3 Transporting the data from collection tier: decoupling the data pipeline

This chapter covers

- Understanding the need for the message queuing tier
- Understanding message durability
- Accommodating offline consumers
- Understanding message delivery semantics
- Choosing the right technology

So far I've talked about the role of handling the incoming data, not the output of data, from the collection tier. This chapter focuses on transporting data from the collection tier to the rest of the streaming pipeline. Although I may mention the collection and analysis tiers in the discussion, the discussion will only be concerned with getting messages from or to those tiers via the message queuing tier. Figure 3.1 shows our streaming architecture with this focus in mind.

After completing this chapter you will have a solid understanding of why we need a message queuing tier, what the core features of the common products used in this tier are, and how to determine which features are important for your streaming data system.

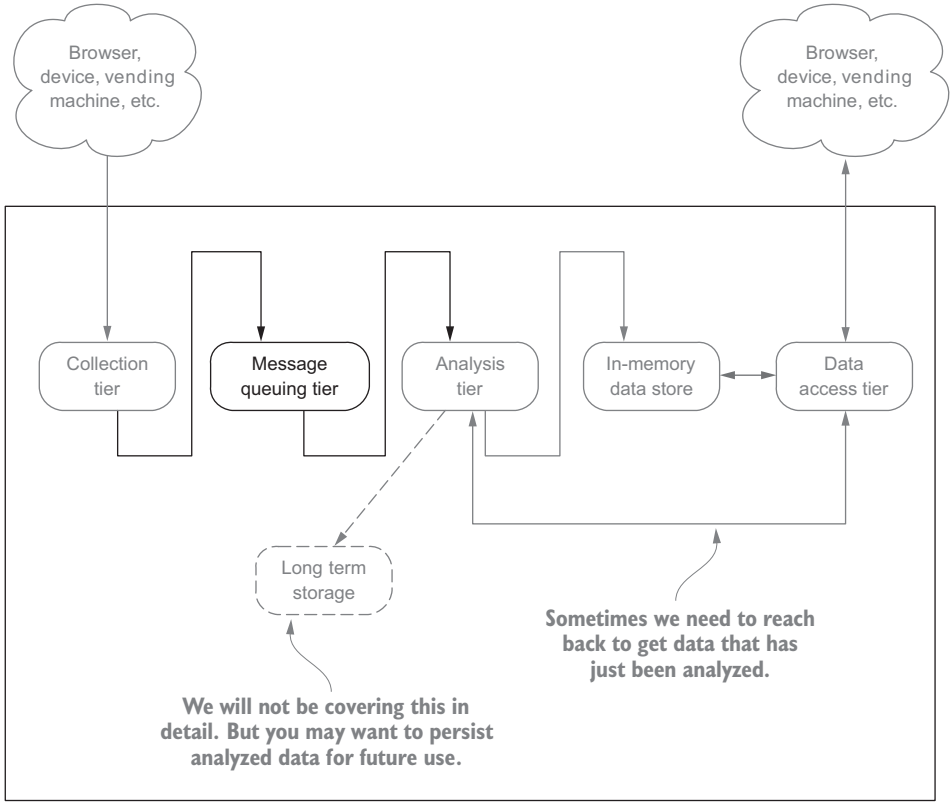

Figure 3.1 The message queuing tier with its input and output as the focus

3.1 Why we need a message queuing tier

If you're new to building data pipelines, and in particular streaming systems, you may look at figure 3.1 and think, "Okay, I figured the output from the collection tier went to the message queuing tier and then the data magically flows to the analysis tier, so what's the big deal? And why do we need this message queuing tier at all?" Those are great questions, so let's imagine for a second that our streaming architecture did not have the message queuing tier as part of it. Figure 3.2 shows a redrawn architecture without this tier.

We may be tempted to say this looks simpler and things should work fine, but don't give in to this desire for what appears to be a simplification of the architecture. Sure, you may be able to bring up an entire streaming system on a single machine and have each layer directly call the next, but your streaming system will span across many machines. When designing a software system, one desirable quality to strive for is the decoupling of the various components. With a streaming system we want the same: to decouple the components in each tier—but more importantly, to decouple the tiers

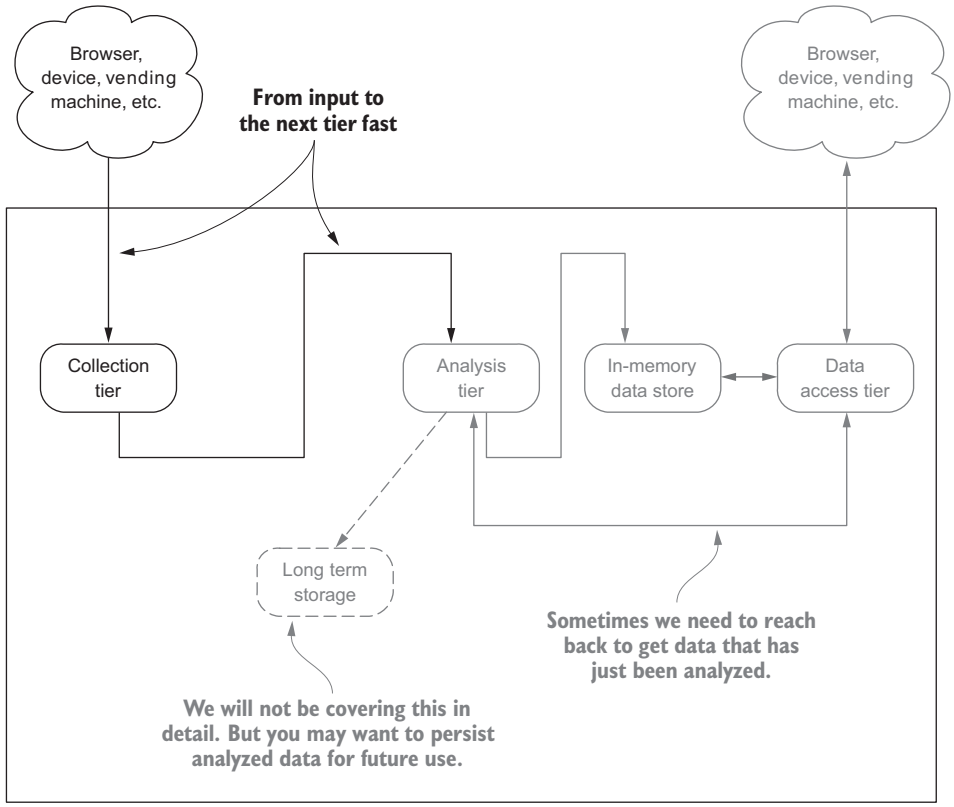

Figure 3.2 From the collection tier straight to the analysis tier

from each other. At the heart of a streaming system, like any distributed system, is the communication between the numerous machines that compose the system. If you look up *interprocess communication* in the literature[1] you will find numerous models; this chapter focuses on the *message queuing model*. By adopting this model, our collection tier will be decoupled from our analytics tier. This decoupling allows our tiers to work at a higher level of abstraction, by passing messages and not having explicit calls to the next layer. These are two good properties to have in any system, let alone a distributed streaming one. As you will see in this and upcoming chapters, this decoupling of the tiers provides wonderful benefits.

3.2 Core concepts

Let's look at the features of a message queuing product that are critical to the success of our streaming system. But first let's get one more formality out of the way—our use of the term *message queuing*. I use such terms broadly to encompass the spectrum

[1] An overview of inter-process communication, https://en.wikipedia.org/wiki/Inter-process_communication.

of messaging services, from the traditional message queuing products (RabbitMQ, ActiveMQ, HornetQ, and so on) to the newer takes on messaging found in NSQ, ZeroMQ, and Apache Kafka. Apache Kafka has grown to embody more than a message system, and I'm going to focus on the fact that it lets us publish and subscribe to streams of records. When you consider this functionality, it is similar to a message queue or enterprise messaging system.

This section discusses where this tier fits in the larger streaming architecture and talks about the core features to consider when selecting a message queuing product—only the ones you want to pay attention to when designing a streaming system. Armed with this information, you will be able to objectively think about the problem we are trying to solve in the context of what is important to your business, making the selection of the correct tool easier.

Before diving into the core features, let's make sure we have an understanding of the components of a message queuing product and how they map to our streaming architecture.

3.2.1 The producer, the broker, and the consumer

In the message queuing world there are three main components: the *producer*, the *broker*, and the *consumer*. Each plays an important role in the overall functioning and design of the message queuing product. Figure 3.3 shows how they fit together in their simplest form.

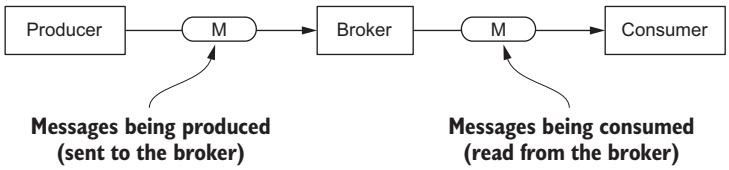

Figure 3.3 The three core parts to a message queuing system

You can see that the producer and the consumer have jobs that closely match their names: the producer produces messages, and the consumer consumes messages. Note in figure 3.3 that the term *broker* is used and not *message queue*. Why the change? Well, it's not so much a change as it is an abstraction, and this abstraction is important because a broker may manage multiple queues. Figure 3.4 shows that the message queue is alive and well but is abstracted away by the broker.

Now the data flow should start to make more sense. If you follow the flow from left to right in figure 3.4, you will see the following steps taking place:

- The producer sends a message to a broker
- The broker puts the message into a queue
- The consumer reads the message from the broker

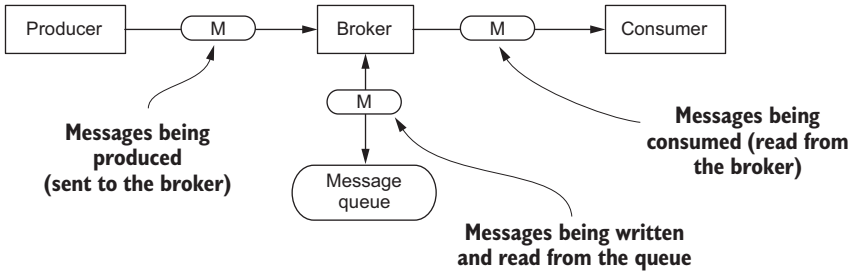

Figure 3.4 The broker with the message queue being shown

To put this in perspective, figure 3.5 shows what overlaying these terms and pieces onto our streaming architecture looks like.

You probably would agree that this seems pretty simple and straightforward, but as the saying goes, the devil is in the details. It is to these details—the subtle interactions between the producer, broker, and consumer, as well as various behaviors of the broker—that we will now turn our attention.

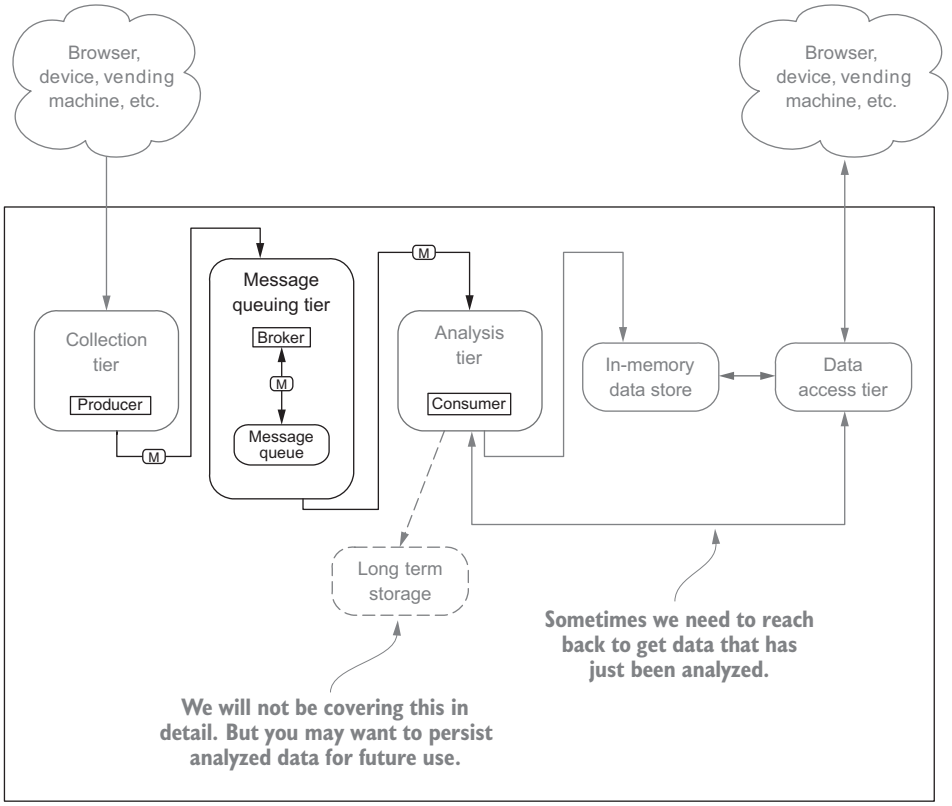

Figure 3.5 Streaming architecture with message queuing tier components shown in context

3.2.2 Isolating producers from consumers

As mentioned, one of our goals is to decouple the different tiers in the system. The message queuing tier allows us to do this with the collection and analysis tiers. Let's explore several reasons why this is a quality we strive for and the benefits we get from it.

