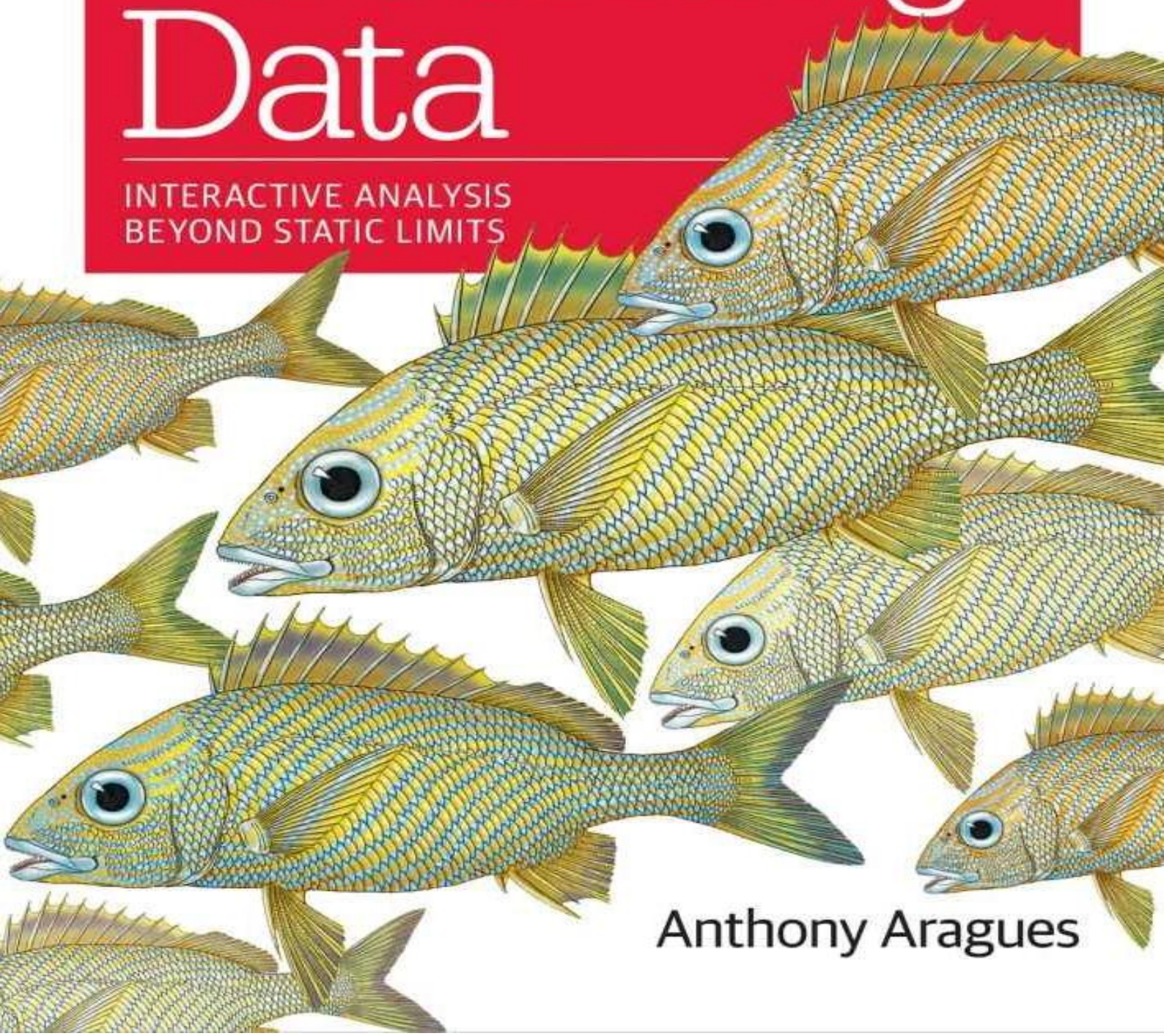
O'REILLY®

Visualizing Streaming Data

INTERACTIVE ANALYSIS
BEYOND STATIC LIMITS



Anthony Aragues

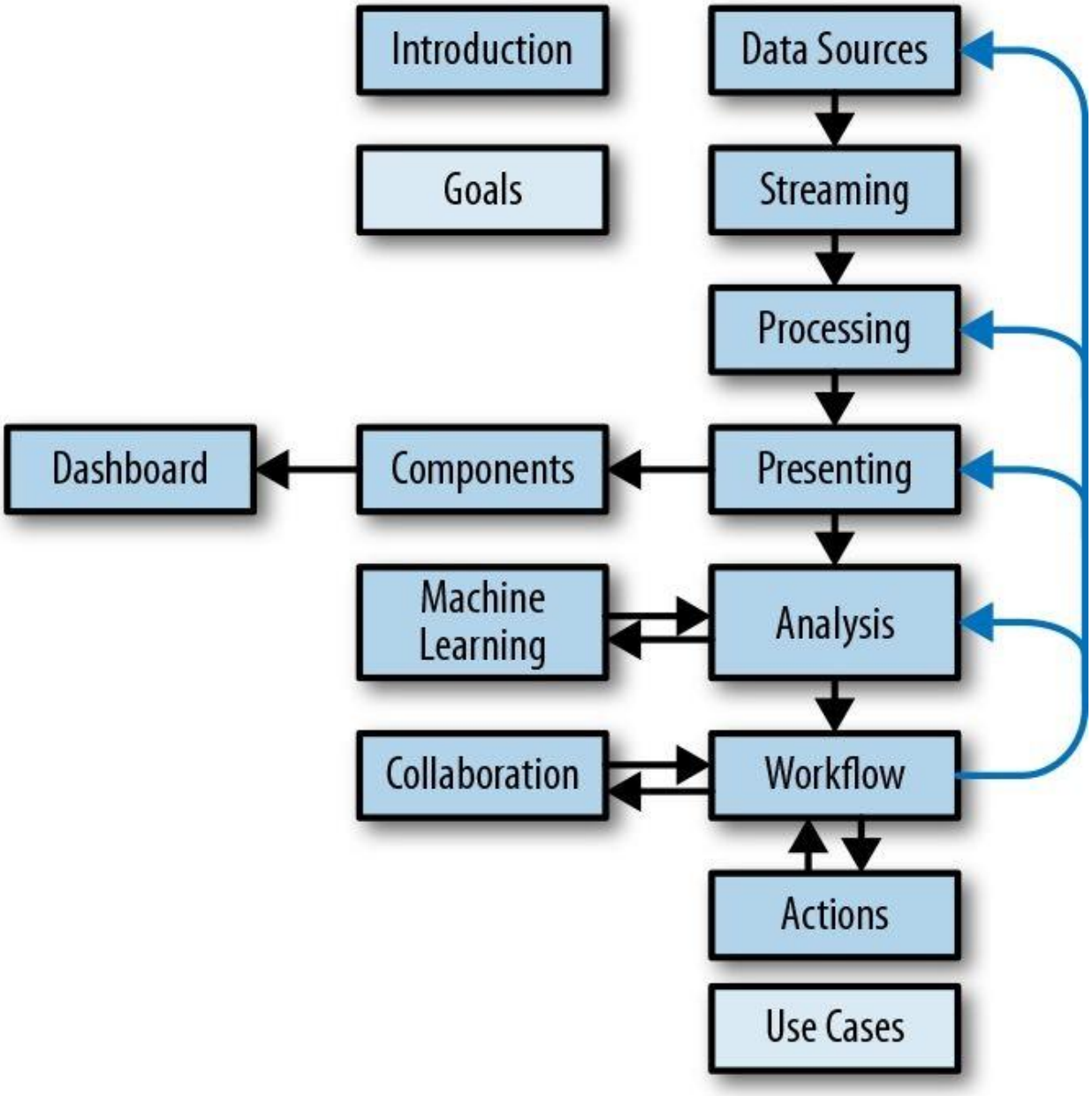


Figure P-2. Data flow of interactive streaming data visualizations

Chapter 1. Introduction

Administrators, analysts, and developers have been watching data fly by on screens for decades. The fast, free, and most common method is to “tail” a log file. `tail` is a standard Unix-like operating system command that allows you to stream all changes to a specified file to the command line. Without any additional options, the logs will display in the console without any filtering or formatting. Despite the overwhelming amount of data scrolling past, it’s still a common practice because the people watching can often catch a glimpse of something significant that is missed by other tools. When filtering and formatting are applied to this simple method, it increases the ease and likelihood of catching significant events that would otherwise be ignored or surfaced only after a significant delay. LNav is an application that represents streaming infromation on a console with some ability to highlight and filter information (see Figure 1-1).

Because of the rate at which information is scrolling by, anything noticed by a human observer with this method will be due to them observing either a pattern or the breaking of a pattern. Statistics, aggregates, groupings, comparisons, and analysis are out of reach for this method at a high data frequency. This method also has a limitation of one log file per command line. In order to progress from this standard of streaming data visualization, this book will explore ways to preserve and build on the effect of noticing something significant in live events. The challenge is how to do this without abstracting the context so far that it becomes another dashboard of statistics that sends the observer back to the tried and true method of command-line scrolling.

