5 Algorithms for data analysis

This chapter covers

- Querying a stream
- Thinking about time
- Understanding four powerful summarization techniques

Chapter 4 covered how the data flows through many stream-processing frameworks, the delivery semantics, and fault tolerance. In this chapter we're going to depart from the architectural views and discuss the algorithmic side of stream processing, often called *streaming analytics* or *stream mining*. We will focus on the *what* and *why* of streaming analysis algorithms and occasionally dip our toes into the detailed *how*. Don't worry if you're looking for the detailed math or code behind the algorithms—ample resources will be provided so that you can continue your learning.

Before we begin, I'll talk about how we perform queries with these tools. In general, there are two types of queries that you may want to execute in a streaming system:

- *Ad-hoc queries*—These are queries asked one time about a stream. For example: What is the maximum value seen so far in the stream? This style of query is the same kind you would execute against an RDBMS.

- *Continuous queries*—These are queries that are, in essence, asked about the stream at all times. For example: Determine the maximum value ever seen in the stream emitted every five minutes and generate an alert if it exceeds a given threshold.

Unfortunately, in the current technology landscape full of so many different stream-processing frameworks, no two systems offer the same query language, and in many cases there is no SQL-like query language available. Instead you express the algorithmic details programmatically. Table 5.1 shows the current state of query language support in each of the popular stream-processing frameworks (subject to change, as many of these projects are being actively developed and are all maturing).

Table 5.1 Stream-processing framework query language support

Product	Query language support
Apache Storm	As of version 1.1.0 Apache Storm has had SQL support (http://storm.apache.org/releases/1.1.0/storm-sql.html). As of this writing it is still considered experimental and not ready for production use
Apache Samza	Since version 0.9 of Apache Samza there has been a JIRA open for adding query language support. As of this writing, that JIRA is still open, and Samza does not have any query language support: https://issues.apache.org/jira/browse/SAMZA-390.
Apache Flink	Table API supporting SQL-like expressions (http://ci.apache.org/projects/flink/flink-docs-release-0.9/libs/table.html).
Apache Spark Streaming	SparkSQL/Hive language support (http://spark.apache.org/docs/latest/sql-programming-guide.html).

Given the current state of SQL-like support in the market today, I won't show implementation details for each product because they're all different. But I will provide guidance on implementing each algorithm with each stream-processing framework. With a high-level understanding of the general way we may have to perform different stream-mining activities, let's discuss the constraints we must keep in mind.

5.1 Accepting constraints and relaxing

As you know from previous chapters, one of the unique aspects of a streaming system is that we can't store the entire stream because it's unbounded and never-ending. Our goal is to continually provide results to queries online. As data reaches the analysis tier, the results must be recomputed or updated and potentially emitted. On the surface, answering these types of queries may seem easy, but when you consider or design algorithms that will process a stream, it is important to take into consideration the following constraints:

- *One-pass*—You must assume that the data is not being archived and that you only have one chance to process it. This can have significant consequences on your algorithmic development. For example, many traditional data-mining algorithms are iterative and require multiple passes over the data. To work in a streaming scenario, each of these needs to be modified accordingly. I find it helpful to remember that you only get to touch the data one time.

- *Concept drift*—This is a phenomenon that may impact your predictive models. Concept drift may happen over time as your data evolves and various statistical properties of it change. Depending on the type of analysis you are doing and the predictive models you have developed, you may need to take this into consideration.
- *Resource constraints*—For many data streams we have little to no control over the arrival rate of the data. There may be times when, due to a temporary peak in the data speed or volume and the resources at our disposal, an algorithm may have to drop tuples that can't be processed in time, called *load shedding*. This constraint is almost universal in streaming systems, but surprisingly few algorithms take it into account. There are two general types, random and semantic; the latter makes use of properties of the stream and quality-of-service parameters.[1]
- *Domain constraints*—Whereas the other constraints are almost universal to all data streams, these are particular to your business domain. For example, if our social network had 100,000,000 users and we wanted to do an analysis of all emails sent between users, we would need to be able to store double that amount of email addresses. Our storage requirements are easily in the multiple-petabyte range. Being able to do simple statistics or distinct counts about this stream would be challenging. This may appear to be a resource constraint, but it's our business data that causes the constraint.

It is because of these constraints that virtually every streaming method uses some form of synopsis. The basic idea we will see employed is an online synopsis that is used for analysis. Many different kinds of synopsis can be created; as you will see, the exact kind used will have a strong influence on the type of questions that can be answered. Before we dig into these different mining activities, let's look at time as it relates to stream processing and its impact on streaming analysis.

5.2 Thinking about time

If you've worked with a data system where the data is static, such as Hadoop or an RDBMS, you probably thought about time as you were executing queries. In a static world you execute your MapReduce job, Spark job, Hive query, SQL query, or in some other fashion query the data set and perhaps provide a time range in the where clause, and you know the resulting data is all the data that is loaded within a given time range. In contrast, with a streaming system, along with our constraints, the data is constantly flowing. It may be out of order when we see it or delayed—and we can't query all the

[1] For more information, see "Load Shedding in a Data Stream Manager" in *Proceedings of the 29th International Conference on Very Large Data Bases* (2003, pages 309–320), http://dl.acm.org/citation.cfm?id=1315479.

data at once, because the stream never ends. Don't worry—all is not lost. I'll discuss concepts and approaches to thinking about time and solving common problems when analyzing a stream of data.

STREAM TIME VS. EVENT TIME

Stream time is the time at which an event enters the streaming system. *Event* time is the time at which the event occurs. Imagine we are collecting data from a fitness-tracking device such as a Fitbit, and the data is flowing into our streaming system. Stream time would be when the fitness event enters the analysis tier; event time would be when it takes place on the device. Thinking back to our overall architecture, stream time is when the event first enters the analysis tier. If the streaming analysis you're doing relies on event time, realize that it's often not the same as stream time. Often there will be a variance, called *time skew*, sometimes significant, between when an event is created and when it enters the system, as shown in figure 5.1.

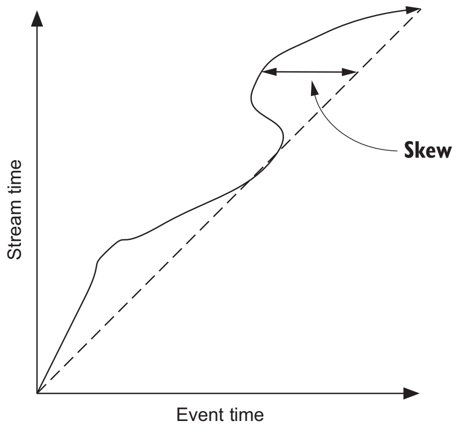

Figure 5.1 Time skew between event time and stream time

Taking into consideration our working example, how would this impact our analysis of the data? How will the drift impact the average speed for the runners we're tracking? Our ability to answer these questions is directly related to the next topic: windowing techniques found in stream-processing systems. Keep the concept of time skew in mind, and we will come back to these questions.

WINDOWS OF TIME

Due to its size and never-ending nature, the stream processing engine can't keep an entire stream of data in memory. This means we can't perform traditional batch processing on it. How then do we perform computations on it? The answer is: by using windows of data. A *window* of data represents a certain amount of data that we can perform computations on. Figure 5.2 shows that a window of data is a small amount of the data flowing through the system at a given point in time.

In figure 5.2, you see that the window is indeed a small part of the entire stream of data. It is a little more complex than that, but not much. The added complexity comes

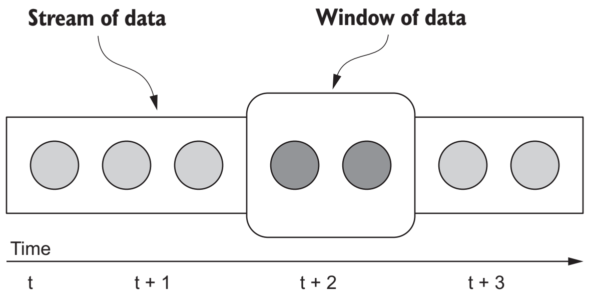

Figure 5.2 A window of data in perspective to the rest of the stream

by the way of two attributes common to all windowing techniques: the trigger and eviction policies. The *trigger* policy defines the rules a stream-processing system uses to notify our code that it's time to process all the data that is in the window. The *eviction* policy defines the rules used to decide if a data element should be evicted from the window. Both polices are driven by either time or the quantity of data in the window. The distinction between the two policies and how time or the count of items come into play will become clearer as we discuss windowing techniques, of which the two most prominent in practice are sliding and tumbling.

5.2.1 Sliding window

The *sliding* window technique uses eviction and trigger policies that are based on time. The two policies are manifested in the window length and sliding interval, as shown in figure 5.3.

The window length represents the eviction policy—the duration of time that data is retained and available for processing. In figure 5.3 the window length is two seconds; as new data arrives, data that is older than two seconds will be evicted. The sliding interval defines the trigger policy. In figure 5.3, the sliding interval is one second.

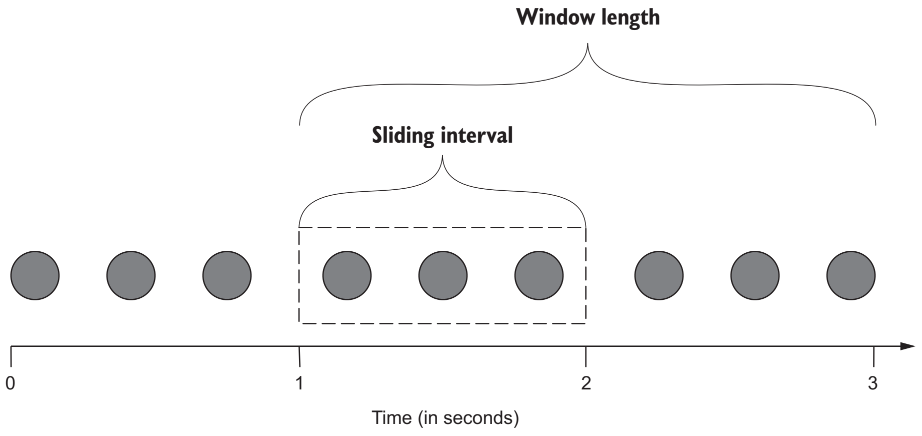

Figure 5.3 Sliding window showing the slide interval and the window length

This means that every second our code would be triggered, and we would be able to process the data in the sliding interval as well as the entire window length.

EXAMPLE USAGE

Going back to our Fitbit example, remember that we have the data flowing into our streaming system. The head of product marketing has asked us to build a dashboard that shows the average speed for all runners broken down by age groups, such as 12–17, 18–24, 25–34, and so on. The dashboard should be updated every 5 seconds, and the averages should represent data for the last 30 minutes. Don't worry about the dashboard aspect; concentrate on the streaming analysis. How you would you handle this using the sliding window technique?

