18 Lambda Architecture in depth

This chapter covers

- Revisiting the Lambda Architecture
- Incremental batch processing
- Efficiently managing resources in batch workflows
- Merging 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 various 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.

18.1 Defining data systems

We started with a simple question: "What does a data system do?" The answer was also simple: a data system answers questions based on data you've seen in the past. Or put more formally, a data system computes queries that are functions of all the data you've ever seen. This is an intuitive definition that clearly encapsulates any data system you'd ever want to build:

```
query = function(all data)
```

There are a number of properties you're concerned about with your queries:

- *Latency*—The time it takes to run a query. In many cases, your latency requirements will be very low—on the order of milliseconds. Other times 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, whereas 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 of 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 trade-offs with the preceding properties. For example, there's 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).

Because data systems are dynamic, changing systems built by humans and with new features and 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's 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 can't corrupt existing data, so your system is far more robust.

This leads us to the basic model of data systems:

- A master dataset consisting of an ever-growing set of data
- Queries as 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.

18.2 Batch and serving layers

Computing queries as functions 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 you can make to such an architecture is to query precomputed views rather than the master dataset directly. These precomputed views can be tailored for the queries so that the queries are as fast as possible, whereas the views themselves are functions of the master dataset.

In chapters 2 through 9, 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 done in the *serving layer*, which 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 6).

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.

18.2.1 Incremental batch processing

In chapter 6 we discussed the trade-offs between incremental algorithms and recomputation algorithms. You saw how one of the primary benefits of the batch layer is 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 trade-offs in order 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 repeatedly recomputing over the entire master dataset. 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 from their public profiles. The profile doesn't 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 18.1. Imagine you crawl the profile of Tom on January 4, 2012, and see his age is 23. Then you crawl his profile again on January 11, 2012, and see his age is 24. You can deduce that his birthday happened sometime between those two dates. Likewise, if you crawl the profile of Jill on October 20, 2013, and see she is 43, and then crawl it again on November 4, 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 that someone's birthday is within a small range of dates.

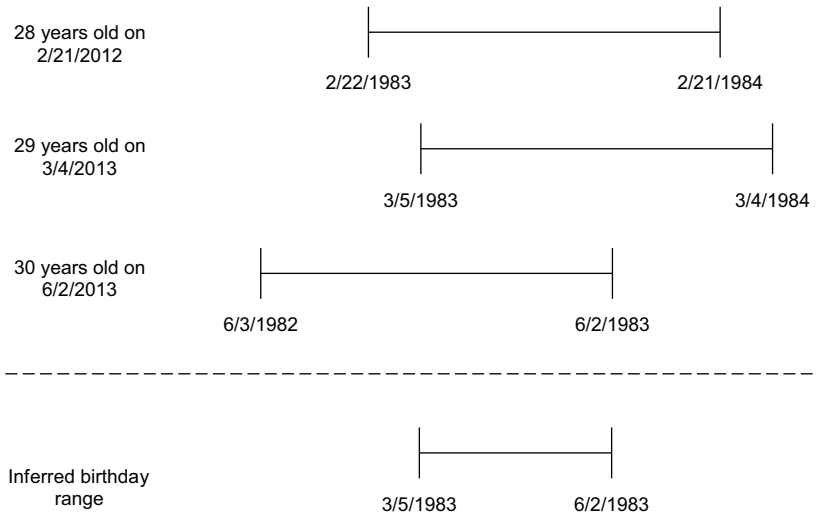

Figure 18.1 Basic birthday-inference algorithm

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 has been 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. But 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 do the following:

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 step 1.
3. Recompute the birthdays for all people in step 1 using the age samples from step 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. But it avoids most of the cost of a full recomputation by ignoring anyone who hasn't 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 entities 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.

The first step to retrieving relevant data is to 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 those entities that exist 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. But a data structure called a *Bloom filter* can make this work for 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 trade-off, query operations on it are probabilistic. A Bloom filter will sometimes incorrectly tell you that an element exists in the set, but it will never tell you an element that was added to the set 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 18.2. If you use a Bloom filter to retrieve relevant data from the master dataset, you'll filter out the vast majority of the master dataset. Due to the false positives, though, some data will be emitted that you didn't want to retrieve. 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 because the vast majority of the master dataset was already filtered out, getting rid of the false positives is not an expensive operation.

Let's now make some estimates as to how much of a latency improvement an incremental batch layer based on partial recomputes offers 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, here are the figures:

- It takes 2 hours to get the relevant data from the master dataset.
- It takes a few minutes to compute the new birthdays for the entities in the current batch.
- It takes 1 hour to update the existing serving layer views with the newly computed birthdays.