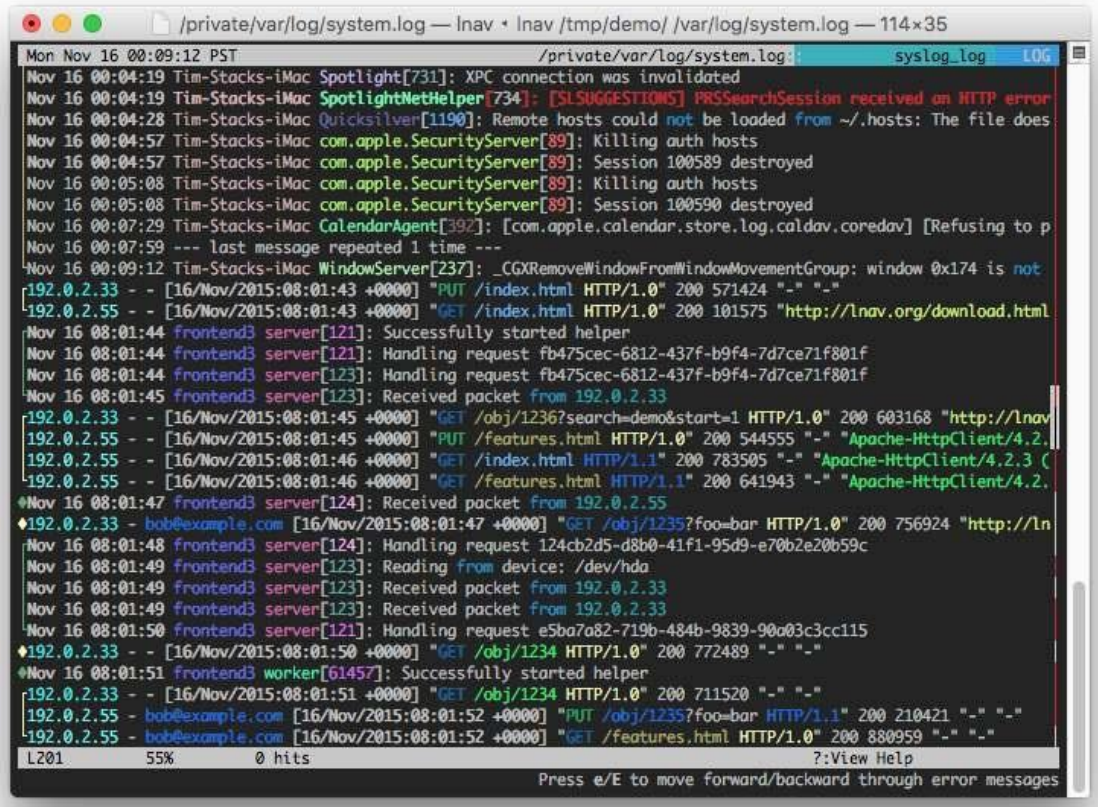


Figure 1-1. A log file viewer (source: http://lnav.org)

A great analogy for visualizing streaming data is visualizing operational intelligence, described in a Netflix database project as follows:

> *Whereas business intelligence is data gathered for analyzing trends over time, operational intelligence provides a picture of what is currently happening within a system.*

Operational intelligence relies heavily on streaming data. The data is usually automatically processed, and alerts are sent when anything goes outside of a defined threshold. Visualizing this information allows people to better understand what's occurring, and whether any automated decisions should be created, deleted, or adjusted.

Why Visualizations

Visualizations certainly can be eye candy, but their value isn't just in attracting eyeballs and mesmerizing people. In general, visualizations can give you a new perspective on data that you simply wouldn't be able to get otherwise. Even at the smaller scale of individual records, a visualization can speed up your ingestion of content by giving you visual cues that you can process much faster than reading the data. Here are a few benefits of adding a visualization layer to your data:

- Improved pattern/anomaly recognition
- Higher data density, allowing you to see a much broader spectrum of data
- Visual cues to understand the data faster and quickly pick out attributes
- Summaries of the data as charted statistics
- Improved ability to conquer assumptions made about the data
- Greater context and understanding of scale, position, and relevance

On top of all that, visualizations also help products sell, get publicity, and screenshot well. Visualizations attract people and entice them to make sense of what they see. They become necessary when trying to understand more complex data such as the automated decisions behind an organization's operational intelligence.