Depending on your business need and the design of your streaming system, you may find yourself in a situation where the producer (collection tier) is generating messages faster then the consumer (analysis tier) can consume them. Often this is because your analysis tier is more processing-intensive than the collection tier and may not be able to process the data as quickly, which presents the following challenge: What if our consumers can't consume data fast enough from the collection tier? How do we prevent our analysis tier from being overwhelmed by surges in the events produced in the collection tier?

To me this conjures an old cartoon picture of a hose with the end plugged and the water spigot opened—it starts to swell and eventually explodes from the backpressure. Taking that example to our realm of data, figure 3.6 shows a time-lapse of the data flowing from the collection tier to the analysis tier.

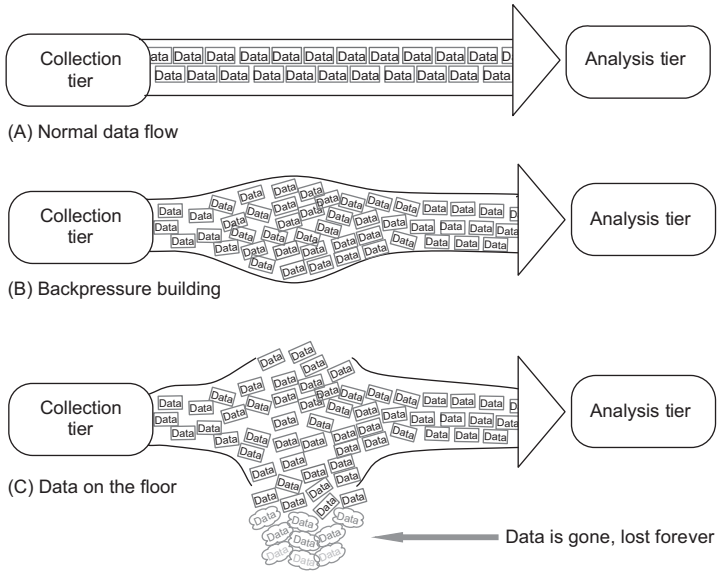

Figure 3.6 The three stages of data flowing without a message queue. We don't want step C.

Let's take figure 3.6 step-by-step:

- *Step A*—This looks pretty normal and is what we would like to see.
- *Step B*—We can tell something is not quite right—backpressure is building.
- *Step C*—Our data pipe broke under pressure, and data is now virtually dropping onto the floor and is gone forever.

Ouch! This is not a good situation because we are now losing data, and for some businesses that can be catastrophic. At first blush, you may think this is a consumer problem, and all we have to do is add more consumers or make them faster so they can keep up and life will be good. But this is not a consumer problem at all; it is perfectly acceptable in many use cases for consumers to read slowly or be offline from time to time. For example, consumers may only want to read all the messages on an hourly basis to support a batch-processing use case; they will be offline from time to time and then connect and read the last hour's worth of data.

Remember, not all message queuing systems provide this type of producer flow control, leaving it up to you, the application developer, to control the rate at which your collection tier is producing messages. If you don't, you may overwhelm your consumer and the broker. The ability to support a consumer reading messages slowly or being offline from time to time is provided by message queuing products that support durable messaging.

3.2.3 Durable messaging

Why do you need to worry about durable messages and offline consumers in a book about building streaming data systems? Great question! Let's look at what else we get in exchange for using a message queuing product that supports offline consumers. Imagine that the data centers for your business are geographically dispersed. You have two data centers: one in Amsterdam, the Netherlands, and the other in San Diego, CA, as shown in figure 3.7.

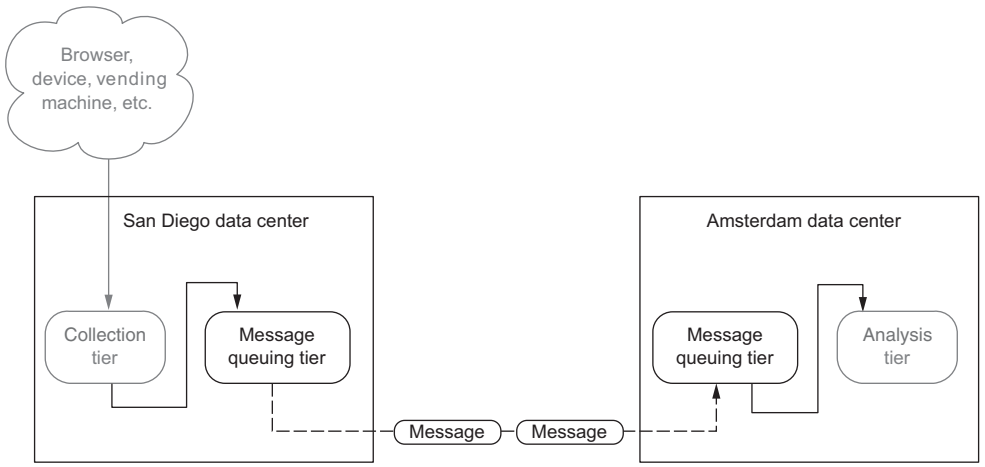

Figure 3.7 Two data centers with data flowing between them

In the San Diego data center you have the collection tier running, and in the Amsterdam data center you're running the analysis tier. I haven't talked about the analysis

tier yet, but it needs the data from the collection tier. All right, you are collecting data in San Diego and analyzing it in Amsterdam. Things are running smoothly and business is good. But as luck would have it, right as you were about to leave for the weekend on a beautiful Friday afternoon, a construction worker accidently put a backhoe through a fiber optic line, cutting off communication between your data centers, as illustrated in figure 3.8.

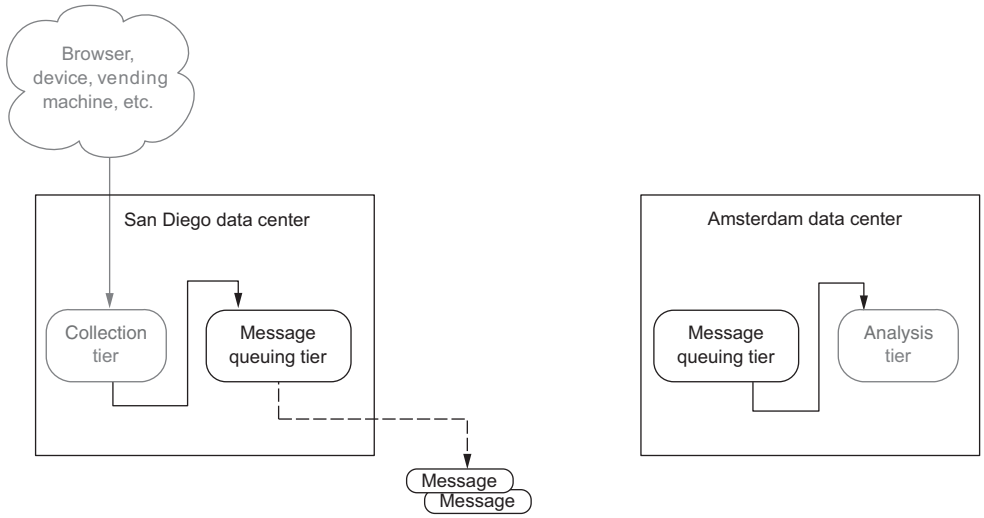

Figure 3.8 Two data centers with data flowing into the ocean

The telecom company that owns the fiber line says its best guess is that it may take two or three days to repair it. What would be the impact on your business? How much data can your business tolerate losing from your collection tier? If you couldn't tolerate losing days of data, make sure you choose a message queuing technology that can persist messages for the long term. Figure 3.9 shows how durable messaging fits in with this tier and some of the types you may find.

Having durable messages not only provides a degree of fault tolerance—and therefore disaster recovery—it allows for the offline consumer scenario mentioned earlier.

Imagine you've built a real-time traffic-routing system that allows people driving around any city to use your smartphone app to get updates and be re-routed based on up-to-the-moment traffic conditions. Three months pass, and now your business wants to offer a historical traffic-replay product that lets a user pick a city and replay the traffic data for a given day, week, or month. If your architecture is similar to figure 3.10, then as the analysis tier consumes messages they are discarded from the message queue—in essence, they are gone, and you can't provide your historical traffic replay.

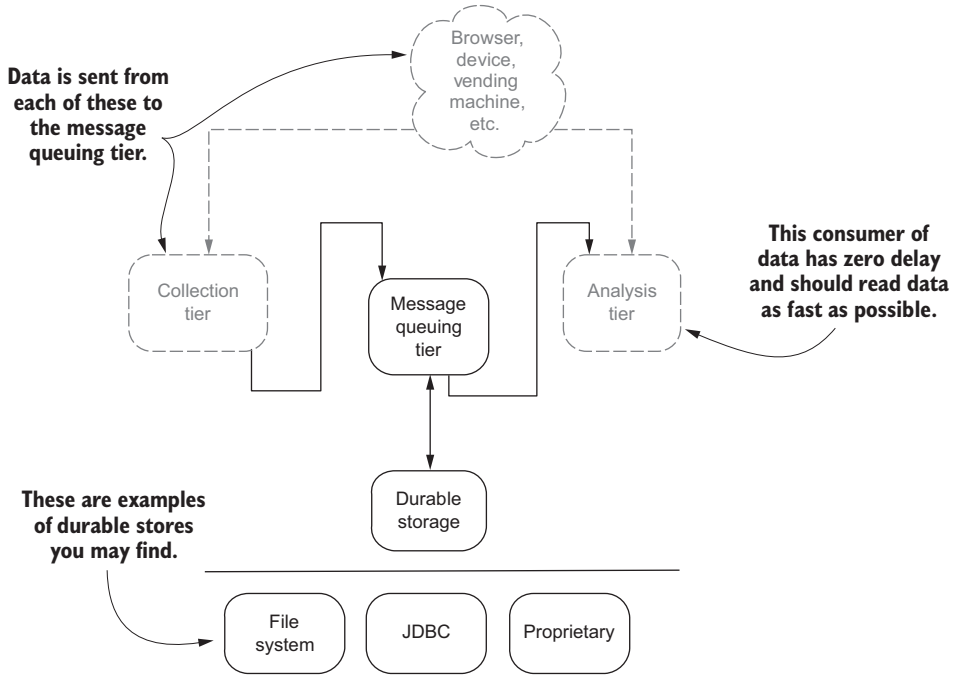

Figure 3.9 Durable messages—where they fit and how they may be stored

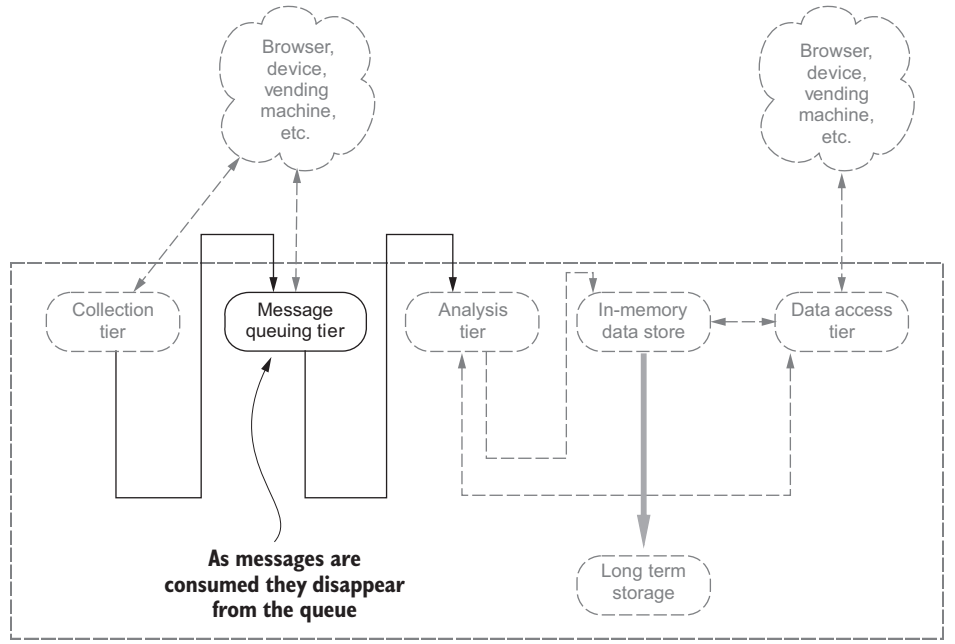

Figure 3.10 Transient messages get discarded after the analysis tier consumes them.

