have been acked, when the event is replayed it will be out of order. With a multi-consumer queue, you can always rewind the stream and replay everything from the point it failed, allowing you to see events in the order they came. So you have more flexibility with replays with multi-consumer queues. And it is exactly this flexibility which is needed to achieve exactly-once processing semantics in the speed layer, as will be discussed in depth in the next chapter.

The benefits of multi-consumer queues are huge, and they don't really have any

drawbacks as compared to single consumer queues. So using a queue such as Apache Kafka is highly recommended.

9.2 Stream processing

Once you have your incoming events feeding your multi-consumer queues, the next step is to process those events and update your realtime views. This is called stream processing.

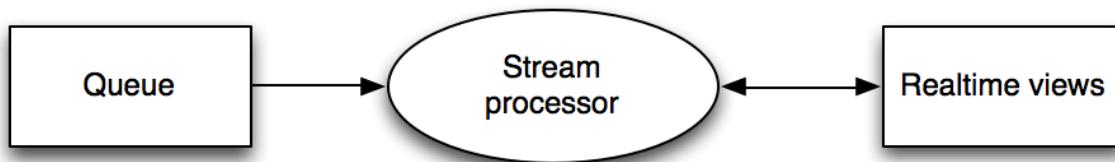


Figure 9.4 Stream processing

There are two models of stream processing that have emerged in recent years: "one-at-a-time" and "micro-batched". There are tradeoffs between these approaches, each has its strengths and weaknesses. They are very much complementary – some applications are better suited towards one-at-a-time stream processing, and others are better suited towards micro-batch stream processing.

	One-at-a-time	Micro-batched
Lower latency	✓	
Higher throughput		✓
At-least-once semantics	✓	✓
Exactly-once semantics	Only in some cases	✓
Simpler programming model	✓	

Figure 9.5 Comparison of stream processing paradigms

One of the big advantages of one-at-a-time stream processing is it can process

streams with much lower latency than micro-batched processing. Example applications that really benefit from this include alerting and financial trading.

In this chapter we'll focus on one-at-a-time stream processing, and in the next we'll cover micro-batch stream processing. We'll build towards a general model of one-at-a-time stream processing by first observing the old way of doing it: the queues and workers model. The problems plaguing this approach will motivate a more general purpose way of doing one-at-a-time stream processing.

9.2.1 Queues and workers

The "queues and workers" paradigm is a widespread way of doing one-at-a-time stream processing. It is an obvious and straightforward approach but is fraught with problems.

The basic idea is to divide your processing into worker processes, and put queues between the worker processes. So if one worker has a failure or the worker process restarts, it can always continue where it left off by reading from its queue. This is illustrated in Figure 9.6. The queues in that diagram could potentially be distributed queues as well.

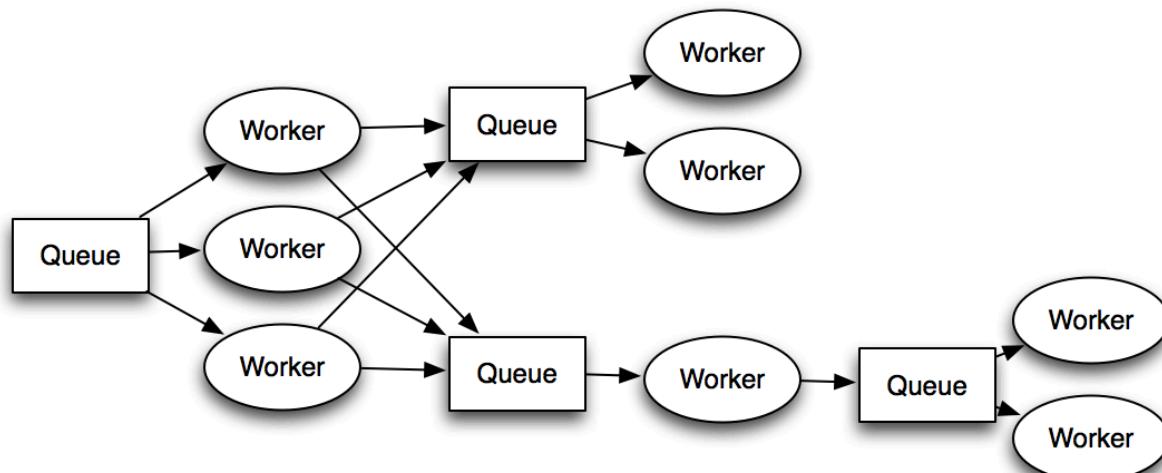


Figure 9.6 Queues and workers

For example, let's say you're implementing pageviews over time with the queues and workers approach. The first set of workers reads pageview events off a set of queues, applies basic validation to filter out invalid URLs, and then passes the remaining events to a second set of workers that will update the pageview counts in the database (let's say Cassandra). Each set of workers is parallelized among many worker processes to handle the throughput of the incoming stream.

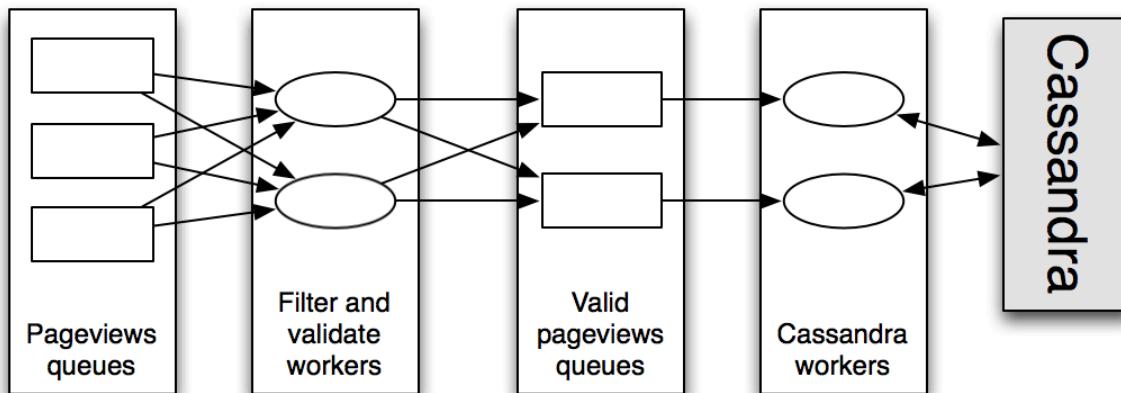


Figure 9.7 Pageviews over time with queues and workers

One subtlety in this scheme is the need to ensure that only one process is updating the pageview counts for one URL at any given time. Otherwise, there would be a race condition and writes would be trampled in the database. So in order to ensure this, the first set of workers partitions its outgoing stream by the URL. So a pageview event for any given URL will always go to the same queue, while the entire set of URLs will be spread among queues. A simple way to implement this is to choose the target queue by hashing the URL and modding the hash by the number of target queues.

The queues and workers approach works great at low scale, when you have few workers and few queues. But what about when you need dozens of workers at each stage of processing? Each worker process must be manually deployed. What happens if one machine goes down? If that was one of the workers updating the pageview counts in the database, now nothing is updating the database for that portion of the stream. You have to manually start that worker somewhere else, or build a custom system to do so. So the fault-tolerance of the queues and workers approach isn't very good.

Another problem is the operational complexity of having queues between every set of workers. What if you need to change your topology of processing? You'll have to coordinate things so that the intermediate queues are cleared before you redeploy, otherwise you'll lose data. Going through a queue to get from one worker to another is also slow – the event is forced to go through a third party and it has to be persisted to disk! This adds latency and decreases throughput. Plus, those intermediate queues need to be managed and monitored and it's yet another layer that needs to be scaled.

Finally, perhaps the biggest problem with the queues and workers approach is how tedious it is to build. So much of your code ends up dedicated to how objects are serialized and deserialized to and from queues, where to read messages from, where to send messages to, and how to deploy workers. Your actual business logic ends up being a very small percentage of your codebase. The fact that the workers work in tandem towards some higher goal, and have so many coordination details in their code, indicates strongly the need for a higher level abstraction.

9.3 Higher level one-at-a-time stream processing

The higher level one-at-a-time stream processing approach you'll learn now is a generalization of the queues and workers model without any of the complexities of queues and workers. Like queues and workers, the code that processes tuples receives one tuple at a time, though the code runs in parallel across the cluster so the system is scalable and high throughput. This is in contrast to "micro-batched" stream processing, which processes the stream in small batches and will be covered in the next chapter. There are tradeoffs between these two approaches and neither is "better" than the other.

Remember, the goal of the speed layer is to process a stream and update the realtime views. That's it. Our goal here is to accomplish that with a minimum of hassle and have strong guarantees on the processing of data.

Like how MapReduce is a model for scalable batch computation, and Hadoop is a specific implementation of that model, a similar model exists for one-at-a-time stream processing. There's no short catchy name for this model like MapReduce, so we'll refer to it as the "Storm model" after the project that originated these techniques. Let's now go over this model and see how it alleviates the complexities of queues and workers.

9.3.1 Storm model

The idea of the Storm model is to represent the entire streaming computation as a graph of computation. This graph is called a "topology". Rather than program each node of the topology as a separate program, like you do with queues and workers, and then have to manually connect them all together, instead everything is coded in a single program and a Storm cluster takes care of distributing that code across the cluster. So you could filter in one node, aggregate in another, and then update your realtime view databases in another. Serialiation, message passing, task discovery, and fault-tolerance is handled automatically for you by Storm. Whereas before you had to explicitly program for those things, now you can focus on your business logic. One of the great benefits of the Storm model is that it enables you to do stream processing with very low latency – achieving average latencies of 10ms or less from receiving a tuple to finishing processing it is very doable.

Let's build to the Storm model from the ground up. At the core of the Storm model are "streams". A stream, illustrated in Figure 9.8, is an infinite sequence of tuples, and a tuple is a named list of values. The Storm model is all about transforming streams into new streams, possibly updating databases along the way.

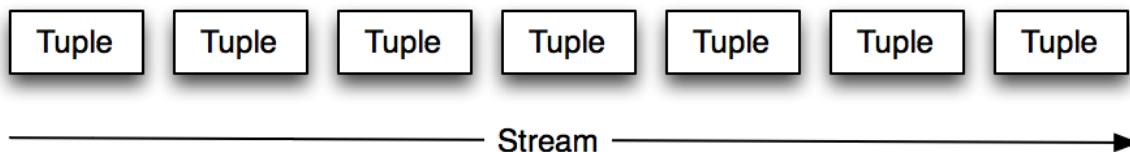


Figure 9.8 Stream

The next abstraction in the Storm model is the "spout". The spout is a source of streams in a Storm application. So you might have a spout that reads from a Kestrel or Kafka queue and turns that into a stream. Or perhaps you have a timer spout that emits a tuple into its output stream every 10 seconds.

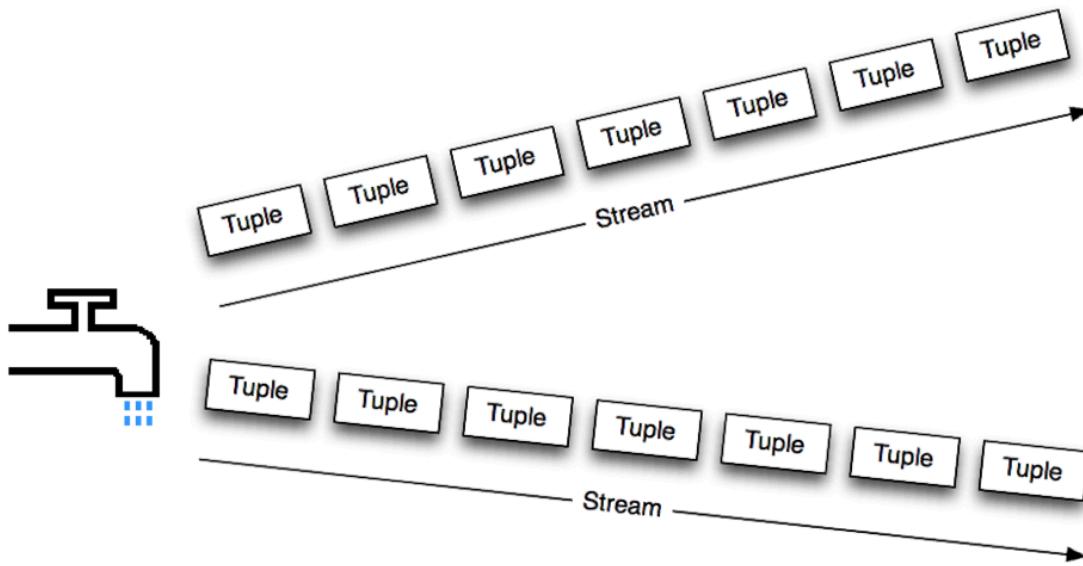


Figure 9.9 Spout

The next abstraction is called a "bolt". A bolt takes as input any number of input streams and produces as output any number of output streams. Bolts are where most of the logic goes in a Storm application – bolts run functions, do filtering, do streaming aggregations, do streaming joins, update databases, and so on.

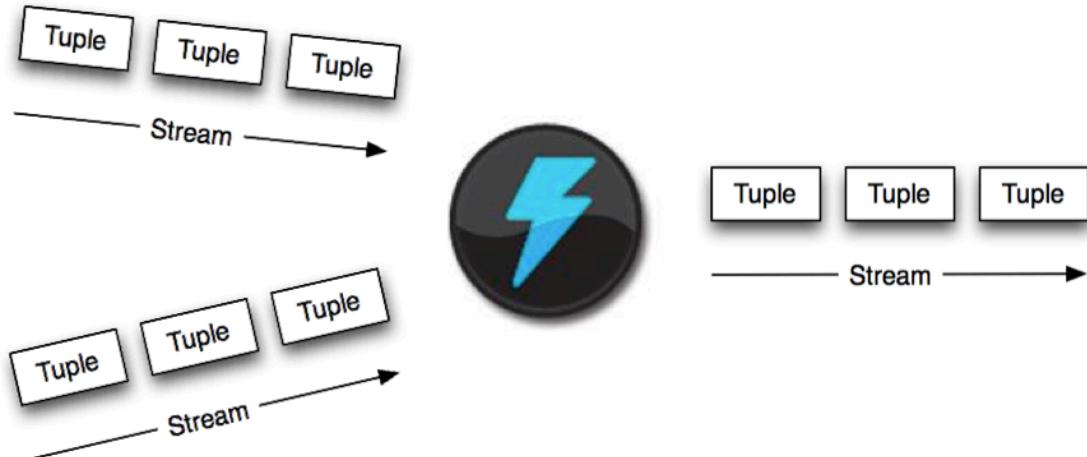


Figure 9.10 Bolt

Finally, the top-level abstraction in the Storm model is called a "topology". A topology is a network of spouts and bolts, with each edge indicating a bolt

subscribing to the output stream of another spout or bolt.

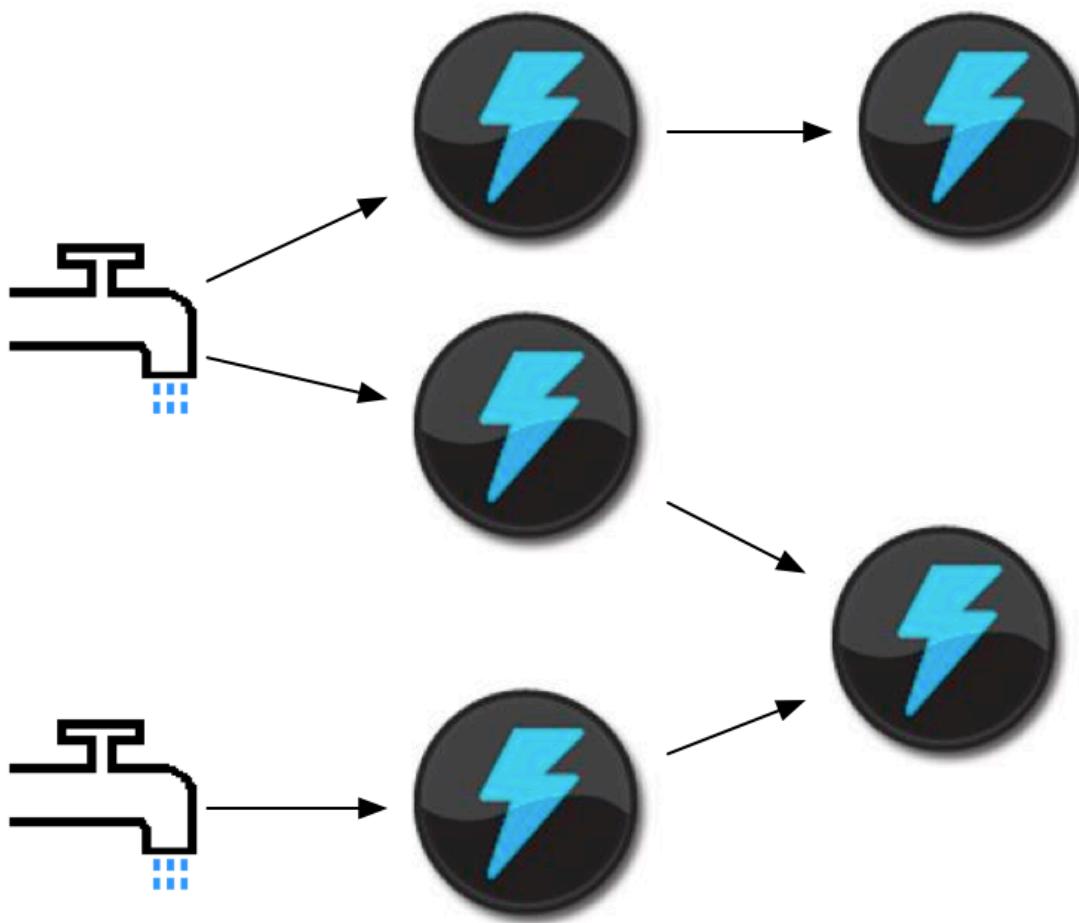


Figure 9.11 Topology

The key to Storm is that spouts and bolts are inherently parallel, just like how map and reduce tasks are inherently parallel in MapReduce. Each instance of a spout or bolt is called a "task". So the tuple flow through a topology looks like Figure 9.12.