The Standard

The processes and applications that we accept as tried and true were written for a different set of circumstances than we are faced with today. There will continue to be a place for them for the problems they were developed to solve, but they were not designed for the volume, frequency, variance, and context that we are seeing now and that will only increase over time.

There are recent highly scalable solutions for processing and storing this data, but visualizing the data is left behind as we resign ourselves to the idea that humans can't possibly review all of it in time to have an impact. Visualizing the data is required only for *people* to understand it. As processes are developed to deal with this post-human scale, visualizations are falling by the wayside—and along with them our ability to gain immediate insights and make improvements to the applications. The same problem occurs if too many processing steps are hidden from view. Examples of this effect are the inverse of the defined goals of streaming data visualization:

- Missing a significant pattern that can be intuitively found by a person but that would be difficult to predict ahead of time and develop into an application
- Missing something anomalous that would justify an immediate action
- Seeing a security-related event as an alert and out of the surrounding context
- Seeing a threshold pass as an alert, with a limited view of what led to it
- Only preprogrammed understanding of the evolution of the data over time

Data Formats

There are a lot of different formats that raw data can come in. We need to work with whatever format is output and transform it into the format that we need for any downstream processes, such as showing it in a visualization. The first significant attribute of a data format is whether it's humanreadable. Table 1-1 shows examples of formats that are humanreadable, and Table 1-2 shows examples of formats that are not.

Table 1-1. Examples of humanreadable data formats

Format	Description	Example
UTF-8	Unstructured but readable text.	There was a modification to the English Wikipedia page for the Australian TV series *The Voice* from an unknown user at the IP address 82.155.238.44.
CSV	Data is flat (no hierarchy) and consistent. The fields are defined in the first row, and all of the following rows contain values. Fields are delimited by a character such as a comma.	Link,item,country,user,event "https://en.wikipedia.org/w/index.php?diff=742259222&oldid=740584413", "The Voice (Australian TV series)","#en.wikipedia","82.155.238.44","wiki modification"
XML	An early, verbose, and highly versatile format standardized to have a common approach to overcome CSV's limitations.	<xml> <link> https://en.wikipedia.org/w/index.php?diff=742259222&oldid=740584413 </link> <item> The Voice (Australian TV series) </item> <country> #en.wikipedia </country> <user> 82.155.238.44 </user> <event>

		<event> wiki modification </event> </xml>

JSON	A format designed to be more succinct than XML while retaining the advantages over CSV.	{
		“link”:”https://en.wikipedia.org/w/index.php?diff=742259222&oldid=740584413“, “item”:"The Voice (Australian TV series)”, “country”:"#en.wikipedia”, “user”:"82.155.238.44”, “event”:"wiki modification”
		}

Key/value pairs	A commonly used format for an arbitrary set of fields.	Link=”https://en.wikipedia.org/w/index.php?diff=742259222&oldid=740584413“, Item="The Voice (Australian TV series)”, Country=#en.wikipedia”, User="82.155.238.44”, Event="wiki modification”

Table 1-2. Examples of data formats that are not humanreadable

Format	Description	Example
Binary	The conversion of anything to a 0 or 1, or on/off state. This is rarely something necessary to work with for visualizing data.	01111011000010100010001001101100011010010 11...
Hex	Similar to binary, but instead of base 2, it’s base 16. Hexadecimal values use the characters 0–9 and a–f.	7B0A226C696E6B223A2268747470733A2F2F656E2E77696B69
Base64	Similar to hex, but with 64 characters available.	ewoibGluayI6Imh0dHBzOi8vZW4ud2lraXBlZGlhLm9yZy93L2...

Data Visualization Applications

Applications that visualize data can be divided into two categories: those that are created for specific data and those that allow visualizing any data they can attach to. General-purpose data visualization applications will allow you to quickly take the data that you have and start applying it to charts. This is a great way to prototype what useful information you can show and understand the gaps in what might be meaningful. Eventually, a design is chosen to best make decisions from, and a context-specific visualization is created in a purpose-built application.

Another distinction we will make for this book is how the visualization application handles constantly updating data. Options include the following:

- A static visualization that uses the data that is available when the visualization is created. Any new data requires a refresh.
- A real-time visualization that looks like the static one but updates itself constantly.
- A streaming data visualization that shows the flow of data and the impact it has on the statistics.

Assumptions and Setup

This introductory chapter only hints at the variations of data and the processes for manipulating it. A common set of data sources and processes will be established for reference in the rest of this book so that they can be consistently built upon and compared. The data sources are available for free and are live streams (Table 1-3). These are ideal sources to test the ideas put forth in this book. They will also provide a much-needed context focus, which is essential for effectively visualizing data.

Table 1-3. Public test data streams

Data	Description	Storage	Volume
Wikimedia edits	All edits to Wikimedia as a public stream of data	Document store	300/second
Throttled Twitter feed by PubNub	A trickle of the Twitter firehose provided as a public demo by PubNub	Distributed storage	50/second
Bitcoin transactions	Bitcoin transactions with information for tracking and analyzing	Database	20/second

You will need to establish your own standards for formats, storage, and transport so that you have a set of tools that you know work well with each other. Then, when you run into new data that you need to work with, you should transform it from the original format into your standard as early in the workflow as possible so that you can take advantage of your established toolset.

The data format for the rest of the book will be JSON. Even if you are working with another format, JSON is flexible enough to be converted to and from various formats. Its balance between flexibility, verbosity, and use within JavaScript makes it a popular choice.

Node.js will be the primary server technology referenced. Its primary advantage is that it runs on JavaScript and can share libraries with browsers. It also happens to be a great choice for streaming data solutions that are not so large that they require dozens of servers or more.

Angular.js is the main client library used in the book. Both Angular.js and React are common and appropriate choices to show event-based data in the browser.

This combination of components is often referred to as a *MEAN stack* for MongoDB, Express.js, Angular.js, and Node.js. MongoDB is a popular document store, and Express is a web server built on Node. Mongo and Express aren't as essential to the discussion of this book, though we will review storage considerations in more detail. Several other libraries will be mentioned throughout this book as needed that build on this technical stack.