We would want a window length of 30 minutes and a sliding interval of 5 seconds. Remember to take into consideration stream time versus event time. Will your analysis make sense if the window length and sliding interval are based on stream time?

FRAMEWORK SUPPORT

Not all current stream-processing frameworks support sliding windows or provide the same level of support. Table 5.2 identifies the level of support for sliding windows in each of the popular frameworks.

Table 5.2 Sliding window support in popular stream-processing frameworks

Framework	Sliding window	Event or stream time	Comments
Spark Streaming	Yes	Stream time	Spark Streaming doesn't allow custom policies.
Storm	No	N/A	Storm doesn't provide native support for sliding windowing, but it could be implemented using timers.
Flink	Yes	Both	Flink allows a user to define a custom policy and trigger policies.
Samza	No	N/A	Samza doesn't provide direct support for sliding windows.

The details of windowing support for Spark Streaming and Flink are both well documented on their respective project sites. Note that Spark Streaming only supports windowing using stream time. If your application is sensitive to the differences between stream time and event time, you will need to make sure your windowing sizes and algorithms account for this.

For both Apache Storm and Apache Samza, it may be possible to implement sliding window support, but it's not natively supported by either of those tools. So, the work you would have to do may be substantial and not as efficient as a framework that natively supports sliding windows. Delving into the details of implementing this support in either framework is beyond the scope of this text. If that's something you need, check the latest additions of each as well as their JIRA tickets and email lists for discussions on windowing support. Considering that they're all open source projects, you may also contribute enhancements to one of the projects.

5.2.2 *Tumbling window*

A tumbling window offers a slight twist on the windowing concept. The eviction policy is always based on the window being full, the trigger policy is based on either the count of items in the window or time, and they break down into two distinct types: count-based and temporal-based. First let's consider count-based tumbling; figure 5.4 shows how this works.

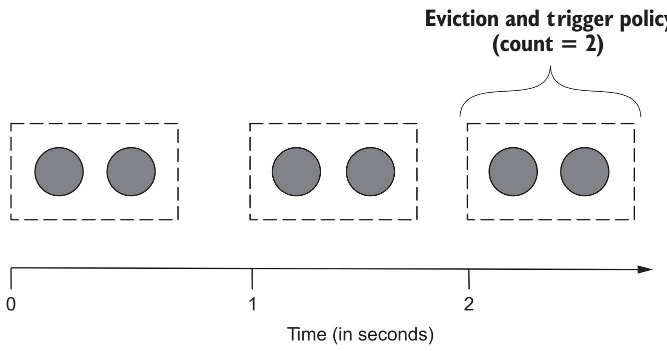

Figure 5.4 Count-based tumbling window with an eviction and trigger policy of two

In figure 5.4 both the eviction and trigger policies are equal to two: when two items are in the window, the trigger will fire, and the window will be drained. This behavior is irrespective of time—whether it takes one second or five hours for the window to fill, the trigger and eviction polices will still execute when the count is reached.

Compare that to the temporal tumbling window in figure 5.5, a tumbling window with an eviction and trigger policy of two seconds.

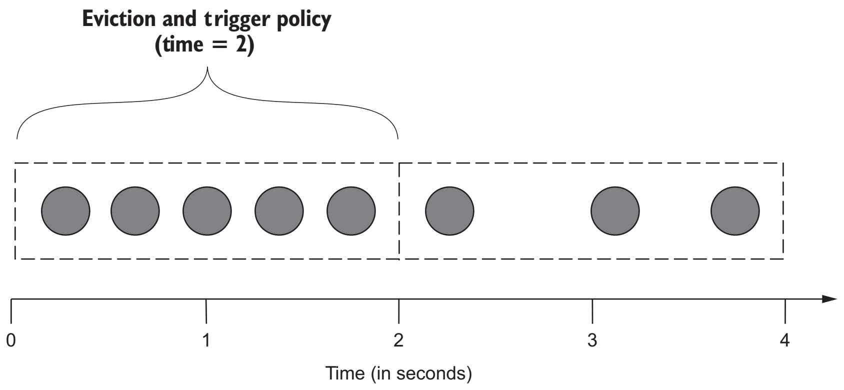

Figure 5.5 Temporal tumbling window with an eviction and trigger policy of two seconds

In the case of figure 5.5, both policies are based on a two-second time frame. In this case it doesn't matter if there are three tuples or five tuples in the window. When the time lapses, the trigger and eviction policies will fire, and the window will be drained. This is distinctly different from the sliding window described in the preceding section.

EXAMPLE USE

Let's imagine that we manufacture a bicycle that is equipped with various sensors, which emit data points such as GPS coordinates, current speed, current direction, ambient temperature, and humidity. From this data set we are interested in understanding two metrics. First, we want to know the average speed of all our bikes every 30 seconds throughout the day. We may break this down by geography, but for now we want a global count. Second, we want to know every time there are more than 100 people riding one of our bikes in a city. Take a moment and jot down how you would handle these two scenarios using tumbling windows.

How did you do? For the first metric, we want our code to be triggered every 30 seconds. To ensure this, I would create a stream that contains only the speed measurement from our sensors and set up a temporal tumbling window of 30 seconds. When our code is triggered, we compute the average using all the tuples in the window at that time. To break this down by geography later, we have a couple of options. One way would be to not pre-filter the stream to contain only the speed measurement, but have it contain the full message sent from the bicycle. Then every 30 seconds we can extract the speed and we would also have the GPS coordinates in hand that we could use to segment the data by any geographic boundary we wanted.

A second way would be to do more filtering. Taking this approach, we would have our collection tier split the data out by geography first and then send it through the rest of the tiers. Then we would have a specific stream: speed with geography. This could be problematic and fairly inflexible. We would need to determine ahead of time the geographic boundaries we used for segmentation and have a strategy for how to handle changes to them.

Let's now consider the second metric we want to capture: every time there are 100 people riding our bicycles in a city. To support this we would need to do two things. First, we create a stream (that may or may not start from our collection tier) for every new city we see in the data and then set up a count-based tumbling window using a window size of 100. When the trigger policy executes, we would have all the tuples for each city that reached 100 cyclists.

Okay, we've worked through two fairly simple examples. Now let's take a look at the current framework support for tumbling windows.

FRAMEWORK SUPPORT

Not all current stream-processing frameworks support tumbling windows or provide the same level of support. Table 5.3 shows the level of support for tumbling windows in each of the popular frameworks.

Table 5.3 Tumbling window support in popular stream-processing frameworks

Framework	Count	Temporal	Comments
Spark Streaming	No	No	Currently you would need to build this.
Storm	Yes	Yes	Although Storm does not have the native windowing support, we can easily implement this.
Flink	Yes	Yes	Flink has built-in support for both types of tumbling windows.
Samza	No	Yes	Samza does not provide direct support for sliding windows.

At the time of this writing Apache Flink is the only framework that has built-in support for tumbling windows, both count- and temporal-based. For the other frameworks the level of effort to implement tumbling window support varies. As with all software, the features available when you evaluate it will likely have changed, so if you need tumbling windows to solve your business problem, double-check the feature set of your chosen tool.

We have now taken a look at the two most common types of windowing found in modern stream-processing frameworks. This information is important to keep in mind as we discuss summarization techniques.

5.3 Summarization techniques

In this section we are going to explore four summarization techniques that form the basis for many different types of analysis you may perform as well as other data-mining techniques you may use. You may wonder why we need to talk about summarizing a stream and question why we need to settle for non-exact answers to questions. The answer lies in the nature of stream processing. Remember, we don't know if the stream will ever end, nor can the entirety of it fit in memory. That makes it extremely difficult to provide exact answers to questions about the data in the stream. In many cases, having a high degree of confidence that the answer to a question is correct or correct enough is adequate. Admittedly you may run into situations where an exact answer must be known, but providing that level of exactness will come at a cost of processing speed and/or implementation. When you are approached with a request to provide exact numbers, it is important to dig in and find out whether a good estimate would work.

NOTE I once worked on a streaming analytics project where we were told our numbers had to be exact because that is how things had always been done in the past (in the pre-streaming world). But due to how the clients were consuming the data, they could not end up with exact metrics. Do you know what happened? You're right—nothing, because the reality was the picture of the business did not change. As humans, we are good at seeing patterns, and if the data being emitted from a stream-processing application is representative of the events occurring in a business—but down-sampled so there is less data—the picture will have the same shape when visualized.

Some of the techniques I cover next are a little deeper. Take your time and if you need to, take it slowly, section by section. Ready? Good, let's now dig into our first summarization technique: random sampling.

5.3.1 Random sampling

Often you may want to take a random sample from a stream. Pretend that we have built a popular advertising network and our ad servers receive 10 million ad views per minute. That's great, but now we want to perform a statistical analysis of the ad serving as it is happening. On the surface that seems pretty easy, but as you think about it you realize that this data is moving fast, it never stops, and it doesn't fit into memory. A viable solution would be to sample the stream as it is flowing. How do we take a random sample from a data set that you can't hold in memory or on disk? How do we know it's random?

A common approach to solving this problem is to use a technique called reservoir sampling. *Reservoir sampling* is based on the notion that we can hold a predetermined number of stream values (the reservoir), and when a new one arrives we can probabilistically determine whether to add it to our collection or randomly select one of the values already in the reservoir as the random sample. Figure 5.6 shows the general flow of reservoir sampling; as new data arrives it goes through a sampling algorithm, and a random sample is determined.

Let's look at what is happening at each step in figure 5.6. Remember, our goal is to ensure that after we process the 16th item, the elements in the reservoir represent a random sample of all the data we have seen, and we have selected a random value. No

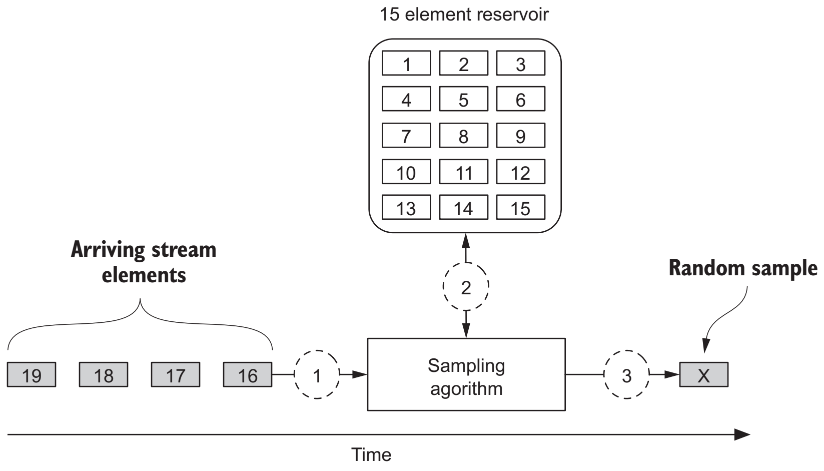

Figure 5.6 General flow of reservoir sampling with first new data item about to be processed

matter how many elements have been consumed from the stream, each element has the same probability of being included in the reservoir. Keep in mind that figure 5.6 shows the state of the reservoir after we have processed the first 15 items. We are using 15, but the general rule is the reservoir is always filled with the first *x* values in the stream, where *x* is the size of the reservoir. After the reservoir is filled and our application is running for a while, we would expect the reservoir to contain a more distributed but random data set.