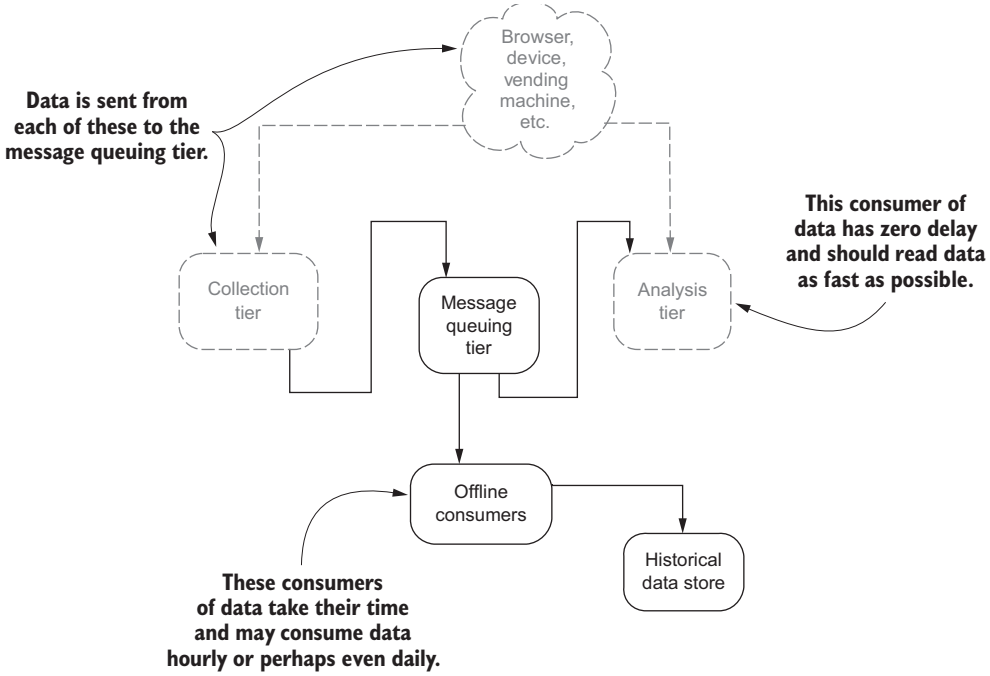

Figure 3.11 Offline consumers persisting data for historical reporting/analysis

To solve this problem, you need an architecture more like figure 3.11.

If your business case requires supporting offline consumers, or you want to be sure your producers and consumers can be completely decoupled, look for a product that supports durable messaging.

3.2.4 Message delivery semantics

Remember, a producer sends messages to a broker, and the consumer reads messages from a broker. That's a pretty high-level description of how message delivery works. Let's go deeper and explore the common semantic guarantees found in messaging products. The following are the three common semantic guarantees you will run into when looking at message queuing products:

- *At most once*—A message may get lost, but it will never be reread by a consumer.
- *At least once*—A message will never be lost, but it may be reread by a consumer.
- *Exactly-once*—A message is never lost and is read by a consumer once and only once.

If you had to pick the one you wanted, which would it be? If you said *exactly-once* you're not alone. In fact, most people want a system in which messages are never lost and each message is delivered to a consumer once and only once. Who wouldn't

want that? If only it were that simple—such a system comes with caveats and risks. Figure 3.12 shows possible points of failure.

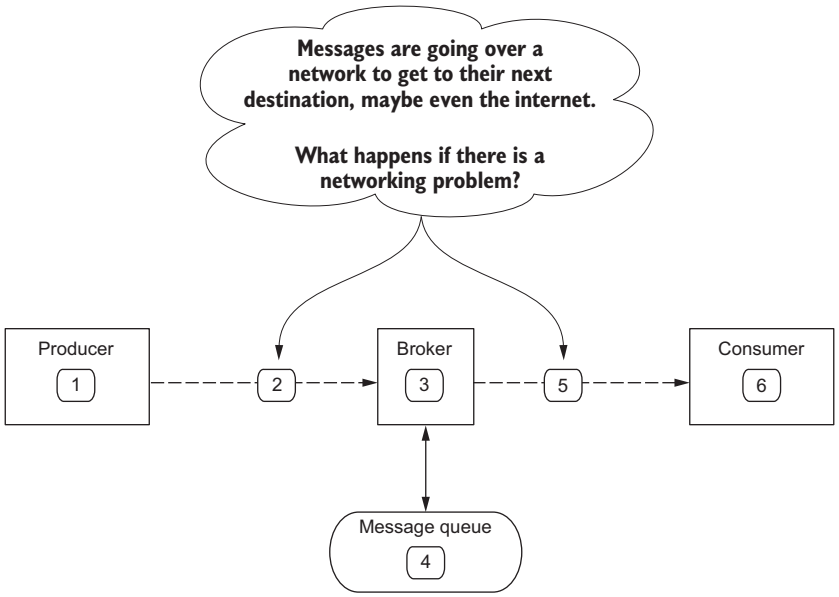

Figure 3.12 Possible points of failure that need to be considered

Wow! It seems as if almost every spot in the diagram is a possible point of failure. But don't worry, it's not all doom and gloom. Let's walk through them and understand what the risks are and what each numbered item in figure 3.12 means:

1. *Producer*—If the producer fails after generating a message but before sending it over the network to the broker, we will lose a message. There is also a chance that the producer may fail waiting to hear back from the broker that it did receive the message, and the producer after it recovers may send the same message a second time.
2. *The network between the producer and broker*—If the network between the producer and the broker fails, the producer may send the message, but the broker never receives it or the broker does receive it but the producer never gets the response acknowledging it. In both cases the producer may send the same message a second time.
3. *Broker*—If the broker fails with messages that are still held in memory and not committed to a persistent store, we may lose messages. If the broker fails before sending an acknowledgment to the producer, the producer may send the message a second time. Likewise if the broker tracks the messages consumers have read and fails before committing that information, a consumer may read the same message more than once.

4. *Message queue*—If the message queue is an abstraction over a persistent store, then if it fails trying to write data to disk we may end up losing messages.
5. *The network between the consumer and broker*—If the network between the consumer and the broker fails, the broker may send a message and record that it was sent, but the consumer may never get it. From the consumer side, if the broker waits for the consumer to acknowledge receipt of a message but that acknowledgment never gets to the broker, it may send the consumer the same message a second time.
6. *Consumer*—If the consumer fails before being able to record that it processed a message, either by sending an acknowledgment to the broker or to a persistent store, it may request the same message from the broker. Another twist here is the case where there are multiple consumers and more than one of them reads the same message.

I know that is a lot to consider and it may seem a little overwhelming, but don't worry. This won't be the last time we discuss these types of semantics. In the context of a message queuing system, we need to keep these failure scenarios in our back pocket so that when a messaging system claims to provide exactly-once deliver semantics, we can understand whether it truly does. As with so many things, the choice of which technology to use in this case involves various tradeoffs, as listed in table 3.1.

Table 3.1 Tradeoffs of a message queuing system

Less complexity, faster performance, and weaker guarantees	vs.	More complexity, a performance hit, and a strong guarantee

Where to compromise should be based on the business problem you're trying to solve with the streaming system. If you're building a streaming web analytics product, missing a message here or there is not going to have much of an impact on your product. Conversely, for a streaming fraud detection system, missing a message can have an undesirable effect.

As you look at messaging systems, you may find that the one you want to use doesn't provide exactly-once guarantees, such as Apache Kafka and Apache ActiveMQ. But don't despair. Often the system will provide enough metadata about the messages that you can implement exactly-once semantics with some coordination between producer(s) and consumer(s). Let's explore the techniques we would use to solve this problem. Figure 3.13 illustrates them graphically.

Figure 3.13 identifies the producer and consumer techniques. Let's talk about those in more detail:

- *Do not retry to send messages*—This is the first technique we must use. To do this, you need to have in place a way to track the messages your producer(s) sends to the broker(s). If and when there is no response or a network connection is interrupted between your producer(s) and the broker(s), you can read data from

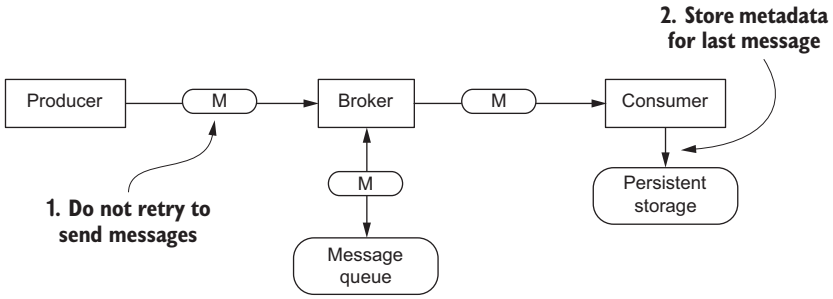

Figure 3.13 The two ways to have exactly-once semantics if the messaging system doesn't provide it

the broker to verify that the message you didn't receive an acknowledgment for was received. By having this type of message tracking in place, you can be sure your producer only sends messages exactly-once.

- *Store metadata for last message*—The second technique we must use involves storing some data about the last message we read. The metadata you store will vary by messaging system. If you're using a JMS-based system, you may store the JMSMessageID; if you're using Apache Kafka, you would store the message offset. In the end, what you need is data about the message so you can be sure your consumer doesn't reprocess a message a second time. Figure 3.13 shows the metadata being stored in a persistent store. One thing to take into consideration is what to do if there's a failure storing the metadata.

If you implement these two techniques, you can guarantee exactly-once messaging. You may not have noticed during this discussion, but by doing so you also get two nice little bonuses—sorry, not that type of bonus. I was thinking more about the data quality and robustness of your system. Look again at figure 3.13 and the subsequent discussion. What do you think the bonuses are? There may be more, but the ones I was thinking of are message auditing and duplicate detection.

From a message auditing standpoint, you're already going to keep track of the messages your producer sends via metadata. On the consumer side you can use this metadata to keep track of not only messages arriving but also the max, min, and average time it takes to process a message. Perhaps you can identify a slow producer or slow consumer. Regarding duplicate detection, we already decided that our producer was going to do the right thing to make sure a message was only sent to a broker one time. On the consumer side, we said it was going to check to see whether a message has already been processed.

One extra thing to keep in mind: on the consumer side, don't merely keep track of metadata related to the messaging system (some will expose a message ID of some sort so you know if you processed a message by the same ID). Also be sure to keep track of metadata that you can use to distinctly identify the payload of a message. This will make the de-duplication of messages and auditing of data easier.

Now you know how to ensure exactly-once semantics, and you're on your way to providing message auditing and detecting message duplication. In this book you'll run into these concepts again, and you may see other ways to apply message auditing through the entire streaming architecture.

3.3 Security

Up until now we have only been concerned with making sure we don't lose data or overwhelm the broker or consumers and that we understand how we can recover from a failure. This is all good, because now we know how to provide a robust message queuing tier. We have one more hurdle to cover: security. In this day and age, not only is it important that we secure the data in-flight and at rest, but that we also can ensure that a producer is allowed to produce messages and a consumer is allowed to consume them. Figure 3.14 shows all the places we need to think about when securing the message-queuing tier.