The client components, when mentioned, will be browser-based. A modern browser with at least WebSockets and WebGL is assumed. What these are and why they make sense will be detailed later, but it's a good idea to check that your browser supports them before getting started. You can do this by following these links:

- *http://caniuse.com/#feat=webgl*
- *http://caniuse.com/#feat=websockets*

The Analyst Decision Queue

An *analyst queue* is a place where you present all of the information necessary to make a decision. You then treat the analyst like your most valuable data classifier. Here are a few things to consider to accomplish that:

1. Anything that can wait until later, should.
2. Automate as much as possible before presenting information to an analyst.
3. Present analysts with the decisions that are the most critical and difficult to get right with automation.
4. Attempt to correct any processing and machine learning algorithms you have based on analyst input.
5. Peer-review decisions and allow analysts to collaborate as much as possible.

Each of the steps in Figure 2-9 represents an opportunity for analyst inclusion and would benefit from their involvement.

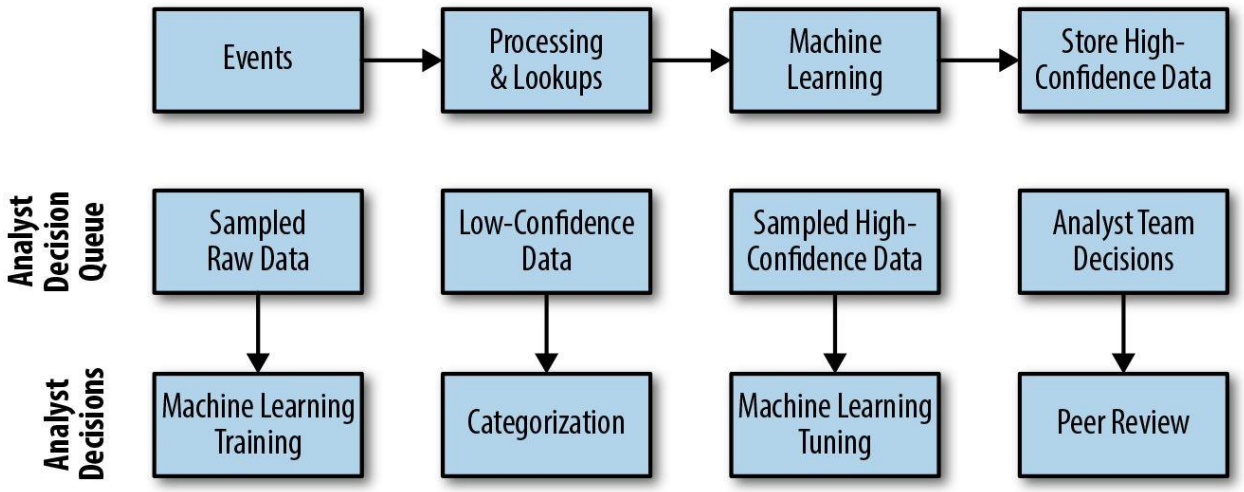


Figure 2-9. Analyst queue workflow

The goal is to use automation where it's best suited, and analysts where it's not. Create a process that doesn't require an army of analysts or trust the automation to do everything, but augments the analyst's role with intelligence in a

complementary fashion. We will discuss this more in Chapter 9, *Streaming Analysis*.

Data Pipeline Visualization

When you have a complex processing pipeline, it helps to know how the processes connect and how many are at each step. This can help you get a sense of where to scale bottlenecks, where items might be dropping, and what the overall health of the system is. Figure 2-10 shows a possible solution for visualizing a processing workflow as data moves across components of a system.

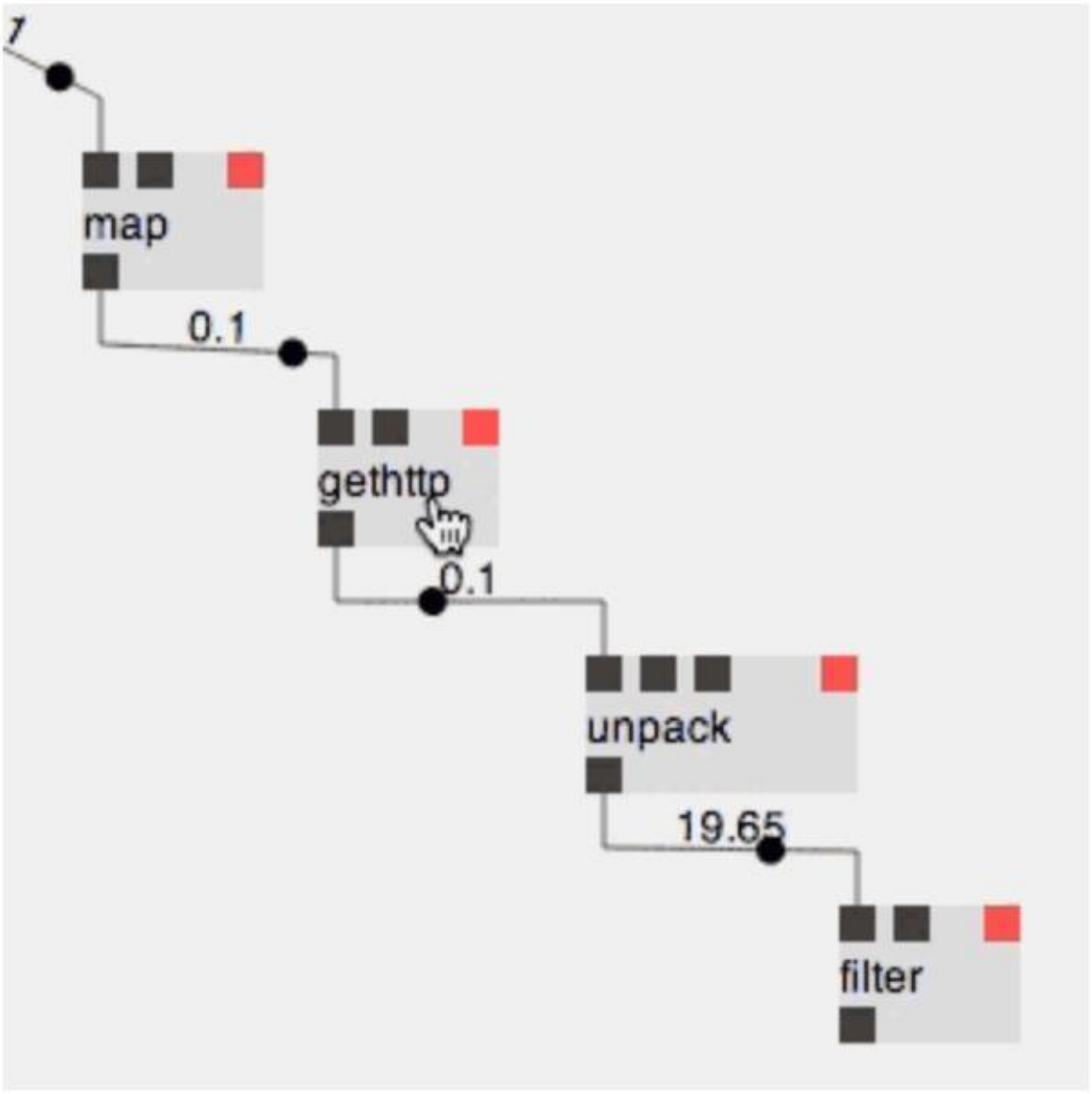


Figure 2-10. A streaming data presentation of a processing workflow (source: http://nytlabs.com/streamtools/)