With that in mind, let's discuss the steps identified in figure 5.6:

1. When the 16th data item arrives, we need to determine if it should be added to the reservoir with a probability of *k/n*, where *k* is the size of the reservoir and *n* is the data element number we are processing. Using these values, the probability that this element should be inserted into the reservoir is 15/16, because we have a reservoir of 15 and we are processing the 16th element.
2. To decide if we add element 16, we generate a random number between 0 and 1. If it is less than 15/16, then we add it to the reservoir and displace one of the items already in the reservoir. If the random number is greater than 15/16, then item 16 becomes our random sample.
3. If element 16 is added in step 2, then we randomly select any element in the reservoir and replace it with the 16th element. The item selected is the random number we use.

That's reservoir sampling. Our next step would be to integrate it into our streaming analysis framework of choice. Currently this algorithm is not provided out of the box with any of the frameworks we have been discussing (Spark Streaming, Storm, Samza, or Flink), but implementing this with any one of them should be fairly straightforward. To learn more about reservoir sampling, the original paper, Jeffrey Vitter's "Random Sampling with a Reservoir" (*Association for Computing Machinery Transactions on Mathematical Software*, 1985, available at www.cs.umd.edu/~samir/498/vitter.pdf), is a great place to start.

5.3.2 Counting distinct elements

You may want to count the distinct items in a stream, but remember we are constrained by memory and don't have the luxury of storing the entire stream. In this section we continue with our ad network example from section 5.3.1, where we have an ad network that is serving 10 million ad views per minute. We're going to try and answer this question: How many distinct ads were shown in the last minute?

The preceding section showed how to take a random sample of that data flowing, but if we wanted to count the distinct ads shown every minute, how would we do that? You may be thinking, "It's only 10 million items—I can store that in a hash table or other data structure that provides search capabilities, and the problem is solved." That may be the case for our ad server, but what if we were building a network intrusion detection system that had to operate at 40 Gbps (~78 million packets per second, assuming 64-byte packets)? In that case, and in any case where we can't store the entire stream, we need to rely on probabilistic algorithms to generate our distinct counts.

There are two general categories of algorithms used to solve this problem:

- *Bit-pattern-based*—The algorithms in this class are all based on the observation of patterns of bits that occur at the beginning of the binary value of each element of the stream. Using the bit pattern—more specifically, the leading zeros in the binary representation of a hash of the stream element—the cardinality is determined. Some of the algorithms you would find in this category are LogLog, HyperLogLog, and HyperLogLog++.
- *Order statistics-based*—The algorithms in this class are based on order statistics, such as the smallest values that appears in a stream. MinCount and Bar-Yossef are two algorithms you would find in this category.

In modern practice the bit-pattern algorithms are most commonly used and are the focus of the remainder of this section.[2]

Let's now turn our attention to the bit-pattern-based algorithms; the most popular and prevalent in practice are HyperLogLog and HyperLogLog++. Conceptually, HyperLogLog and HyperLogLog++ are the same, so I will refer to them collectively as HyperLogLog for this discussion. Figure 5.7 shows the general flow of the algorithm.

Figure 5.7 shows the general flow of processing a new element with the HyperLogLog algorithm. Let's walk through it from the top.

- In step 1 is the ad ID that was viewed. In this case I've used a UUID—there's nothing special about using a UUID; for your data, if you have IDs, you could use them.
- In step 2 the string from step 1 is passed through a hash function, resulting in the hashed value you see before step 3.
- Step 4 is where the magic begins. Here we take the binary string of the hashed value from step 3 and determine which register value, often called the bin, to update and the value to update it with. The six least significant bits are used to determine which register value position will be updated. The number of bits used is called the *precision*; I chose six arbitrarily. If you use this algorithm for your analysis, make sure you understand the precision implications. The binary value of those bits 100010 is 34. Therefore, we are going to be updating the value at index 34.

[2] To learn more about the order statistics–based algorithms, a couple of good jumping off points are Ziv Bar-Yossef's "Counting Distinct Elements in a Data Stream" (*Randomization and Approximation Techniques*, 2002) at https://link.springer.com/chapter/10.1007/3-540-45726-7_1, and Frederic Giroire's "Order Statistics and Estimating Cardinalities of Massive Data Sets" (*International Conference on Analysis of Algorithms*, 2005) at www.emis.ams.org/journals/DMTCS/pdfpapers/dmAD0115.pdf.

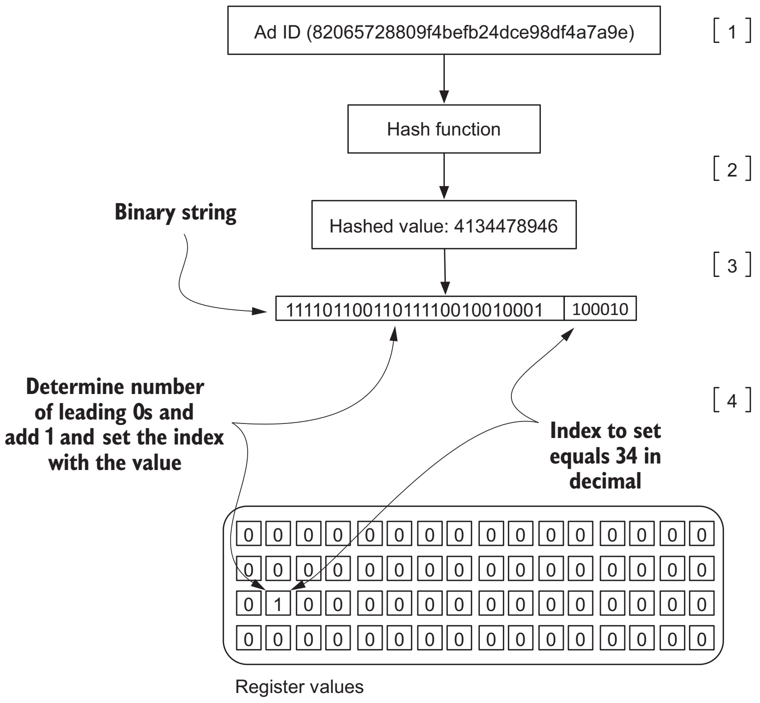
Ad ID (82065728809f4befb24dce98df4a7a9e)
[1]
Hash function
[2]
Binary string
Hashed value: 4134478946
[3]
11110110011011110010010001
100010
Determine number of leading 0s and add 1 and set the index with the value
[4]
Index to set equals 34 in decimal
Register values

Figure 5.7 Processing a single stream element with the HyperLogLog algorithm

- Now that we know the index that will be updated, we determine the number of leading zeros, starting from the right, for the rest of the bit string and add 1 to it. In this case there are no 0s, so we end up with 0 + 1, and we update index position 34 with the value of 1.
- At this point you can determine the distinct counts (again, it's an approximation) by taking the harmonic mean of all the register values.

That is the general flow of the algorithm. With this algorithm keep in mind that the count of leading zeros in a bit string is used to estimate the cardinality of a stream. Then to increase accuracy, the average of many estimates is taken to reduce bias and the harmonic mean is used to reduce the impact of outliers. These algorithms have their start with, and are enhancements to, the original work by Philippe Flajolet and G. Nigel Martin's "Probabilistic Counting Algorithms" (*Journal of Computer and Systems Science*, 1985) and more recently Durand and Flajolet's "LogLog Counting of Large Cardinalities" (*Annual European Symposium on Algorithms*, 2003).

HyperLogLog++ provides several improvements over HyperLogLog, namely in the reduction of memory usage and an increase in accuracy for a range of cardinalities. Our focus has been on how these algorithms work conceptually so you know how to think about and use them.[3] In practice this algorithm isn't hard to implement, and in fact you may be able to find implementations readily available in the language you're using.

A couple of other things to keep in mind regarding HyperLogLog are that it uses little space and is distributable. From a size and space standpoint, according to the authors of the papers I mentioned, you can count one billion distinct items with an accuracy of 2% using only 1.5 K of memory, which is quite impressive. From the distributed standpoint it is easy to perform a union operation between two HyperLogLog structures. When doing stream analysis, this will enable you to maintain summarizations on each node that is analyzing data and then join them to determine an overall approximate, distinct count.

You should also be able to integrate this into any of the streaming frameworks we've been looking at. With this information you can now determine the approximate distinct counts for your stream. In the next section we will look at an algorithm that helps us answer a slightly different question.

5.3.3 Frequency

The preceding section discussed determining the distinct count for a stream. In this section we'll try to answer this question: How many times has stream element X occurred?

See chapter 9 section 9.3.2 and code listing 9.25

The most popular algorithm for answering this type of question is called Count-Min Sketch.[4] This algorithm can be used any time you need count-based summaries of your data stream. In general Count-Min Sketch is designed to provide approximate answers to the following types of questions:

- *A point query*—You are interested in a particular stream element.
- *A range query*—You are interested in frequencies in a given range.
- *An inner product query*—You are interested in the join size of two sketches. For our ad example we may use this to provide a summarization to this question: What products were viewed after an ad was served?

These three types of questions are fundamental to a lot of streaming applications. In our ad-serving example, we may want to ask how often ad X has been viewed. You will also find that similar questions are fundamental to network monitoring and analysis, where millions of packets per second are processed and there is a strong desire to prevent malicious intent such as a Denial Of Service (DOS)

[3] To understand the inner workings of these algorithms I encourage you to read Flajolet, Fusy, Gandouet, and Meunier's "HyperLogLog: The Analysis of a Near-optimal Cardinality Estimation Algorithm" (*Conference on Analysis of Algorithms*, 2007) at http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf and Huele, Nunkesser, and Hall's "HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm" (*Proceedings of the EDBT*, 2013 Conference) at https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/40671.pdf.

[4] Graham Cormode and S. Muthu Muthukrishnan first published an article on this algorithm in the *Journal of Algorithm* (2004) titled "An Improved Data Stream Summary: The Count-Min Sketch and Its Applications." You can read it at http://dimacs.rutgers.edu/~graham/pubs/papers/cm-full.pdf.

attack.[5] I'm sure you can come up with many more examples of when the Count-Min Sketch algorithm could be useful; for now let's dig into how this works.