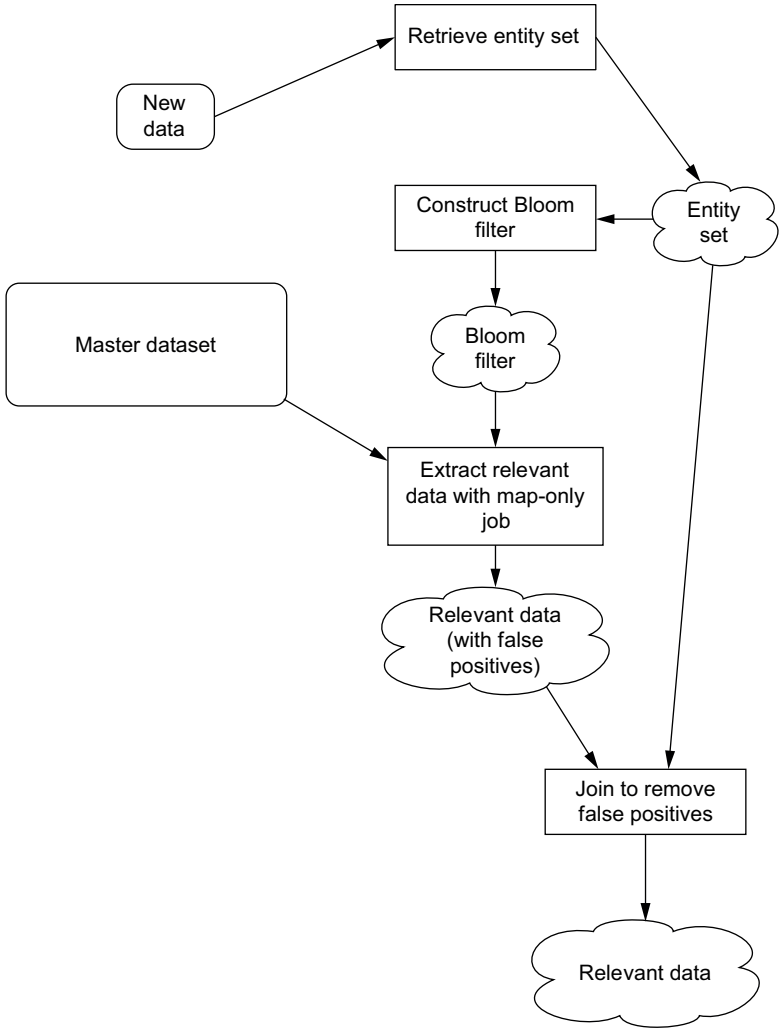

Figure 18.2 **Bloom join**

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 four 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. But 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're 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 real time is impossible. But for the batch layer, partial recomputes are 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. But this would negate the many advantages of serving layer databases (discussed in chapter 10) that result from not supporting random writes:

- *Robustness*—Not having random writes means the codebase is simpler and less likely to have bugs.
- *Easier to operate*—Fewer 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 a read/write database occasionally needs to perform compaction to reclaim unused parts of the index, 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 called ElephantDB in chapter 11.

The crux of how ElephantDB works is that 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 as follows. 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'd 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.

In the basic Lambda Architecture, the batch layer loosens the performance requirements of the speed layer; similarly, 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 because the cost of copying the views won't dominate.

The other benefit to having multiple batch layers is that 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 of 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.

18.2.2 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, so 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. You've now 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. Because 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.

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's independent of the data being processed. This can include things like setting up processes, copying code around the cluster, and so on.
- 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's assumed that the rate of incoming data is fairly constant. But the conclusions we'll make are not dependent on this.
- P—The dynamic processing time. This is the number 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:

$$T = O + P \times H$$

Of course, H will vary with every iteration of the workflow, because 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:

$$T = O + P \times T$$
$$T = \frac{O}{(1 - P)}$$

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 time, P. One implication of this is that there are diminishing returns on performance gains with each machine added to the cluster.

What happens if *P* is greater than or equal to 1?

You may be wondering what would happen if your dynamic processing time, P, is greater than or equal to 1. 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's incredibly important to keep P below 1.

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 time, 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 afterward, you get these two equations:

$$T1 = \frac{O}{(1 - P)}$$
$$T2 = \frac{O}{(1 - P/2)}$$

Solving for the ratio $T2/T1$ nets you this equation:

$$\frac{T2}{T1} = \frac{1 - P}{(2 - P)}$$

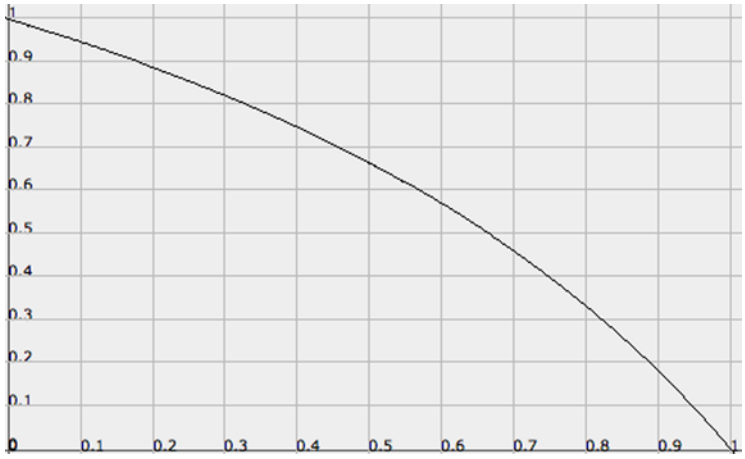

Figure 18.3 **Performance effect of doubling cluster size**

Plotting this, you get the graph in figure 18.3.

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'd retry those 10 tasks. However, on average 1 of those will also fail, so you'll need to retry that one too.) Because 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%.

As in the last analysis, let's call $T1$ the runtime before the failures start happening and $T2$ the runtime afterward:

$$T1 = \frac{O}{(1 - P)}$$

$$T2 = \frac{O}{(1 - 1.11 \times P)}$$

The ratio $T2/T1$ is now given by the following equation:

$$\frac{T2}{T1} = \frac{(1 - P)}{(1 - 1.11 \times P)}$$

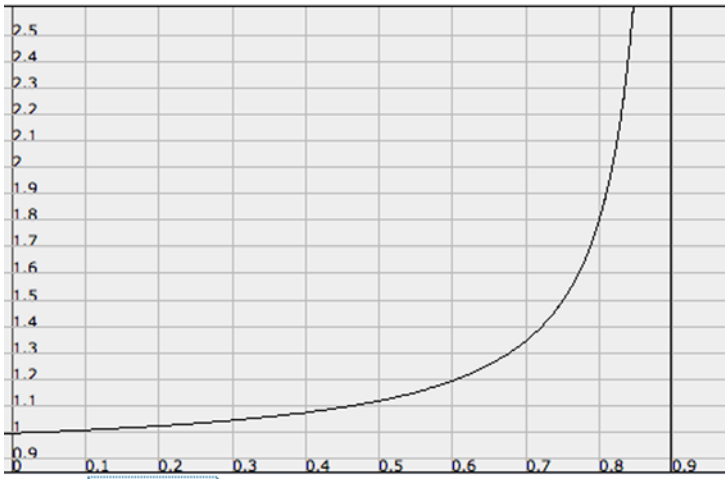

Figure 18.4 Performance effect of 10% increase in error rates

Plotting this, you get the graph in figure 18.4.

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 (such as 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% more 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 $T = O + P \times 0$, allowing you to easily solve for *O*. You could then use that value to solve for *P* in the equation $T = O / (1 - P)$. But this approach tends to be inaccurate. For example, on Hadoop, a job typically has many more tasks than there are 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 stabilizes, you'll 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:

$$O = \frac{T1}{(T2 - T1)}$$
$$P = \frac{(1 - 1)}{(T2 - T1)}$$

Of course, don't forget to remove the artificial overhead once you've completed your measurements!

When building and operating a Lambda Architecture, you can 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 inadvertent 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.

18.3 Speed layer

Because the serving layer updates with high latency, it's always out of date by some number of hours. But 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 toward the side of performance in the trade-offs you make—incremental algorithms instead of recomputation algorithms and mutable read/write databases instead of the kinds of databases preferred in the serving layer. You need to do this because you need the low latency, and the lack of human-fault tolerance in these approaches doesn't ultimately matter. Because 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. But because there's no batch layer underpinning it, it's 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. Because 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 12 through 17 you saw the intricacies and variations on building a speed layer, involving queuing, synchronous versus asynchronous speed layers, and one-at-a-time versus micro-batch stream processing. You saw how for difficult problems you can

make approximations in the speed layer to reduce complexity, increase performance, or both.

18.4 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 of batch views and 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, such as 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 that's naturally split on time like this will have a similar merge strategy.

You'd 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's ignored. This incremental strategy is fast and simple but doesn't deal with messy data well. That's fine, though, because 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're 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 later. 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 because it's based on more 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 that 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.

18.5 Summary

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, low-latency reads, and low-latency 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. Although 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 are 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 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 be far simpler than building systems based on traditional architectures. The Lambda Architecture completely solves the normalization versus denormalization problem, something that plagues traditional architectures, and it also has human-fault tolerance built in, something we consider to be non-negotiable. Additionally, it avoids the plethora of complexities brought on by architectures based on monolithic read/write databases. Because it's based on functions of all data, the Lambda Architecture is by nature general-purpose, giving you the confidence to attack any data problem.