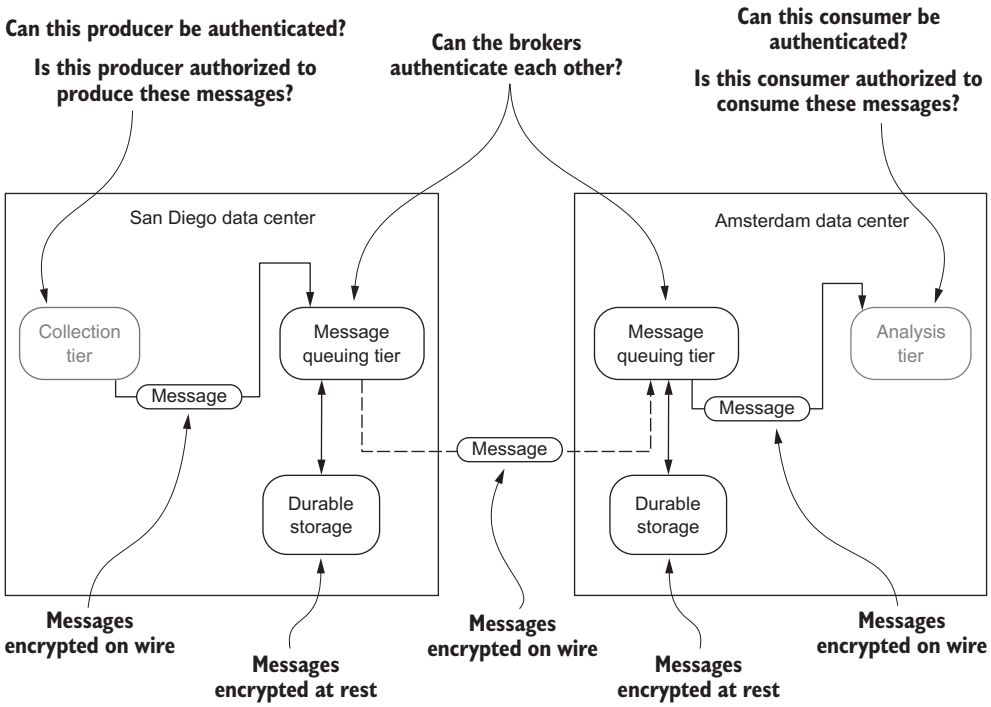

Figure 3.14 Security points that need to be considered for the message queuing tier

It may seem as if we have a daunting task ahead of us to secure the message queuing tier. At a minimum, you need to think through these aspects, and if you work with a security group, it would be good to engage them as you undertake securing this and the other tiers. At a high level I'm sure they all make sense, but as I've mentioned, the

devil is in the details, including figuring out how to work around some of the limitations when the message-queuing product you chose doesn't support everything you may need. There are many great resources to consult regarding security as you proceed further. A good place to start for in-depth coverage of security in distributed systems is Ross Anderson's *Security Engineering: A Guide to Building Dependable Distributed Systems* (Wiley, 2008).

3.4 Fault tolerance

Now that our collection and analysis tiers are isolated and we're sending messages through the message queuing tier, you need to understand what happens to the data when things go wrong, and it's not a matter of *if* things will go wrong, but *when*. For the multiple data center architecture we've been talking about, how many places can you identify where we may lose data? Look at figure 3.15 and compare notes.

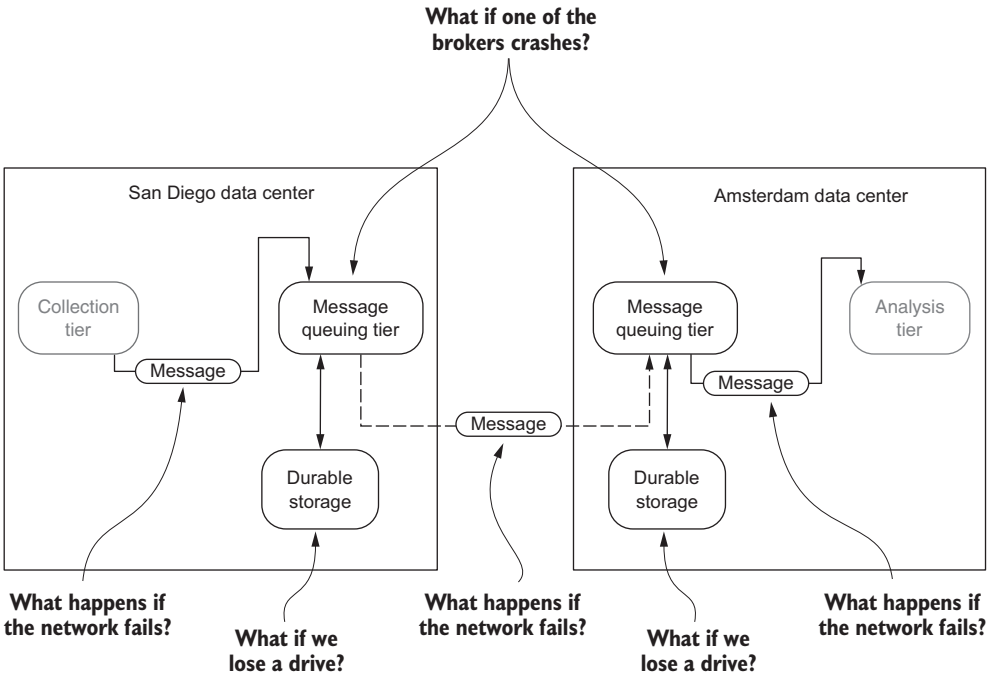

Figure 3.15 A high-level view of where things can go wrong

In figure 3.15 several callouts should be familiar to you from chapter 2, notably the producer having to handle network failures. The network failing or being unavailable between data centers is a reality that can often be mitigated with a broker that uses durable storage. If the message queuing product that meets your business needs doesn't use a durable store, you will either need to live with data loss or find another way to mitigate the risk of losing network connectivity. The callout in the figure on the

consumer side regarding the network failing while the analysis tier is consuming the message is covered in chapter 4.

Let's dig a little deeper into what can go wrong in this tier. Figure 3.16 shows a zoomed-in view of the message queuing tier.

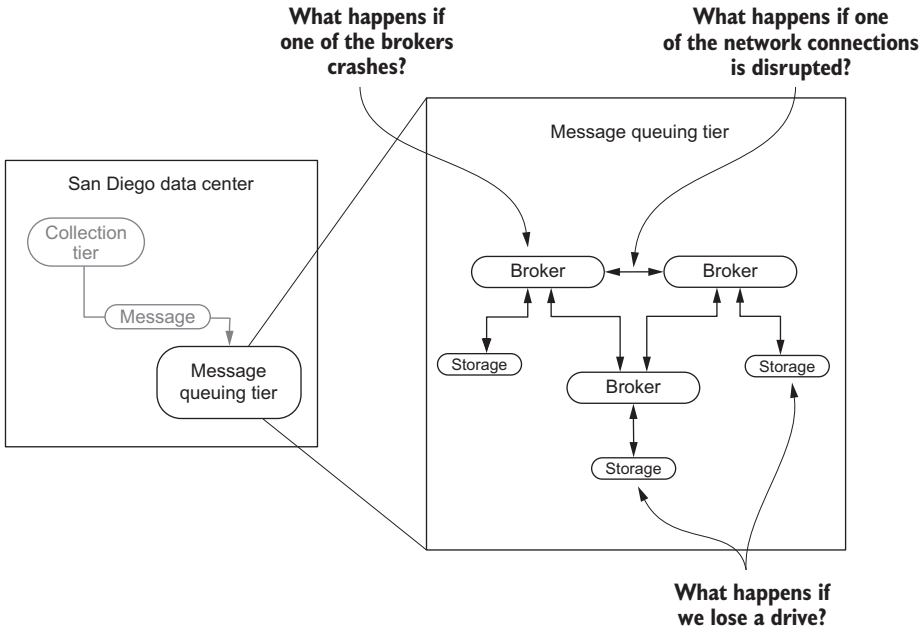

Figure 3.16 Zoomed in view of brokers and what can go wrong

In this zoomed-in view you can see we have three brokers—the number of brokers isn't as important as what's going on between them and where things can fail. When it comes to particular message queuing products, the number of brokers you deploy will matter, and you should have a thorough understanding of how best to utilize the product you chose. For now the goal is to make sure you know where things can break and what questions you may need to answer when evaluating various products.

Let's now tackle the questions identified in figure 3.16. These questions and the following discussion are not exhaustive but should serve as a good starting point when thinking through this problem in greater detail.

- *What happens if a broker crashes?*—This is an interesting problem. If the broker uses durable storage, we would hope that the only data at risk are those messages that are in memory when the broker crashes. There are ways we can mitigate this risk:
  - You can apply the lessons of chapter 2: to guarantee the message was delivered, our producer should wait for an acknowledgment that the data was indeed written to disk.

  - You can have the message queuing product replicate the message to more than one broker. In this case you still have risk, but it's been reduced significantly because now two or more brokers must crash at the same point in time before the messages are written to disk.
  - You can configure the broker to hold as little data in memory as possible. This method may have performance implications, so you are trading potential performance for durability.
- *What happens if the network between two brokers is interrupted?*—For many of the message queuing systems that provide replication, your data is safe because it is on more than one broker. Also, once the network connection is restored, the broker will rejoin the cluster and synchronize the messages it missed from other brokers. Think through the following related questions with the message-queuing product you chose:
  - Is a different broker chosen as the new replica?
  - What happens if the network connection is restored?
  - Is there an ability to configure the delay before the network is deemed to be "down"?
  - What happens to the data if the network connection isn't restored?
  - What happens to the data if a producer was in the middle of sending a message to the broker that got severed from the cluster?
- *What happens if we lose a drive?*—Although this has the potential to be a drastic loss, many operations people are used to dealing with it. In the case of a message-queuing product, if multiple brokers are involved, then these questions follow:
  - Are there replicas of the data that was lost?
  - What if there is data that was being replicated and it wasn't written to disk when the drive was lost—is that data lost?
  - How can you recover a broker?

Armed with these questions, I hope you will be able to apply them when assessing the fit of a particular product for your business problem.

3.5 Applying the core concepts to business problems

Now that we have covered the core concepts to keep in mind regarding the message queuing tier, let's see if we can apply them to different business scenarios.

FINANCE: FRAUD DETECTION

Paul's company provides real-time fraud-detection services that help detect fraud as it is happening. To do this he needs to collect credit card transactions from all over the web as they are occurring, apply some pretty cool algorithms against the data, and then send back approved or declined messages to customers while a purchase is happening. Bearing in mind the core concepts we've discussed, table 3.2 lists things that may come up when designing the architecture for Paul's business.

Table 3.2 Fraud detection scenario questions

Question	Discussion
What would be the impact to Paul's business if the communication between the collection tier and analysis tier were interrupted for an extended period of time?	The impact would be catastrophic. Paul's business would not be able to offer its service and may cause a detrimental impact to his customers' businesses.
How many days' worth of data can Paul's business tolerate losing?	Zero. In fact, I would argue that given the nature of Paul's business and the type of data he's dealing with, losing data is not an option.
Would you anticipate that Paul would need to store historical data?	I would expect that at least one customer or an executive in Paul's business has asked to see a report detailing how their service has performed over time.
What type of message delivery semantics does Paul's streaming system need?	I would expect that his business needs exactly-once semantics. Without that, he may miss a message, and therefore miss a fraudulent transaction. Could he get by with at least once? Perhaps. It may make the consumer more complex, but it is possible.

That was fun. Now let's take two more totally different businesses and see if you can answer the questions for them.

INTERNET OF THINGS: A TWEETING COKE MACHINE

Frank's business owns thousands of Coke vending machines and would like to make them social. He wants his machines to tweet and send push notifications with special offers to consumers who are geographically close. As if that weren't enough, there's a twist: If the closest vending machine doesn't have stock to offer, it should recommend the next closest vending machine that can offer the customer a deal. How would you answer the questions posed in table 3.3? I've added some things to think about in the discussion column. Considering the totally different business requirements, don't be surprised if your answers are quite different from the previous example. The idea is to spend time thinking through the problem.

Table 3.3 Internet of Things scenario questions

Question	Discussion
What would be the impact to Frank's business if the communication between the collection tier and analysis tier were interrupted for an extended period of time?	Obviously, Frank would lose money or could lose money if a social vending machine has a significant impact on his revenue. What if he also used this for inventory management? Would the impact be larger?
How many days' worth of data can Frank's business tolerate losing?	That may depend on how he handles inventory and the popularity of the machines. Perhaps this is locale-specific.
Would you anticipate that Frank would need to store historical data?	Perhaps—if he wanted to do reporting on how well his social vending machines are performing.

Table 3.3 Internet of Things scenario questions *(continued)*

Question	Discussion
What type of message delivery semantics does Frank's streaming system need?	In this case, at least once should suffice, although I think you could make an argument for at most once. What do you think would be the impact if a message were processed more than one time? What if a message was missed entirely?