Count-Min Sketch, as its name implies, was designed to count first and compute the minimum next. Let's get a of couple of definitions out of the way before we see how this works diagrammatically. Count-Min Sketch is composed of a set of numeric arrays, often called *counters*, the number of which is defined by the width w and the length of each is defined by the length m. Each array is indexed starting at 0 and has a range of $\{0 \ldots m - 1\}$. Each counter must be associated with a different hash function, which must be pairwise independent—otherwise the algorithm won't work as designed. How this all comes together is shown in figure 5.8.

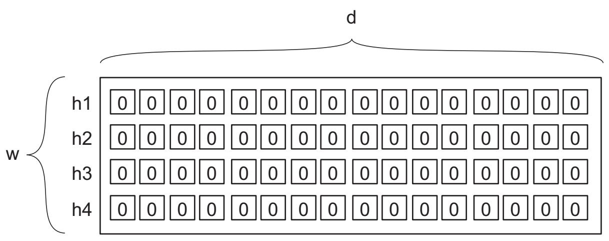

Figure 5.8 Setup of the Count-Min Sketch algorithm

As figure 5.8 shows, this is a 2-dimensional array with all elements initialized to 0 and each row associated with a different hash function. Using different hash functions increases the accuracy of the summary while also reducing the probability of bad estimates, as the chance of hash collisions has been reduced. For our ad network example, the sketch will represent a probabilistic summarization of how many times an ad was served. If our sketch looked like figure 5.8, we would have a 4 x 16 2-dimensional array. Each row is independent and represents a bit array that we'll use to keep count.

Now let's walk through the process of updating the sketch as ad view data is streaming into our system, as shown in figure 5.9.

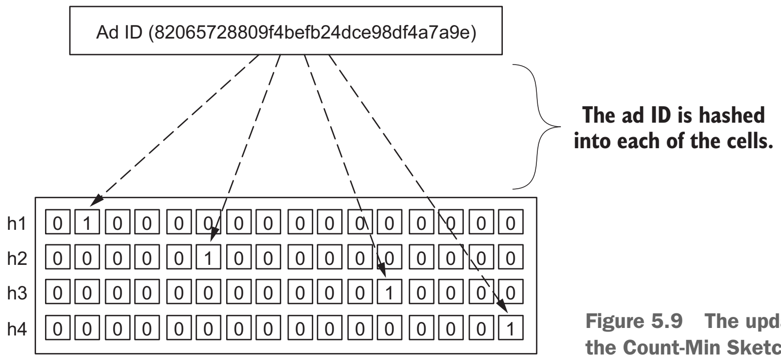
Ad ID (82065728809f4befb24dce98df4a7a9e)
The ad ID is hashed into each of the cells.
h1
h2
h3
h4
0
1

Figure 5.9 The update process for the Count-Min Sketch algorithm

[5] For an idea of how this type of algorithm is used in network monitoring and analysis, a good place to start is with Cormode and Muthukrishnan's "What's New: Finding Significant Differences in Network Data Streams (INFOCOM, 2004) at http://infocom2004.ieee-infocom.org/Papers/33_1.PDF.

In figure 5.9 we have an ID of an ad that we want to add to the sketch. The first step is to hash the value using the hash function for each respective row and then increment the count for the cell the value hashes to by 1. For our example, all the values were 0, so the result counts are all 1. As ad view data continues to flow through our streaming system, we will repeat this process of updating our sketch. After some time has passed, we want to estimate how many times our ad from figure 5.9 was viewed. To get this frequency estimate, we would use the following equation:

ESTIMATED COUNT =
min(h1(82065728809f4befb24dce98df4a7a9e),h2(82065728809f4befb24dce98df4a7a9e),
h3(82065728809f4befb24dce98df4a7a9e),h4(82065728809f4befb24dce98df4a7a9e))

In that function we're hashing the ID of the ad we're interested in. That gives us the four cells to look at. That is a salient point that may not be obvious from the example equation.

Specifically the result of h1 is the hash that determines the counter to look up. This is the same for h2, h3, and h4. We then take the minimum value from the four cells. This value represents the approximate count for the number of times the ad was viewed. Keep in mind that this algorithm will never undercount, but could overcount. How accurate is this? In the original paper, the authors show that with a width of 8 and a count of 128 (a 2-dimensional array of 8 x 128) the relative error was approximately 1.5%, and the probability of the relative error being 1.5% is 99.6%.

I find it fascinating that we can do this with little space and with little computational cost. This is a pretty straightforward algorithm that can be used to answer a lot of questions. To learn more and gain a deeper understanding of the why behind it, read the award-winning paper by Cormode and Muthukrishnan, "An Improved Data Stream Summary: The Count-Min Sketch and Its Applications (*Journal of Algorithm*, 2004).[6]

Up next we're going to talk about a sketch that is closely related to the Count-Min Sketch, except this one is used when you want to determine whether you've seen a stream element before.

5.3.4 Membership

The question we're asking now is: Has this stream element ever occurred in the stream before?

That may seem like a tall order to fill. We know from earlier discussions that we can't store the whole stream—realistically we can't even store an ID for every element we've seen in the stream. You may wonder how then are we going to pull this off? Simple. We're going to use a data structure that you should look to when trying to answer membership type queries: a *Bloom filter*. A Bloom filter is tailor made for this specific task. As with the other algorithms we've seen in this chapter, the Bloom filter's accuracy is probabilistically bound, and as expected, this is configurable.

[6] The paper can be found at http://dimacs.rutgers.edu/~graham/pubs/papers/cm-full.pdf.

A unique feature of Bloom filters is that false positives are possible, but false negatives are not. What exactly does that mean? It means that if the filter reports that the stream element has not been seen, then that will always be true. But if the filter reports that the element *has* been seen before, then that may or may not be true. In the literature there are various advanced Bloom filters, but for our discussion we'll stick to the good old plain one. Once you understand how it works, you'll be ready to take on more complex ones.

A Bloom filter is composed of a binary bit array of length *m* and is associated with a set of independent hash functions. Does that sound familiar? Remember from our discussion of the Count-Min Sketch that it's composed of multiple arrays, each of width *w* and length *m*—pretty interesting, huh? It doesn't take many changes to go from one to the other. Similar to the Count-Min Sketch, the elements of the bit array are indexed starting at 0 ending at $(m - 1)$, and because the Bloom filter is a binary bit array of length *m*, the space requirements are $m/8$ bytes.

Figure 5.10 shows how this algorithm works.

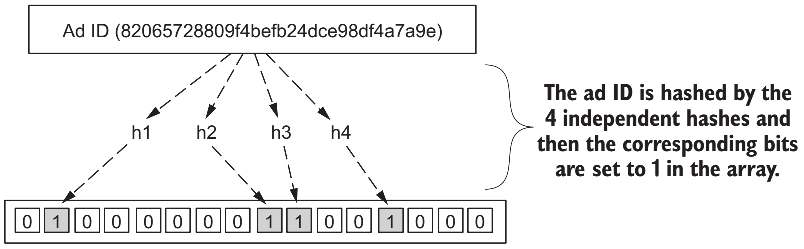
Ad ID (82065728809f4befb24dce98df4a7a9e)
h1
h2
h3
h4
0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 0
The ad ID is hashed by the 4 independent hashes and then the corresponding bits are set to 1 in the array.

Figure 5.10 A Bloom filter showing one stream element being processed

That's it—quite straightforward. You may have already realized there will undoubtedly be collisions as you process all your data, and those can lead to the false positives I mentioned. When that happens, if the bit in the array is already set, it remains set. Querying a Bloom filter to check the membership of a stream element is also quite simple and comes down to this:

$$MEMBERSHIP\ of\ Stream\ Element\ Z\ = AND(h1(Z),h2(z),h3(z),h4(z));$$

With this, we compute each of the hashes and then check the array to see if all the elements are 1, and if any of them is 0 we are guaranteed that the element has never been seen before. To dig deeper into Bloom filters, a great place to start is the original article by Burton Bloom titled "Space/Time Trade-offs in Hash Coding with Allowable Errors" (*Communications of the ACM*, 1970), there are many papers that have been published since then that discuss more advanced bloom filters.[7]

[7] Bloom's article can be found at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.2080.

This data structure can be used to determine whether you have ever seen a stream element—before incurring the cost of performing an expensive computation that may involve querying an external data store. Maybe you're building a network-monitoring application that keeps track of known bots and/or bad hosts. As you watch traffic flow you can query a Bloom filter, and if it comes back positive that the packet was from a malicious host, *then* you can perform the more costly operation to confirm if it is indeed a packet that should be rejected. Maybe you're not building either of these, but I think you get the general idea here. It may not come as a surprise that this data structure is called a *filter,* as that is the most common use case.

5.4 Summary

In this chapter we took a step back from discussing architecture and dove into how to think about querying a stream, considered the problems with time, and dug into four popular summarization techniques. You learned about the following:

- The different types of queries
- How to think about time when dealing with a streaming system
- Four powerful stream summarization techniques that form the basis of a lot of streaming analysis programs.

I understand that some of this may have been a little deep. Don't worry about it. As you start to build out a streaming system, a lot of this will start to crystalize. You may want to pick one of the summarization techniques and apply it to one of the problems you're trying to solve. The architecture may be fun, but the exciting part comes when you apply what you've learned in this chapter. My hope is that you're ready to start asking questions of the data you're working with.

The next chapter covers how to store the results of the analysis you learned to perform in this chapter. This may be a good time to refill your coffee.

6 Storing the analyzed or collected data

This chapter covers

- Understanding why we need to store the data
- Learning what to store
- Choosing your storage solution

Up to this point we have spent time discussing the architecture and the algorithms commonly used in stream-processing applications. This chapter focuses on what to do with the data after you have processed it. Our focus will be less on the performance of any persistent store we use and more on how to choose a persistent store if one is needed for your application.

First let's recap where we are in the overall architecture. Figure 6.1 shows our over-arching architecture with the focus of this chapter highlighted.