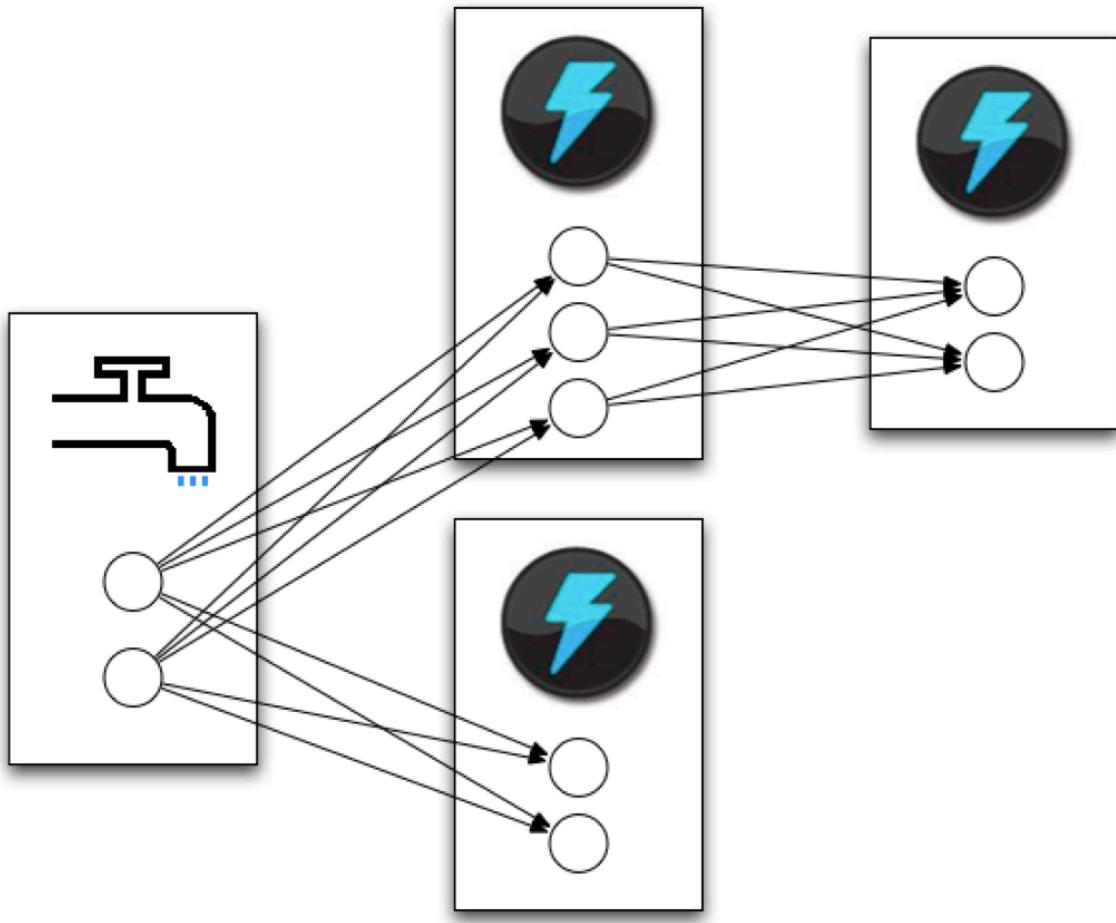


Figure 9.12 Topology tasks

Of course, not all tasks for a given spout or bolt will necessarily run on the same machine. They'll be spread around the cluster among different workers, like in Figure 9.13.

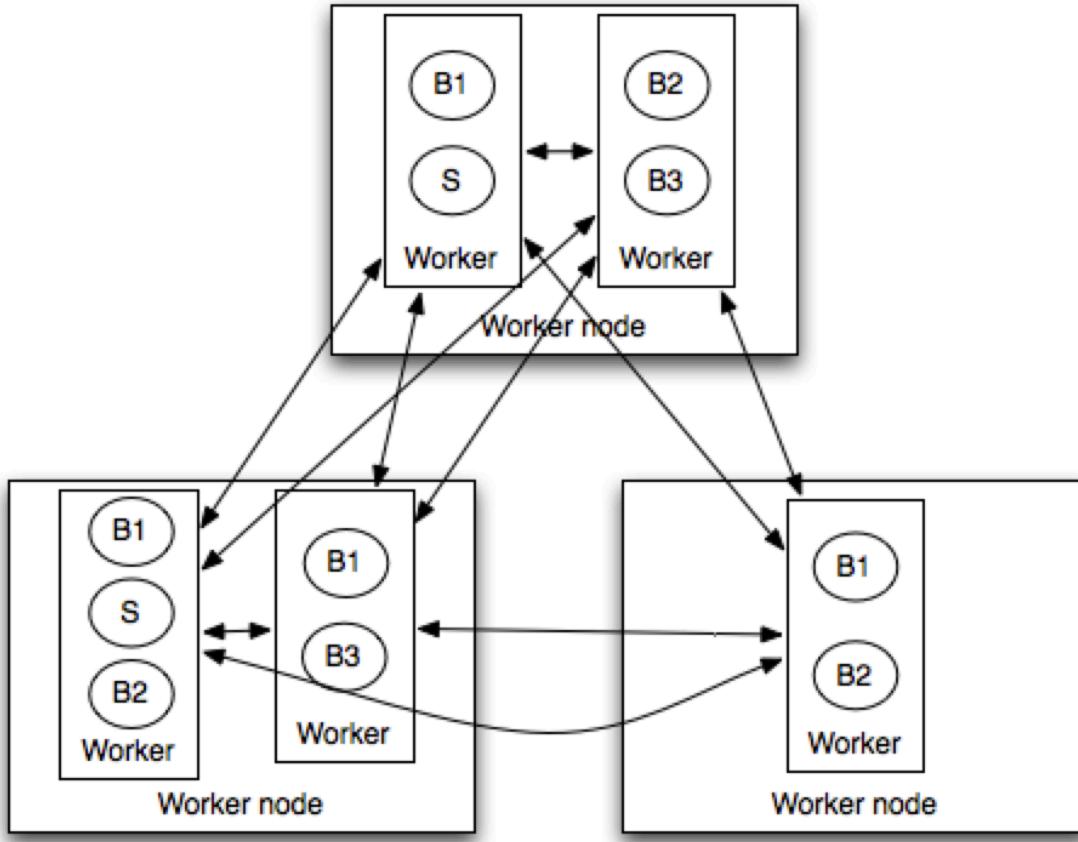


Figure 9.13 Physical view of topology tasks

The fact that spouts and bolts run in parallel brings up a key question: when a task emits a tuple, to which of the consuming tasks should it send the tuple to? The user provides the answer by providing something called a "stream grouping". A stream grouping tells Storm how to partition tuples in a stream among consuming tasks.

The simplest kind of stream grouping is called a "shuffle" grouping, which distributes tuples among the consuming tasks using a random round robin algorithm. This algorithm randomly distributes the tuples but ensures all consumers receive an equal amount of tuples. This has the effect of evenly distributing the load of processing. A more interesting kind of grouping is called a "fields grouping" which distributes tuples by taking a hash of a subset of the tuple fields and then modding by the number of consuming tasks. So if you did a fields grouping by the "word" field, all tuples with the same "word" field would go to the

same task, but different words would be distributed across all the consuming tasks. If you go back to the topology diagram, to complete the picture every subscription edge is annotated with the kind of stream grouping used, as shown in Figure 9.14.

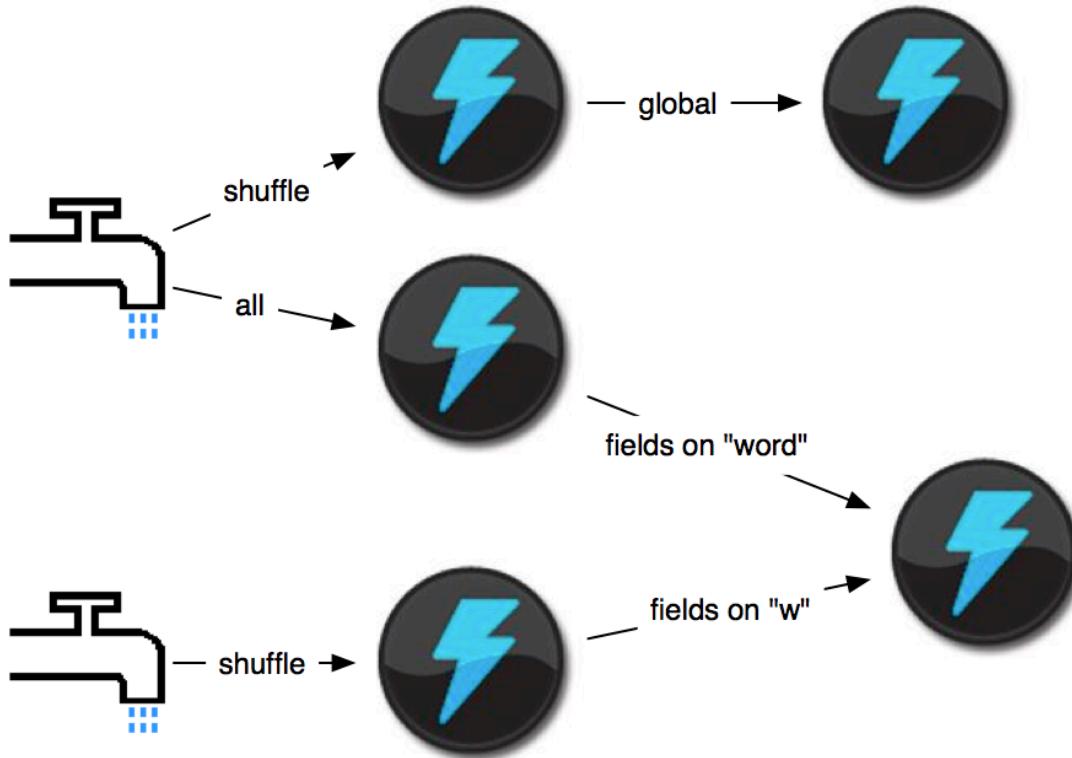


Figure 9.14 Topology with stream groupings

Let's solidify this example by looking at a basic example of a Storm topology. Like how word count is the de facto MapReduce example, let's see how to implement a streaming version of word count on top of Storm. We'll start by wiring up all the spouts and bolts into a topology and then look at the implementation of those spouts and bolts.

First, you create a `TopologyBuilder` object that exposes a nice Java API for defining topologies:

```
TopologyBuilder builder = new TopologyBuilder();
```

Next, let's add a spout that will emit a stream of sentences:

```
builder.setSpout(
    "sentence-spout",
```

```
new RandomSentenceSpout(),
8);
```

This spout is named "sentence-spout" and is given a parallelism of 8. So 8 tasks will be spawned across the cluster to execute the spout.

Now let's add a bolt that consumes a stream of sentences and transform it into a stream of words:

```
builder.setBolt("splitter", new SplitSentence(), 12)
    .shuffleGrouping("sentence-spout");
```

This bolt is called "splitter" and has a parallelism of 12. This bolt declares that it consumes all the output from the "sentence-spout" using a shuffle grouping. So the load of splitting sentences will be evenly distributed across all 12 tasks.

The last bolt consumes a stream of words and produces a stream of word counts:

```
builder.setBolt("count", new WordCount(), 12)
    .fieldsGrouping("splitter", new Fields("word"));
```

This bolt is called "counter" and has a parallelism of 12. This bolt declares that it consumes all the output from the "splitter" bolt using a fields grouping on the "word" field. You might be wondering why it uses a fields grouping rather than a shuffle grouping. If it used a shuffle grouping, then the same word can go to multiple tasks. So no task will have a complete view of the number of times that word has appeared. If the counts are kept in memory, then no task will be able to determine the correct count. If the counts are kept in a database, then there will be race conditions on updating the counts. By doing a fields grouping on the word field, this ensures that only one task is responsible for any particular word, while still distributing the load of processing to all tasks.

Here's the implementation of the "splitter" bolt. As you can see, it's extremely simple. It grabs the sentence from the first field of the tuple and emits a new tuple for every word in the sentence. It also declares the names of the fields for the stream it emits.

```
public static class SplitSentence extends BaseBasicBolt {
    public void execute(
        Tuple tuple, BasicOutputCollector collector) {
```

```

        String sentence = tuple.getString(0);
        for(String word: sentence.split(" ")) {
            collector.emit(new Values(word));
        }
    }
    public void declareOutputFields(
        OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}

```

Here's the implementation of the "counter" bolt. This particular implementation just keeps the word counts in an in-memory hashmap, but you could easily have this communicate with a database instead.

```

public static class WordCount extends BaseBasicBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();
    public void execute(
        Tuple tuple, BasicOutputCollector collector) {
        String word = tuple.getString(0);
        Integer count = counts.get(word);
        if(count==null) count = 0;
        count++;
        counts.put(word, count);
        collector.emit(new Values(word, count));
    }
    public void declareOutputFields(
        OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}

```

Finally, let's take a look at a spout implementation. You could easily just use one of the pre-built spout implementations to read the sentences off of a queue like Kafka, but instead lets look at implementing a spout that emits a stream without talking to any external source:

```

public static class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;
    public void open(
        Map conf,
        TopologyContext context,
        SpoutOutputCollector collector) {
        _collector = collector;
        _rand = new Random();
    }
    public void nextTuple() {
        Utils.sleep(100);

```

```

        String[] sentences = new String[] {
            "the cow jumped over the moon",
            "an apple a day keeps the doctor away",
            "four score and seven years ago",
            "snow white and the seven dwarfs",
            "i am at two with nature"};
        String sentence = sentences[_rand.nextInt(sentences.length)];
        _collector.emit(new Values(sentence));
    }
    public void declareOutputFields(
        OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}

```

This spout emits a random one of its sentences every 100ms, creating an infinite stream of sentences. Storm calls the nextTuple method in a loop.

9.3.2 Guaranteeing message processing

One of the problems with the queues and workers model was the need for intermediate queues between every stage of processing. One of the beauties of the Storm model is that it eliminates this need. The key to how Storm accomplishes this is by keeping track of something called a "tuple DAG" (tuple directed acyclic graph). Take a look at the example tuple DAG in Figure 9.15 that could be produced from the streaming word count topology.

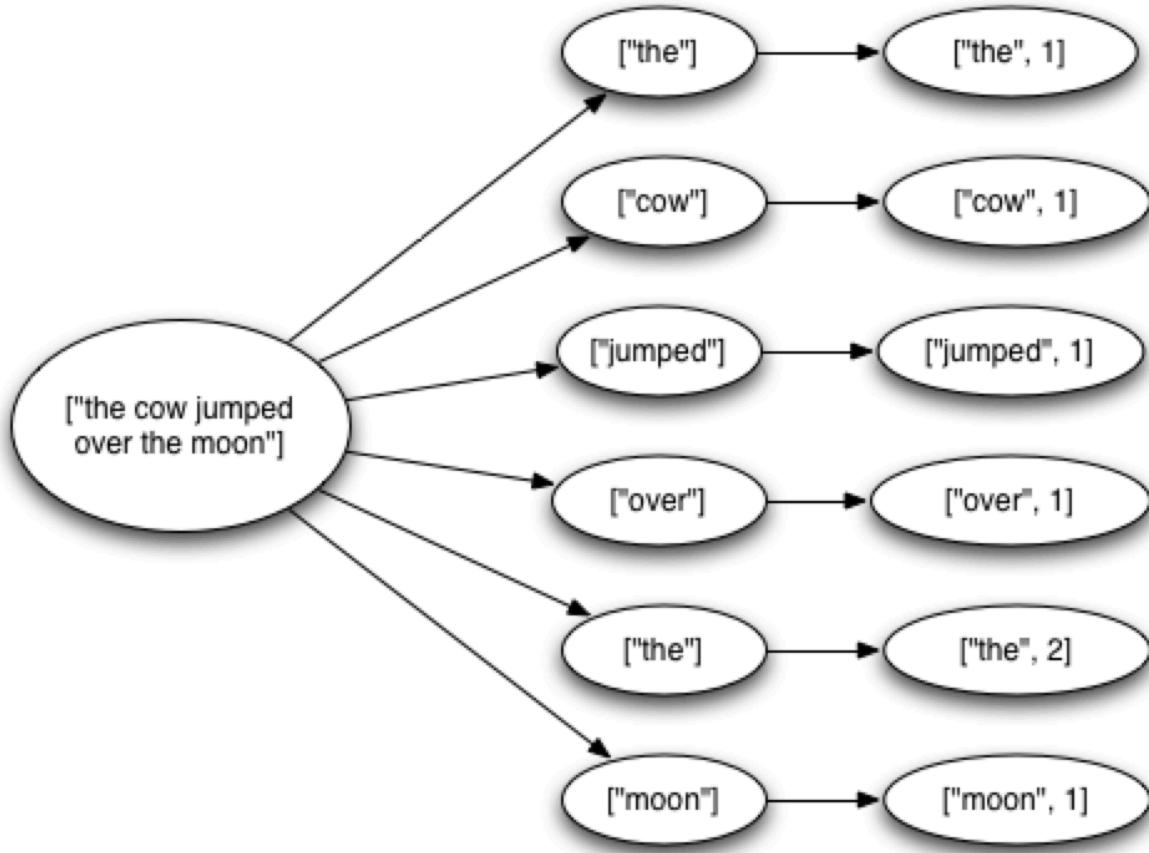


Figure 9.15 Tuple DAG

A tuple DAG tracks dependencies between tuples, each edge showing which tuples were based on which other tuples. For example, the spout emits a sentence, which then creates a tuple for each word. Each of those word tuples is based on that sentence tuple. Then each word tuple goes on to create a new word count tuple. You could imagine these tuple DAGs getting to be very large if there's a lot of processing happening.

Storm's message processing guarantee all has to do with the spout tuples. If the tuple DAG is not completed – meaning all nodes created and each node marked as processed – within a pre-specified timeout, the spout tuple at the root of the DAG is considered failed and is retried. So unlike queues and workers, where retries happen in the middle of the topology, in Storm retries always happen from the source. Then the spout tuple will be replayed, and the whole tuple DAG will be regenerated and hopefully succeed in completing the next time through.

This may seem to be a step backward. By retrying from the source, stages that had successfully completed will be retried again. But in actuality, this still isn't any

different than before. With queues and workers, a stage could succeed in processing, fail right before taking the message off the queue, and then retry again. So the processing guarantee is still an at-least once processing guarantee. As you'll see in the next chapter, exactly-once semantics can be achieved by building on top of this at least once guarantee – and at no point are intermediate queues needed.

You have to do two things as a user to take advantage of this message processing guarantee. You have to tell Storm whenever you're creating one of those dependency edges, called anchoring, and you have to tell Storm when you're finished with a tuple. Finishing a tuple is called "acking". Let's look at the sentence splitter code from streaming word count with all the tuple DAG logic in it:

```
public static class SplitSentenceExplicit extends BaseRichBolt {
    OutputCollector _collector;
    public void prepare(
        Map conf,
        TopologyContext context,
        OutputCollector collector) {
        _collector = collector;
    }
    public void execute(Tuple tuple) {
        String sentence = tuple.getString(0);
        for(String word: sentence.split(" ")) {
            _collector.emit(tuple, new Values(word));
        }
        _collector.ack(tuple);
    }
    public void declareOutputFields(
        OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

When the new tuple is emitted, the tuple it's based on is included as the first argument. This anchors the new tuple to the other tuple. After the new tuples are emitted, the sentence tuple is then acked since all processing with it is finished.

The logic of this bolt is actually identical to the original implementation in the previous section. Bolts that extend a "BasicBolt" class automatically anchor all outgoing tuples to the input tuple and then ack the input tuple at the end. This is a very common pattern.

The cases where you wouldn't do this are when you're doing some sort of aggregation or joining. Perhaps you're waiting for 100 tuples to process them all

together. In those cases you could hold on to all 100 tuples in a buffer, and then anchor the output tuple to all 100 tuples. Then, you would ack the 100 tuples. This is illustrated by the following code which emits the sum of every 100 tuples.