E-COMMERCE: PRODUCT RECOMMENDATIONS

Rex runs a high-end fashion e-commerce business and is trying to increase the conversion rate on his site. He feels that perhaps social influence is one way to do this. To support this effort, Rex has asked you to architect a system that will let him show visitors to his site what other people have recently added to their cart or purchased. If I'm looking at a pair of jeans, and other customers on the site added shoes along with the same jeans to their cart or purchased a shirt with the same jeans, I should see the shoes and shirt recommended for me to buy along with the jeans. Imagine the messaging being something like this: "Ten other people just purchased these [product name] with the jeans you're looking at." Or "Five people also added this shirt to their cart along with these jeans." To summarize, the requirements we are trying to fulfill are:

- Keep track of all purchases in real time
- Keep track of all shoppers carts in real time
- Show on every product page the products recently purchased together
- Show on every product page the related products current shoppers have in their carts right now

With our mission in mind, answer the questions in table 3.4 so we can design the correct system for Rex. I have added input to the discussion column that may aid in your analysis.

Table 3.4 E-commerce requirements questions

Question	Discussion
What would the impact be to Rex's business if the communication between the collection tier and analysis tier were interrupted for an extended period of time?	Not being able to have the marketing opportunities to up-sell and cross-sell would cause this feature to not function. Imagine if you went to Amazon and were not shown the product recommendations. It would be hard for us to quantify the monetary value for Frank, but for sure it would be lost opportunity and lost sales.
How many days' worth of data can Rex's business tolerate losing?	Perhaps we could provide an old-fashioned batch-based recommendation system as a backup, but the freshness of the data may be an issue.
Would you anticipate that Rex would need to store historical data?	Perhaps for reporting purposes and for refinement of any algorithms that we develop.

Table 3.4 E-commerce requirements questions *(continued)*

Question	Discussion
What type of message delivery semantics does Rex's streaming system need?	Having at least once should suffice for this use case. Exactly-once is overkill. Do you think we would need at most once?

3.6 Summary

See section 9.2.1

In this chapter we've explored how to decouple the data being collected from the data being analyzed by using a message queuing tier in the middle. During this exploration we did the following:

- Learned why we need the message queuing tier
- Developed an understanding of message durability
- Learned how to accommodate offline consumers
- Learned about the different message delivery semantics
- Learned how you can choose the right technology for your business problem

At this point we've developed a good understanding of how to decouple the data being produced by our collection tier from the analysis tier. As you move through the other chapters, you'll see some of the terms and concepts popping up again, so don't worry if this is a bit overwhelming right now—it won't be by the last time we talk about message delivery semantics. If you're interested in learning more about the numerous aspects of messaging systems and the variety of integration patterns, see Gregor Hohpe and Bobby Woolf's *Enterprise Integration Patterns* (Addison-Wesley, 2003).

Now let's get ready to have fun with the data we've collected. The next chapter takes us through the analysis tier—our message consumer. Ready? Let's go.

4 Analyzing streaming data

This chapter covers

- In-flight data analysis
- The common stream-processing architecture
- Key features common to stream-processing frameworks

In chapter 3 we spent time understanding and thinking through the importance of the message queuing tier. That tier is designed to gather data from the collection tier and make it available to be moved through the rest of the streaming architecture. At this point the data is ready and waiting for us to consume and do magic with. In this chapter you're going to learn about the analysis tier. Our goal is to get to know the underlying principles of this tier, and in chapter 5 we'll dive into all the ways to use this tier to perform magic on the data. With that frame of reference in mind, consult our navigational aid in figure 4.1 to make sure you're oriented with respect to the flow of data.

Notice in figure 4.1, unlike in chapter 3 which discussed the input and output of the data, here we're only going to concern ourselves with the input. We'll hold off on talking about where the data goes from this tier until the next chapter. After finishing this chapter you'll know the core concepts found in all the modern tools

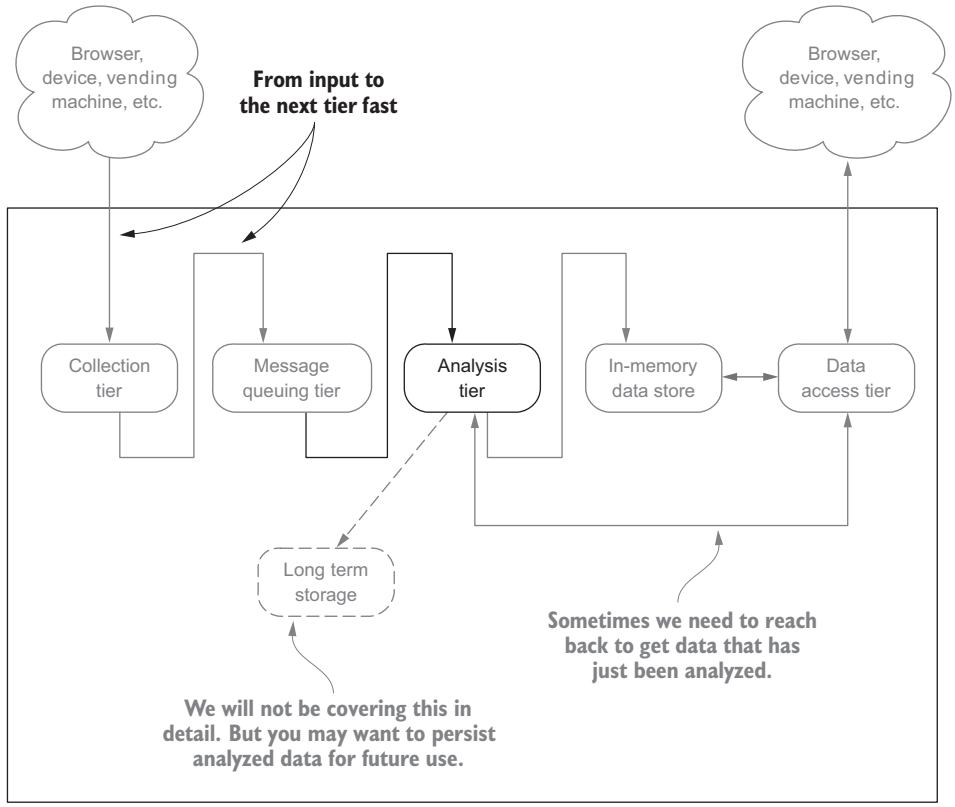

Figure 4.1 The streaming data architecture with the analysis tier in focus

used for this tier and you'll be ready to learn how to perform various operations on the data.

Grab a quick coffee refill, and let's get going.

4.1 Understanding in-flight data analysis

Key to understanding the features we'll discuss in this chapter is coming to grips with what *in-flight data* means and grasping the concept of *continuous queries*. If *in-flight* makes you think of something moving in the air, that's the right idea. When it comes to data, *in-flight* refers to all the tuples in the system, from the input source (the message queuing tier), to the output, to a client (the next tier). Data is always in motion and never at rest, meaning it's never persisted to durable storage. (Data *at rest* means the data is stored on disk or another storage medium.) Figure 4.2 shows how this plays out in our streaming architecture.

Figure 4.2 should make it clear that our goal in this tier is to *pull* the data from the message queuing tier as fast as possible; ideally the analysis tier should be able to keep up with the rate at which the collection tier is pushing data into the message queuing

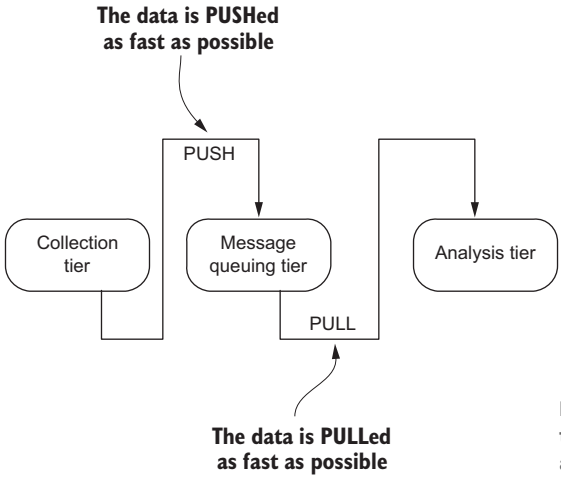

Figure 4.2 Data being pushed from the collection tier and pulled from the analysis tier

tier. How is this different from a non-streaming system? Say, one built with a traditional DBMS (RDBMS, Hadoop, HBase, Cassandra, and so on)? In those non-streaming systems the data is at rest, and we query it for answers. In a streaming system we turn that on its head—the data is moved through the query. This model is called the *continuous query model*, meaning the query is constantly being evaluated as new data arrives.

Let's imagine that you run a large news agency and you want to know if an article is trending or if a link to it is broken so that you can adjust your marketing campaign or fix your site. If you were using traditional DBMS technologies, you would have to do the following:

1. Gather the data from your site
2. Load the data into the DBMS
3. Execute a query to determine if the link is broken or the article is trending
4. Take action
5. Rinse and repeat every *x* minutes or, more likely, hours

Compare that to the steps you might take if you were using a streaming system:

1. Collect the stream of data
2. Start a query that determines if the link is broken or the article is trending
3. Take action

I think you'll agree that it would be hard for your business to react to changes happening now using a traditional system, whereas with the streaming system the query is always executing against the data, and you can react in real time to trends or problems. In a streaming system a user (or application) registers a query that's executed every time data arrives or at a predetermined time interval. The result of the query is then pushed to the client.

Here are the key differences to remember:

- In the architecture of traditional database management systems, when a user (the active party) wants an answer to a question, they submit a query to the system (the passive party), and an answer is returned. This is always based on data that has been loaded into the system before it's queried; in essence, the data set is static.
- In a streaming system a query is started and is continuously (this could be triggered on an interval or another event) executed over the data as it is flowing. The answer to the query is then pushed to the next tier, which may be a user or application.

This inverts the traditional data management model by assuming users to be passive and the data management system to be active. Figure 4.3 shows this inversion graphically.

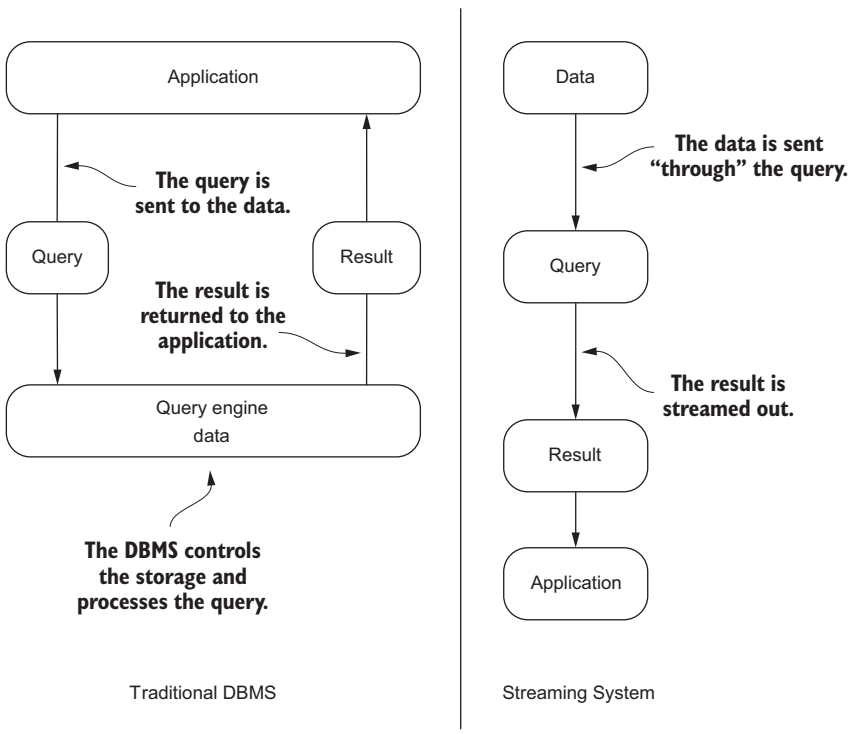

Figure 4.3 Turning things on their head: non-streaming versus streaming

As shown on the left side of figure 4.3, in the traditional DBMS the query is sent to the data and executed, and the result is returned to the application. In a streaming system, as illustrated on the right, this model is completely changed—the data is sent "through" the query, and the result is sent to an application. In the case of the streaming

system the data is being pulled or pushed through our system in a never-ending stream; this undoubtedly has implications on both the design and the way we query these systems. To give you a better feel for these differences, table 4.1 highlights some of the main differences between a traditional DBMS and streaming system.