We'll talk about the storage options from a streaming perspective, the key attributes of the most popular products, and things to consider when you use one. To set the stage, let's begin with the four options we have when our analysis is done and the data is ready to be consumed. We can do any of the following:

- Analyze and discard the data
- Analyze and push the data back into the streaming platform
- Analyze and store the data for real-time usage
- Analyze and store the data for batch/offline access

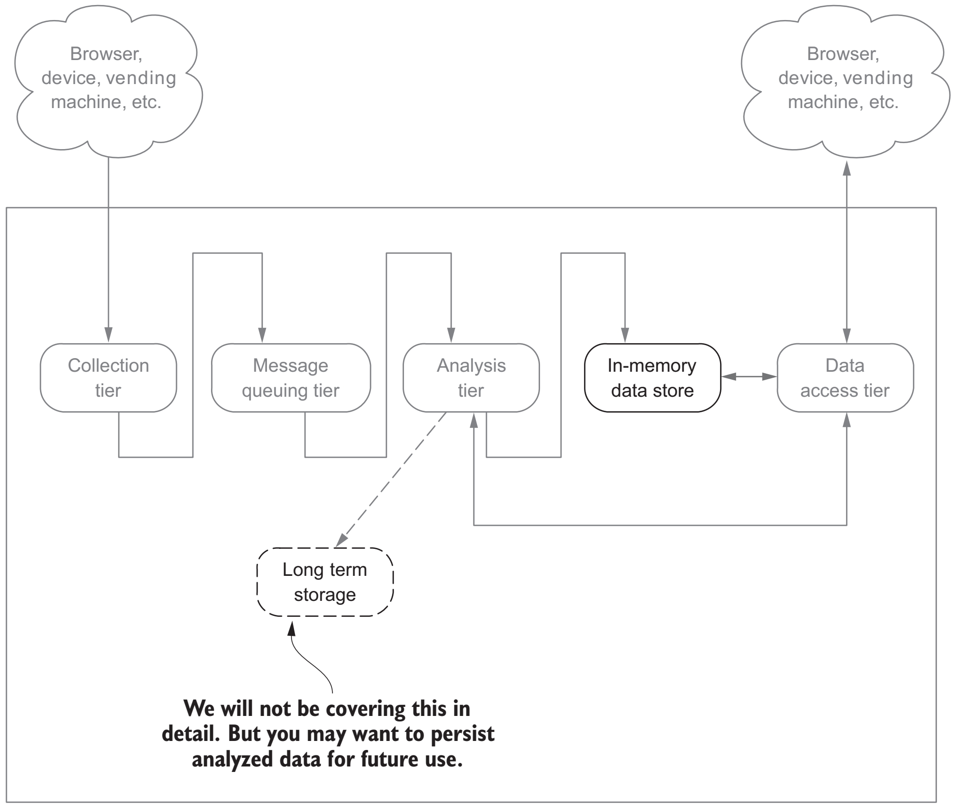
Browser, device, vending machine, etc.
Browser, device, vending machine, etc.
Collection tier
Message queuing tier
Analysis tier
In-memory data store
Data access tier
Long term storage
We will not be covering this in detail. But you may want to persist analyzed data for future use.

Figure 6.1 Overall streaming architecture blueprint with the chapter focus identified

Let's walk through each of these briefly. Discarding the data may be a realistic use case. You may perform an analysis of the data and subsequently discard it if it doesn't meet a certain threshold. You may not need me to remind you, but if you're only processing the data as a stream and don't have another copy of it—either from the source or from processing it via a more traditional batch process—then when you discard it, it's gone for good. This option's implementation and impact are quite obvious.

Pushing data back into the streaming platform is an interesting option. The output from one streaming analysis can become the input to another. This type of pattern is common when chaining multiple stream-processing workflows. In fact, this pattern is at the core of Apache Samza.

Then we have the last two options: analyzing and storing the data for real-time access and analyzing and storing the data for batch/offline access. Why in a book on streaming systems are we spending time discussing storing data for batch/offline access? Great question. We won't be digging into how to choose one of these data stores; many books and online resources are dedicated to that topic. Instead, we will be looking at these storage options from the viewpoint of a stream and developing our understanding of the delicate balance between the two systems. After that we will turn our focus for the rest of the chapter toward the technologies and ideas we need to keep in mind for our in-memory store. By the end you will know how to handle both storage scenarios and be poised to set up a streaming system with an in-memory store.

6.1 When you need long-term storage

There will be times when you need to have the data you're processing in your streaming system stored in a storage system designed for a non-streaming scenario, perhaps for more traditional batch or offline access. For example, writing to Amazon S3, HDFS, HBase, or many traditional RDBMSes would be considered writing to a non-streaming data store. When the need arises to write to these stores, you have the three options shown in figure 6.2.

As you'll see in later sections, some modern in-memory technologies and techniques let you write to a long-term store in different ways than described in figure 6.2.

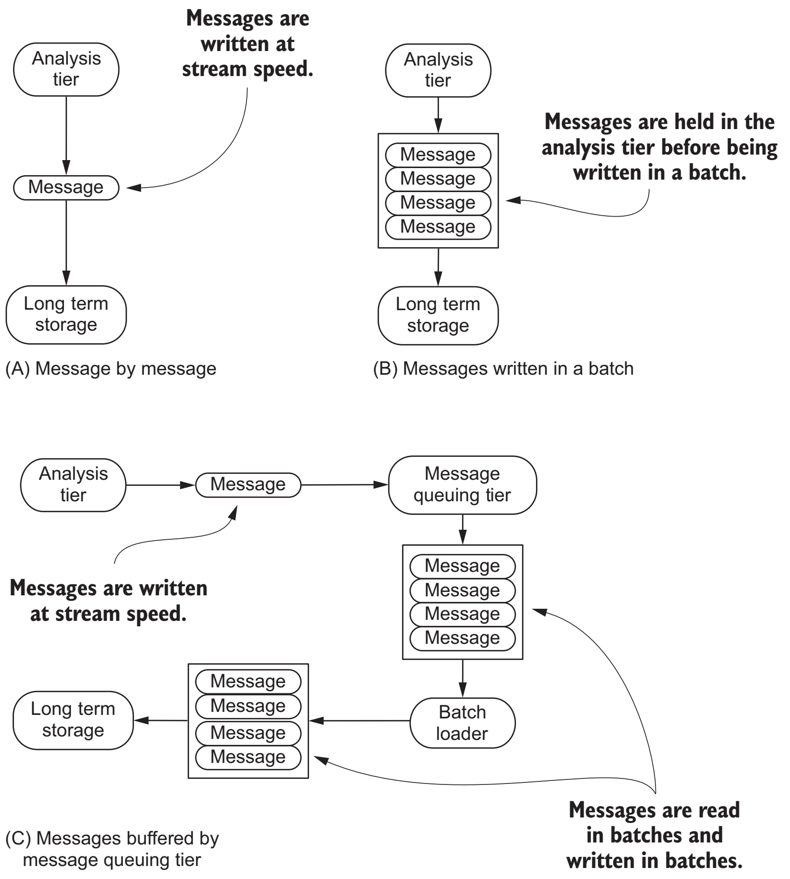
Analysis tier
Message
Long term storage
Messages are written at stream speed.
(A) Message by message
Analysis tier
Message
Message
Message
Message
Long term storage
Messages are held in the analysis tier before being written in a batch.
(B) Messages written in a batch
Analysis tier
Message
Message queuing tier
Messages are written at stream speed.
Message
Message
Message
Message
Batch loader
Long term storage
Message
Message
Message
Message
Messages are read in batches and written in batches.
(C) Messages buffered by message queuing tier

Figure 6.2 Three ways to write data to long-term storage

But for now we will focus on the options we have if we're only interested in populating a long-term store and not in-memory.

DIRECT WRITING

Start with options A and B in figure 6.2, both of which I consider to be forms of direct writing, even though they're slightly different in that the analysis tier is writing directly to the long-term store. Option A is the most direct method of getting data to a long-term store. Following this pattern, you write each message as it is processed—in essence writing at stream speed. In option B you build up the messages in the analysis tier until either a certain batch size is reached or a certain period of time passes and then write them to long-term storage.

There are potential issues and risks in options A and B. If the data store you're writing to can't keep up with the rate at which you're processing data—our stream time—then that may have a negative impact on your ability to process the data in the stream. If you're processing data at speed, it becomes harder and harder to tune a batch-oriented storage system as your data volume and velocity increases. And that makes perfect sense because these are systems that are often designed for situations where data is loaded by batch and then processed. Although option B involves building up batches of messages before writing them, it is an attempt to optimize for the slow write speed (*slow* is relative—in this case it's related to stream speed).

INDIRECT WRITING

In figure 6.2's option C the approach is quite different and falls into a category I'll call *indirect* writing. In this case we decouple the storage of the stream-processed data from the long-term store. We write the data out to a message queuing tier as an intermediary. This helps decouple our streaming analysis tier from the performance of the long-term storage system. We've added a little complexity, but that's outweighed by the benefit of offloading the data storage to a more appropriate place. The added complexity we need to take on in this case is in the form of another component that needs to be used, called in figure 6.2 the *batch loader*. The goals of the batch loader are to read batches of messages from the message queuing tier and write them to our long-term storage.

This approach has two benefits. First, as we know from earlier chapters, the message queuing tier is designed to handle the speed and volume of a stream, eliminating the risk of not being able to write fast enough. Secondly, we can use a component that is dedicated to bulk loading data, ensuring that we can keep our streaming analysis focused on analyzing the stream. Although we won't focus on bulk loading tools in great detail, it is important to understand the overall capabilities these tools provide. From a capability standpoint, the most common features that you would expect from data ingestion extract, transform, and load (ETL) tools are as follows:

- Job/task scheduling
- Error handling
- Data quality checking
- Data publishing

- Monitoring/metrics
- Horizontal scaling
- Fault tolerance
- Extensibility
- Strong consistency

Although it's easy to look at that feature set and reason about how we would have some of that in our stream processing, consider that we often want to write to a long-term store like HDFS or S3, neither of which is ideal for continuous high-volume writing, having been designed for writing large amounts of data in a batch-oriented fashion. From an architectural standpoint the long-term store is also cleaner because we have a separation of concerns and components focused on a single task.

Two of the most prominent tools used for performing this type of batch loading are Goblin and Secor. Linked-In's Goblin project (http://gobblin.readthedocs.io/en/latest/) is a universal data ingestion framework, the latest built by LinkedIn after years of experience building and running ingestion frameworks. Another popular one is Pinterest's Secor project (https://github.com/pinterest/secor). Both projects provide good options for ingesting data from Kafka or another data source and publishing it to HDFS and/or S3. If your project requires batch loading of data into a long-term store, I encourage you to consider one of these. Make sure you look closely at their feature sets and ensure that they match your requirements.

At this point we've discussed the two main options for getting data from your streaming analysis to a long-term data store. Now let's look at the in-memory options and how we can make our data available for real-time use. Get a refill and then come back and get ready to explore the world of in-memory data storage.