Data Source Types

Any data can be streamed, but some types lend themselves better to downstream actions like visualizing than others. It's a good idea to know what you are working with prior to diving in. The easiest data to work with as a stream is atomic and structured. *Atomic* means that it has a clear beginning and end. We will refer to atomic data as "messages." *Structured* data is parsed into various fields and values with a consistent schema. A schema defines the structure for the data: what fields exist, what data they contain, and if there is any hierarchy or relationships. Structured data allows for the most mapping of visual elements to values.

The minimal structure that is applied to a message is an action. Actions or events can be categorized when nothing else is structured because the action can be appended at the time of creation based on its source. For example, the errors from a web server can be streamed as "error" actions. Even adding this minor level of structure to the data makes a big difference. Categorized actions allow a division of channels and simple statistics per action. We will define a message that has a categorized action as an "event." When most of the message is structured, we will call it a "record." Records are the ideal when processing and visualizing data. We will discuss how to get more records to work with out of messages later. Table 3-1 shows the difference between which data source types you choose to use. There is a tradeoff between volume and how machine-readable it is.

Table 3-1. Data source comparison table

	Categorized actions	Structured data
Message		
Event	**X**	
Record	**X**	**X**

What to Stream

Once you've identified the data you need to stream, you'll have a few options for how to stream it into the service you have chosen. Unless you already have data streaming from the source you need, you will need to build that bridge. The major areas we will consider are attaching to existing logs, event-based monitoring built into applications, and polling APIs and databases.

Each one has different challenges and advantages. In most environments, you can choose one of many locations to get to the same information. The more consistent you are in your approach to getting the data, the better it will work as an overall system. Reuse as many tools as possible, and try to pick the data collection method that is the best fit and most consistent for your solution.

Logs are the most common, but their formats will vary and they will require inline parsing. Tailing log files is the easiest way to create a new data stream to work with. Existing monitoring systems will already have a streaming data architecture that you may be able to tap into. If they have an open API, this can allow you to take advantage of already centralized, normalized, and categorized data. Similarly, if your systems already have an API for the information that you need to stream, you can poll the API for updates. This is not technically difficult if the API is able to handle the increased frequency of calls. Finally, if you have an old or simple data system that outputs only a CSV file of data, you can stream through the results after the fact. Table 3-2 compares efforts of streaming data methods

Table 3-2. Streaming data methods

	Difficulty	Online	Resource intensity	Data structure
Logs	Easiest	Disconnected		Events
Monitoring	Hardest	Connected	Lowest impact	Depends on development effort
Polling	Abstracted		Highest impact	Structured records
CSV	Easy	Disconnected	Low impact	Flat records

Data Storage Considerations

The data must be stored at various stages in its life cycle. How it's stored and where has a big impact on what is possible to do with it.

Table 3-3 compares some common storage techniques. Complex environments will have several of these implemented at once. In addition to these considerations for choosing data persistence methods, you should consider what is already being used in your environment that you can leverage.

Table 3-3. Considerations for common data persistence methods

Storage	Examples	Strengths	Weaknesses	Use cases
Files	Text, JSON, CSV	Portable, compressible	Not indexed for fast searching Manual	Raw data Data transport between systems
Database	SQL	Indexed, structured relationships Fast aggregates	Extremely structured Difficult to scale beyond the limitations of a single server	Relational data Consistent structure
Document store	MongoDB Elasticsearch NoSQL	Flexible Moderately scalable	Not relational Slow aggregates	Inconsistent data Atomic more than relational
Distributed storage	HBase Cassandra	Highly scalable	Complex maintenance Inefficient	Beyond the scale of other systems

Files are perfect for long-term storage and moving data between systems. When a file is imported into a system, it will probably need some level of parsing and processing (see Chapter 5) in order to be used on the new system. Some systems, such as Hadoop, will allow you to keep a collection of files and work with them if they are in a certain format. It is possible to stream a file by sending a certain number of records at an interval (see "Buffering"). This can be useful when a workflow needs to occur for each record and there is an advantage to watching it occur instead of seeing the results of a large batch.

Relational databases are the most common method of storing data with indexes on fields that need to be referenced quickly. You can stream results out of a database using a buffer, but the most benefit can be seen from streaming the

binlog of the database. The *binlog* is a record of changes, used primarily for replication. By streaming the binlog, you are able to see information as it occurs without needing to poll the database through an application or API. Streaming the binlog into a pipeline and visualizing it brings new capabilities to an established technology. You can also store results from analyst decisions back into a database.

A *document store* requires no schema to be used. A mixed set of messages with varying data and structure can be stored here. This can be a good place to cache unpredictable data before being able to process it or stream it to a client.

Distributed storage is an ambiguous term, but here it represents large systems built on top of something like Hadoop that are beyond the scale of most other systems. These highly redundant and scalable platforms have their own streaming technologies that they work well with, such as Kafka. The larger the scale of the streaming data, the more consideration needs to be given as to which records are most worth displaying to an analyst and when.

Managing Multiple Sources

Your data streams and sources won't often map to your use cases. The two major considerations are the number of sources and schemas. As we saw, a schema defines the structure of the data. Applications usually require specific schemas to handle their functions and output a specific schema. When you run the same application in multiple instances, you'll have multiple sources with the same schema. When you have different applications, you'll have different schemas. This ends up being very important when processing the data. Here are scenarios to consider when you have multiple sources:

- Many to one, same schema
- Many to one, different schemas
- Many to many, different schemas

Figure 3-1 shows what the different mappings might look like for one or multiple data sources.

If you are taking advantage of streaming data in multiple applications or from different sources, you need to give some thought to how you will divide or combine the streams in such a way that they can be effectively monitored and visualized. There are a few things to consider:

Goals

: How does combining or splitting streams best suit your goals?

Schema variation

: Are the schemas the same?

Events

: Are there ways of dividing events that are more significant than by their source?

Volume

: Is the volume high enough to warrant splitting, or low enough that combining makes sense?

combining makes sense?