```
List<Tuple> _buffer = new ArrayList<Tuple>();
int _count = 0;
public void execute(Tuple tuple) {
    _count+=tuple.getInteger(0);
    if(_buffer.size() < 100) {
        _buffer.add(tuple);
    } else {
        _collector.emit(_buffer, new Values(_count));
        _buffer.clear();
        _count = 0;
    }
}
```

You might be wondering about the efficiency of tracking tuple DAGs. It seems quite expensive – especially if the tuple DAGs get to be very large. It might seem like tracking a tuple DAG containing thousands or millions of tuples might use a lot of memory. As it turns out, there's a way to track tuple DAGs by not explicitly keeping track of the DAG – only about 20 bytes of space are needed. This is true regardless of how big the tuple DAG is – it could even have trillions of tuples in it and only 20 bytes of space would be needed. We won't get into the algorithm here, though it is documented extensively in Storm's online documentation. The important takeaway is that the algorithm is very efficient, and that efficiency makes it a practical way to track failures and initiate retries during stream processing.

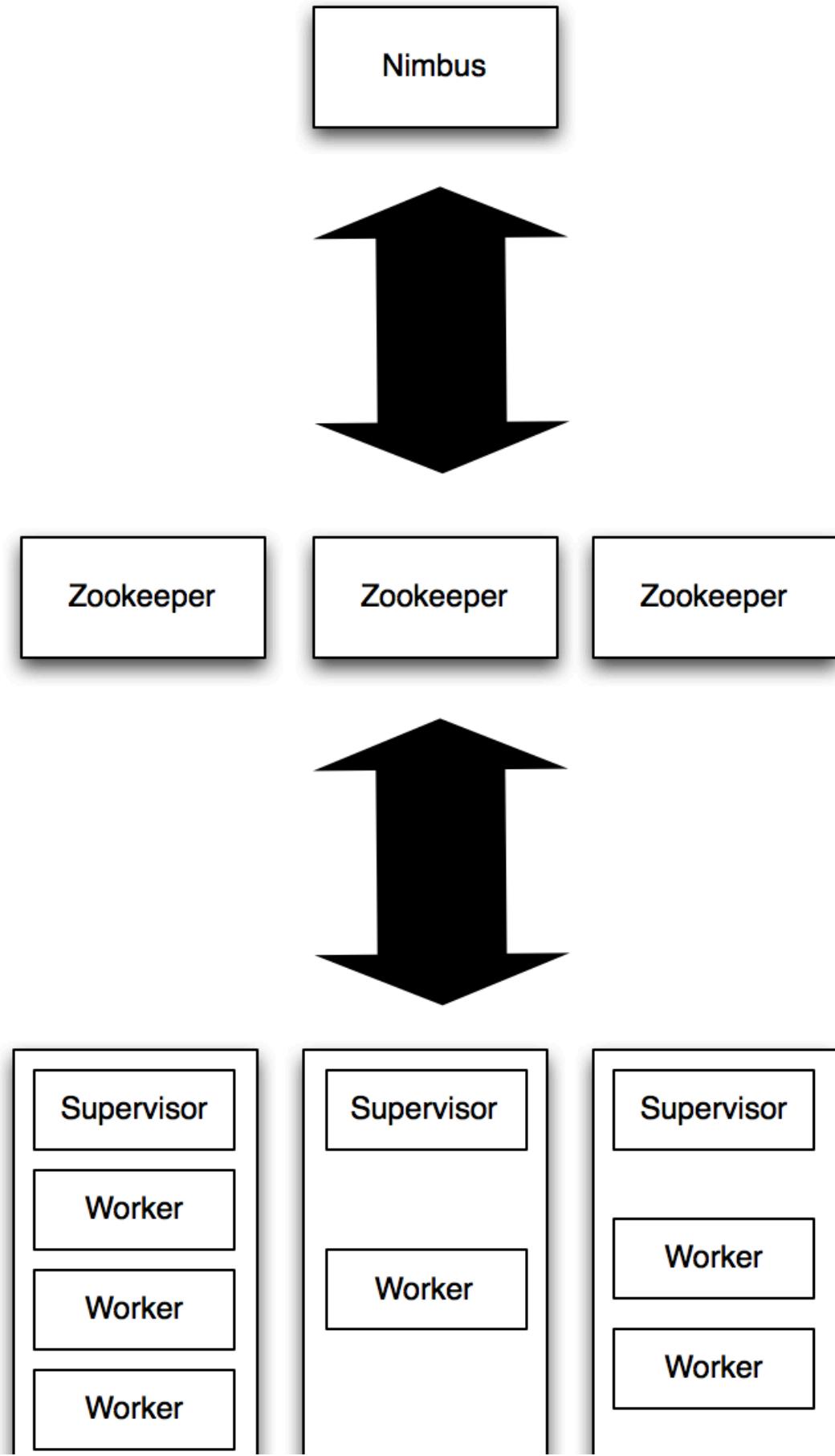
Finally, let's discuss a few more things about the at-least once guarantees that you have with the Storm model. If the operations in a topology are idempotent – that is, applying them more than once for the same tuple will not change the result, then that topology will have exactly-once semantics. An example of this is doing a "set" operation based on a timestamp in the tuple – only set the value to the new one if the tuple's timestamp is greater than what's stored in the database. Another example of an idempotent operation is adding an element to a set. No matter how many times you do that, you'll get the same result.

Another thing to keep in mind is that you might just not care about a little inaccuracy when you're using non-idempotent operations. Failures are relatively rare, so the inaccuracy should be small. And since the serving layer replaces the speed layer anyway, any inaccuracy will be automatically corrected. By sacrificing

some latency, you can achieve exactly-once semantics (as will be discussed in the next chapter), but if lower latency is more important to you than temporary inaccuracy, then using non-idempotent operations with the Storm model is a fine tradeoff.

9.4 Using Apache Storm

Apache Storm is an open-source project that originated the Storm model. The code examples you've seen utilize the Storm API, so there's not that much left to understand how to use it. After getting a grasp of using Storm, you'll see some practical examples of using Storm and Cassandra together to implement part of the speed layer for SuperWebAnalytics.com.



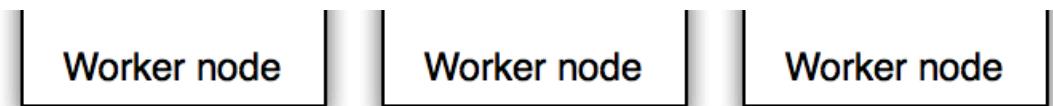


Figure 9.16 Apache Storm architecture

The architecture of a Storm cluster is outlined in Figure 9.16. Storm has a master node called "Nimbus" that manages running topologies. Nimbus is where you submit topologies for execution, and Nimbus will assign workers around the cluster to run that topology. Nimbus is responsible for detecting when workers die and reassigning them to other machines if necessary.

In the center of that diagram is "Zookeeper". Zookeeper is another Apache project that is good at keeping small amounts of state and has semantics perfect for cluster coordination. Zookeeper is where Storm keeps information about where workers are assigned and other topology configuration information. The typical Zookeeper cluster size for a Storm cluster is 1 to 5 nodes.

Then you have all your worker nodes. Each worker node runs a daemon called the "Supervisor" that communicates with Nimbus through Zookeeper to determine what should be running on the machine. The Supervisor then starts or stops workers as necessary as dictated by Nimbus.

Lastly, on each machine you have worker processes. Each worker runs a subset of a single topology. Workers discover the location of other workers through Zookeeper and pass messages to each other directly.

Let's see how to deploy the word count topology that we constructed in the previous section. The code to do this looks like this:

```
Config conf = new Config();
conf.setNumWorkers(3);
StormSubmitter.submitTopology(
    "word-count-topology",
    conf,
    builder.createTopology());
```

First a "topology configuration" is set up to make any configurations that apply to the topology as a whole. The only configuration set up here is the number of workers in the topology. This is set to 4 in this code sample. This means there will be 4 workers spawned around the cluster to run the topology. When defining the topology, you saw that a parallelism was included for each spout and bolt: the sentence spout had a parallelism of 8, the splitter bolt had a parallelism of 12, and

the counter bolt also had a parallelism of 12. So the total parallelism for the topology is 32. The parallelism at the spout and bolt level indicates the number of threads that should be spawned for that spout or bolt. In contrast, setting the number of workers sets the number of actual Java processes for the topology. By default, Storm evenly distributes tasks across the workers, and evenly distributes workers across the cluster, but this is all completely customizable by plugging in a new scheduler into Nimbus.

Another important aspect of stream processing is flow control. If there's a burst of incoming events, it's important that the resources of your stream processor don't get overwhelmed and crash (e.g. running out of memory). Storm has a simple mechanism for managing flow control based on its guaranteed message processing features. A configuration setting called "topology max spout pending" controls the maximum amount of tuples that can be emitted but un-acked on spout tasks. Once this limit is reached, spout tasks will stop emitting tuples until tuples either get acked, fail, or time out. Here's an example of setting this config:

```
conf.setMaxSpoutPending(1000);
```

This code tells Storm that the most number of tuples that can be pending on any one spout task is 1000. If the spouts have a combined parallelism of 20, then the most number of pending tuples in the entire topology is 20,000.

Now that you understand the basics of using Storm, let's see how to use Storm to implement part of the SuperWebAnalytics.com speed layer.

9.5 SuperWebAnalytics.com speed layer

Recall that there are three separate queries we're implementing for SuperWebAnalytics.com:

- Number of pageviews over a range of hours
- Unique number of pageviews over a range of hours
- Bounce rate for a domain

We will implement the "unique number of pageviews" query in this section and the other two queries in the next chapter.

The goal of this query is to be able to get the number of unique visitors for a URL over a range of hours. Recall that when implementing this query for the batch/serving layers the HyperLogLog algorithm was used for efficiency:

HyperLogLog produces a compact set representation that can be merged with other sets, making it possible to compute the uniques over a range of hours without having to store the set of visitors for every hour. The tradeoff is that HyperLogLog is an approximate algorithm, so the counts will be off by a small percentage. However, the space savings are so enormous it was an easy tradeoff to make, since perfect accuracy is not needed for SuperWebAnalytics.com. The same tradeoffs exist in the speed layer, so we will make use of HyperLogLog for the speed layer version of uniques over time.

Also recall that SuperWebAnalytics.com is quite intricate and can track visitors beyond just IP addresses and also takes into account login information. So if a user is logged in on both her phone and computer, visits the page on the computer and 10 minutes later visits the same page on her phone, that should only count as a single visitor. In the batch layer this was accounted for by using the "equiv" edge relationships to keep track of which identifiers represented the same person and then normalizing all identifiers for one person into a single identifier.

Handling the multiple identifier problem in the speed layer is much more complex than handling it in the batch layer. In the batch layer, all the equiv analysis could happen at once before the uniques over time computation ever took place. In the speed layer, an equiv edge could come in **after** the two separate visitors are accumulated. For example, consider the following sequence of events:

- IP address "11.11.11.111" visits "foo.com/about" at 1:30pm
- User "sally" visits "foo.com/about" at 1:40pm
- An equiv edge between "11.11.11.111" and "sally" is discovered at 2:00pm

So to have the most accurate stats possible, the speed layer would have to reduce the uniques for that hour by one.

Let's now consider what it would actually take from a computational standpoint to do this in realtime. First of all, you would need to keep track of the graph of equivs in realtime, meaning that the entire graph analysis performed in chapter 6 would have to be done incrementally. Second, you need to be able to go back and discover if the same visitor was counted multiple times. This means you have to store the entire set of visitors for every hour – HyperLogLog is a compact representation of a unique count and is unable to help with this. So handling the equiv problem completely precludes you from taking advantage of HyperLogLog

in the first place. Besides all of that, of course, the incremental algorithm to do the equiv graph analysis and adjust already computed unique counts seems rather complex.

There are other options available. Rather than strive for doing the perfect uniques over time computation, let's see what can be accomplished by trading off accuracy. Remember, one of the benefits of the Lambda Architecture is that you can tradeoff accuracy in the speed layer without trading off accuracy permanently – since the serving layer continuously overrides the speed layer, these inaccuracies go away and the system exhibits "eventual accuracy". After looking at how much inaccuracy these alternative approaches will introduce and judging that against the value of computational and complexity benefits, we can decide on the appropriate approach to take. Of course, we're already trading off accuracy by using HyperLogLog, so we'll see what can be accomplished by trading off additional accuracy.

The first alternative approach would be to not do the equiv analysis in realtime. Instead, the results of the batch analysis can be made available to the speed layer through a key/value serving layer database with identifier keys and normalized identifier values. In the speed layer, identifiers are first normalized through that database before the uniques over time computation is done. Of course, the precomputed equiv analysis will be out of date by a few hours, so any equivilents arriving in the last few hours will not be accounted for.

The benefit of this approach is that you no longer have to undo prior computations, allowing you to take advantage of HyperLogLog. This is massively simpler to implement and also much less resource intensive.

Now let's consider where this approach will be inaccurate. It's only inaccurate for cases where a user navigates to a page, registers for a user id (upon which an equiv edge is captured), and then navigates to the same page in the same hour from a different IP address (like a different device). In this case the relationship between the IP addresses and the user id are not accounted for. Note however that the inaccuracy only happens with **brand new** users – if the person has been registered for a few hours (or however long the batch cycle is), then the user id normalization will still happen and that person's visit will be properly recorded. Overall this seems like a very slight amount of inaccuracy to tradeoff for big savings.

Let's go even further. Another approach is to ignore equivilents completely and calculate realtime uniques solely based on individual identifiers. This simplifies the implementation further at the cost of increasing inaccuracy. Now cases like a

person visiting a page, logging in, and then visiting the same page within the same hour won't prevent those visits from being recorded as additional uniques.

The right thing to do is to run batch analyses to actually quantify how much inaccuracy the various approaches actually generate. Then you can make an informed judgement as to the best approach to take. Intuitively, it seems like even the approach of ignoring equivilents completely in the speed layer won't introduce that much inaccuracy, so in the interest of keeping examples as simple as possible that's the approach we'll demonstrate. If implementing this in real life though the right thing to do is to quantify the different approaches – and continue to quantify them even after implementation – so that you can make sure the approach taken is always optimal. Let's also emphasize again that this inaccuracy is temporary – the entire system as a whole is eventually accurate.

9.5.1 Topology structure

Let's now implement the uniques over time speed layer by ignoring equivilents. There are three steps to this:

- Consume a stream of pageview events that contain a user identifier, a URL, and a timestamp.
- Filter out any invalid URLs
- Update a Cassandra database containing a nested map from domain to hour to a HyperLogLog set.

Figure 9.17 illustrates a topology structure to implement this.

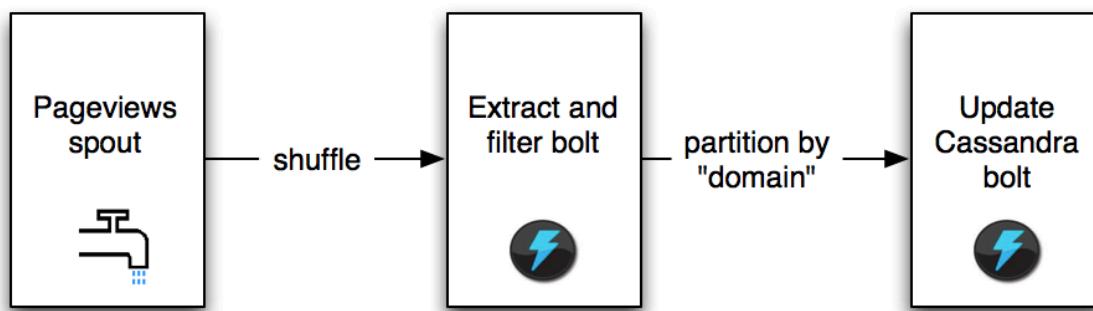


Figure 9.17 Uniques over time topology

A key aspect of this topology is partitioning tuples in the Cassandra bolt by 'domain' – this ensures that only one thread is updating the unique counts for a domain at a time, avoiding any race conditions with the database.

9.5.2 Topology implementation

To implement the topology, let's start with the spout. Here's the code to initialize a Kafka spout (from the storm-kafka library) to read the pageviews from a cluster of Kafka servers. The pageviews are assumed to be stored on Kafka as the Thrift 'Data' objects defined in Chapter 2.

```
TopologyBuilder builder = new TopologyBuilder();
SpoutConfig spoutConfig = new SpoutConfig(
    new KafkaConfig.ZkHosts("zkserver:1234", "/kafka"),
    "pageviews",
    "/kafcastorm",
    "uniquesSpeedLayer"
);
spoutConfig.scheme = new PageviewScheme();
builder.setSpout("pageviews",
    new KafkaSpout(spoutConfig), 16);
```

Most of this code is setup to configure the location of the Kafka cluster as well as where the spout should record in Zookeeper what it has consumed so far from Kafka. Note the setting of a scheme in the spout configuration to tell the spout how to reserialize binary records into full-fledged Data objects.

The next step is extracting the URL, domain, timestamp, and user identifier from the pageview event and filter out any malformed URLs. The code to do this is as follows:

```
public static class ExtractFilterBolt extends BaseBasicBolt {
    private static final int HOUR_SECS = 60 * 60;
    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        PersonID user = (PersonID) tuple.getValue(0);
        String url = tuple.getString(1);
        int timestamp = tuple.getInteger(2);
        try {
            String domain = new URL(url).getAuthority();
            collector.emit(new Values(
                domain,
                url,
                timestamp / HOUR_SECS,
                user));
        } catch(MalformedURLException e) {
        }
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("domain", "url", "bucket", "user"));
    }
}
```

```

    }
}
```

Finally, the last step is to update the HyperLogLog sets stored in Cassandra. Let's start with a simple version of this:

```

public static class UpdateCassandraBolt extends BaseBasicBolt {
    ColumnFamilyTemplate<String, Integer> _template;
    @Override
    public void prepare(
        Map conf,
        TopologyContext context) {
        Cluster cluster = HFactory.getOrCreateCluster(
            "mycluster", "127.0.0.1");
        Keyspace keyspace = HFactory.createKeyspace(
            "superwebanalytics", cluster);
        _template =
            new ThriftColumnFamilyTemplate<String, Integer>(
                keyspace,
                "uniques",
                StringSerializer.get(),
                IntegerSerializer.get());
    }
    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String url = tuple.getString(1);
        int bucket = tuple.getInteger(2);
        PersonID user = (PersonID) tuple.getValue(3);
        HColumn<Integer, byte[]> hcol =
            _template.querySingleColumn(
                url,
                bucket,
                BytesArraySerializer.get());
        HyperLogLog hll;
        try {
            if(hcol==null) hll = new HyperLogLog(800);
            else hll = HyperLogLog.Builder.build(hcol.getValue());
            hll.offer(user);
            ColumnFamilyUpdater<String, Integer> updater =
                _template.createUpdater(url);
            updater.setByteArray(bucket, hll.getBytes());
            _template.update(updater);
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
    }
}
```

This code retrieves the HyperLogLog set corresponding to that pageview,

updates the set, and then writes the set back into Cassandra.

Finally, for completeness, here's the rest of the code wiring the topology together:

```
builder.setBolt("extract-filter",
    new ExtractFilterBolt(), 32)
    .shuffleGrouping("pageviews");
builder.setBolt("cassandra",
    new UpdateCassandraBolt(), 16)
    .fieldsGrouping("extract-filter",
        new Fields("domain"));
```

Note that the topology shown so far is completely fault-tolerant. Since spout tuples are only considered acked until after the database has been updated, any failure will cause that spout tuple to be replayed. And since adding to a HyperLogLog set is an idempotent operation, there won't be any overcounting if the same item is added to the same set multiple times. So failures/retries do not affect the accuracy of the system.

The problem with the Cassandra code shown that there's a lot of overhead with retrieving sets from Cassandra and writing them back. It would be more efficient to batch updates together, especially if the same set can be updated multiple times at once. The following code shows a template for this batching approach, writing to Cassandra every hundred tuples or once per second, whichever comes first:

```
public static class UpdateCassandraBoltBatched
    extends BaseRichBolt {
    List<Tuple> _buffer = new ArrayList();
    OutputCollector _collector;
    @Override
    public void prepare(
        Map conf,
        TopologyContext context,
        OutputCollector collector) {
        _collector = collector;
        // set up Cassandra client here
    }
    @Override
    public void execute(Tuple tuple) {
        boolean flush = false;
        if(tuple.getSourceStreamId().equals(
            Constants.SYSTEM_TICK_STREAM_ID)) {
            flush = true;
        } else {
            _buffer.add(tuple);
            if(_buffer.size()>=100) flush = true;
        }
    }
}
```

```
        if(flush) {
            // batch updates to Cassandra here
            for(Tuple t: _buffer) {
                _collector.ack(t);
            }
        }
    }

@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
}

@Override
public Map<String, Object> getComponentConfiguration() {
    Config conf = new Config();
    conf.put(Config.TOPOLOGY_TUPLE_FREQ_SECS, 60);
    return conf;
}
```

A key aspect of this code is that tuples are buffered and not acked until after the corresponding updates have been batched into Cassandra. This ensures that replays will happen if there are any failures. In order to implement the "once per second" updates, a Storm feature called tick tuples are used. A tick tuple is set up in "getComponentConfiguration" that will be fed to the bolt once per second. When one of those tuples comes in, whatever is currently buffered is written to the database. We've left out the Cassandra portions of the code because it's somewhat hairy and distracts from the stream processing aspects of the code.

There's a lot more work that could be done to make this code even more efficient. Consider the following list of possible optimizations:

- A batch computation could be done to estimate the size of the HyperLogLog set needed for different domains (domains with more uniques need larger HyperLogLog sets). Most domains need very small HyperLogLog sets, so there's a lot of savings to be had by knowing that in advance.
 - You could implement a custom scheduler for your Storm cluster so that the Cassandra bolt tasks are co-located with the Cassandra partitions they update. This would eliminate network transfer between the updater tasks and cassandra.
 - If Cassandra were to implement HyperLogLog natively, then the HyperLogLog sets wouldn't have to be transferred back and forth.

Implementing all these optimizations are beyond the scope of this book – these are just suggestions on techniques that can be used to improve this particular speed layer.

9.6 Conclusion

You should now have a good understanding of all the pieces of the speed layer – queues, stream processors, and realtime views. The speed layer is by far the most complex part of any architecture due to its incremental nature, and contrasting the incremental code in this chapter with the batch code in previous chapters demonstrates this.

What's left in learning the speed layer is to learn micro-batched stream processing, the other paradigm of stream processing besides one-at-a-time stream processing that was mentioned. Micro-batched stream processing makes a few tradeoffs compared to one-at-a-time processing, such as sacrificing latency, but enables some powerful things such as exactly-once processing semantics for a much more general set of operations.

Speed layer: micro-batch stream processing

10

This chapter covers:

- Exactly-once processing semantics
- Micro-batch processing and its tradeoffs
- Implementing micro-batch processing with Trident

You've learned the main concepts of the speed layer in the last two chapters: realtime views, incremental algorithms, stream processing, and how all those fit together. There are no more fundamental concepts to learn about the speed layer – instead, in this chapter, we will focus on a different method of stream processing that makes certain tradeoffs to get benefits like improved accuracy and higher throughput.

The one-at-a-time stream processing you've learned is very low latency and simple to understand. However, it can only provide an at-least-once processing guarantee during failures. While this does not affect accuracy for certain operations, like adding elements to a set, it does affect accuracy for other operations such as counting. In many cases, this inaccuracy is unimportant because the batch layer overrides the speed layer, making that inaccuracy temporary. However, there are other cases where you want full accuracy all of the time and temporary inaccuracy is unacceptable. In those cases, micro-batch stream processing can give you the fault-tolerant accuracy you need, at the cost of higher latency on the order of hundreds of milliseconds to seconds.

After diving into the ideas underlying micro-batch stream processing, you'll see how to implement micro-batching with Storm's Trident API. Then we'll use Trident to finish the speed layer for SuperWebAnalytics.com.

10.1 Achieving exactly-once semantics

With one-at-a-time stream processing, tuples are processed relatively independent of each other. Failures are tracked at an individual tuple level, and replays also happen at an individual tuple level. The world of micro-batch processing is different. Instead, small batches of tuples are processed at a time, and if anything in a batch fails, the entire batch is replayed. In addition, strict ordering is enforced in how the batches are processed. This approach allows you to make use of new techniques in order to achieve exactly-once semantics in your processing, rather than relying on inherently idempotent functions as one-at-a-time processing does. Let's see how this works.

10.1.1 Strongly ordered processing

Suppose you just want to compute a count of all tuples in realtime, and you want that count to be completely accurate regardless of how many failures you sustain during processing. To do this with one-at-a-time processing, your code would look something like this (pseudo-code):

```
process(tuple) {
    counter.increment()
}
```

What happens when there are failures? Tuples will be replayed, but when it comes time to increment the count you have no idea if that tuple was processed already or not. It's possible you incremented the count but then crashed immediately before acking the tuple. The only way to know is if you were to store the id of every tuple you've processed – but now you'd be storing a huge amount of state instead of just a single number. So that's not a very viable solution.

The key to achieving exactly-once semantics is to enforce a strong ordering on the processing of the input stream. Let's see what happens when you only process one tuple at a time from the input stream, and you don't move onto the next tuple until the current one is successfully processed. Of course, this isn't a scalable solution, but it illustrates the core idea behind micro-batch processing. In addition, let's assume that every tuple has a unique id associated with it that's always the same no matter how many times it's replayed.

The key idea is rather than just store a count, store the count along with the id

of the latest tuple processed. Now, when updating the count, there are two cases:

1. The stored id is the same as the current tuple id: In this case, we know that the count already reflects the current tuple, so we do nothing.
2. The stored id is different from the current tuple id: In this case, we know that the count does not reflect the current tuple. So we increment the counter and update the stored id. The keys here are that tuples are processed in order and that the count/id are updated atomically.

This update strategy is resilient to all failure scenarios. If the processing fails after updating the count, then the tuple will be replayed and the update will be skipped the second time around. If the processing fails before updating the count, then the update will occur the second time around.

10.1.2 Micro-batch stream processing

As mentioned though, processing one tuple at a time is highly inefficient. So a better approach is to process the tuples as discrete batches, as illustrated in Figure 10.1. This is known as micro-batch stream processing.

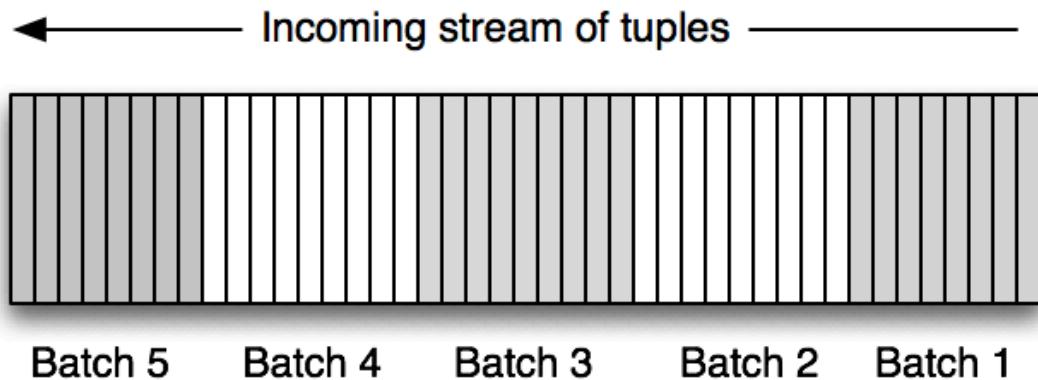


Figure 10.1 Tuple stream divided into batches

Just like before, these batches are processed in order, and each batch has a unique id which is always the same on every replay. However, now that many tuples are processed per iteration rather than just one, the processing can be parallelized and thus made scalable. Finally, for simplicity's sake, let's assume that only one batch is processed at a time – it must be processed completely successfully before moving onto the next batch.

Let's see how the global counting example works with micro-batch processing. Again, rather than just storing the count, the count is stored along with the latest

batch id involved in updating that count. For example, suppose the current value of the stored state is as shown in Figure 10.2.

count	112
batchid	3

Figure 10.2 Count state including batch id

Now suppose batch 4 comes in with 10 tuples. The state will be updated to look like Figure 10.3.

count	122
batchid	4

Figure 10.3 Result of updating count state

Now let's say after updating the state in the database, something fails in the stream processor and the message that the batch was finished never gets completed. The stream processor will then retry batch 4. Now when it comes time to update the state in the database, it sees that the state has already been updated by batch 4. So rather than increment the count again, it does nothing and moves on to the next batch.

Now let's look at a more complicated example than global counting to further drive home how micro-batch stream processing works.

10.1.3 Micro-batch processing topologies

Let's suppose you want to build a streaming application that consumes a massive stream of words and computes the top 3 most frequently occurring words. Micro-batch processing can accomplish this task as well, while being fully parallelized, fault-tolerant, and accurate.

There's two parts to this while processing each batch of words. First, you have to keep state on the frequency of each word. This can be done using a key/value

database. Then, if any of the words you just processed has a higher frequency than one of the current top 3 most frequent words, then the top 3 list must be updated.

Let's start with updating the word frequencies. Just like how MapReduce and one-at-a-time stream processing partition data and process each partition in parallel, the same is done with micro-batched processing. So processing a batch of words looks like Figure 10.4. As you can see, a single batch includes tuples from all partitions in the incoming stream.

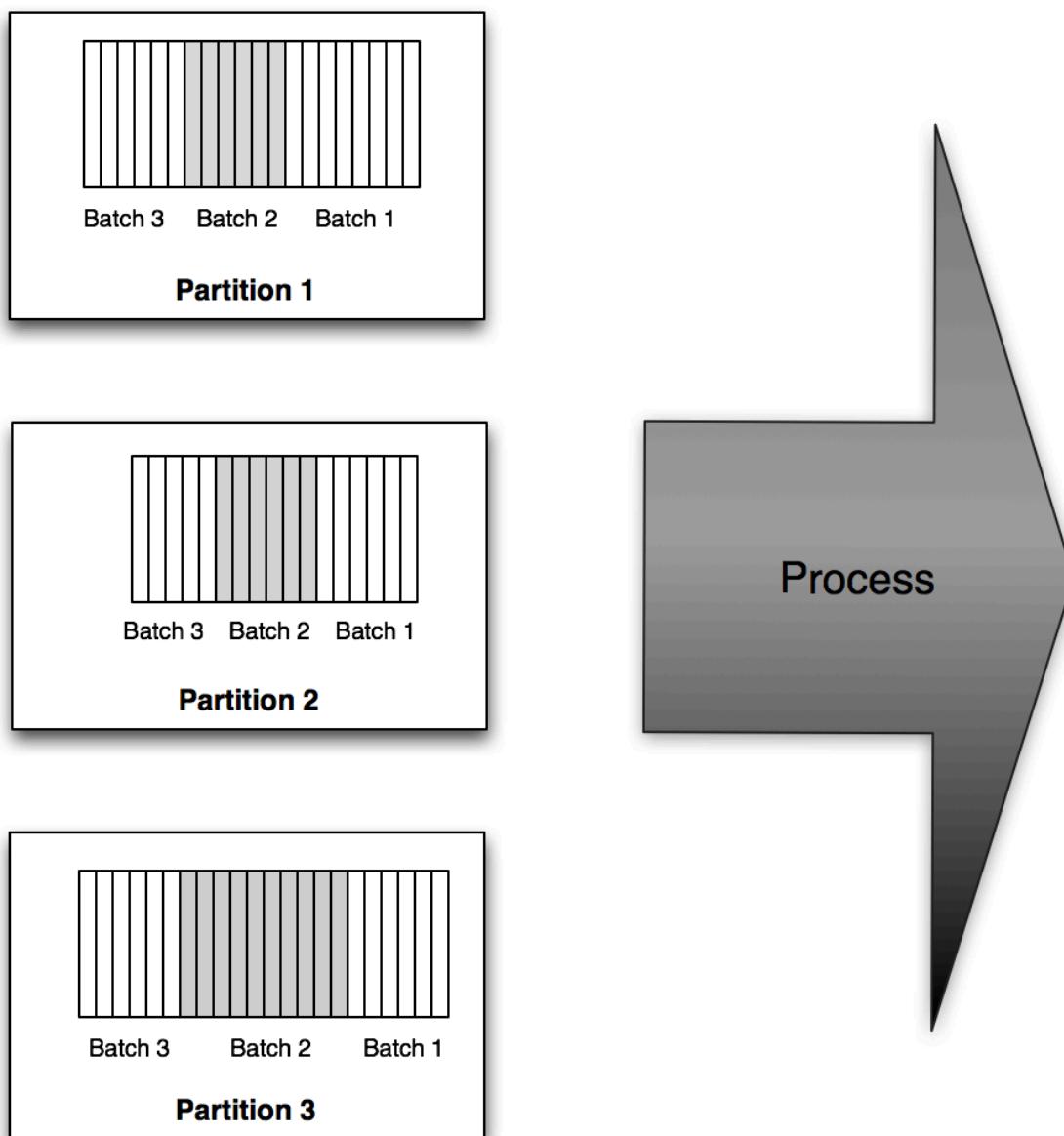


Figure 10.4 Each batch includes tuples from all partitions of incoming stream

Now to update the word frequencies, the words must be repartitioned so that the same word is always processed by the same task. This ensures that when updating the database, only one thread will be updating the values for that word and there will be no race conditions. This is illustrated in Figure 10.5.

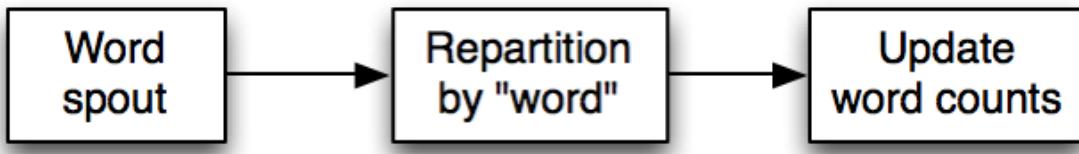


Figure 10.5 Word counting topology

Now you need to figure out what to store as the state for each word. Just like how storing just a count was not enough for global counting, it is also not enough to store just a count for each word. If a batch is retried you won't know if the current count reflects the current batch or not. So just as the solution for global counting was to store the batch id with the count, the solution for the word counts is to store the batch id with the count for each word, as illustrated in Figure 10.6.

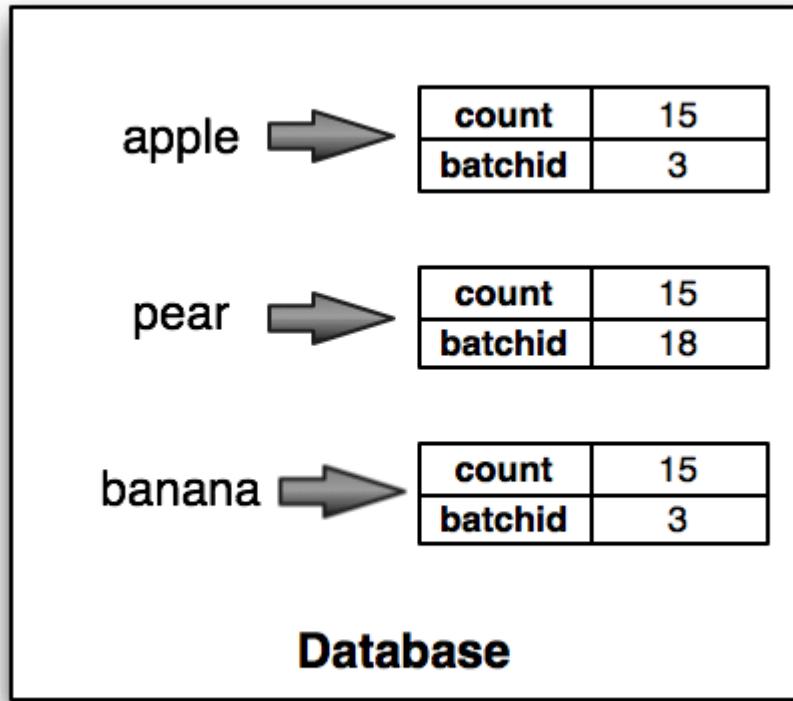


Figure 10.6 Storing word counts with batch id

Now let's consider a failure scenario. Suppose a machine dies in the cluster while a batch is being processed, and only some partitions update the database while others don't make it. So some words will have counts reflecting the current batch, while others won't be updated yet. When the batch is replayed, the words that have state including the current batch id won't be updated (because they have the same batch id as the current batch), while the words that haven't been updated yet will be updated like normal. So just like global counting, the processing is completely accurate and fault-tolerant.

Let's now move on to the second part of the computation: computing the top 3 most frequent words. One solution would be to send the new counts for every word to a single task and have that task merge those word counts into its top 3 list. The problem with this approach is that it's not scalable. The amount of tuples sent to that single top 3 task could be nearly the same size as the entire input stream of words.

Fortunately there's a better solution that does not have this bottleneck. Instead of sending every updated word count to the top 3 task, each word counting task can

compute the top 3 words in the current batch, and then send its top 3 list to the task responsible for the global top 3. The global top 3 task can then merge in all those lists into its own list. Now the amount of data transferred to that global top 3 task in each batch is proportional to the parallelism of the word counting tasks, not the entire input stream.

Now let's consider how failures might affect the top 3 portion of the computation. Suppose there's a failure that results in one of the top 3 lists not being sent to the global top 3 task. In that case, the global top 3 list will not be updated, and when the batch is replayed it will be updated normally.

Now suppose there's a failure after the top 3 list is updated (the message indicating the batch completed never gets through). In this case the batch will be replayed, and the same top 3 lists will be received by the global top 3 task. This time, the global top 3's list already incorporates the current batch. However, since merging those lists is an idempotent operation, re-merging them into an already updated list won't change the results. So the trick used before of including the batch id with the state is not needed in this case to achieve fully accurate processing.

10.2 Core concepts of micro-batch stream processing

From the examples shown some core concepts should be emerging. There are two main aspects to micro-batch stream processing:

- Batch-local computation: There's computation that occurs solely within the batch, not dependent on any state being kept. This includes things like repartitioning the word stream by the "word" field and computing the count of all the tuples in a batch.
- Stateful computation: Then there's computation which keeps state across all batches, such as updating a global count, updating word counts, or storing a top 3 list of most frequent words. This is where you have to be really careful about how you do state updates so that processing is idempotent under failures and retries. The trick of storing the batch id with the state is particularly useful here to force idempotence.

Finally, micro-batch stream processing relies on a stream source that can replay a batch exactly how it was played before. It turns out that queues like Kafka, covered in the last chapter, have the perfect semantics for this. Because Kafka exposes an API very similar to files, when a batch is emitted the consumer can remember which offsets from which partitions were read for a particular batch. Then, if a batch has to be replayed, the exact same batch can be emitted. These kinds of spouts are called "transactional spouts" and a Kafka implementation exists as an open-source project that manages all this bookkeeping for you.

SIDE BAR**Beyond transactional spouts**

There are also ways to achieve exactly-once processing semantics without transactional spouts. In a transactional spout, the exact same batch must be emitted on every replay. The problem with transactional spouts is that if a batch fails and a partition of the batch becomes unavailable, processing will be unable to continue because the batch can't be exactly replayed.

A less restrictive kind of spout, called "opaque spouts", must simply ensure that each tuple is successfully processed in only one batch. So it allows for the following sequence of events:

- Batch A is emitted with tuples from partitions 1, 2 and 3.
- Batch A fails to process.
- Partition 3 becomes unavailable.
- Batch A is replayed only with tuples from partitions 1 and 2.
- Sometime later on, partition 3 becomes available.
- Those tuples that were in a failed batch before now succeed in a later batch.

In order to achieve exactly-once semantics with opaque spouts, more complicated logic is needed when doing state updates. It's no longer sufficient to only store the batch id with whatever state is being updated. A small amount of additional information must be tracked – you can find more information about opaque topologies can be found in the Storm documentation.

The latency and throughput characteristics are different for micro-batch processing as compared to one-at-a-time processing. For any individual tuple, the latency from when it's added to the source queue and fully processed is much higher in micro-batch processing. There's a small but significant amount of overhead to coordinating batches that increases latency, plus instead of waiting for just one tuple to complete, processing needs to be done on many more tuples. In practice this turns out to be latency on the order of hundreds of milliseconds to seconds.

However, micro-batch processing can have higher throughput than one-at-a-time processing. Whereas one-at-a-time processing must do tracking on

an individual tuple level, micro-batch processing only has to track at a batch level. So less resources are needed on average per tuple, allowing micro-batch processing to have higher throughput than one-at-a-time processing.

10.3 High-level API for micro-batch processing

A lot of the nitty gritty details you've seen, like storing batch ids and deciding whether or not to skip an update by comparing the current batch id with the stored batch id, are things that you'd rather have automated by a higher level API. Ideally, the API would let you express the computation as if every tuple gets processed exactly one time without any failures, and then the API translates your code into something that is fault-tolerant and idempotent under retries.

An example of an API like this is Storm's Trident API. Trident is Java API that translates micro-batch processing topologies into the spouts and bolts of Storm. Trident looks very similar to the batch processing idioms you're already familiar with – it has joins, aggregations, grouping, functions, and filters. In addition to that, it adds abstractions for doing stateful processing across batches using any database or persistence store.

Let's look at how to implement streaming word count with Trident. For the purposes of illustration, this example will read from an infinite stream of sentences from the following source:

```
FixedBatchSpout spout = new FixedBatchSpout(
    new Fields("sentence"),
    3, // number of tuples in each batch
    new Values("the cow jumped over the moon"),
    new Values("the man went to the store"),
    new Values("four score and seven years ago"),
    new Values("how many apples can you eat"),
    new Values("to be or not to be the person"));
spout.setCycle(true); // repeats these tuples forever
```

This spout emits three sentences every batch and cycles through the sentences ad infinitum. Here's the definition of a Trident topology that implements word count:

```
TridentTopology topology = new TridentTopology();
topology.newStream("spout1", spout)
    .each(new Fields("sentence"),
          new Split(),
          new Fields("word"))
    .groupBy(new Fields("word"))
    .persistentAggregate
```

```

new MemoryMapState.Factory(),
new Count(),
new Fields("count"));

```

Let's go through the code line by line. First a TridentTopology object is created, which exposes the interface for constructing Trident computations. TridentTopology has a method called newStream that creates a new stream of data in the topology reading from an input source. In this case, the input source is just the FixedBatchSpout defined from before. If you wanted to read from Kafka, you would instead use a Kafka spout for Trident. Trident keeps track of a small amount of state for each input source (metadata about what it has consumed) in Zookeeper, and the "spout1" string here specifies the node in Zookeeper where Trident should keep that metadata. This metadata contains the information about what was in each batch, so that if the batch has to be replayed the exact same batch will be emitted the next time.

The spout emits a stream containing one field called "sentence". The next line of the topology definition applies the Split function to each tuple in the stream, taking the "sentence" field and splitting it into words. Each sentence tuple creates potentially many word tuples – for instance, the sentence "the cow jumped over the moon" creates six "word" tuples. Here's the definition of Split:

```

public static class Split extends BaseFunction {
    public void execute(TridentTuple tuple,
                        TridentCollector collector) {
        String sentence = tuple.getString(0);
        for(String word: sentence.split(" ")) {
            collector.emit(new Values(word));
        }
    }
}

```

Trident operations work very differently than Storm bolts. A Storm bolt takes entire tuples as input and produces entire tuples as output. Trident operations, on the other hand, take in partial tuples as input and have their output values appended to the input tuple. So after executing the Split function above, the tuples contain the fields "sentence" and "word". This is identical to how JCascalog operations work as discussed in Chapter 5 (as well as identical to most other batch processing abstractions). Behind the scenes, Trident compiles as many operations as possible together into single bolts.

The rest of the topology computes word count and keeps the results persistently

stored. First the stream is grouped by the "word" field. Then, each group is persistently aggregated using the Count aggregator. The persistentAggregate function knows how to store and update the results of the aggregation in a source of state. In this example, the word counts are kept in memory, but this can be trivially swapped to use Memcached, Cassandra, or any other persistent store.

Let's see how to get this code to store the word counts in Cassandra instead. Here is the code to do so:

```
CassandraState.Options opts =
    new CassandraState.Options();
opts.globalCol = "COUNT";
opts.keySerializer = StringSerializer.get();
opts.colSerializer = StringSerializer.get();
stream.groupBy(new Fields("word"))
    .persistentAggregate(
        CassandraState.transactional(
            "127.0.0.1",
            "mykeyspace",
            "mycolumnfamily"),
        new Count(),
        new Fields("count"));
```

This CassandraState implementation allows grouped aggregation to be done with either 1-tuple groups or 2-tuple groups. In the 1-tuple case, like shown above, the value in that tuple corresponds to the Cassandra key and the column used will be the "globalCol" specified in the options. With 2-tuple groups, the first element of the grouping tuple will be the Cassandra key and the second will be the Cassandra column.

SIDE BAR More information on provided CassandraState

The accompanying source code for this book provides a simple implementation of a CassandraState. It's not ideal however, as it does database operations one at a time instead of batching them. So the potential throughput of this CassandraState is much lower than it could be. However, the code is much easier to follow this way, so we hope it can serve as a reference implementation for making States to interact with whatever database you choose to use.

Here is the definition of the count aggregator:

```
public static class Count
    implements CombinerAggregator<Long> {
```

```

public Long init(TridentTuple tuple) {
    return 1L;
}
public Long combine(Long val1, Long val2) {
    return val1 + val2;
}
public Long zero() {
    return 0L;
}
}

```

As you can see, it's a straightforward implementation of Count, similar to how parallel aggregators are defined in JCascalog. Notice in particular that nowhere in all this code is the tricky batch id logic to achieve exactly-once semantics. Trident takes care of that behind the scenes automatically. In this case, it automatically stores the batch id with the count, and if it detects the stored batch id to be the same as the current batch id, it will not do any updates to the persistent store.

10.4 Finishing speed layer for SuperWebAnalytics.com

Let's now finish the speed layer for SuperWebAnalytics.com by making use of micro-batch stream processing. Portions of Trident that have not been covered yet will be explained as we go along. To get deeper information on the Trident API, we refer you to the Storm documentation available online. Our goal here is to show how micro-batch processing can be applied to practical problems, not to get lost in every last detail on how these APIs work.

Let's start with the pageviews over time speed layer.

10.4.1 Pageviews over time

Remember that the goal of the pageviews over time query is to get the total number of pageviews to a URL over any range of hours. To make this query in the serving layer, we computed hourly, daily, weekly, and monthly rollups of pageviews so that large ranges could be computed efficiently. This optimization is not needed in the speed layer because the speed layer only deals with recent data – since the speed layer only deals with data since the last serving layer update, most of those rollups would be pointless. If the batch cycle is very long, like over a day, then having a daily rollup might be useful. However, for simplicity's sake we'll implement the pageviews over time speed layer with just hourly rollups.

Like uniques over time from the last chapter, Cassandra can be used as the realtime view. And just like uniques over time, the key will be a URL, the column a time bucket, and the value the number of pageviews for that URL and time

bucket.

The first step is to define a spout to read the pageviews from Kafka. The following code accomplishes this:

```
TridentTopology topology = new TridentTopology();
TridentKafkaConfig kafkaConfig =
    new TridentKafkaConfig(
        new KafkaConfig.ZkHosts(
            "zkstr", "/kafka"),
        "pageviews"
    );
kafkaConfig.scheme = new PageviewScheme();
```

Configuring a Trident Kafka spout is similar to configuring a regular Storm Kafka spout as shown in the last chapter. Note the setting of a scheme which will deserialize pageviews into three fields: "url", "user", and "timestamp".

Now here's the first part of the topology that normalizes URLs and converts timestamps into the appropriate hour bucket:

```
Stream stream =
    topology.newStream(
        "pageviewsOverTime",
        new TransactionalTridentKafkaSpout(
            kafkaConfig))
    .each(new Fields("url"),
        new NormalizeURL(),
        new Fields("normurl"))
    .each(new Fields("timestamp"),
        new ToHourBucket(),
        new Fields("bucket"))
```

As you can see, it's just a function for each task. Here's the implementation of those functions:

```
public static class NormalizeURL extends BaseFunction {
    public void execute(TridentTuple tuple,
                        TridentCollector collector) {
        try {
            String urlStr = tuple.getString(0);
            URL url = new URL(urlStr);
            collector.emit(new Values(
                url.getProtocol() +
                "://" +
                url.getHost() +
                url.getPath()));
        } catch(MalformedURLException e) {
```

```

        }
    }

public static class ToHourBucket extends BaseFunction {
    private static final int HOUR_SECS = 60 * 60;
    public void execute(TridentTuple tuple,
                        TridentCollector collector) {
        int secs = tuple.getInteger(0);
        int hourBucket = secs / HOUR_SECS;
        collector.emit(new Values(hourBucket));
    }
}

```

The logic is no different than what was used in the batch layer, and in fact it would be superior to just share code between the layers (it's duplicated here only so it's easier to follow).

Finally, all that's left is rolling up the pageview counts into Cassandra and ensuring this is done in an idempotent manner. First, let's configure the CassandraState:

```

CassandraState.Options opts =
    new CassandraState.Options();
opts.keySerializer = StringSerializer.get();
opts.colSerializer = IntegerSerializer.get();
StateFactory state =
    CassandraState.transactional(
        "127.0.0.1",
        "superwebanalytics",
        "pageviewsOverTime",
        opts);

```

The appropriate serializers are set for the keys (URLs) and columns (time buckets). Then the state is configured to point at the appropriate cluster, keyspace, and column family. Now let's see the definition of the remainder of the topology:

```

stream.groupBy(new Fields("normurl", "bucket"))
    .persistentAggregate(
        state,
        new Count(),
        new Fields("count"));

```

The grouping key contains two fields, the "url" and the "bucket". When using CassandraState, the first grouping key corresponds to the Cassandra key, and the second grouping key corresponds to the Cassandra column. The

`persistentAggregate` method is used to apply the built-in Count aggregator to roll up the counts. Trident automatically stores the batch id with each count so that any failures and retries can be done in an idempotent manner. As you can see, the implementation of the pageviews over time speed layer is very straightforward.

10.4.2 Bounce rate analysis

Let's now move on to the speed layer for computing bounce rates. Remember that a visit was defined as a bounce if the user visited a page and did not visit another page on that domain within 30 minutes of the first visit. So a "bounce" occurring is based on an event not happening.

Like uniques over time, you have to take into account equivs to compute bounces completely accurately. If a user browses your website on their computer and then ten minutes later browses from her phone, that should count as one visit that's not a bounce. If you didn't know those two user identifiers were the same person, that would count as two visits and two bounces. And like uniques over time, taking into account equivs massively complicates the computation. To see why, just consider what you need to do when you receive a new equiv. Here's an illustrative series of events:

- A) Pageview received at minute 0 from "alice-computer"
- B) Pageview received at minute 10 from "alice-phone"
- C) Equiv received at minute 140 between "alice-computer" and "alice-phone"

Before event C, those two pageviews would have counted as two visits and two bounces. You didn't know they were equivalent so there's nothing else you could possibly compute in realtime. But event C changes everything. Now you have to go back and fix the bounce rates computed over two hours earlier. The only way to do this is to retrieve all visits in the past for "alice-computer" and "alice-phone" and recompute the appropriate bounces. This should be compared with what was computed the first time around, and then the appropriate increments/decrements can be made to the database that stores bounce counts and visit counts. In addition to all this, the equiv graph analysis needs to be done in realtime.

Besides being rather complicated to accomplish, taking into account equivs also massively increases storage costs for the speed layer. Rather than just store bounce and visit counts per domain, you also need to be able to retrieve all the pageviews for a user since the last serving layer update. Since an equiv could come in at any time for any user, this means all speed layer pageviews need to be indexed. This is not infeasible, but it certainly is a lot more expensive.

Just like uniques over time, there's an easy way to simplify all this: just ignore

equivs in the speed layer. It doesn't seem like this should create too much innaccuracy, and due to the fact the batch/serving layers will automatically correct any speed layer innaccuracy, this seems like a great tradeoff. So that's the approach we will use. Of course, as always, you should verify the inaccuracy created by any approach by comparing both approaches via batch analytics jobs.

There are three steps to doing the bounce rate analysis in realtime:

- Consume a stream of pageview events that contain a user identifier, a URL, and a timestamp.
- Identify bounces by detecting when a user has visited a single page on a domain and no other pages within 30 minutes.
- Update a Cassandra database containing a map from domain to bounce rate. The bounce rate will be stored as a pair of numbers [number of bounces, number of visits].

Consuming the stream of pageviews is identical to how it was done for pageviews over time: just create a Trident Kafka spout. Identifying bounces is more interesting, in particular because it's a time-based event. At any given moment, a bounce could occur, and that would be based on pageviews from 30 minutes prior. Since in stream processing you only have access to events that have just occurred, we will need to keep state about what has happened in the past 30 minutes.

The main idea is to track each "visit" (a [domain, user] pair) until that visit is complete. We've defined a visit to be complete when 30 minutes has elapsed with no further pageviews by that user on that domain. Once that visit is complete, the bounce rate for that domain can be updated. The "number of visits" is incremented, and the "number of bounces" is incremented if there was only one pageview in that visit. At the topology level, in order to track these visits we'll maintain a map from [domain, user] to the first time and the last time a pageview occurred in that visit. Once a minute, we'll iterate through the entire map to determine which visits have completed, and then remove those visits from the map and update the bounce rate information for the corresponding domains.

This strategy requires tracking all visits in the last 30 minutes. If you are at large scale, this could be on the order of hundreds of millions or even billions of visits at once. If you figure tracking a visit requires about 100 bytes, what with the domain, user id, timestamps, and memory usage by the map, you're looking on the order of a terabyte of memory required. This is doable, but expensive. After finishing the memory-based implementation, we'll look at ways to reduce and even eliminate the memory requirements.

SIDEBAR Windowed stream processing?

You may have heard the term "windowed stream processing" before, which refers to breaking an incoming stream into windows of time, such as 30 seconds, 1 minute, 5 minutes, or 30 minutes. Sometimes the window is "rolling" and always refers to the last X seconds of time. Other times the windows are fixed and occur right after one another.

At first glance, the bounce rate analysis may seem like a good fit for windowed stream processing due to the time-oriented nature at it. However, on a closer glance it does not fit windowed stream processing at all. Any particular visit could span an indefinite period of time. For example, if someone visits a page on a domain every 10 minutes for 16 hours, then that visit must remain in the map for all 16 of those hours (until 30 minutes with no activity has elapsed). Windowed stream processing does not handle computations like this – it's meant to answer questions like "how many pageviews have I received in the last 15 minutes?"

Let's start with the topology itself and then dive into the details of implementing the state and corresponding functions. The beginning of the topology looks like this:

```
topology.newStream(
    "bounceRate",
    new TransactionalTridentKafkaSpout(kafkaConfig))
.each(new Fields("url"),
      new NormalizeURL(),
      new Fields("normurl"))
.each(new Fields("normurl"),
      new ExtractDomain(),
      new Fields("domain"))
```

There's nothing new here. It consumes the stream of pageviews from Kafka and runs a couple functions to extract the domain from the URL. Here's the next part of the topology which analyzes visits and determines when bounces occur:

```
.partitionBy(new Fields("domain", "user"))
.partitionPersist(
    new BounceState.Factory(),
    new Fields("domain", "user", "timestamp"),
    new PageviewEventToBounceState(),
    new Fields("domain", "isBounce"))
.newValuesStream()
```

This is probably a lot shorter than you expected! Actually, most of the logic exists in the "BounceState" implementation which we'll cover in a bit. This part of the topology starts off by partitioning by "domain" and "user".

Trident goes a bit further than most batch processing abstractions by giving you control over how tuples are partitioned at all points in the processing. Most abstractions only go so far as letting you do "groupBy" operations. Here we repartition so that all tuples with the same domain and user fields will go to the same task, and all domain/user combinations are evenly distributed across all tasks. Tuples are partitioned this way because bounce detection is based on all visits by a user to a domain, so all those tuples have to go to the same task. This also could have been accomplished by just partitioning by the "domain" field. However, that might lead to skew if you have just a few domains dominating the visits in your dataset. By partitioning by "user" and "domain", the distribution will almost certainly be even since it's extremely unlikely a single person is dominating the visits to a single domain in your dataset.

The "partitionPersist" line both updates the state kept on visits as well as emitting tuples when a bounce or non-bounce is detected. partitionPersist lets you run an arbitrary function that takes in a State and a batch of tuples. That function can update the State any way it wants.

States are allowed to emit their own stream called a "new values stream". In this case the Bounce State uses that stream to emit tuples whenever enough time has elapsed on a visit that it can be marked as a bounced visit or a non-bounced visit. The operation given to the partitionPersist invocation will use the timestamps in the pageview events to determine when that time has elapsed. The new values stream contains 2-tuples, with the first field being the domain, and the second field a boolean indicating whether it was a bounce or not.

Finally, here's the rest of the topology definition:

```
.each(new Fields("isBounce"),
      new BooleanToInt(),
      new Fields("bint"))
.groupBy(new Fields("domain"))
.persistentAggregate(
    CassandraState.transactional(
        "127.0.0.1",
        "superwebanalytics",
        "bounceRate",
        opts),
    new Fields("bint"),
    new CombinedCombinerAggregator()
```

```

        new Count(),
        new Sum())),
new Fields("count-sum"));

```

This part of the topology simply consumes the stream of ["domain", "isBounce"] and aggregates it into Cassandra to determine for each domain the number of visits and the number of bounces. First "isBounce" is converted to a 0 if it's false and a 1 if it's true using the BooleanToInt function. Then a standard persistentAggregate is done to update Cassandra. We actually need to do two aggregations: a count to determine the number of visits, and a sum of the isBounce integers to determine the number of bounces. So the Count and Sum aggregators are combined into one using the "CombinedCombinerAggregator" utility. This utility is defined as follows:

```

public static class CombinedCombinerAggregator
    implements CombinerAggregator {
    CombinerAggregator[] _aggs;
    public CombinedCombinerAggregator(
        CombinerAggregator... aggs) {
        _aggs = aggs;
    }
    public Object init(TridentTuple tuple) {
        List<Object> ret = new ArrayList();
        for(CombinerAggregator agg: _aggs) {
            ret.add(agg.init(tuple));
        }
        return ret;
    }
    public Object combine(Object o1, Object o2) {
        List l1 = (List) o1;
        List l2 = (List) o2;
        List<Object> ret = new ArrayList();
        for(int i=0; i<_aggs.length; i++) {
            ret.add(
                _aggs[i].combine(
                    l1.get(i),
                    l2.get(i)));
        }
        return ret;
    }
    public Object zero() {
        List<Object> ret = new ArrayList();
        for(CombinerAggregator agg: _aggs) {
            ret.add(agg.zero());
        }
        return ret;
    }
}

```

Now let's take a look at how BounceState is implemented, which is where most of the interesting logic lies. States must implement the "State" interface, which is very minimal:

```
public interface State {
    void beginCommit(Long txid);
    void commit(Long txid);
}
```

States are told when batches begin and when they are successfully completed. All the rest of the methods – how the state is read and updated, is completely up to you. For BounceState, we want it to receive pageview events and then emit a stream of visits marked as bounced or not bounced. Additionally, the state is responsible for being idempotent and producing the exact same answer no matter how many times the same batch is replayed.

The BounceState needs to do two things: first, keep track of the timestamps of the first and last pageviews in a visit. Then, when it's been 30 minutes since the last pageview in a visit, that visit should be cleared and emitted as either a "bounce" or "not a bounce". To accomplish the first task, we'll keep a map from visit ([domain, user]) to a pair of [start timestamp, last timestamp]. For the second task, the state will iterate through all visit information once a minute to determine which ones have "timed out". Then it can clear the completed visits and emit the "bounce" or "no bounce" events. For simplicity, we'll use the timestamps of the incoming pageviews to determine when 30 minutes have elapsed. This assumes, of course, that you have a steady stream of pageviews coming in.

SIDE BAR**Time and out of order messages**

One of the assumptions made in the bounce rate analysis code is that the timestamps in tuples are always increasing. However, what if tuples come in out of order?

This is not just a hypothetical question – tuples are being generated all over the cluster and then placed together on your queue servers. It's very likely tuples won't be in perfect order. Further than that, if you have network partitions or errors and tasks get delayed writing their tuples to your queues, tuples could be out of order by much longer, even on the order of minutes.

For many computations, such as pageviews over time, this doesn't matter. But for the bounce rate analysis, which uses time to trigger checking for completed visits, this does matter. For example, a pageview for a visit could come in after checking for completed visits that could turn a completed visit into a still active visit.

The way to deal with out of order tuples is to introduce latency into your computations. With the bounce rate analysis code, you could change the definition of a completed visit to "more than 45 minutes has passed since last visit and there are no additional pageviews in the 30 minutes after the last pageview". This strategy will handle out of order tuples that come up to 15 minutes late.

Of course, there's no perfect way to deal with out of order tuples. You could theoretically receive a tuple that was generated two days ago, but it's not reasonable for your bounce rate analysis code to wait indefinitely to see if there are any out of order tuples. Otherwise your code wouldn't be able to make any progress. So you have to place a limit on how long you're willing to wait. As usual, it's prudent to do measurements to determine the distribution and rate of out of order tuples.

Like many things in the speed layer, this is another example of something that's fundamentally difficult to deal with in realtime but not a problem in batch. Since the Lambda Architecture has a batch layer that overrides the speed layer, any inaccuracy introduced by out of order tuples is weeded out over time.

Let's start with storing the visit information. All you need for that is a map, and fortunately Trident comes built-in with a map that already implements all the batch id logic to be idempotent under replay. So setting up the bounce state and the information it stores internally is done as follows:

```
public static class BounceState implements State {
```

```

static final String LAST_SWEEP_TIMESTAMP = "lastSweepTs";
static final int THIRTY_MINUTES_SECS = 30 * 60;
public static class Factory implements StateFactory {
    MemoryMapState.Factory _factory =
        new MemoryMapState.Factory();
    public State makeState(Map conf,
                           IMetricsContext context,
                           int partIndex,
                           int numPartitions) {
        return new BounceState(
            (MemoryMapState) _factory.makeState(
                conf,
                context,
                partIndex,
                numPartitions));
    }
}
MemoryMapState _state;
public BounceState(MemoryMapState state) {
    _state = state;
}

```

The BounceState itself simply stores a "MemoryMapState" internally which is the idempotent map mentioned built-in to Trident. Trident topologies themselves are set up with State factories, so a factory to create bounce states is provided here (the reason for factories is that every partition of the state needs its own State instance, so it needs a factory to create those instances once the code is deployed to the cluster). The map itself will store the [domain, user] pairs as keys and the [start timestamp, last timestamp] pairs as values. In additionally, it will also keep track of the last time the map was iterated through to check for completed visits using the key "lastSweepTs". The map is re-used for this to take advantage of the idempotent semantics.

Next up is the definition of the keys and values for the map, called "Visit" and "VisitInfo":

```

static class Visit extends ArrayList {
    public Visit(String domain, PersonID user) {
        super();
        add(domain);
        add(user);
    }
}
static class VisitInfo {
    public int startTimestamp;
    public Integer lastVisitTimestamp;
    public VisitInfo(int startTimestamp) {
        this.startTimestamp = startTimestamp;
        this.lastVisitTimestamp = startTimestamp;
    }
}

```

```

        }
        public VisitInfo clone() {
            VisitInfo ret = new VisitInfo(this.startTimestamp);
            ret.lastVisitTimestamp = this.lastVisitTimestamp;
            return ret;
        }
    }
}

```

Now is the real meat of the `BounceState`, the method that receives pageview events and implements all the necessary logic:

```

public void pageviewEvent(TridentCollector collector,
                           final String domain,
                           final PersonID user,
                           final int timestampSecs
                           ) {
    Visit v = new Visit(domain, user);
    update(_state, v, new ValueUpdater<VisitInfo>() {
        public VisitInfo update(VisitInfo v) {
            if(v==null) {
                return new VisitInfo(timestampSecs);
            } else {
                VisitInfo ret = new VisitInfo(
                    v.startTimestamp);
                ret.lastVisitTimestamp = timestampSecs;
                return ret;
            }
        }
    });
    Integer lastSweep =
        (Integer)get(_state, LAST_SWEEP_TIMESTAMP);
    if(lastSweep==null) lastSweep = 0;
    List<Visit> expired = new ArrayList();
    if(timestampSecs > lastSweep + 60) {
        Iterator<List<Object>> it = _state.getTuples();
        while(it.hasNext()) {
            List<Object> tuple = it.next();
            Visit visit = (Visit) tuple.get(0);
            VisitInfo info = (VisitInfo) tuple.get(1);
            if(info.lastVisitTimestamp >
                timestampSecs + THIRTY_MINUTES_SECS) {
                expired.add(visit);
                if(info.startTimestamp ==
                   info.lastVisitTimestamp) {
                    collector.emit(new Values(domain, true));
                } else {
                    collector.emit(new Values(domain, false));
                }
            }
        }
        put(_state, LAST_SWEEP_TIMESTAMP, timestampSecs);
    }
    for(Visit visit: expired) {

```

```

        remove(_state, visit);
    }
}

```

There's two parts to this method. First, the visit info in the MemoryMapState is updated. Then the timestamp of the pageview is compared against the last "sweep time" to determine if the map needs to be checked for completed visits. If so, it iterates through the map and determines which visits have completed and emits the visit even with the corresponding flag about whether it was a bounce or not. Finally, at the end, expired visits are removed from the map.

At no point in the code do you have to deal with messy logic to handle idempotence under replays. Because all state is managed through the MemoryMapState, you get that essentially for free. Finally, the last part of the bounce state is implementing the actual State interface, which is done as follows:

```

@Override
public void beginCommit(Long batchid) {
    _state.beginCommit(batchid);
}
@Override
public void commit(Long batchid) {
    _state.commit(batchid);
}

```

All you have to do is pass the events on to the underlying MemoryMapState.

The last piece of the puzzle is telling the Trident topology how to interact with the state. This was done by giving it a "StateUpdater" in the partitionPersist method. As mentioned before, a "StateUpdater" is similar to a Function except it can also interact with a State. That is done as follows:

```

public static class PageviewEventToBounceState
    extends BaseStateUpdater<BounceState> {
    public void updateState(
        BounceState state,
        List<TridentTuple> tuples,
        TridentCollector collector) {
        for(TridentTuple t: tuples) {
            state.pageviewEvent(
                collector,
                t.getString(0),
                (PersonID) t.get(1),
                t.getInteger(2));
        }
    }
}

```