6.2 Keeping it in-memory

When building a streaming analysis system, the goal is to be able to take action on the data in real time, when an event occurs, as soon as possible. Imagine we're running a power company that has smart meters deployed across the world. Each meter reports its status every 30 seconds, and there are many meters connected to a single transformer. We can easily analyze the data about the meters, but what if we could take the data from all the meters connected to a transformer and watch for trends? Perhaps using a 30-minute window we see that the transformer is starting to trend toward malfunctioning and we can immediately shut it down before it malfunctions and possibly destroys other components. To be able to make this type of decision we need the current and recent data to be accessible in real time—we need it in-memory.

Not too long ago, keeping a lot of data in-memory seemed like a good idea for caching, but not for analytics. Times have changed, and the writing has been on the wall for a while. In 2006 the late Jim Gray gave a talk titled "Tape Is Dead, Disk Is Tape, Flash Is Disk, RAM Locality Is King" (http://research.microsoft.com/en-us/um/people/gray/talks/Flash_is_Good.ppt). More recently, Gartner analysts have been quoted as saying, "RAM is the new disk, disk is the new tape." There *is* still tape, but its disappearance is only a matter of time. Today you can buy servers with 32 TB or even 64 TB of memory. For approximately $5,000 a month you can have a cluster in Amazon EC2 with 1 TB of DRAM. No matter how you look at it, the idea of keeping the entire working set for your streaming system in-memory is now a reality and is something you should seriously consider as you begin to build a streaming system.

You may be thinking, "I have SSD drives—isn't that fast enough?" Perhaps, but a single seek on an SSD drive today takes approximately 100,000 nanoseconds (ns); it only takes 100 ns to reference main memory, and that goes down to 0.5 ns to access an L1 cache reference. No matter how you slice it, accessing data in DRAM is still significantly faster than even SSDs. It may seem like 100,000 ns is fast enough, and for a single access it may be. But when you're processing a continuous stream of data, all such little costs add up to be quite significant. It's no surprise that since we can now economically store entire data sets in-memory, all of a sudden every vendor has an in-memory offering, and each is better than its competitors. The question is: How do we match them to our use cases?

Let's look at the categories these technologies fall into and at some products you will find in each category. After that we'll walk through examples to help you choose among them.

6.2.1 *Embedded in-memory/flash-optimized*

See chapter 9 section 9.4

In this category are products designed to be embedded into your software. As such they are focused toward a single node, don't provide any way to access data across nodes, and don't provide any of the management-related features you find in non-embeddable systems. These products are not a good fit for the distributed streaming systems we have been talking about building. Remember, we want to store the analyzed data so that we can make it accessible to clients in real time. Using an embedded database means we have to find a way to get access to our analysis nodes. Figure 6.3 illustrates this.

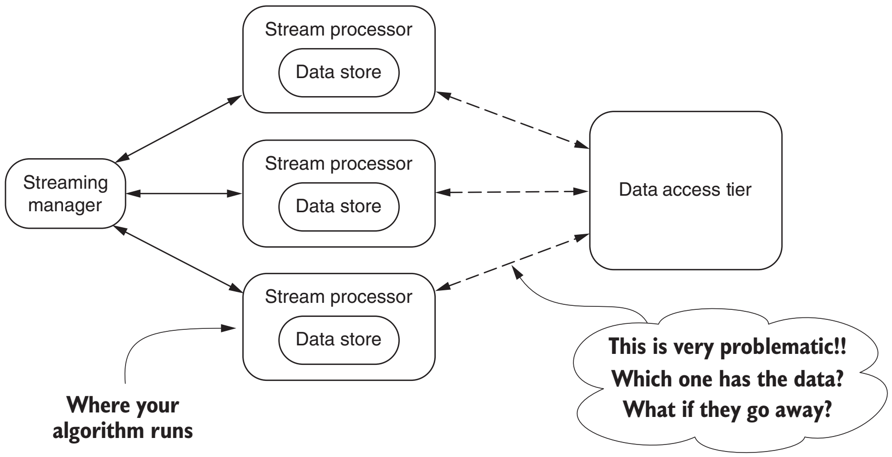

Figure 6.3 Architectural view of using an embedded data store with our analysis nodes

In figure 6.3 you can see that the embedded data store will live on the same node as our stream processor. In this case we will run into several problems that will need to be addressed, two of which are called out in the figure. It doesn't take long to realize this type of design is not ideal; it's fragile, prone to error, and doesn't follow good software architecture design patterns. The stream processor nodes are performing many roles: stream processing, local data storage, and serving data to a data access tier. For these reasons I won't spend much time discussing this approach, but I realize there may be a time and use case in which your streaming system requires this type of architecture. The following list suggests products you may want to consider:

- *SQLite (www.sqlite.org)*—This is a serverless embeddable database designed for local storage for applications and/or devices. Supports most of the common features of the SQL language; for a list of what isn't supported, see www.sqlite.org/omitted.html.
- *RocksDB (www.rocksdb.org)*—An embeddable key-value store designed for fast storage that builds on LevelDB. It can be used as the basis for building a traditional client-server solution.
- *LevelDB (https://github.com/google/leveldb)*—The precursor to RocksDB, an embedded key-value store that provides ordered mapping from keys to values.
- *LMDB (http://symas.com/mdb/)*—An embeddable key-value store developed as a replacement for Berkley DB. It is fully transactional, uses memory-mapped files, and is designed for read-heavy workloads.
- *Perset (www.mcobject.com/perst)*—A fully transactional object store for Java and .NET, designed for speed, easy of use, and transparent storage between supported languages and the data store.

This list is by no means exhaustive. You will find different options as you research this topic.

6.2.2 Caching system

Products in this category may also be called by other names, such as an object caching system, in-memory store, or even in-memory key-value store. The key features are that they are designed to store data in memory, there often is no option to store the data outside of DRAM, and often the API is key-value based. Caching systems use many different strategies; the ones related to persisting cache entries and keeping the cache fresh are most relevant to our discussion in this section. Let's go over the common strategies employed by most of the products in this category.

READ-THROUGH

With this strategy the caching system reads data from a persistent store when it's asked for a cached entry that isn't in the cache. Read-through incurs the cost of reading from the persistent store the first time a cache entry is asked for that doesn't already exist in the cache. This is transparent to the client of the cache, but there is the performance impact of the caching system having to read the data from a persistent store, as well as the updates having to be written to the persistent store. Figure 6.4 illustrates this strategy.

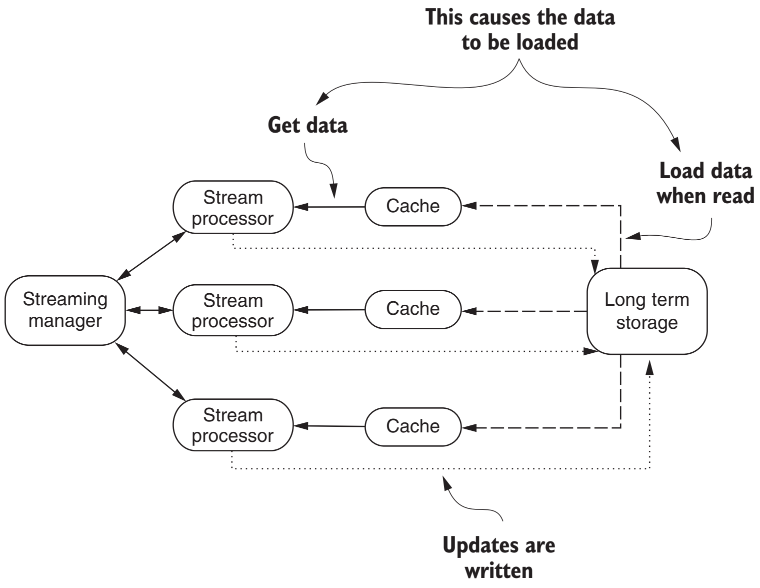

Figure 6.4 Read-through caching strategy showing how the data is loaded from long-term storage if it's not in the cache

REFRESH-AHEAD

The goal of this strategy is for the cache to refresh recently accessed data before it's expired and evicted. This attempts to avoid the read-through performance penalty of having to retrieve data from a persistent store when a cache entry expires and is evicted from the cache. If refresh-ahead is configured to closely match the update frequency of data in the backing store, then it may enable the cache to return the most current value to a client and thereby keep it in sync with the backing store. Keeping this level of fine-grained coordination may be hard to do with a stream of data. Figure 6.5 graphically depicts this strategy.

WRITE-THROUGH

This strategy has the caching system write updated data through to the backing store, eliminating the need for an out-of-band process to write data to the backing store or load changes into the cache. With this feature a caching system doesn't acknowledge the write as being successful until it's written to the backing store. With this you incur the latency of writing to the backing store, which can be deemed a disadvantage compared to the other strategies. This strategy is depicted in figure 6.6.

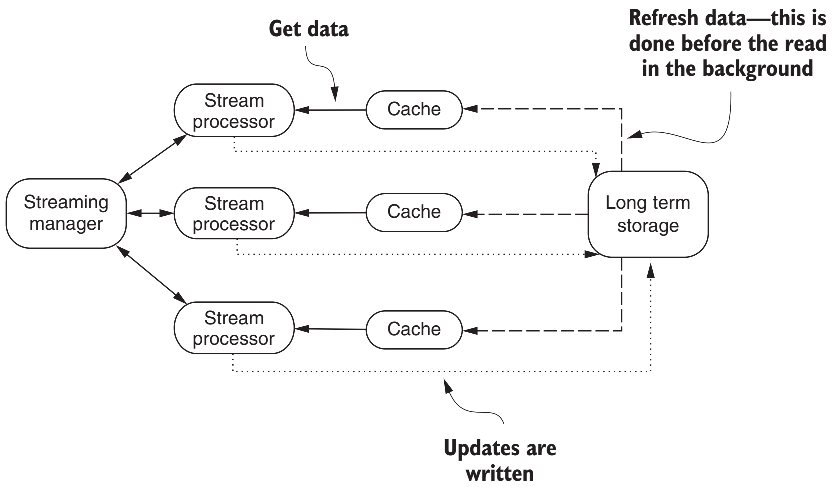

Figure 6.5 Refresh-ahead cache strategy, showing how data is refreshed from long-term memory

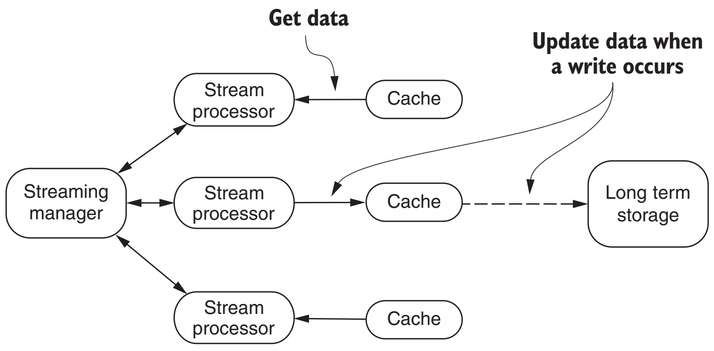

Figure 6.6 Write-through cache strategy showing how data is written all the way through to the long-term storage

Write-around

With this strategy the idea is that the process of updating a persistent store that the cache is representing happens out of band of the cache. *Out of band* refers to the updating of the cache happening in the background—the dotted line in figure 6.6. Often out of band refers to an activity happening in a secondary pathway other then the primary. In this case the caching system doesn't know about the changes being written to the persistent store and relies on another process to update the cache after the persistent store is updated. Here you're adding the complexity of having to update two data stores: the persistent store and the cache. But the advantage is that the caching system doesn't have to talk to the persistent. Figure 6.7 shows this graphically.