Examples 3-1 through 3-3 show the various methods that can be used to organize the data streams in all of the combinations shown in Figure 3-1.

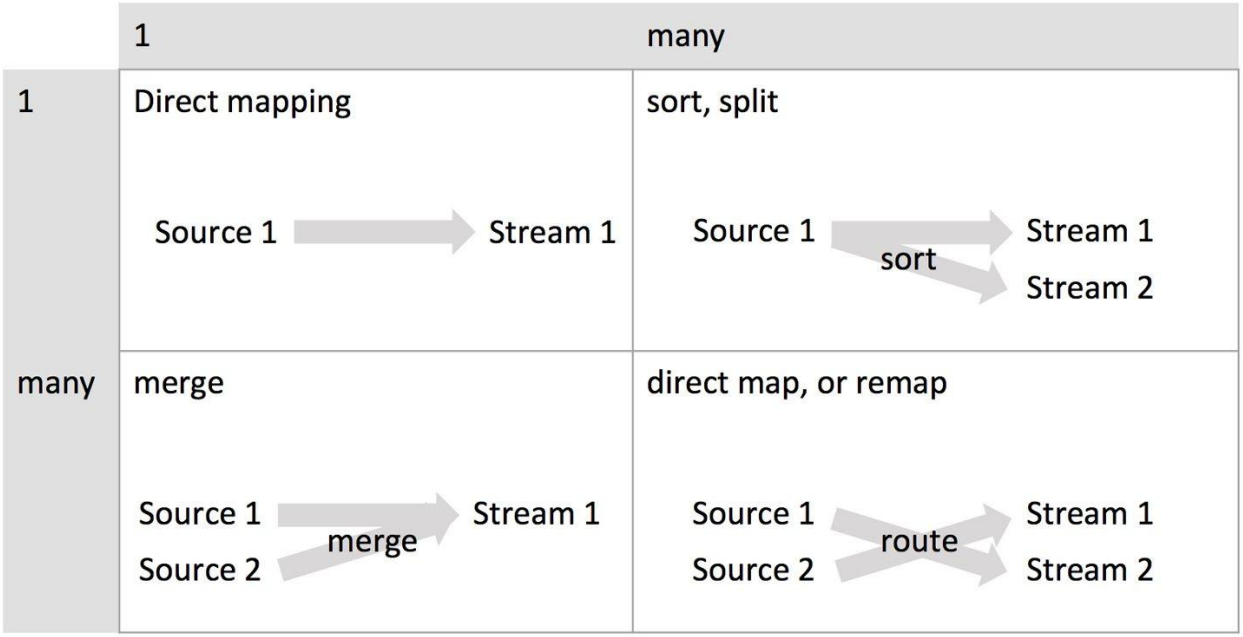


Figure 3-1. Source-to-stream mapping

Example 3-1. Split stream, simple round-robin

```
// set stream count
var intStreams=4;
// current stream
var intCurrentStream=1;
var fnOnMessage=function(objMsg){
  // assuming this function will send to the stream specified
  fnSendMessage(intCurrentStream,objMsg);
  // update the stream pointer
  if(intCurrentStream === intStreams){
    // reset
    intCurrentStream=1;
  }else{
    //Increment
    intCurrentStream++;
  }
}
```

Example 3-2. Stream sort/map

```
var objMap={
  "production":1,
  "development":2,
  "staging":3,
```

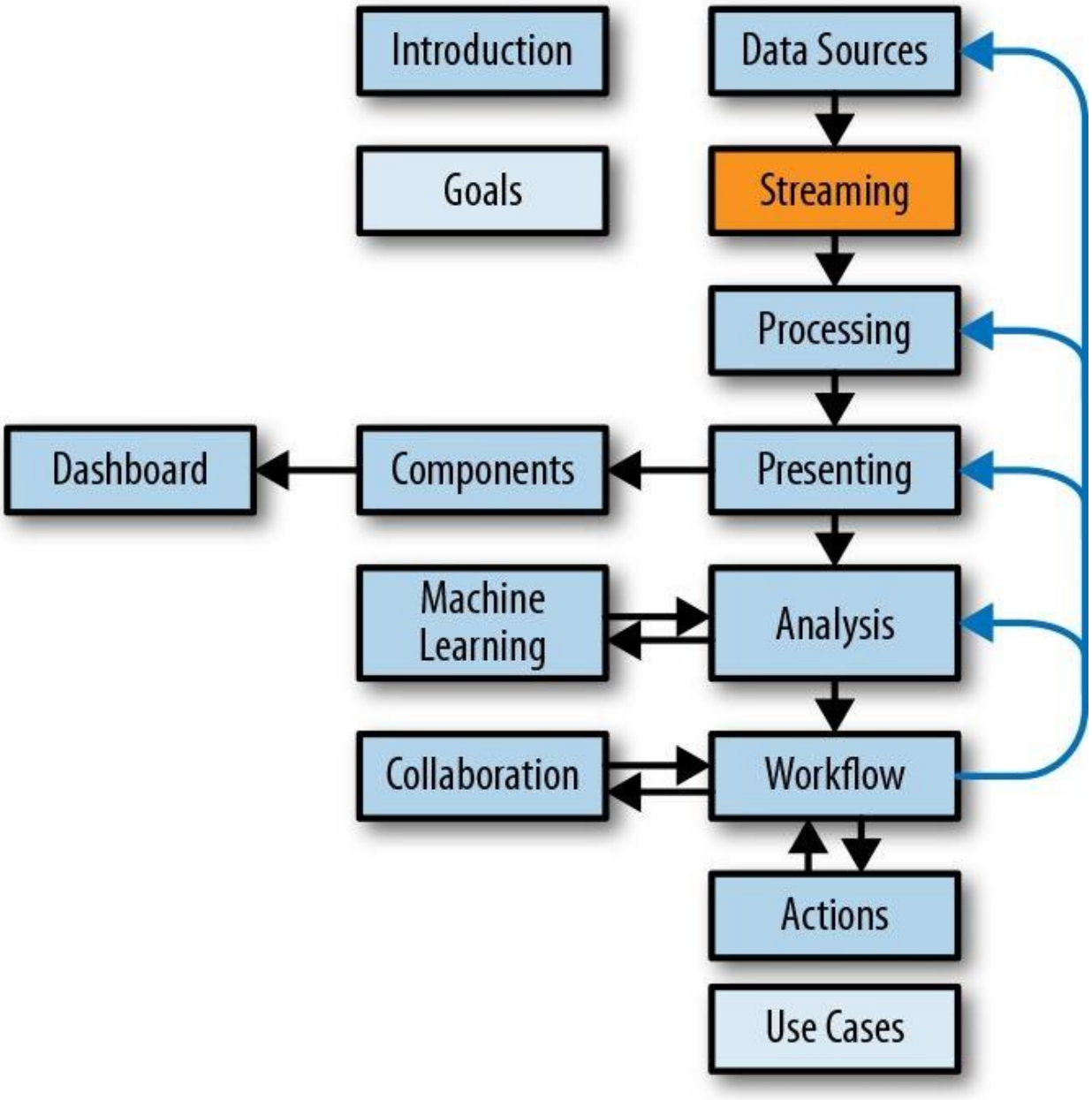
```
  "certification":4
};
var fnOnMessage(objMsg){
  // assuming this function will send to the stream specified
  // map the environment data by the map
  fnSendMessage(objMap[objMsg.environment],objMsg);
}
```