```

    }
}
```

It simply passes the pageview information on to the `BounceState`. And that actually completes the implementation of the speed layer for realtime bounce rate analysis.

There is a weakness to this implementation though – all the state is kept only in memory. State is not persisted/replicated anywhere. So if a task carrying state dies, that state is lost. One way to deal with this weakness is to just ignore it, accept the small amount of inaccuracy it introduces, and rely on the batch layer to correct that inaccuracy down the road.

Alternatively, it is possible to do stream processing with in-memory state that is tolerant to failures. There are two ways to accomplish this.

The first is to make use of the standard database technique of keeping a commit log in a replicated store. Good technologies for this are HDFS file appends or Kafka. Whenever you make an update to your state, you write what that update was to your log. Figure 10.7 illustrates what a commit log might look like.

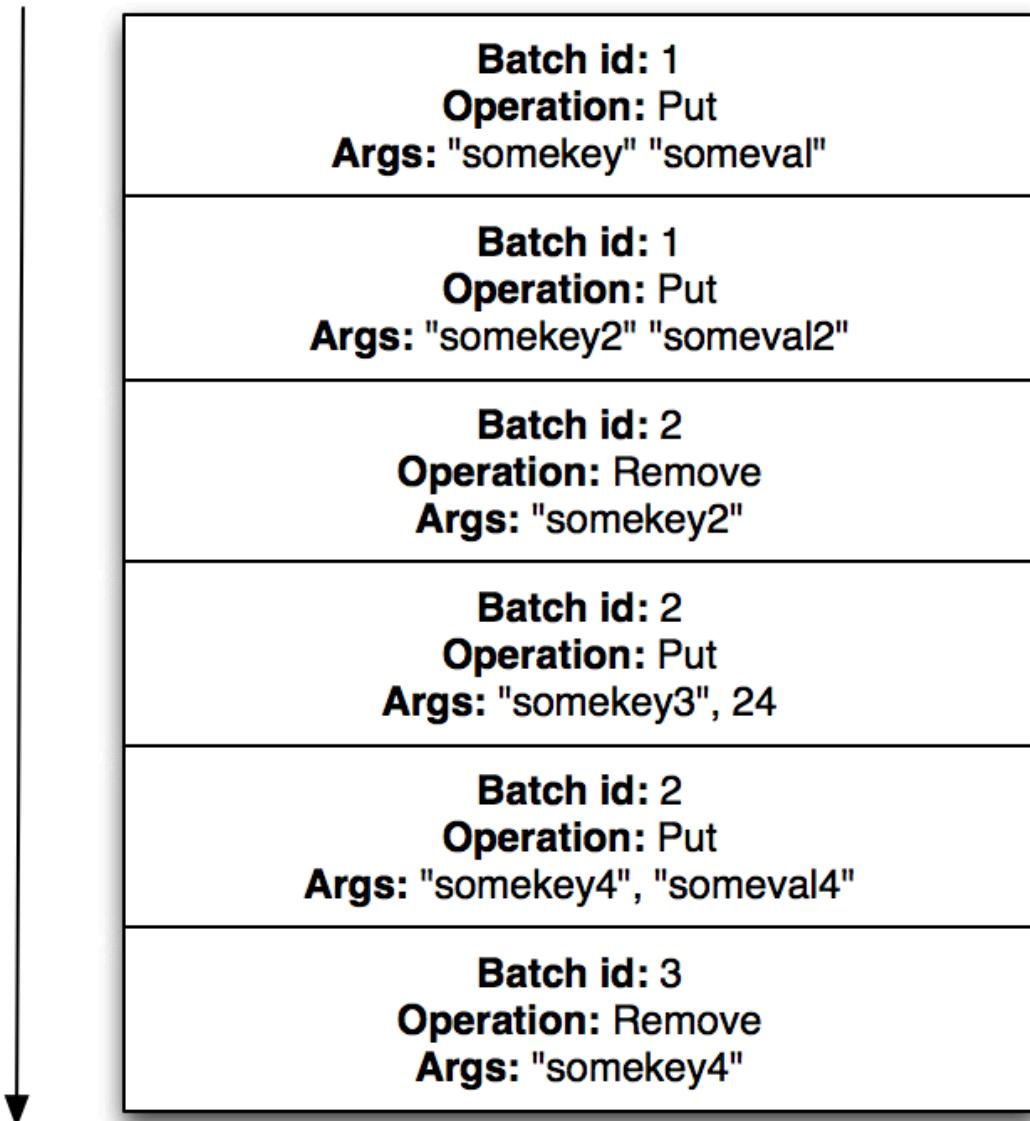


Figure 10.7 Commit log

When a task starts up, it replays the commit log to rebuild the internal state. Of course, the commit log grows indefinitely, so rebuilding a state based on the log will get more and more expensive. You can fix this problem by periodically compacting the log. Compaction is the process of persisting the entire state itself, and then deleting all commit log elements involved in the construction of that state. A great technology for storing the state in its entirety is a distributed filesystem. Your strategy around compaction could be as simple as doing it once a minute, or after the commit log grows to a certain size.

There's another technique for accomplishing persistent in-memory state that

doesn't involve a commit log at all. Recall that the way Trident works is it processes batches in a strong order, and by keeping the batch id stored with the state it can detect if a batch has been processed before and achieve exactly-once semantics. But what if your computation system could retry batches beyond just the last one – say batches a few minutes in the past? This lets you do some cool new things.

The idea is to periodically checkpoint any state kept in memory by writing it out somewhere (like a distributed filesystem). You might checkpoint once a minute. You also remember up to which point in your source stream that state represents. Then, you continue processing batches like normal. Now let's say you have a failure 45 seconds later and one of the partitions holding your state dies. Now you have all your tasks with state current up to the most recent batch, while task that died only has state current up to the batch from 45 seconds ago (when the last checkpoint was). Now, what you do is replay the computation from the past 45 seconds, but only for the partition in question that you're trying to repair. So you skip computing partitions for which you already have the up to date state. Like the first approach, this strategy requires the state to be periodically written out in full. However, it requires no commit log to be written out.

This strategy requires extensions to the micro-batch stream processing model that Storm/Trident do not currently implement. Another system, however, called Spark Streaming does implement this approach. More information about Spark Streaming can be found in the sidebar.

SIDE BAR **Spark**

Spark is a computation system focused on the smart usage of memory. It has two modes of operation. First, it can be used as a batch system, where it implements MapReduce and also adds operators for caching datasets in memory. For many iterative batch algorithms, like graph and machine learning algorithms, this enables it to achieve much higher performance than the purely disk based approach taken by MapReduce. Spark's second mode of operation is called Spark Streaming which implements the micro-batch stream processing approach with periodic checkpointing of internal state. Whereas Trident is focused on integrating with external databases, Spark Streaming is focused on computing state to be kept in-memory.

A good way to categorize computation systems is by the computation styles they support. The three main computation styles are batch processing, low latency one-at-a-time processing, and micro-batch processing. Hadoop does only batch processing, Storm does one-at-a-time and micro-batch processing, and Spark does batch and micro-batch processing.

10.5 Another look at the bounce rate analysis example

Earlier you saw that the memory requirements of the bounce rate analysis speed layer were pretty substantial, as at large scale you might be tracking hundreds of millions or even billions of visits at once. Let's see how we can reduce, and in fact, completely eliminate the memory requirement.

The trick is taking a step back and looking at the problem again. Visits are not complete until 30 minutes has passed without activity on that visit. That required 30 minutes of waiting to determine the status of a visit means that bounce rate analysis is fundamentally not a realtime problem. So latency is not that much of a constraint, so you're not forced to use a memory-focused stream processing system at all.

For this problem using a batch processing system for the speed layer is absolutely viable. The logic of the workflow doesn't change – you still maintain a map from visits to information about the visit, mark visits as bounces or not bounces after 30 minutes of inactivity have passed, and then aggregate the bounce and visit information into a key/value database. The difference is the underlying technologies change: for the computation system, you might use Hadoop. And for storing the speed layer view you could use a serving layer database like

ElephantDB. Finally, for the intermediate, key/value state, you could also use something like ElephantDB. This is illustrated in Figure 10.8.

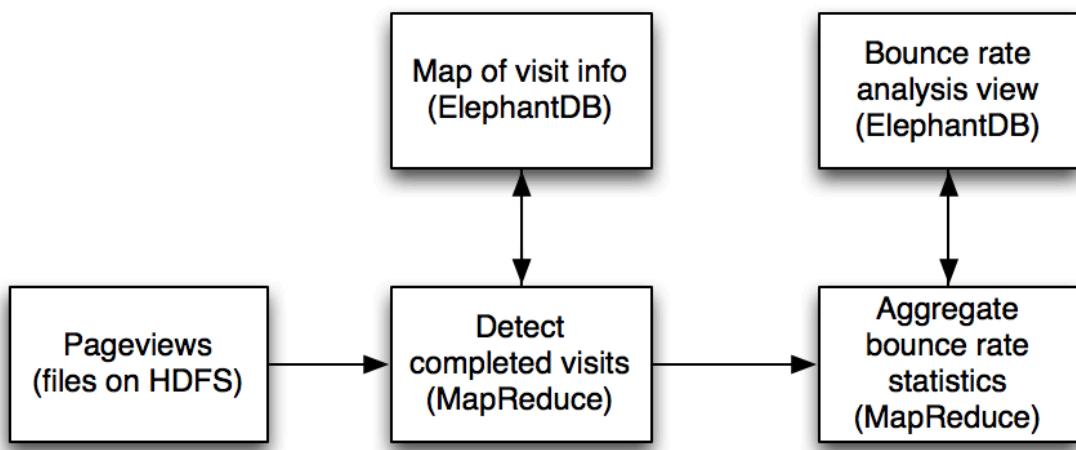


Figure 10.8 Bounce rate analysis using incremental batch processing

So far we've only discussed using batch computation to do full recomputations, where you consume all the data at once to produce a view from scratch. Incremental batch processing works differently, where you consume new data and produce new views based on the last version of your views. This is different than stream processing, which mutates your views in place. The views produced by incremental batch processing workflows are brand new and never modified after creation. We will talk more about incremental batch processing in the next chapter.

10.6 Conclusion

You've seen how by sacrificing some latency, you can go beyond the at-least-once semantics of one-at-a-time stream processing and achieve exactly-once processing semantics. Whereas one-at-a-time stream processing only has exactly-once semantics with inherently idempotent operations, micro-batch processing can achieve it for nearly any computation.

It should also be apparent that speed layer does not necessarily mean realtime, nor does it necessarily entail stream processing. The speed layer is about accounting for recent data – you saw how in the bounce rate analysis example the definition of the problem is inherently not realtime. And so accounting for recent data allows for other approaches like incremental batch processing.

Lambda Architecture *in-depth*

This chapter covers:

- Revisiting the Lambda Architecture
- Incremental batch processing
- Efficiently managing resources in batch workflows
- Merge logic between batch and realtime views

In Chapter 1 you were introduced to the Lambda Architecture and its general purpose approach for implementing any data system. Every chapter since then has dived into the details of the different components of the Lambda Architecture. As you've seen, there's a lot involved in building big data systems that not only scale, but are robust and easy to understand as well.

Now that you've had a chance to dive into all the different layers of the Lambda Architecture, let's use that newfound knowledge to review the Lambda Architecture once more and achieve a better understanding of it. We'll fill in any remaining gaps and explore variations on the methodologies that have been discussed so far.

11.1 Defining data systems

We started with the simple question: "what does a data system do?" The answer was very simple: a data system answers questions based on data you've seen in the past. Or put more formally, a data system computes queries which are functions of all the data you've ever seen. This is an intuitive definition which clearly encapsulates any data system you'd ever want to build.

query = function(all data)

Figure 11.1 General formulation of queries

There's a number of properties you're concerned about with your queries. These include:

- **Latency:** the time it takes to run a query. In most cases you wish your latency requirements will be very low – on the order of milliseconds. Sometimes it's okay for a query to take a few seconds. When doing ad-hoc analysis, your latency requirements are often very lax, even on the order of hours.
- **Timeliness:** how up to date the query results are. A completely timely query takes into account all data ever seen in the past, while a less timely query may not include results from the recent minutes or hours.
- **Accuracy:** In many cases, in order to make queries performant or scalable, you must make approximations in your query implementations.

A huge part to building data systems is making them fault-tolerant. You have to plan for how your system will behave when you encounter machine failures. Oftentimes this means making tradeoffs with the properties above. For example, there is a fundamental tension between latency and timeliness. The CAP theorem shows that under partitions a system can either be consistent (queries take into account all previous written data) or available (queries are answered at the moment). Consistency is just a form of timeliness, and availability just means the latency of the query is bounded. An eventually consistent system chooses latency over timeliness (queries are always answered but may not take into account all prior data during failure scenarios).

As data systems are dynamic, changing systems, built by humans and with new features/analyses deployed all the time, humans are an integral part of any data system. And like machines, humans can and will fail. Humans will deploy bugs to production and make all manner of mistakes. So it is critical for data systems to be human fault-tolerant as well.

You saw how mutability – and associated concepts like CRUD – are fundamentally not human fault-tolerant. If a human can mutate data, then a mistake can mutate data. So allowing updates and deletes on your core data will inevitably lead to corruption.

The only solution is to make your core data **immutable**, with the only write operation allowed being appending new data to your ever growing set of data. You

can do things like set permissions on your core data to disallow deletes and updates – this redundancy ensures that mistakes cannot corrupt existing data, so your system is far more robust.

This leads us to the very, very basic model of data systems:

- A master dataset consisting of an evergrowing set of data.
- Queries which are functions that take in the entire master dataset as input.

Anything you'd ever want to do with data can clearly be done this way, and such a system has at its core that crucial property of human fault-tolerance. If it were possible to implement, this would be the ideal data system. The Lambda Architecture emerges from making the fewest sacrifices possible to achieve this ideal of queries as functions of an ever-growing immutable dataset.

11.2 Batch and serving layers

Computing queries as a function of all data is not practical because it's not reasonable to expect queries on a multi-terabyte dataset, much less a multi-petabyte dataset, to return in a few milliseconds. And even if that were possible, queries would be unreasonably resource-intensive. The simplest modification to make to such an architecture is to make queries on precomputed views rather than directly on the master dataset. These precomputed views can be tailored for the queries so that the queries are as fast as possible, and the views themselves are functions of the master dataset.

In Chapters 2 through 7, you saw the details of implementing such a system. At the core is a batch processing system that can compute those functions of all data in a scalable and fault-tolerant way – hence, this part of the Lambda Architecture is called the "batch layer".

The goal of the batch layer is to produce views that are indexed so that queries on those views can be resolved with low latency. The indexing and serving of those views is called the "serving layer", and the serving layer is tightly connected to the batch layer. In designing your batch and serving layers, you must strike a balance between the amount of precomputation done in the batch layer with the size of the views and the amount of computation needed at query time (discussed extensively in Chapter 4).

Let's now go beyond this basic model of the batch and serving layers of the Lambda Architecture to explore more options you have available to you in designing them. A key performance metric of these layers is how long it takes to

update the views. As the speed layer must compensate for all data not represented in the serving layer, the longer it takes the batch layer to run, the larger your speed layer views must be. Needing larger clusters of significantly more complex databases greatly increases your operational complexity. In addition, the longer it takes the batch layer to run, the longer it takes to recover from bugs that are accidentally deployed to production. One way to lower the latency of the batch layer is to incrementalize it.

11.2.1 Incremental batch processing

In Chapter 4 we discussed the tradeoffs between incremental algorithms and recomputation algorithms and how one of the primary benefits of the batch layer was its ability to take advantage of recomputation algorithms. So you may be surprised at the suggestion to incrementalize the batch layer. Like all design issues, you must consider all the tradeoffs to come to the best design.

Let's consider an extreme case, where the only view you're producing is a global count of all records in the master dataset. In this case, incrementalizing the batch layer is a clear win, as the incremental view is no bigger than a recomputation-based view (just a single number in both cases) and it's not complex to incrementalize the code. You save a huge amount of resources by not recomputing over the entire master dataset over and over. For instance, if your master dataset contains 100 terabytes of data, and each new batch of data contains 100 gigabytes, your batch layer will be orders of magnitude more efficient. Each iteration only has to deal with 100 gigabytes of data rather than 100 terabytes.

Now let's consider another example where the choice between incremental and recomputation algorithms is more difficult: the "birthday inference" problem. Imagine you're writing a web crawler that collects people's ages off of their public profiles. The profile does not contain a birthday but only what that person's age is at the moment you crawled that web page. Given this raw data of [age, timestamp] pairs, your goal is to deduce the birthday of each person.

The idea of the birthday inference algorithm is illustrated in Figure 11.2. Imagine you crawl the profile of "Tom" on January 4th, 2012 and see his age is 28. Then you crawl his profile again on January 11th, 2012 and see his age is 29. You can deduce that his birthday happened sometime between those two dates. Likewise, if you crawl the profile of "Jill" on October 20th, 2013 and see she is 43, and then crawl it again on November 4th, 2013 and see she is still 43, you know her birthday is not between those dates. The more "age samples" you have, the better you can infer someone's birthday to a small range of dates.

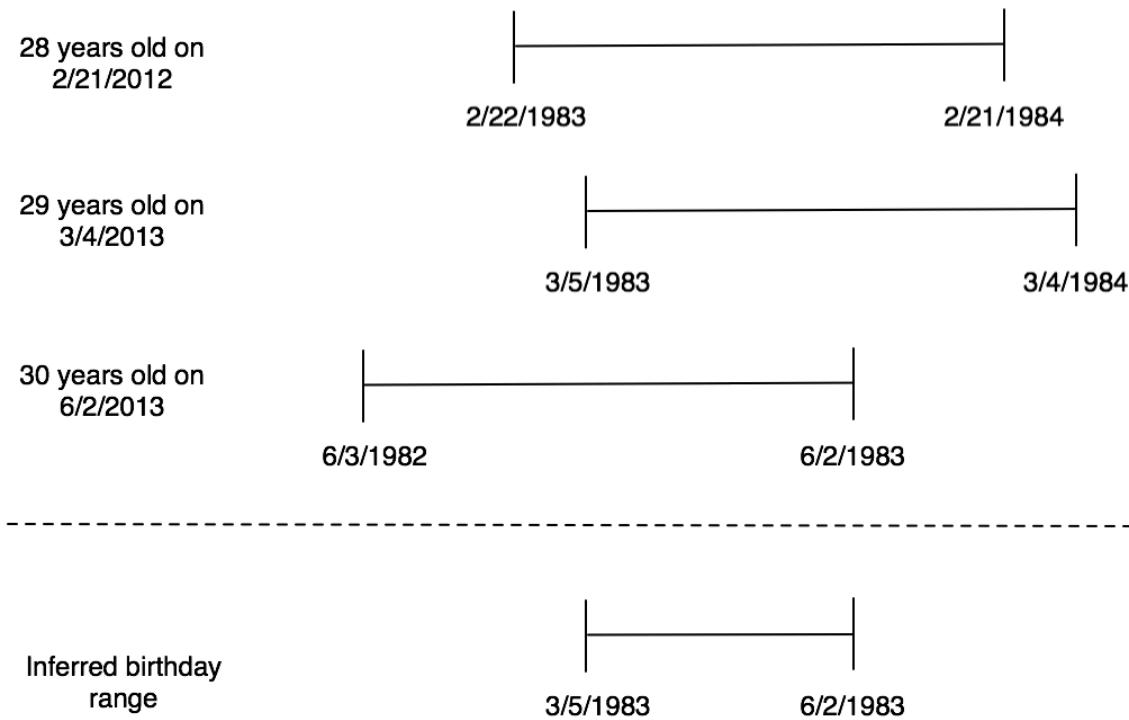


Figure 11.2 Basic birthday inference algorithm

In the real world, of course, the data can get messy. Someone may have incorrectly entered their birthday and then changed it at a later date. This may cause your age inference algorithm to fail to produce a birthday because every day of the year got eliminated as a possible birthday. You might modify your birthday inference algorithm to search for the smallest number of age samples it can ignore to produce the smallest range of possible birthdays. The algorithm might prefer to use recent age samples over older age samples.

If you implement a birthday inference batch layer using recomputation, it's easy. Your algorithm can look at all age samples for a person at once and do everything necessary to deal with messy data and emit a single range of dates as output. However, incrementalizing the birthday inference batch layer is much trickier. It's hard to see how you can deal with the messy data problem without having access to the full range of age samples. Incrementalizing this algorithm fully would be considerably harder and may require a much larger and more involved view.

There's an alternative that blurs the line between incrementalization and recomputation and gets you the best of both worlds. This technique is called "partial recomputation".

PARTIAL RECOMPUTATION

Recomputing every person's birthday from the age samples every single time the batch layer runs is wasteful. In particular, if a person has no new age samples since the last time the batch layer ran, then the inferred birthday for that person will not change at all. The idea behind a birthday inference batch layer based on partial recomputation is to:

1. For the new batch of data, find all people who have a new age sample.
2. Retrieve all age samples from the master dataset for all people in (1)
3. Recompute the birthdays for all people in (1) using the age samples in (2) and the age samples in the new batch
4. Merge the newly computed birthdays into the existing serving layer views

This is not fully incremental, because it still makes use of the master dataset. However, it avoids most of the cost of a full recompute by ignoring anyone who has not changed in the latest set of data.

You can easily see how partial recomputes as applied to the birthday inference problem could apply to many problems. The key idea is to retrieve all the relevant data for the people that changed, run a normal recompute algorithm on the retrieved data plus the new data, and then merge those results into the existing views. The nice thing about partial recomputes is that they can be implemented very efficiently. The most expensive step – looking over the entire master dataset to find relevant data – can be done relatively cheaply.

The key to making it efficient is to avoid having to repartition the entire master dataset, as this is the most expensive part of batch algorithms. For example, repartitioning happens whenever you do a group by operation or a join. Partitioning involves serialization/deserialization, network transfer, and possibly buffering on disk. In contrast, operations that don't require partitioning can quickly scan through the data and operate on each piece of data as it's seen. Retrieving relevant data for a partial recompute can be done using the latter method.

First, you construct a set of all the entities for which you need relevant data. You then scan over the master dataset and only emit data for whose entity exists in the set (each task would have a copy of that set). In a batch processing system like Hadoop this would correspond to a "map-only job".

You're limited by memory, so your set can only be so big. A data structure called a "bloom filter" can make this work for much, much larger sets of entities. A bloom filter is a compact data structure that represents a set of elements and allows

you to ask if it contains an element. A bloom filter is much more compact than a set but as a tradeoff query operations on it are probabilistic. A bloom filter will sometimes tell you an element exists in the set that was never added to it, but it will never tell you an element that was added to it is not in the set. So a bloom filter has false positives but no false negatives.

Using a bloom filter to optimize retrieving relevant data is illustrated in Figure 11.3. If you use a bloom filter to retrieve relevant data from the master dataset, you will filter out the vast majority of the master dataset but some data will be emitted that you didn't want to retrieve (because of the false positives). You can then do a join between the retrieved data and the list of desired entities to filter out the false positives. A join requires a partitioning, but since the vast majority of the master dataset was already filtered out, getting rid of the false positives is not an expensive operation.

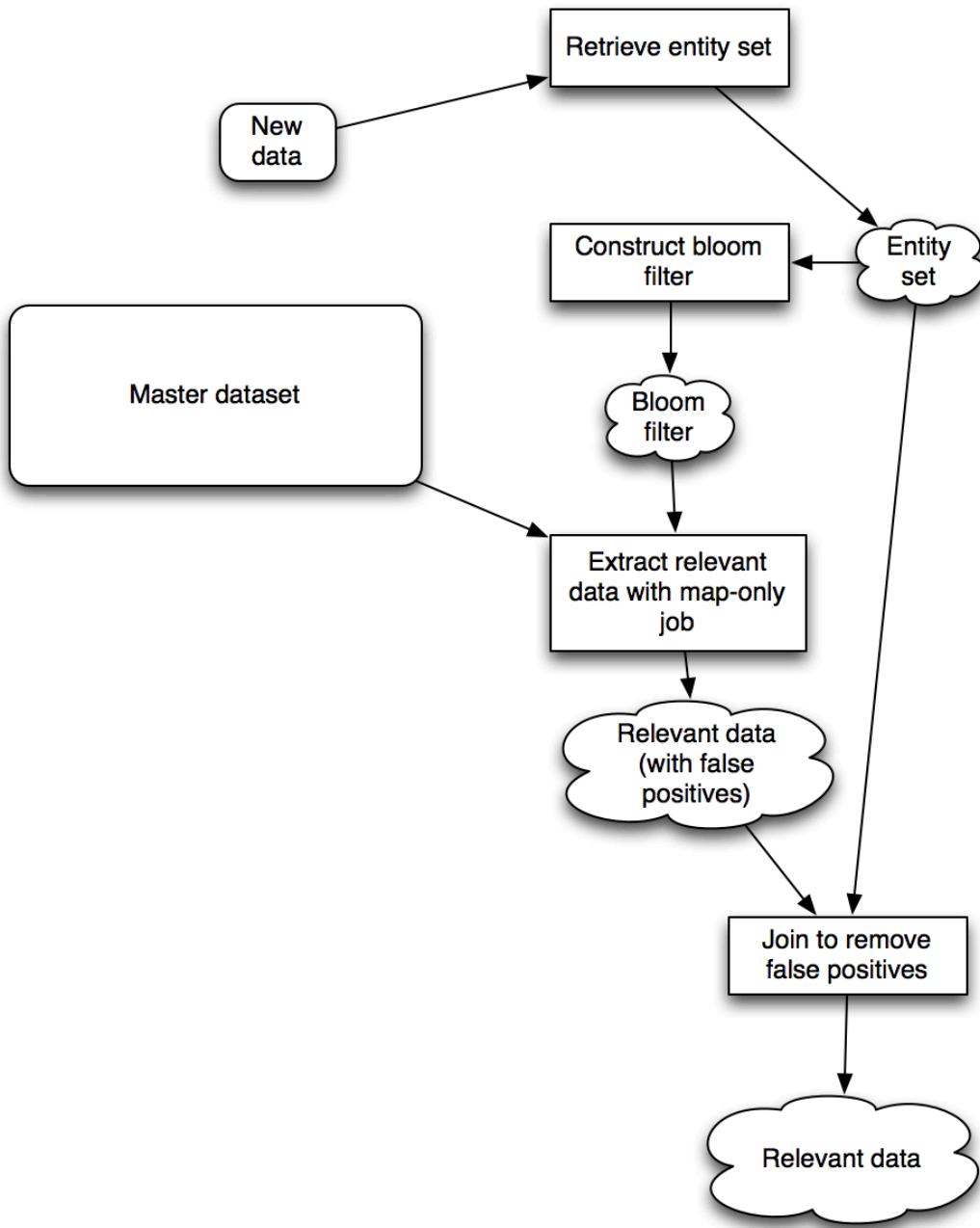


Figure 11.3 Bloom join

Let's now make some estimates as to how much of a latency improvement an incremental batch layer based on partial recomputes gets compared to a fully recomputation-based batch layer. Let's say computing birthday inference requires one full MapReduce job with partitioning, and that the following facts exist about your cluster and your data:

- Your master dataset contains 100 terabytes of data

- A partial recompute-based approach will have 50 gigabytes of new data each batch
- A MapReduce job with partitioning takes 8 hours on the full master dataset
- A map-only job (without any partitioning) takes 2 hours on the full master dataset (the 4x speed difference is typical in MapReduce clusters)
- Creating brand new serving layer views takes 2 hours in the full recompute
- Updating the serving layer views takes 1 hour in the partial recompute

With these numbers, recomputing all the birthday inference views from scratch would take 8 hours for the computation plus 2 hours to build the serving layer views. That's 10 hours total. For a partial recompute, it takes:

- 2 hours to get the relevant data from the master dataset
- A few minutes to compute the new birthdays for the entities in the current batch
- 1 hour to update the existing serving layer views with the newly computed birthdays

So the partial recompute based approach would only take about 3 hours, which is 70% faster than the full recompute based approach. These numbers are estimates, but it should give you a good idea of the kinds of performance improvements you could expect from a partial-recompute based batch layer. For more complex batch computations that require more than one partitioning step, the savings would be much greater. For instance, if the recomputation algorithm required 4 partitioning steps, the full recompute job would require 34 hours, while the partial recompute job would still only require about 3 hours.

Another benefit of partial recomputes is that they give you a certain amount of power to correct for human mistakes. If bad data is written that corrupts certain entities, you could fix your serving layers by doing a partial recompute of just the affected entities. This lets you get your application fixed in far less time than doing a complete full recompute. However, partial recomputes only help fix mistakes as long as you can identify all affected entities. For this reason partial recomputes are much more effective at fixing mistakes related to writing bad data than mistakes related to deploying buggy code that corrupts your views. The human fault-tolerance of partial recomputes is in between full recomputes and full incrementalization.