Table 4.1 Comparison of traditional DBMS to streaming system

	DBMS	Streaming system
Query model	Queries are based on a one-time model and a consistent state of the data. In a one-time model, the user executes a query and gets an answer, and the query is forgotten. This is a pull model.	The query is continuously executed based on the data that is flowing into the system. A user registers a query once, and the results are regularly pushed to the client.
Changing data	During down time, the data can't change.	Many stream applications continue to generate data while the streaming analysis tier is down, possibly requiring a catch-up following a crash.
Query state	If the system crashes while a query is being executed, it's forgotten. It's the responsibility of the application (or user) to re-issue the query when the system comes back up.	Registered continuous queries may or may not need to continue where they left off. Many times it's as if they never stopped in the first place.

When you think of all the data zipping around you all day long, from myriad connected devices and appliances to online activity, the questions you could ask and problems you could solve if it all passed through a streaming analysis tier are amazing. Here are some categories and examples to get you going:

- *Tracking behavior*—Imagine being able to provide personalized advertising based on a customer's location, the weather, and their previous buying habits and preferences. McDonald's did this using the VMob platform. In one case study, McDonald's in the Netherlands realized a 700% increase in offer redemptions, and customers using the app returned twice as often and spent on average 47% more (http://mng.bz/1p0t).
- *Improving traffic safety and efficiency*—According to the European Commission (http://ec.europa.eu/transport/themes/urban/urban_mobility/index_en.htm), congestion in the European Union (EU) in and around urban areas costs nearly €100 billion, or 1% of EU GDP annually. According to the Federal Highway Administration (http://ops.fhwa.dot.gov/program_areas/reduce-non-cong.htm), 25% of traffic congestion is caused by traffic incidents. Imagine you were able to employ roadway vehicle sensors (see www.fhwa.dot.gov/policyinformation/pubs/vdstits2007/03.cfm for an introduction); based on our analysis of the traffic data, we can provide drivers with updated traffic conditions and reroute traffic accordingly to maximize driving efficiency. For real-world examples, see Blip Systems

(www.blipsystems.com/traffic/), which has examples of how some cities have solved lots of traffic problems.

- *Real-time fraud analytics*—Every time a credit card is swiped, a complex series of algorithms must be executed to determine whether the attempted transaction is valid or fraudulent. According to FICO (www.fico.com/en/node/8140?file=5582), U.S. fraud losses on credit cards have declined by 70% as a percentage of credit card sales since real-time fraud analytics have been deployed.

These examples are the tip of the iceberg and hopefully have whetted your appetite for what's possible. They also may help you realize that understanding how to build these systems to harness the numerous data streams available in the world today is becoming an essential skill.

But let's not get ahead of ourselves; we have our work cut out for us learning about the core features of an analysis tier. Let's begin our journey by discussing the general architecture of a stream-processing system and then move onto the key features and see how each of the features plays a role in the decision to use a particular framework.

4.2 Distributed stream-processing architecture

It may be possible to run an analysis tier on a single computer, but the velocity and volume of the data at some point make this a nonviable option. Instead of tracking trending or broken links to articles, for example, imagine we were interested in analyzing the performance of a gas turbine in real time to determine if it was functioning correctly. According to General Electric, a single turbine engine can produce approximately 1 TB of data per hour. Clearly, using a single computer will quickly become a nonviable option for us. Therefore, we're going to concentrate on the tools and technologies involved in building a distributed analysis tier.

As you survey the technology landscape, you'll find numerous technologies designed for stream processing. At the time of this writing the most popular open source products are Spark Streaming, Storm, Flink, and Samza, all from Apache. I'm not going to go into detail on each of them but will discuss each briefly after going over the generalized streaming architecture so you can see how each fits into it.

A GENERALIZED ARCHITECTURE

Spark, Storm, Flink, and Samza all have the following three common parts:

- A component that your streaming application is submitted to; this is similar to how Hadoop Map Reduce works. Your application is sent to a node in the cluster that executes your application.
- Separate nodes in the cluster execute your streaming algorithms.
- Data sources are the input to the streaming algorithms.

Taking these central ideas and their respective architectures into consideration, we can generalize this into a single common architecture, as shown in figure 4.4.

There are other streaming systems on the market today that have not achieved the same level of popularity as those discussed in the previous section, and undoubtedly

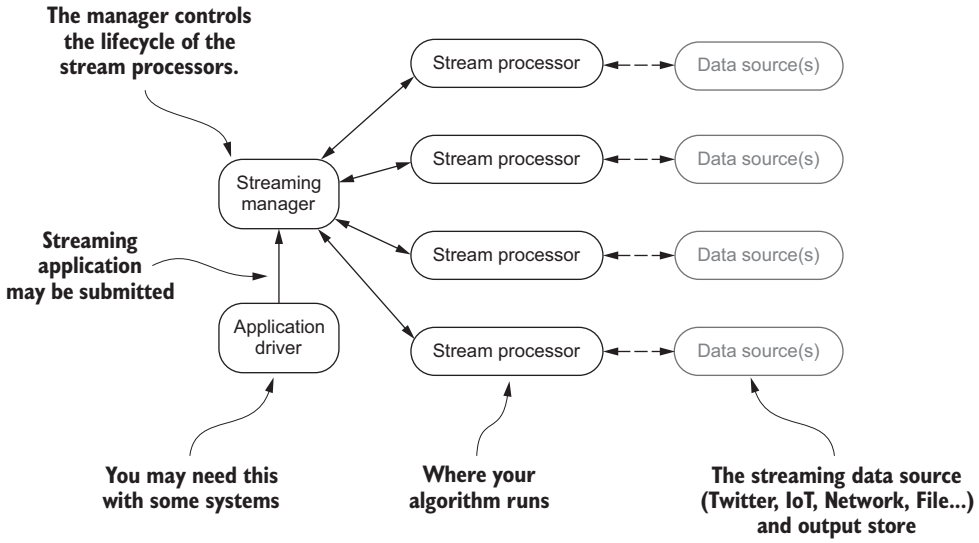

Figure 4.4 Generic streaming analysis architecture you will find with many products on the market

there will be more in the future. In many cases other products will map onto this common architecture and help in your understanding of how they work. To make sure we're on the same page, consider the common architectural pieces shown in figure 4.4:

- *Application driver*—With some streaming systems, this will be the client code that defines your streaming programming and communicates with the streaming manager. For example, with Spark Streaming your client code is broken into two logical pieces: the driver and the streaming algorithm(s) or job. The driver submits the job to the streaming manager, may collect results at the end, and controls the lifetime of your job.
- *Streaming manager*—The streaming manager has the general responsibility of getting your streaming job to the stream processor(s); in some cases it will control or request the resources required by the stream processors.
- *Stream processor*—This is where the rubber meets the road, the place where your job runs. Although this may take many shapes based on the streaming platform in use, the job remains the same: to execute the job that was submitted.
- *Data source(s)*—This represents the input and potentially the output data from your streaming job. With some platforms your job may be able to ingest data from multiple sources in a single job, whereas others may only allow ingestion from a single source. One thing that may not be obvious from the architectures is where the output of the jobs goes. In some cases you may want to collect the data in your driver; in others you may want to write it out to a different data source to be used by another system or as input for another job.

Now that you know about the various architectures and have boiled them down to our common architecture, let's go back to our example of monitoring the performance of gas turbines to determine if they're functioning correctly and map that to our common architecture. In figure 4.5 you can see our common architecture (simplified to be less busy) with our business problem mapped to it.

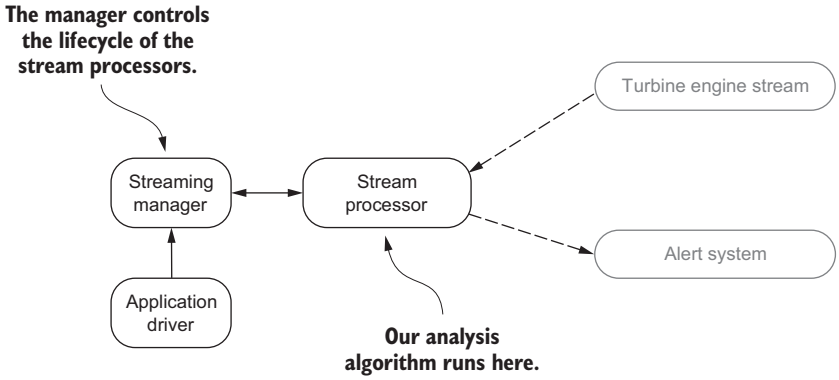

Figure 4.5 Turbine engine monitoring on our common architecture

I realize that this may have been a lot to digest, so take a moment to see if you can map your business problem to the common architecture we've derived. When you're ready, move on to the next discussion of the architecture of the three major streaming systems. Then we'll go a little deeper and talk about some of the key features you'll want to consider when choosing a stream-processing framework.

Apache Spark Streaming

Apache Spark Streaming, often called Spark Streaming, is built on Apache Spark, as depicted in figure 4.6.

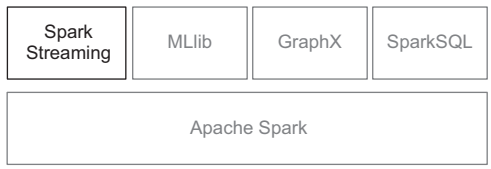

Figure 4.6 Apache Spark Streaming with the basic Spark stack

As you'll notice, there are other features built on top of Apache Spark, which is becoming the de facto platform for general-purpose distributed computation. It provides support for multiple languages (Java, Scala, Python, and R) and at the time of this writing has the following high-level tools built on top of it: Spark Streaming, MLlib (machine learning), SparkR (integration with R), and GraphX (for graph processing). Outside the normal project documentation, a great resource to start

learning more about Spark is Marko Bonaći and Petar Zečević's book *Spark in Action* (www.manning.com/books/spark-in-action). Figure 4.7 illustrates Spark Streaming's overall architecture.

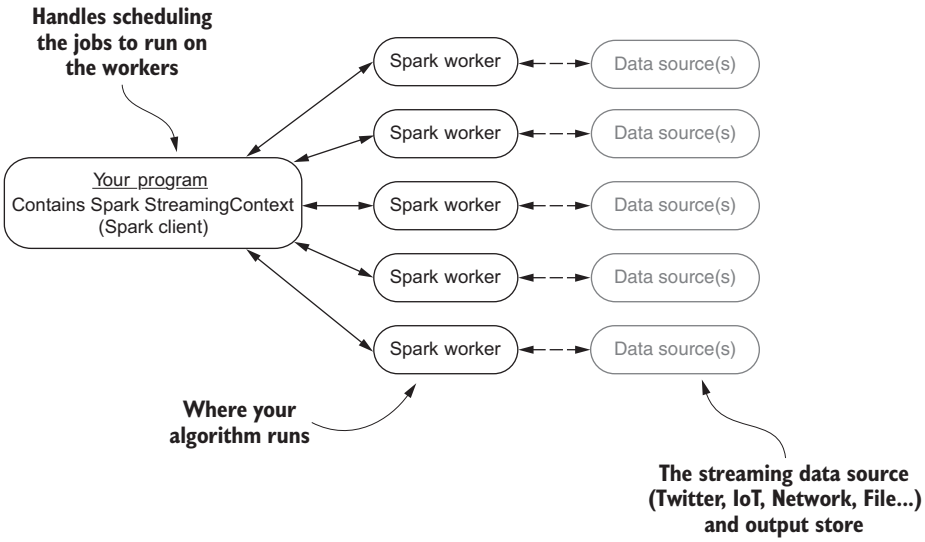

Figure 4.7 Spark Streaming's high-level architecture

Starting from the left in the figure is our program, which contains what is called a Spark StreamingContext; collectively, this is known as *the driver*. Without diving into the details, the Spark StreamingContext contains all the logic to be able to keep track of incoming data, set up the streaming jobs, schedule them on the Spark workers, and execute the jobs. You may notice here that we're talking about jobs and not a stream. Spark and subsequently Spark Streaming operate on batches of work. In the case of Spark Streaming, these batches represent data over a period of time and can be scheduled to run at intervals of less than half a second. A *job* in Spark Streaming is the logic of your program that's bundled and passed to the Spark workers. If you've read about or worked with Hadoop MapReduce, this is the same concept. In the middle of figure 4.7 are the Spark workers, which run on any number of computers (from one to thousands) and are where your job (your streaming algorithm) is executed. As you'll notice, they receive data from an external data source and communicate with the Spark StreamingContext that's running as part of the driver.

APACHE STORM

Apache Storm is tuple-at-a-time stream-processing framework designed for real-time processing of data streams. Storm has so many features I can't cover them in detail. To learn more about Storm, see Sean Allen, Peter Pathirana, and Matthew Jankowski's *Storm Applied* (Manning, 2013). The overall architecture for Storm is shown in figure 4.8.