Example 3-3. Sort by schema

```
// properties to look for in an object
var objMap={
  "ip":1,
  "url":2,
  "sha256":3,
  "regex":4
};
var arrFields=Object.keys(objMap);
var fnOnMessage(objMsg){
  var fFound=false;
  var intStream=0;
  for(var i=0;i<arrFields.length;i++){
    if(fFound===false &&
      typeof objMsg[arrFields[i]] !== 'undefined'){
      intStream=objMap[arrFields[i]];
      // stop on first one found
      fFound=true;
    }
  }
  //assuming this function will send to the stream specified
  fnSendMessage(intStream,objMsg);
}
```

Any data that is collected and useful to you can be streamed. Don’t be limited by what’s officially streamed. Find what you need, and then in the next chapter we’ll talk about ways to attach to it.

Chapter 4. Streaming Your Data



This chapter covers some specifics of how to stream data that you have identified. There are always more ways to stream data, but this covers the most common methods currently in use.

There are two important aspects to consider with any streaming data endeavor: publication and subscription. Both must be in place to effectively stream data for a visualization. Regardless of your data source, you will need to publish the data into a stream. Streams of data are often divided into channels and sometimes further divided into events. A service needs to exist to publish to, and a client needs to be connected to that service to publish events into it. This is an important detail to consider when evaluating data streaming services. First you need to get your data into the service; then you need to subscribe to a data stream. The exact method you use for this will depend on the service you choose. You will then receive the stream of data that is published, filtered by channels and events in some cases. This is the data you will be able to visualize.

Many publishers and many subscribers can be working within the same data stream. This is one of the advantages of streaming data: it lends itself well to a distributed architecture. You could have a large collection of servers publish their errors into a data stream and have multiple subscribers to that data stream for different purposes. Subscribers could include an operations dashboard, event-based fixit scripts, alert generators, and interactive analysis tools. It is also possible to have the service and publishing components combined. This can be easy to deploy but limits options on combining information from multiple sources. `websocketd` is a good example of a service combined with a publisher that is easy to set up and creates a single publisher to a data stream.

Key things to consider in choosing a service for streaming data are cost, scalability, maintenance, security, and any special requirements you have, like programming language support. Table 4-1 lists some popular examples.

Table 4-1. Example services

Service	Licensing	Notes
Socket.IO	Open source	A popular and easy standard for building your own streaming.
SocketCluster	Open source	A distributed version of Socket.IO for scalability.
`websocketd`	Open	Utility to stream anything from a console.

[illegible]	source	[illegible]
WebSockets	Standard	A standard supported by most browsers. The current baseline for all streaming services.
Pusher	Commercial	A complete streaming server solution. Paid service.
PubNub	Commercial	A complete streaming server solution. Paid service.

How to Stream Data

A few scenarios can cover most needs in streaming data. We'll look at each of these in turn:

- Using a publish/subscriber channel or message queue
- Streaming file contents
- Emitting messages from something that is already streaming
- Streaming from the console
- Polling a service or API

A publish/subscriber channel is a mechanism for publishing events into a channel to be consumed by all those who subscribe to it. This is a pattern intended for streaming data and is also known as *pub/sub*. Pub/sub libraries will handle all of the connections for you, but you may need to buffer the data yourself.

Streaming file contents may sound odd because they're static and can be run through a program to analyze at will or in a batch. But you may still want to do this if your file is the beginning of a workflow of multiple components that you want to monitor as it occurs. For example, if you have a long list of URLs in a file to run against several APIs and services, it may take a while to get the results back. When you do, it would be a shame to find out that an error had occurred or you'd gathered the wrong information, or they'd all ended up redirecting to the same spot. These are things that would stand out if you were streaming the lines of the file to a service and seeing the results as they returned.

To stream the contents of a file, you can either have a service hold the file open and stream the contents at an interval per record or dump the contents of the file into a service and allow it to keep it in memory while it emits the records at an interval. If the file is below a record count that can comfortably fit into memory —maybe less than 1 GB of records—then the second method is preferred. If the file is much larger than you want to store in memory, keep it open and stream at the interval you need. If you are able to work with the file in memory, you have

the added advantage of being able to do some sorts on the data before sending. Papa Parse is an easy-to-implement library that you can use to stream your files. It allows you to attach each line to an event so that you can stream it: `Papa.parse(bigFile, { worker: true, step: function(results) { YourStreamingEventFunction(); } });`

Many applications stream data between components. Many of them have a convenient spot to tap into and emit events as they occur. For example, you can stream a MySQL binlog that's used for replication by using an app that watches the binlog as if it were a replication server and emits the information as JSON. One such library for MySQL and Node.js is ZongJi: `var zongji = new ZongJi({ /* ... MySQL Connection Settings ... */ }); // Each change to the replication log results in an event zongji.on('binlog', function(objMsg) { fnOnMessage(objMsg); })`

Similar libraries can be found for other data storage technologies and anything that has multiple components.

Streaming from a console gets back to the original streaming data example of watching tailed logs scroll by. By turning a scrolling console into a streaming data source, you can add a lot of processing and interaction to the data instead of just watching it fly by. `websocketd` is a convenient little open source application that will allow you to turn anything emitted in a console into a WebSocket with the lines sent to the console as messages. These will need to be parsed into some sort of structure to be useful. You can create multiple instances that run in the background or as services. This can be very effective to get a good idea of what's going on and what's useful, but may not be a long-term solution.

To start `websocketd` and create a stream, use a shell command like the following: websocketd --port=8080 ./script-with-output.sh

Then connect to the `websocketd` stream in JavaScript: `// setup WebSocket with callbacks var ws = new WebSocket('ws://localhost:8080/'); ws.onmessage = function(objMsg) { fnOnMessage(objMsg); };`

Finally, polling an API is an attractive option because APIs are everywhere. They are the most prolific method of data exchange between systems. There are several types of APIs, but most of them will return either a list of results or

details on a record returned in a previous response. Both of these work well in an interactive streaming data visualization client. The API polling workflow is as follows:

1. Call the API for a list.
2. Stream the list to the client at a reasonable interval.
3. Run any processing and filters that fit your goals.
4. For any records that pass the conditions to get details, make the details API call.
5. Show the results of the details API call as they return.
6. When you reach the end of the list, make another API call for the list.
7. Stream only the new items from the API call.
8. Repeat.