Partial recomputes, when appropriate, enable you to have a batch layer with far less latency without sacrificing the benefits of recomputation-based algorithms. They are generally not appropriate for realtime algorithms because that would

require indexing your entire master dataset, which would be extremely expensive. And obviously scanning over the entire master dataset in realtime is impossible. But for the batch layer, they're a great tool to have in your toolkit.

IMPLEMENTING AN INCREMENTAL BATCH LAYER

Whether you're doing fully incremental algorithms or partial recompute algorithms in your incremental batch layer, the main difference between an incremental batch layer and a recomputation-based batch layer is the need to update your serving layer views rather than create them from scratch.

It's absolutely viable to build the incremental batch layer similar to a speed layer, with the views being read/write databases that you modify in-place. However, this would negate the many advantages of serving layer databases (discussed in Chapter 7) that result from not supporting random writes, such as:

- *Robustness*: Not having random writes means the codebase is simpler and less likely to have bugs.
- *Easier to operate*: Less moving parts means there's less for you to worry about as an operator of these databases – less configuration and less that can go wrong.
- *More predictable performance*: By not having random writes that happen concurrently with reads, there's no need to worry about any sort of locking inside the database. Likewise, whereas in a read/write database it occasionally needs to compact its write-ahead log, which can significantly degrade performance, a database without random writes never needs to do this.

So let's focus on how to make serving layer databases that preserve these properties as much as possible. You saw one design for a serving layer database in Chapter 7 called ElephantDB.

The crux of how ElephantDB works is the batch layer view is indexed and partitioned in a MapReduce job and those indexes are stored on the distributed filesystem. An ElephantDB cluster periodically checks for new versions of the view and will hot swap the new version once it's available. The key point here is that the creation and serving of views are completely independent and coordinated through a distributed filesystem.

The way to extend this design to enable incremental batch processing is to include the last version of the batch layer view as input to the job that creates the new version of the batch layer view. Then updates are applied to the old version and the new version is written out to the distributed filesystem (ElephantDB implements this). For example, if you're using BerkeleyDB as your indexing system and storing word counts inside it, the job to create the new version of the

view would work like the following. The task for a given partition of the view would download the appropriate partition from the distributed filesystem, open it locally, increment word counts for its batch of data, and then copy the updated view into the distributed filesystem under the folder for the new version. In a strategy like this, all the incrementalization happens on the view creation side. Serving new versions of the view is no different than before.

A strategy like this saves you from redoing all the work that went into creating the prior version of the view. You can also take advantage of the higher latency of the batch layer to compact the indexes before writing them out to the distributed filesystem.

This strategy works better with moderate to smaller sized views. If the views themselves are huge, the cost of the jobs may be dominated by reading and writing the entire view to and from the distributed filesystem. In these cases incrementalization may not help very much. An alternative is to use a serving layer database design that ships "deltas" to the serving databases, and the serving databases merge them in on the fly. Of course, in that case you would also have to do compaction while serving, so the serving layer would look more and more like a read/write database and have many of the associated complexities.

Fortunately, there's a way to minimize the size of your incremental batch layer views so that you're not forced to use the "deltas" strategy or a read/write database for your serving layer. Instead you can keep the benefits of a serving layer where the creation and serving of views are completely independent. The idea is to have multiple batch layers.

MULTIPLE BATCH LAYERS

Instead of just having one batch layer and one speed layer that compensates for the latency of the batch layer, you can have multiple batch layers. For example, you could have one batch layer based on full recomputes that finishes once a month. Then you could have an incremental batch layer that only operates on data not represented in the full recompute batch layer. That might run once every six hours. Then you would have a speed layer that compensates for all data not represented in the two batch layers.

Like how in the basic Lambda Architecture the batch layer loosens the performance requirements of the speed layer, with multiple batch layers each layer loosens the requirements for the layer above it. In the example mentioned, the incremental batch layer only has to deal with two months of data. That means its views can be kept *much smaller* than if the views had to represent all data for all

time. So techniques like making brand new serving layer views based on the old serving layer views are feasible since the cost of copying the views won't dominate.

The other benefit to having multiple batch layers is it helps you get the best of both worlds of incrementalization and recomputation. Incremental workflows can be much more performant but lack the ability to recover from mistakes that recomputation workflows give you. If recomputation is a constantly running part of your system, you know you can recover from any mistake.

The latency of each layer of your system directly affects the performance requirements of the layer above it. So it's incredibly important to have a good understanding at how the latency of each layer is affected by the efficiency of your code and the amount of resources you allocate to them. Let's see how that plays out.

11.2.2 Measuring and optimizing batch layer resource usage

There turns out to be a lot of counterintuitive dynamics at work in the performance of batch workflows. Consider these examples which are based on real-world cases:

- After doubling the cluster size, the latency of a batch layer went down from 30 hours to 6 hours, an 80% improvement.
- An improper reconfiguration caused a Hadoop cluster to have 10% more task failure rates than before. This caused a batch workflow's runtime to go from 8 hours to 72 hours, a 9x degradation in performance.

It's hard at first to wrap your head around how this is possible, but the basic dynamic can be easily illustrated. Suppose you have a batch workflow that takes 12 hours to run, and so therefore it processes 12 hours of data each iteration. Now let's say you enhance the workflow to do some additional analysis, and you estimate the analysis will add two hours to the processing time of your current workflow. Now you increased the runtime of a workflow that operated on 12 hours of data to 14 hours. That means the next time the workflow runs, there will be 14 hours of data to process. Since the next iteration has more data, it will take longer to run. Which means the next iteration will have even more data, and so on.

Determining if and when the runtime stabilizes can be determined with some very simple math. First, let's write out the equation for the runtime of a single iteration of a batch workflow. The equation will make use of the following variables:

- T: the runtime of the workflow in hours

- O: the overhead of the workflow in hours. This is the amount of time spent in the workflow that is independent of the data being processed. This can include things like setting up processes, copying code around the cluster, etc.
- H: the number of hours of data being processed in the iteration. "Hours" are used here to measure the **amount** of data because it makes the resulting equations very simple. As part of this, it is assumed that the rate of incoming data is fairly constant. However, the conclusions we will make are not dependent on this.
- P: the "dynamic processing time". This is the amount of hours each hour of data adds to the processing time of the workflow. If each hour of data adds half an hour to the runtime, then P is 0.5.

Based on these definitions, the following equation is a natural expression of the runtime of a single iteration of a workflow:

$$T = O + P * H$$

Of course, H will vary with every iteration of the workflow, since if the workflow takes shorter or longer to run than the last iteration, the next iteration will have less or more data to process respectively. To determine the stable runtime of the workflow, you need to determine the point at which the runtime of the workflow is equal to the number of hours of data it processes. To do this, you simply plug in $T=H$ and solve for T:

$$T = O + P * T$$

$$T = O / (1 - P)$$

As you can see, the stable runtime of a workflow is linearly proportional to the amount of overhead in the workflow. So if you're able to decrease overhead by 25%, your workflow runtime will also decrease by 25%. However, the stable runtime of a workflow is non-linearly proportional with the dynamic processing rate "P". One implication of this is that there are diminishing returns on performance gains with each machine added to the cluster.

SIDE BAR

What happens if P is greater than or equal to one?

You may be wondering what would happen if your dynamic processing rate P is greater than or equal to one. In this case, each iteration of the workflow will have more data than the iteration prior, so the batch layer will fall further and further behind forever. It is incredibly important to keep P below one.

Using this equation, the counterintuitive cases described earlier make a lot more sense. Let's start with what happens to your stable runtime when you double the size of your cluster. When that happens, your dynamic processing rate "P" gets cut approximately in half, as you can now parallelize the processing twice as much

(technically your overhead to coordinate all those machines also increases slightly but let's ignore that). If "T1" is the stable runtime before doubling the cluster size, and "T2" is the stable runtime afterwards, you get these two equations:

$$T1 = O / (1 - P) \quad T2 = O / (1 - P/2)$$

Solving for the ratio $T2 / T1$ nets you this equation:

$$T2 / T1 = 1 - P / (2 - P)$$

Plotting this, you get the graph in Figure 11.4.

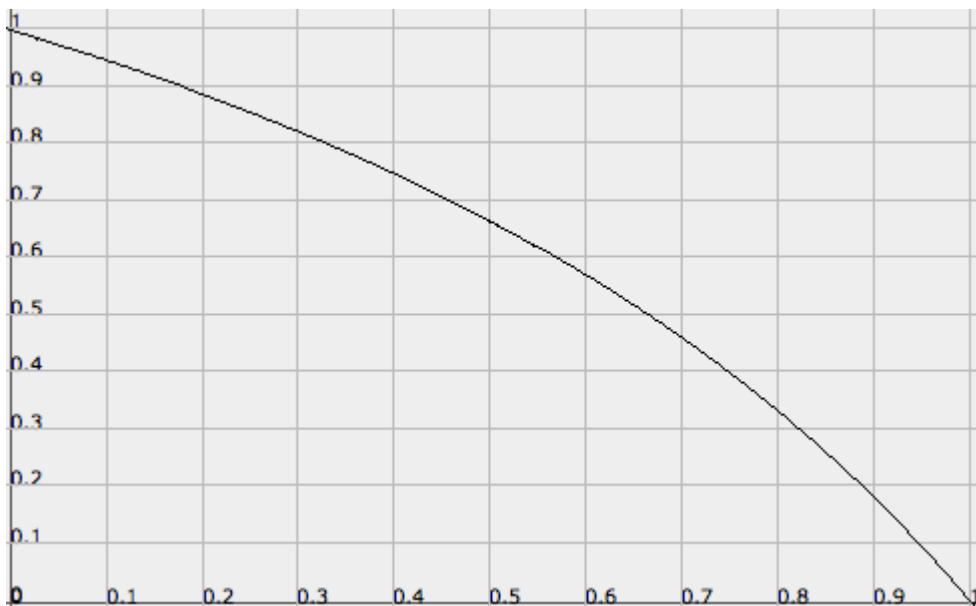


Figure 11.4 Performance effect of doubling cluster size

This graph says it all. If your P was really low, like 6 minutes of processing time per hour of data, then doubling the cluster size will barely affect the runtime. This makes sense because the runtime is dominated by overhead, which is unaffected by doubling the cluster size.

However, if your P was really high, say 54 minutes of dynamic time spent per hour of data, then doubling the cluster size will cause the new runtime to be 18% of the original runtime, a speedup of 82%! What happens in this case is the next iteration finishes much faster, causing the next iteration to have less data, upon which it will finish even faster. This positive loop eventually stabilizes at an 82% speedup.

Now let's consider the effect an increase in failure rates would have on your stable runtime. A 10% task failure rate means you'll need to execute about 11% more tasks to get your data processed. (If you had 100 tasks and 10 of them failed, you would retry those 10 tasks. However, on average 1 of those will also fail so you'll need to retry that one too). Since tasks are dependent on the amount of data

you have, this means your "time to process one hour of data" (P) will increase by 11%. Like the last analysis, let's call T_1 the runtime before the failures start happening and T_2 the runtime afterwards:

$$T_1 = O / (1 - P) \quad T_2 = O / (1 - 1.11 * P)$$

The ratio T_2 / T_1 is now given by the following equation:

$$T_2 / T_1 = (1 - P) / (1 - 1.11 * P)$$

Plotting this you get the graph in Figure 11.5.

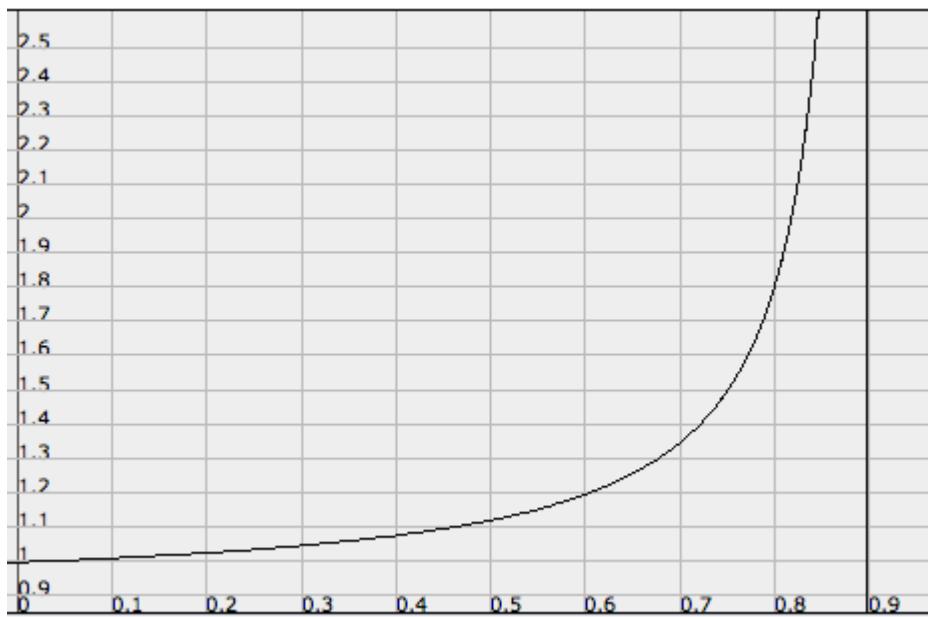


Figure 11.5 Performance effect of 10% increase in error rates

As you can see, the closer your P gets to 1, the more dramatic an increase in failure rates has on your stable runtime. This is how a 10% increase in failure rates can cause a 9x degradation in performance. It's important to keep your P away from 1 so that your runtime is stable in the face of the natural variations your cluster will experience. According to this graph a P below 0.7 seems pretty safe.

By optimizing your code, you can control the values for O and P . In addition, you can control the value for P with the amount of resources (e.g. machines) you dedicate to your batch workflow. The magic number for " P " is 0.5. When P is above 0.5, adding 1% more machines will decrease latency by more than 1%, making it a cost-effective decision. When P is below 0.5, adding 1% machines will decrease latency by less than 1%, making the cost-effectiveness more questionable.

To measure the values of O and P for your workflow, you may be tempted to run your workflow on zero data. This would give you the equation $T = O + P * 0$, allowing you to easily solve for O . You could then use that value to solve for P in

the equation $T = O / (1 - P)$. However, this approach tends to be inaccurate. For example, on Hadoop, a job typically has many more tasks than exist task slots on the cluster. It can take a few minutes for a job to "get going" and achieve full velocity by utilizing all the available task slots on the cluster. The time it takes to "get going" is normally a constant amount of time and so is captured by the O variable. When you run a job with a tiny amount of data, the job will finish before utilizing the whole cluster, skewing your measurement of O .

A better way to measure O and P is to artificially introduce overhead into your workflow, such as by adding a sleep(1 hour) call in your code. Once the runtime of the workflow re-stabilizes, you will now have two measurements T_1 and T_2 for before and after you added the overhead. You end up with the following equations to give you your O and P values:

- $O = T_1 / (T_2 - T_1)$
- $P = 1 - 1 / (T_2 - T_1)$

Of course, don't forget to remove the artificial overhead once you've completed your measurements!

When building and operating a Lambda Architecture, you use these equations to determine how many resources to give to each batch layer of your architecture. You want to keep P well below 1 so that your stable runtime is resilient to an increase in failure rates or an increase in the rate of data received. If your P is below 0.5, then you're not getting very cost-effective use of those machines so you should consider allocating them where they'd be better used. If O seems abnormally high, then you may have identified an inadvertant bottleneck in your workflow.

You should now have a good understanding of building and operating batch layers in a Lambda Architecture. The design of a batch layer can be as simple as a recomputation-based batch layer, or you may find you can benefit from making an incremental batch layer that's possibly combined with a recomputation-based batch layer. Let's now move on to the speed layer of the Lambda Architecture.

11.3 Speed layer

Since the serving layer updates with high latency, it is always out of date by some number of hours. However, the views in the serving layer represent the vast majority of the data you have – the only data not represented is the data that has arrived since the serving layer last updated. All that's left to make your queries realtime is to compensate for those last few hours of data. This is the purpose of the speed layer.

The speed layer is where you tend towards the side of performance in the tradeoffs you make – incremental algorithms instead of recomputation algorithms and mutable read/write databases instead of kinds of databases preferred in the serving layer. You need to do this because you need the low latency, and ultimately the lack of human fault-tolerance of these approaches doesn't matter. Since the serving layer constantly overrides the speed layer, mistakes in the speed layer are easily corrected.

Traditional architectures typically only have one layer which is roughly comparable to the speed layer. However, because there is no batch layer underpinning it, it is very vulnerable to mistakes that will cause data corruption. Additionally, the operational challenges of operating petabyte-scale read/write databases are enormous. The speed layer in the Lambda Architecture is largely free of these challenges because the batch and serving layers loosen its requirements to an enormous extent. Since the speed layer only has to represent the most recent data, its views can be kept very small, avoiding the aforementioned operational challenges.

In Chapters 8 through 10 you saw the intricacies and variations on building a speed layer, from queuing, synchronous vs. asynchronous speed layers, and one-at-a-time vs. micro-batch stream processing. You saw how for difficult problems you can make approximations in the speed layer to reduce complexity and/or increase performance.

11.4 Query layer

The last layer of the Lambda Architecture is the query layer which is responsible for making use of your batch and realtime views to answer queries. It has to determine what to use from each view and how to merge them together to achieve the proper result. Each query is formulated as some function(batch views, realtime views).

The merge logic you use in your queries will vary from query to query. The

different techniques you might use are best illustrated by a few examples.

Queries that are time-oriented have straightforward merging strategies – for example, the pageviews over time query from SuperWebAnalytics.com. To execute the pageviews over time query, you get the sum of the pageviews up to the hour for which the batch layer has complete data. Then you retrieve the pageview counts from the speed views for all remaining hours in the query and sum them with the batch view counts. Any query which is naturally split on time like this will have a similar merge strategy.

You would take a different approach for the "birthday inference" problem introduced earlier in this chapter. One way to do it is as follows:

- The batch layer runs an algorithm that will appropriately deal with messy data and choose a single range of dates as output. Along with the range it also emits the number of age samples that went into computing that range.
- The speed layer incrementally computes a range by narrowing the range with each age sample. If an age sample would eliminate all possible days as birthdays, it is ignored. This incremental strategy is fast and simple but does not deal with messy data well. That's fine though since that's handled by the batch layer. The speed layer also stores the number of samples that went into computing its range.
- To answer queries, the batch and speed ranges are retrieved with their associated sample counts. If the two ranges merge together without eliminating all possible days, then they are merged to the smallest possible range. Otherwise, the range with the higher sample count is used as the result.

This strategy for birthday inference keeps the views simple and handles all the appropriate cases. People that are new to the system will be appropriately served by the incremental algorithm used in the speed layer. It doesn't handle messy data as well as the batch layer, but it's good enough until the batch layer can do more involved analysis. This strategy also handles "bursts of new data" well. If you suddenly add a bunch of age samples to the system, the speed layer result will be used over the batch layer result which is based on less data. And of course, the batch layer is always recomputing birthday ranges so the results get more accurate over time. There are variations on this implementation you might choose to use for birthday inference, but you should get the idea.

Something that should be apparent from these examples is your views must be structured to be mergeable. This is natural for time-oriented queries like pageviews over time, but the birthday inference example specifically added sample counts into the views to help with merging. How you structure your views to make them mergeable is one of the design choices you must make in implementing a Lambda Architecture.

11.5 Conclusion

The Lambda Architecture is the result of starting from first principles – the general formulation of data problems as functions of all data you've ever seen – and making mandatory requirements like human fault-tolerance, horizontal scalability, and low latency reads and updates. As we've explored the Lambda Architecture, we made use of many tools to provide practical examples of the core principles, such as Hadoop, JCascalog, Kafka, Cassandra, and Storm. We hope it's been clear that none of these tools is an essential part of the Lambda Architecture. We fully expect the tools to change and evolve over time, but the principles of the Lambda Architecture will always hold.

In many ways, the Lambda Architecture goes beyond the currently available tooling. While implementing a Lambda Architecture is very doable today – something we tried to demonstrate by going deep into the details of implementing the various layers throughout this book – it certainly could be made even easier. There's only a few databases specifically designed to be used for the serving layer, and it would be great to have speed layer databases that can more easily handle the expiration of parts of the view that are no longer needed. Fortunately building these tools is much, much easier than the wide variety of traditional read/write databases being built, so we expect these gaps will be filled as more people adopt the Lambda Architecture. In the meantime, you may find yourself repurposing traditional databases for these various roles in the Lambda Architecture, and doing some engineering yourself to make things fit.

When first encountering big data problems and the big data ecosystem of tools, it's easy to be confused and overwhelmed. It's understandable to yearn for the familiar world of relational databases that we as an industry have become so accustomed to over the past few decades. We hope that by learning the Lambda Architecture you've learned that building Big Data systems can in fact be far simpler than building systems based on traditional architectures. The Lambda Architecture completely solves the normalization vs. denormalization problem, something that plagues traditional architectures, and also has human fault-tolerance built-in, something we consider to be non-negotiable. Its general purpose nature, due to being based on functions of all data, should give you the confidence to attack any data problem.