See chapter 9 section 9.3.1

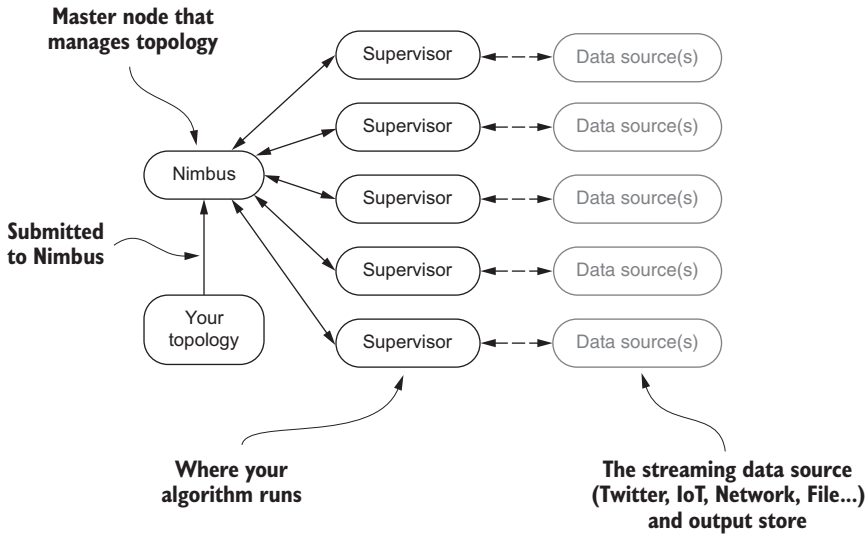

Figure 4.8 High-level overview of Storm architecture showing Nimbus, supervisors, and a data source

You can see that from a high-level Storm is similar to Spark Streaming or perhaps a Hadoop cluster if you change Nimbus to a job tracker and the supervisors to data nodes. Hadoop and Spark use the term *job* to describe the unit of work; with Storm the term is topology. The reasoning behind this is that a *job* will eventually finish, whereas a *topology* will run forever. Let's not get bogged down with semantics; at the end of the day they both represent a way to deploy your program to the worker nodes. With this definition in mind, let's walk through figure 4.8 and discuss the different pieces.

Starting on the bottom left in figure 4.8 you can see the topology is submitted to a component called Nimbus. Nimbus decides how the topology is deployed across the supervisors, assigns different tasks to the supervisors, and monitors the entire system for failures. Moving to the middle of the figure, you see the *supervisor* nodes, where your topology runs. On the right, the *data source* represents the data that will be ingested by the running topology.

Apache Flink

Apache Flink is a *stream first* framework, where everything is viewed as a stream. A program written with Flink is composed of two fundamental building blocks: streams and transformations. A *stream* can be considered an intermediate result as the data flows from a source to a sink. A *transformation* is any operation that takes one or more streams as input and produces one or more streams as output. When an application is composed using these concepts and executed with Flink, it's considered a streaming dataflow. The dataflow begins with the ingestion of data from one or more sources,

usually contains one or more transformations, and ends with the data being written out to one or more sinks. Flink applications run in a distributed environment composed of a single master and workers. The high-level architecture for this is shown in figure 4.9. We are merely touching the surface here; to learn more about Apache Flink, I strongly recommend you read Sameer Wadkar and Hari Rajaram's *Flink In Action* (Manning, 2016).

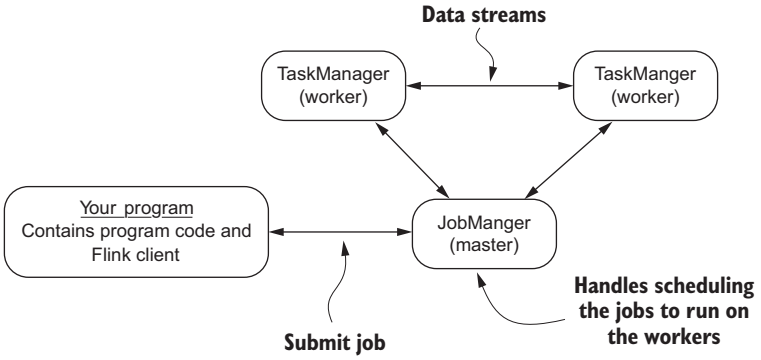

Figure 4.9 High-level Apache Flink architecture

APACHE SAMZA

The streaming model with Apache Samza is slightly different in that it provides a stage-wise stream-processing framework. To do so it uses two prominent technologies found in the Big Data space: Apache Yarn and Apache Kafka. We won't spend much time on those technologies, but I will discuss them briefly as they relate to the high-level Samza architecture shown in figure 4.10.

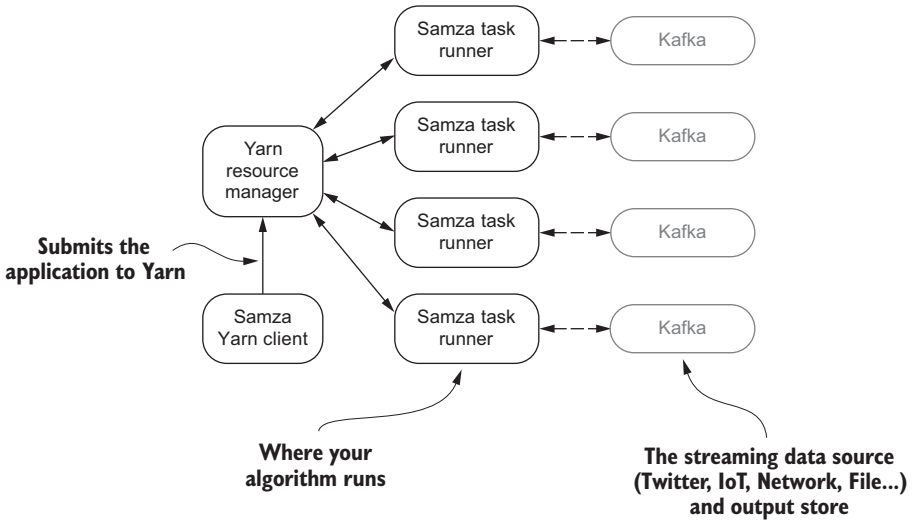

Figure 4.10 High-level Apache Samza architecture

Yarn is a cluster manager designed to handle resource management and job scheduling/monitoring. I know that's a mouthful, but think of it this way: the resource management part is responsible for allocating resources (CPUs, memory, disk, network, and so on) for all applications running on a cluster of computers. The job scheduling/monitoring aspect is responsible for running the job on the cluster. In figure 4.10 you can see that the Samza Yarn client makes a request to the Yarn resource manager asking that the requested resources be allocated for the Samza application to run. Subsequently, after some resource negotiation, the Samza task runners are executed in various nodes in the cluster. This is intentionally simplified because focusing on the Yarn specifics here doesn't add value to our discussion and is subject to change as the project matures. In the center of the figure the Samza tasks are running. In this case all input and output to our Samza tasks will be done using Apache Kafka.

Apache Kafka is a technology that squarely fits into chapter 3's discussion on the message queuing tier and is one we'll revisit in future chapters. For now think of it as a high-speed data store that our streaming tasks will read from and write to. Great resources to learn more about Yarn are Alex Holmes's *Hadoop in Practice*, Second Edition (Manning, 2014) and Chuck Lam, Mark Davis, and Ajit Gaddam's book *Hadoop in Action*, Second Edition (Manning, 2014). For the latest information on Apache Samza, visit http://samza.apache.org.

4.3 Key features of stream-processing frameworks

You can use many different stream-processing frameworks in the analysis tier of our streaming data architecture. We want to pay special attention to a handful of key features when comparing them and deciding whether they're suitable for solving our business problem. The goal is to be able to apply this knowledge when selecting the stream-processing framework you'll use in your streaming data architecture.

4.3.1 Message delivery semantics

In chapter 3 you learned about message delivery semantics with respect to the message queuing tier and the producers, brokers, and consumers. This time the discussion focuses on message delivery semantics on the analysis tier. The definitions don't change, but the implications are a little different.

First, let's refresh your memory on the definitions of the different guarantees:

- *At-most-once*—A message may get lost, but it will never be processed a second time.
- *At-least-once*—A message will never be lost, but it may be processed more than once.
- *Exactly-once*—A message is never lost and will be processed only once.

Those definitions are a slightly more generic version of what you saw before with the message queuing tier. In chapter 3, our focus was on understanding what each of these meant with respect to producing and consuming messages. In this chapter, with the stream-processing tools, we're talking about the continuation of the consumer side of the message queuing tier. How do these manifest themselves in the stream-processing

tools you may use in this tier? Let's overlay them on a data flow diagram and walk through them to see what they mean.

Figure 4.11 shows at-most-once semantics with the two failure scenarios: a message dropping and a streaming task processor failing, the latter of which will also result in message loss until a replacement processor comes online.

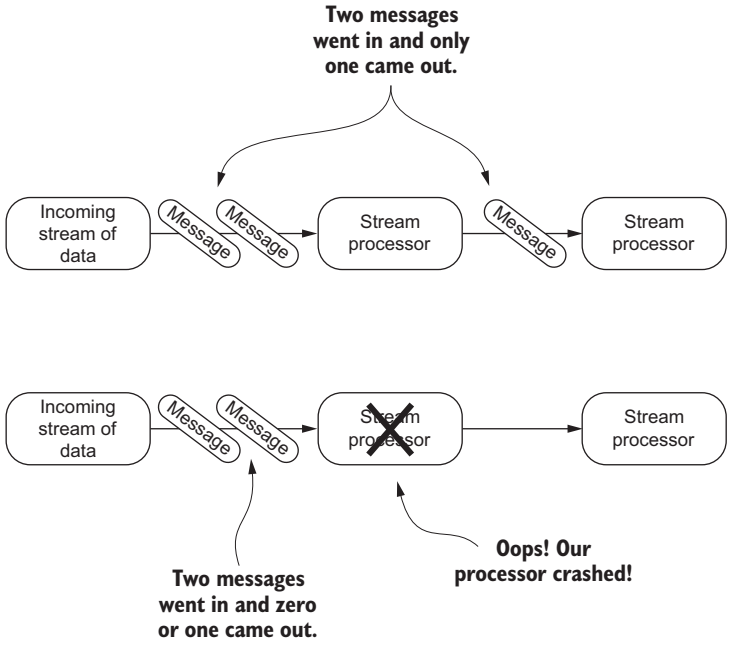

Figure 4.11 At-least-once message delivery shown with the streaming data flow

At-most-once is the simplest delivery guarantee a system can offer; no special logic is required anywhere. In essence, if a message gets dropped, a stream processor crashes, or the machine that a stream processor is running on fails, the message is lost.

At-least-once increases the complexity because the streaming system must keep track of every message that was sent to the stream processor and an acknowledgment that it was received. If the streaming manager determines that the message wasn't processed (perhaps it was lost or the stream processor didn't respond within a given time boundary), then it will be re-sent. Keep in mind that at this level of messaging guarantee, your streaming job may be sent the same message multiple times. Therefore your streaming job must be *idempotent*, meaning that every time your streaming job receives the same message, it produces the same result. If you remember this when designing your streaming jobs, you'll be able to handle the duplicate-messages situation.

Exactly-once semantics ratchets up the complexity a little more for the stream-processing framework. Besides the bookkeeping that it must keep for all messages

that have been sent, now it must also detect and ignore duplicates. With this level of guarantee your streaming job no longer has to worry about dealing with duplicate messages—it only has to make sure it responds with a success or failure after a message is processed. Even though being idempotent with this level of messaging guarantee isn't required of your streaming job, I highly recommend that you approach all your streaming jobs with the expectation that they should be idempotent. It will make troubleshooting and reasoning about them much easier.

You may be wondering which of these guarantees you need; it depends on the business problem you're trying to solve. Take our example from earlier—the turbine engine monitoring system. Remember, we want to constantly analyze how our turbine engine is performing so we can predict when a failure may occur and preemptively perform maintenance. Earlier I said our turbines produce approximately 1 TB of data every hour—keep in mind that's one turbine, and we're monitoring thousands to be able to predict when a failure may occur. Do we need to ensure we don't lose a single message? We may, but it would be worth investigating whether our prediction algorithm needs all the data. If it can perform adequately with data missing, then I'd choose the least complex guarantee first and work from there.

What if instead your business problem involved making a financial transaction based on a streaming query? Perhaps you operate an ad network and you provide real-time billing to your clients. In this case you'd want to ensure that the streaming system you choose provides exactly-once semantics.

I hope you're getting the hang of it and can apply this to your business problem. Now let's move on to talk about state management.