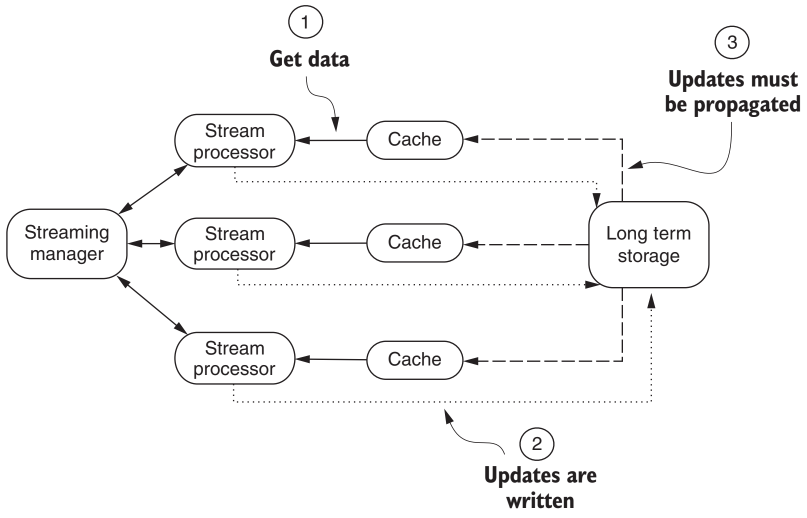

Figure 6.7 Write-around caching strategy showing how data is written to long-term storage and then has to be propagated to the cache

WRITE-BACK (WRITE-BEHIND)

With this strategy, the caching system eventually writes the new data to the persistent store. Unlike write-through, where the data is immediately written, with write-back the write to the cache is acknowledged, and then in the background the updated or new data is written to the persistent store. This has the advantage of not incurring the I/O overhead of the write-through, although for the period of time when the data is only held in memory there is a risk of data loss. Write-back is illustrated in figure 6.8.

With all these strategies you're either in a situation where the data is only in-memory or you pay the cost of having to load data from a persistent store. The fact that there is

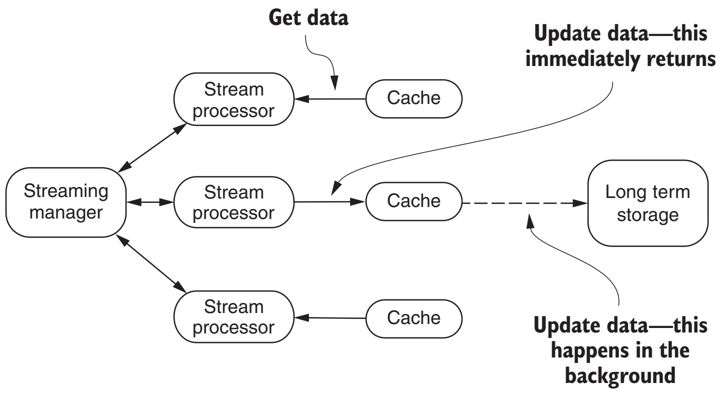

Figure 6.8 Write-back caching strategy

often no option to store the data other than in memory poses a risk as well because if you lose a cache node, you lose your data.

Some systems in this space, such as EHCache (www.ehcache.org), do provide ways of extending it to support read-through, write-through, and write-behind functionality. With a streaming system processing a constant flow of data, this dual writing and propagation of changes will become a significant bottleneck. If your results are transient and never need to be persisted, this bottleneck may not be an issue. In general, though, you'll want to look at historical data—in which case, using a caching system will force you to dual write your streaming analysis results, potentially incurring significant overhead.

If you're interested in exploring these systems in greater detail, here are several of the most popular open source products in this segment:

- *Memcached (http://memcached.org)*—This is a popular cache product, but you'd need to employ the write-around strategy to keep it fresh with data.
- *EHCache (www.ehcache.org)*—This is a sophisticated caching system with support for various usage scenarios—namely write-behind, write-through, and read-through.
- *Hazelcast (http://hazelcast.org)*—This is a robust product with a lot more than caching. On the caching side, it has support for read-through and write-through strategies.
- *Redis (http://redis.io)*—This is a popular option that works well as an in-memory cache. It does have persistence but to its own file formats. You'd need to build any of the strategies discussed earlier.

6.2.3 In-memory database and in-memory data grid

In-memory databases (IMDBs) are sometimes called *in-memory database management systems* and *in-memory data grids (IMDGs)*. Unlike the caching systems, IMDBs and IMDGs are designed to use disk for the non-volatile persistence of data. Although the products in this category use a storage medium other than DRAM, they are designed to use memory first and disk second. In fact, disk storage is often only used for logging and periodic snapshots so the data is available in the face of failure. This subtle design decision is what separates the true IMDBs and IMDGs from the traditional databases that now offer an in-memory option. Traditional databases (Microsoft SQL Server, Oracle, and so on) as well as modern NoSQL ones (such as Apache Cassandra) were all designed with a disk-first, memory-second mindset.

What is a disk-first mindset versus a memory-first mindset? Glad you asked. Figure 6.9 illustrates the stark differences in the approaches.

The disk-first approach shown in figure 6.9 is a simplified picture of how a traditional database and many NoSQL databases have been designed. It should be apparent why products that started as disk-based systems and now claim to have in-memory features were designed this way. The problem is revealed later: When that product adds features to become an in-memory database, the same design principles are applied;

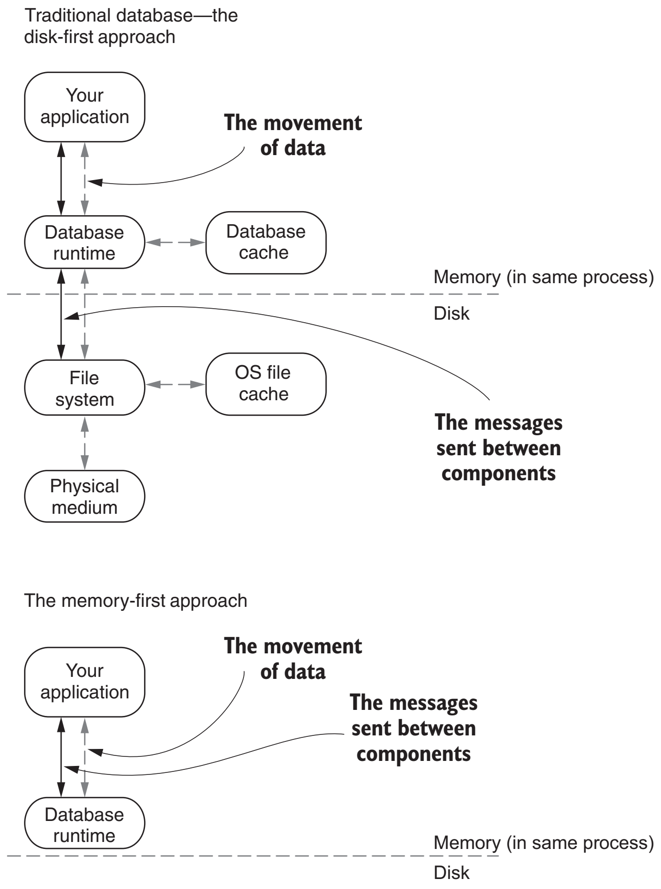

Figure 6.9 The two different approaches to database design: disk-first and memory-first

memory is swapped for disk but treated like another I/O device. That's not to say these products don't use memory from the start; for performance reasons they all happily use memory as a cache.

The memory-first mindset as shown in figure 6.9 completely flips this. With the memory-first approach, the software is designed to only use memory, and then later the disk is used secondarily for reliability. This subtle difference has a profound impact on the performance of the products in this segment. Several products cross over into this category from the caching category; with IMDB and IMDG products, the lines are getting blurry, change frequently, and are at this time fairly fluid.

Traditionally there were several key differences that distinguished between IMDBs and IMDGs. IMDGs used a distributed architecture and allowed computations to be performed on the server close to the data, similar to the stored procedures you're used to in the RDBMS world. IMDBs would often provide API access via SQL, whereas IMDGs usually used a non-SQL API. As the industry has matured in the streaming space, these lines are getting erased, and IMDGs now sometimes find themselves having to offer much richer computation models as products like Apache Spark encroach on the in-memory computation capabilities. As this has been happening, IMDBs have been moving closer to being IMDGs, and IMDGs have been moving toward trying to compete with streaming analysis tools and complex event-processing frameworks. A few of the most common open source products you'll find in this segment are as follows:

- *MemSQL (www.memsql.com)*—This started as an in-memory-only option but now also supports storing data on disk. The main idea is that it is 100% MySQL compatible, but much faster because by default all data is in-memory.
- *VoltDB (www.voltdb.com)*—This is a high-performance SQL-compliant database that offers an in-memory option as well as options for durability, high-availability, and export integrations.
- *Aerospike (www.aerospike.com)*—This is a flash-optimized NoSQL engine with a lot of different features, ranging from geospatial indexing and querying to geographic replication.
- *Apache Geode (http://geode.incubator.apache.org)*—This was originally designed to be in-memory but also offers off-heap storage and a query language called OQL (Object Query Language) that is SQL-like.
- *Couchbase (www.couchbase.com)*—This document database was originally a mix of Memcached and CouchDB but has grown well beyond that. It can be geo replicated and has its own query language called N1QL (pronounced "nickel") that is a SQL extension for JSON data.
- *Apache Ignite (https://ignite.apache.org)*—Billed as an in-memory data fabric, in the end Ignite is an IMDG with the addition of a rich compute grid and SQL support and integrates with Hadoop and Spark. Although a young Apache project, it started life as a product offering from GridGain and was subsequently donated to the Apache Foundation.
- *Hazelcast (https://hazelcast.org)*—This is an IMDG that offers features (at least at the time of this writing) such as queuing, distributed aggregations, Map/Reduce support, and distributed data structures.
- *Inifispan (http://infinispan.org)*—This is an IMDG that also offers features such as integration with Apache Spark and distributed streams and performing complex operations via script.