State management

Once your streaming analysis algorithm becomes more complicated than using the current message without dependencies on any previous messages and/or external data, you'll need to maintain state and will likely need the state management services provided by your framework of choice. We'll take a simple example to help you understand where and perhaps how state needs to be managed.

Pretend you're the marketing manager for a large e-commerce site and you want to know the number of page views per hour for each visitor. I know you're thinking, "An hour—that can be done in a batch process." Instead of worrying about that right now let's focus on the implied state you must keep to satisfy this business question. Figure 4.12 shows how your streaming task processors would be organized to answer this question.

In figure 4.12 it should be obvious where you need to keep state—right there in the stream processor where your job performs the counting by user ID. If your streaming analysis tool of choice doesn't provide state management capabilities, one viable option for you is to keep the data in memory and flush it every hour. This would work as long as you're okay with the potential of losing all the data if the streaming processor or job fails at any time. As luck would have it, your job would be running smoothly and then, one day, start to fail at 59 minutes into the hour. Depending on your busi-

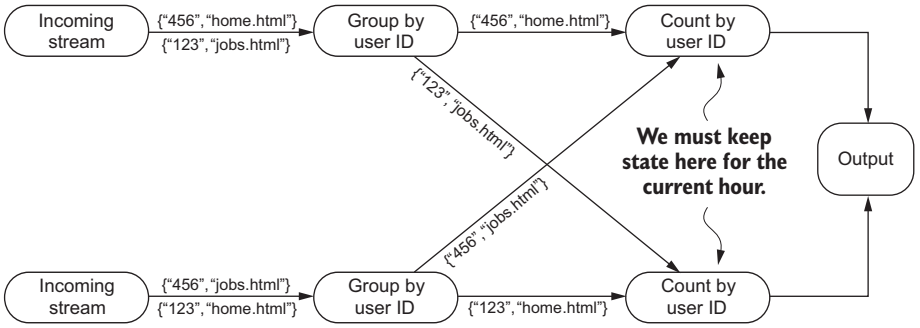

Figure 4.12 Simple example of counting page views per user over an hour

ness case, the risk and possible loss of data by keeping all state in memory may be acceptable. But in many business cases life is not so simple, and you do need to worry about managing state. To help in these scenarios, many stream-processing frameworks provide state management features you can use.

The state management facilities provided by various systems naturally fall along a complexity continuum, as shown in figure 4.13.

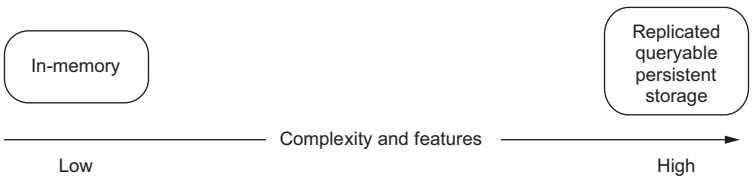

Figure 4.13 State management complexity continuum for stream-processing tools

The continuum starts on the left with a naïve in-memory-only choice, similar to what we used earlier, and progresses to the other end of the spectrum with systems that provide a queryable persistent state that's replicated. If you're saying, "These seem like two totally different slants on state management," you're not alone. The solutions on the low-complexity side only solve the problem of maintaining the state of a computation in the face of failures. For the simple operations of keeping a running count current and not losing track of the current value in the face of failure, these systems are a great fit.

On the other end of the spectrum, the frameworks that offer state management by way of a replicated queryable persistent store help you answer much different and more complicated questions. With these frameworks you can join together different streams of data. Imagine you were running an ad-serving business and you wanted to track two things: the ad impression and the ad click. It's reasonable that the collection

of this data would result in two streams of data, one for ad impressions and one for ad clicks. Figure 4.14 shows how these streams and your streaming job would be set up for handling this.

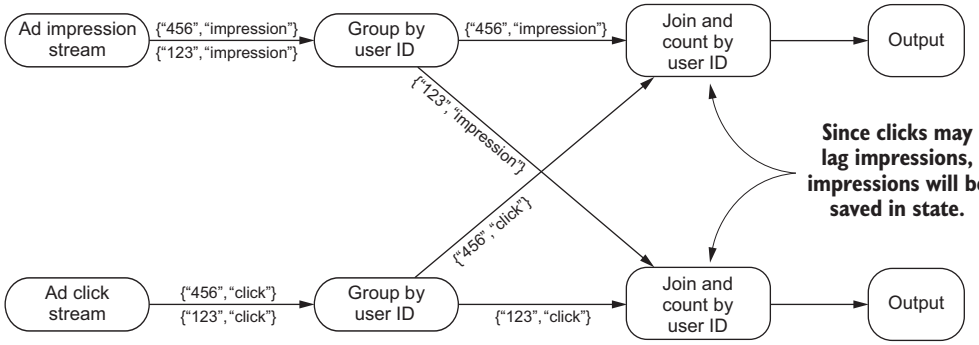

Figure 4.14 Handling ad impression and ad click streams that use stream state

In this example the ad impressions and ad clicks arrive in two separate streams; because ad clicks will lag ad impressions, we'll join the two streams and then count by user ID. Because of the lag in the ad click stream, using a stream-processing framework that persists the state in a replicated queryable data store enables us to join the two streams and produce a single result. I think you'll agree that being able to join streams by using the state management facilities of a stream-processing framework is quite a bit different than making sure the current value of an aggregation is persisted.

If you give additional thought to this example, I'm sure you can come up with other ideas of how to join more than one stream of data. It's a fascinating topic and something we'll look at in more depth in chapter 5. For now, let's continue to the next feature you need to understand when choosing a stream-processing framework.

FAULT TOLERANCE

It's nice to think of a world where things don't fail, but in reality it's not a matter of *if* things will fail, but *when*. A stream-processing framework's ability to keep going in the face of failures is a direct result of its fault-tolerance capabilities. When you consider all the pieces involved in stream processing, there are quite a few places where it can fail. Let's use figure 4.15 to identify all the failure points.

In figure 4.15 are seven points of failure in a simple stream-processing data flow. Go through them and make sure you understand what you'll need from a stream-processing framework with respect to fault tolerance:

1. *Incoming stream of data*—In all fairness, the message queuing tier won't be under the control of the stream-processing framework, but there is the potential for the message queuing system to fail, in which case the stream-processing framework must respond gracefully and not fail if data or the resource isn't available.

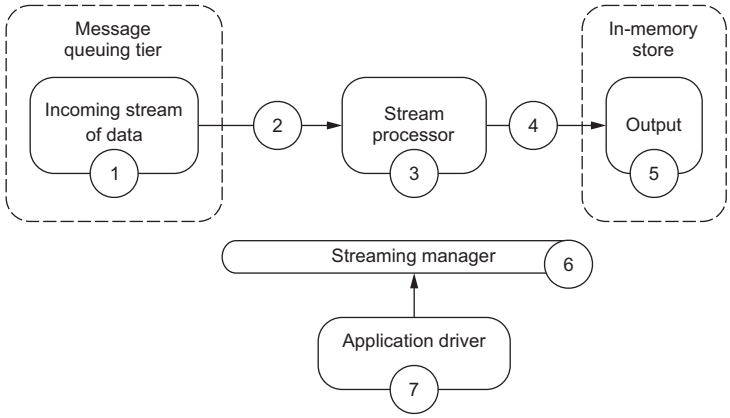

Figure 4.15 The points of failure with stream processing in the context of the streaming architecture

2. *Network carrying input stream*—This is something that the stream-processing framework can't control, but it needs to handle the disruption gracefully.
3. *Stream processor*—This is where your code is running; it should be under supervision of the stream-processing framework. If something goes wrong here—perhaps your software fails or the machine it's running on fails—then the streaming manager should take steps to restart the processor or move the processing to a different machine.
4. *Connection to output destination*—The stream task manager may not be able to control the network path to the output, but it should be able to control the flow of data from the last stream processor so it doesn't become overwhelmed by network backpressure or fail if the network or destination is unavailable.
5. *Output destination*—This wouldn't be under the direct supervision of the stream task manager, but its failing could impact the processing of the stream, so it needs to be considered.
6. *Streaming manager*—If this fails, you end up with a situation called *running headless*—the stream processors would continue to run without being supervised by the streaming manager. If this component fails, there's no supervisor for the data flow and the stream processors—no new ones can be started or failed ones recovered.
7. *Application driver*—This comes in two flavors. With the first, the application driver does nothing more than submit the streaming job to the streaming manager—we're not worried about this type. The second flavor is where the application driver logically contains the streaming manager and in turn is subject to the same risk as the streaming manager.

Let's go through how these problems are solved or could be solved. First, let's boil our problem down a bit. In the preceding list, we can eliminate the incoming stream and

output destination availability from the concerns of the streaming framework. It should go without saying that the streaming framework must not fail if there are failures with the input or output destinations. For this discussion we won't consider those aspects to be fault tolerance–related. If we take the list and consolidate it down to the common elements, we end up with the following:

- *Data loss*—This covers data lost on the network and also the stream processor or your job crashing and losing data that was in memory during the crash.
- *Loss of resource management*—This covers the streaming manager and your application driver, in the event you have one.

In the discussion of fault tolerance in chapter 3, I talk about ways to prevent data loss. When it comes to stream-processing frameworks, all the common techniques for dealing with failures involve some variant of replication and coordination. A common approach would be for the stream manager to replicate the state of a computation (the state of your streaming job) onto different stream processors. If there's a failure, then the streaming manager must coordinate the replicas in order to recover properly from failures. It's common for fault-tolerance techniques to be designed with a tolerance up to a predefined number of simultaneous failures, in which case you'll hear of a system being called *k-fault tolerant*, where *k* represents the number of simultaneous failures.

In general there are two common approaches—state-machine and rollback recovery—a streaming system may take toward replication and coordination. Both assume that we've designed and thought about our streaming algorithm in an idempotent way. If you recall from before, for our streaming job to be idempotent it means that two non-faulty stream processors that receive the same input in the same order will produce the same output in the same order. If we can ensure that, then we can refer to those two stream processors, and hence our streaming job, as idempotent and consistent if they generate the same output in the same order.

The first approach used by stream-processing systems is known as *state-machine*. With this approach the stream manager replicates the streaming job on independent nodes and coordinates the replicas by sending the same input in the same order to all. This approach requires $k + 1$ times the resources of a single replica, where k is the number of stream processors our streaming job is running on. But this allows for quick failover, resulting in little disruption. For some applications, such as an intrusion-detection system that has low-latency requirements at all times, the extra resource cost may be justifiable.

The second approach is known as *rollback recovery*. In this approach the stream processor periodically packages the state of our computation into what is called a *checkpoint*, which it copies to a different stream processor node or a nonvolatile location such as a disk. Between checkpoints, the stream processor has to keep track of the computation. Given the relative high latency of disks, once they're introduced the latency of our streaming computation will go up. It therefore may not be unreasonable for a

stream-processing framework to instead decide to take the approach of copying the checkpointed state to other stream processor nodes and also maintain logs in memory. In this case, if a stream processor fails, the stream manager would need to reconstruct the state from the most recent checkpoint and replay the log to recover the exact pre-failure state of the streaming job. Compared to the first approach, this approach has a lower overhead, but it's more expensive in terms of time to recover when a failure does happen. This approach is useful in situations where fault tolerance is important and rare moderate latencies are acceptable.

As you investigate which stream-processing framework to use to solve your business problem, you'll find that if they offer fault-tolerance, they'll all be some variant on these two common approaches. If you're interested in taking a deeper dive into either of these approaches, you may find the following articles of interest: Elnozahy, Alvisi, Wang, and Johnson's "A Survey of Rollback-Recovery Protocols in Message-Passing Systems"[1] and Schneider's "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial."[2]

4.4 Summary

This chapter dove into the common architecture of stream-processing frameworks that you'll find when surveying the landscape and covered the core features that you need to consider.

You learned about the following:

- The common architecture of stream-processing frameworks
- What message delivery semantics mean for this tier
- What state is and how it can be managed
- What fault tolerance is and why you need it

I understand that some of this may seem fuzzy or fairly abstract; don't worry about it at all. In chapter 5 we'll focus on how to perform analysis and/or query the data flowing through the stream-processing framework. Some may say that's where the fun begins, but to effectively be able to ask questions of the data, you need the understanding you developed in this chapter.

Are you ready to start asking questions of the data? Great! Turn the page and get started.

---

[1] *ACM Computing Surveys*, 34(3):375–408 (2002, www.cs.utexas.edu/~lorenzo/papers/SurveyFinal.pdf.

[2] *ACM Computing Surveys*, 22(4):299–319 (1990), www.cs.cornell.edu/fbs/publications/smsurvey.pdf.