6.3 Use case exercises

We've covered a lot of ground, going over different aspects of in-memory storage. Given all the product options and the never-ending list of new contenders, it's impossible to discuss all of them. But we're now ready to walk through several use cases and tease apart which product category may suit our needs. Without a doubt your requirements may be drastically different than those for our use cases, but my hope is that you will be able to take the information and apply it to your use case. I know I haven't talked about the data-access side of streaming analysis, so the upcoming uses cases keep the focus on making the data available for being accessed and may make some assumptions about how it is accessed.

6.3.1 In-session personalization

For this use case, imagine that we've built a stream-processing system behind a popular e-commerce website called TheOceana.com, where the product catalog is as vast as the world's oceans. We have a stream of all activity on our site and we want to personalize the site for our users while they're in the midst of shopping. This is often termed *in-session personalization*, because it's occurring during the user's current session. Our goal is to change the page content at request time based on the activity in the user's current session. Imagine a user's session looked like this:

1. Browsed sunglasses
2. Added yellow sunglasses to cart
3. Browsed motorcycle helmets
4. Removed sunglasses from cart
5. Browsed jackets
6. Browsed motorcycle gloves

Now, we want to personalize this experience in the following ways:

- When they go to the motorcycle helmets in step 3, we want to show them helmets that would go with their yellow sunglasses.
- When they start to browse the motorcycle gloves in step 6, we want to show them a coupon for the sunglasses they removed from their cart a couple pages ago.

Figure 6.10 shows this flow along with the personalization actions we want to take.

I'm sure you can think of many other ways to personalize the page, perhaps not based on a single user's session but maybe for all active users, or maybe you can think of ways to combine the current session with the visitor's history. For now, let's focus on the two ways we want to personalize the page content: first by showing related content and then offering a coupon for a product that was removed from their cart. To do this we need to keep track of and analyze the active visitor sessions. In our scenario, a visitor session has a lifespan of 30 minutes and contains each action taken by a user on the site, often called *click-stream data*. Think about how you might solve this problem using each of the in-memory options we've discussed. All right, let's walk through each of our in-memory options and see how we can solve the problem.

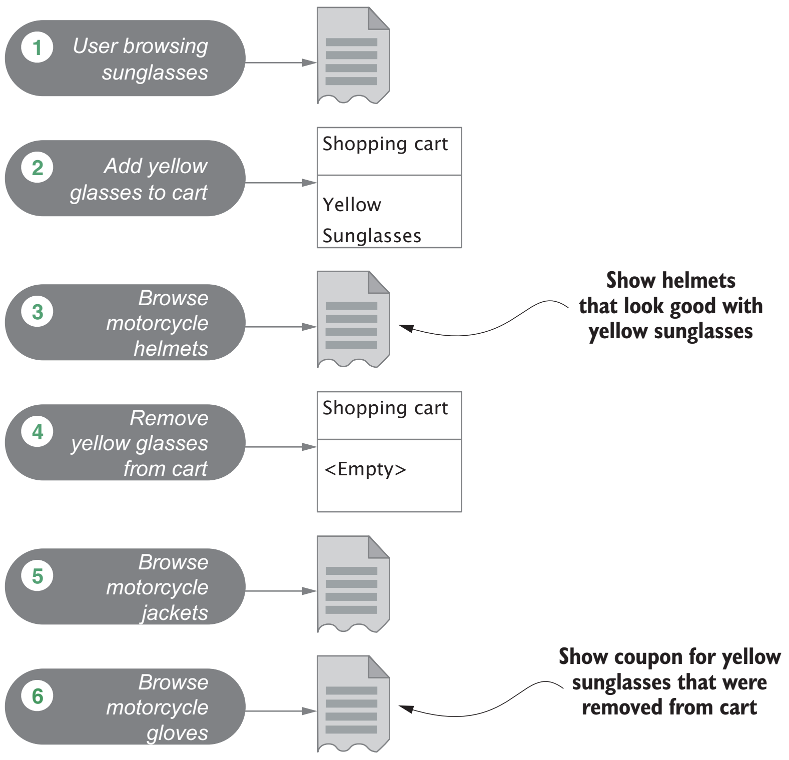
1 User browsing sunglasses
2 Add yellow glasses to cart
Shopping cart
Yellow Sunglasses
3 Browse motorcycle helmets
Show helmets that look good with yellow sunglasses
4 Remove yellow glasses from cart
Shopping cart
<Empty>
5 Browse motorcycle jackets
6 Browse motorcycle gloves
Show coupon for yellow sunglasses that were removed from cart

Figure 6.10 In-session personalization page flow

EMBEDDED IN-MEMORY/FLASH-OPTIMIZED

Storing the data in-memory on the analysis nodes presents some challenges for this use case. Namely, we need to ensure several things to be able to make this work correctly:

- We need to store the entire session for all visitors across all the analysis nodes.
- Our analysis nodes will need to know when a session ends and how to evict the data from memory.
- We need a way for a client to query our analysis nodes to get a visitor's session. Unless we had a way to perform a distributed query, the client would need to know which node has a particular visitor's session.
- We need a way to ensure that when the analysis node storing a visitor's session crashes, we won't lose all the data.

Considering these requirements and/or risks, to use an embedded database we'd need to not only satisfy the requirements, we would also need to mitigate the inherent risks of storing the data in memory. Therefore, it's not a good fit. In fact, if you spend time trying to handle them with any of the leading stream-processing products we've looked at—Apache Storm, Spark, Samza, or Flink—you'll quickly realize that the design is fragile, not scalable, and from an architectural standpoint not clean at all. I would say this type of storage option is not a good fit.

CACHING SYSTEM

Would a caching system be a good fit for this use case? To use a caching system we need to ensure at least the following:

- That we don't lose data if a cache node crashes
- Clients can query the cache cluster using the visitor ID as a key and get back the entire session
- Our analysis nodes can constantly update the session
- The session expires from the cache after 30 minutes

Can we accomplish all of these? Ensuring that we don't lose data when a cache node crashes means we need a caching system that redundantly stores data across nodes. Many products in this category don't support replication, so this may be a risk we can't mitigate. Providing the ability for a client to query the cache using the visitor ID is easy for all caching systems because they operate with a key-value API and will return the data stored at a given key—in this case, the visitor's session. With the continuous updating that needs to be done to this data, we have to consider that we won't only be replacing a value stored with a key—we need to append new data to the value. In that case, how do we handle updating a visitor session as they are browsing our site? We are left with one option because we'll need to use the visitor ID as the key. We will need to constantly read the entire visitor session from the cache, update it, and then write it back to the cache. This is a lot of thrashing that will undoubtedly have a significant impact from a performance standpoint. Perhaps we can do better.

IMDB OR IMDG

What about an IMDB or IMDG? Would it be a good fit for this use case? Consider the same basic requirements we saw for the caching system:

- We need to ensure we don't lose data if a single node crashes.
- We need to provide a way for a client to be able to query the system for the visitor's session using the visitor ID as a key.
- We need a way for our analysis nodes to constantly update the session.
- We need a way for the session to be expired from the cache after 30 minutes.

The first one, ensuring that the data doesn't disappear when a single node crashes, is easy for many of these products because they seamlessly take care of scaling horizontally and ensuring that data is replicated across nodes in the cluster. Querying the cluster using a visitor ID to get back the full session is also pretty straightforward using many of the IMDB or IMDG products on the market. The last requirement—being able to expire a session after 30 minutes—can be accomplished easily with Aerospike, which acts like a NoSQL store that has the ability to add a "time to live" value for a record that is written. With other IMDBs or IMDGs, specifically MemSQL and VoltDB, you need to determine the session expiration outside of the data store and subsequently delete the data. Overall, satisfying this use case using an IMDB is fairly straightforward, and the technology is a good match.

TAKING IT TO THE NEXT LEVEL

Some of our choices may have seemed quite easy. Taking it one step further, would your assessment change if we also wanted to support the following two additional requirements?

- Instead of using a single visitor's session, we wanted to take into consideration all visitors' sessions for that current day.
- Along with the visitor's current session we also wanted to include their entire visitor history when making our decisions.

Next we'll set up another use case, but this time I'll ask you to work through the questions independently.

6.3.2 *Next-generation energy company*

Imagine we're building a next-generation energy company, one that can help avoid brownouts and blackouts that plague certain U.S. states during the summer months. Utility companies have long suffered from this type of Saturday afternoon scenario: it's hot out, and all our customers are running their air conditioners. They also go to the fridge to grab a cold drink, and since they're in the kitchen they decide to wash the dishes. It's so hot outside they keep doing things inside. Perhaps they start a load of laundry and turn on the TV. As you can imagine, all of this spikes the demand on our energy company, and when this demand exceeds our capacity, trouble ensues.

We decide to change this: We're going to build a smart grid and let our consumers participate in the solution. Our first step is to deploy smart meters at our customers' homes. These meters will report the energy use of the home every five minutes, along with information on which appliances are consuming the energy. From all this data now streaming in, we would like to offer our customers variable pricing in real time, based on when and how they run their major appliances. If they run their air conditioner at a slightly warmer temperature or run their dishwasher at night, for example, we would offer them a discount.

To succeed, we need to perform streaming analysis in two places: at the power plant and at the home. First let's consider the features we need for the power plant:

- Ability to analyze the data, taking into consideration weather and historical data, and store it every five minutes.
- Ability to store pricing for the next hour of use (perhaps we decided to offer hourly pricing rates).
- A way to query the data store for the pricing information for a customer.
- A way to ensure we cannot lose customer data.
- A way to ensure this data is available for further analysis.

Take a moment and work through these questions for each of the in-memory data stores we discussed. How would your decisions change if you were to move the processing of this data to the meter? It may seem far-fetched, but a system like this is where energy is headed.[1]

6.4 Summary

In this chapter we looked at different options for storing data in-memory during and after analysis. We didn't delve into the use of disk-based, long-term storage solutions because they are often used out of band of a streaming analysis and don't offer the performance of the in-memory stores.

In this chapter we:

- Learned about the different types of caching approaches
- Compared different types of in-memory data stores
- Discussed how to choose the right one

Some of this may have seemed like we're only seeing half of the picture because part of choosing the correct store also involves taking into consideration how we access data. Chapter 7 focuses on that aspect of accessing and making available the data that we have stored. The key takeaway from this chapter is that many different in-memory options are available, and the idea of keeping your entire working set in-memory is no longer a dream; with modern hardware, software, and the continuous drop in pricing, it is a reality today. When you're ready, go ahead and turn the page and let's get going on accessing the data we've stored.

[1] A good starting point to read more about this is a publication from the U.S. Department of Energy titled *The SmartGrid: An Introduction*, www.smartgrid.gov/files/The_Smart_Grid_Introduction_200804.pdf.