Using one of the options described here, you can stream just about anything you would like to and get a live view of what is transpiring.

Buffering

If we suppose that an analyst can understand up to 20 records per second while intently watching the screen, it may not make sense to fly past that limit when data is coming in faster than that. *Buffering* allows data to be cached before going to the next step; it means you can intelligently stream records to a downstream process instead of being at the mercy of the speed of the incoming data. This makes a big difference in the following instances:

- When the data is streaming in much faster than it can be consumed
- When the data source is unpredictable, and the data can come in large bursts
- When the data comes in all at once but needs to be streamed out

Cross these scenarios with the likely data needs of an analyst, and you start to have some logic that maintains the position of the analyst's view within the buffer as well as the buffer itself. The analyst's views in the buffer are

- Newest data (data as it arrives)
- Oldest data (data about to leave the buffer)
- Samples of data (random selections from the buffer)

Any of these buffer views can be mixed with conditions on what records are worth viewing to filter out anything unneeded for the current task. To meet all of these conditions, the following rules can be applied in your buffering logic:

- Put all incoming data into a large cache.
- If the data is larger than the cache, decide what to do with it. You can
    1. Replace the entire contents of the cache with the new data.
    2. Accept a portion of the new data if you have not seen enough of the previous data.
    3. Grow the cache.

- Set the viewable items within the cache to correspond to the buffer view required by the analyst.
- When new data is added at a rate that can be easily consumed, move it into the cache and the views where appropriate.
- When data is added too quickly to be consumed intelligently, trickle it in at a pace that is suitable.
- When data is not arriving at all, start streaming records that haven't been seen yet.

Figure 4-1 shows the relationship between a changing cache and moving views within it.

Figure 4-1. Buffer views and movement in relation to a cache for streaming data

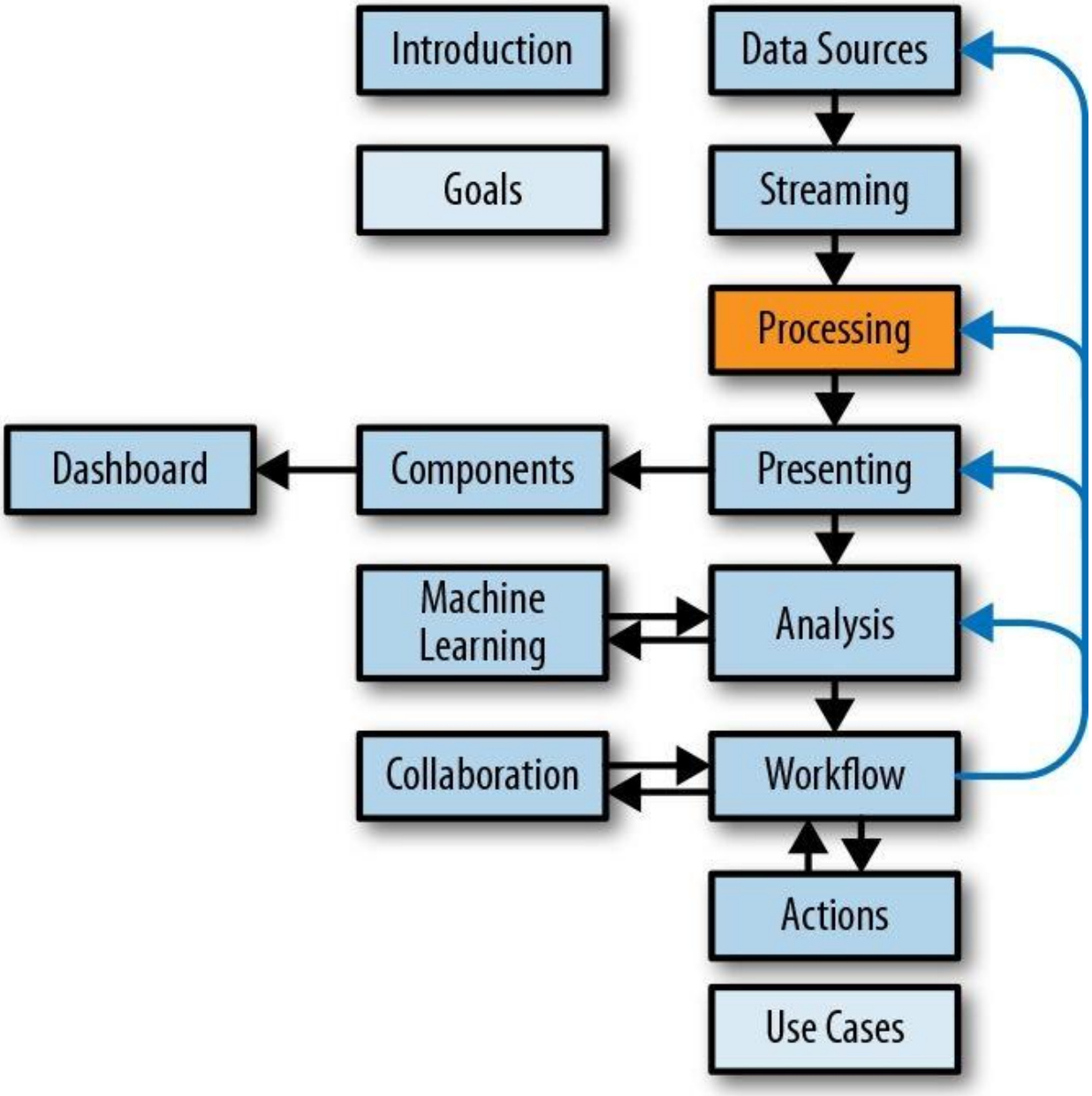
Streaming Best Practices

These are guidelines to consider implementing in your data streaming strategy:

- Stream data as something that is fire-and-forget, not as blocking for any process or as a reliable record.
- Use a protocol that encrypts the data in transport for anything sensitive. Without an encrypted transport such as WSS, a simple packet sniffer can see the data in plain text passively.
- Create multiple channels or queues to organize streams of data. Each channel can be assigned a different purpose and set of processes.
- Don’t mix data schemas in the same channel. This makes it much easier to process anything within that channel.
- Separate analyst feedback events into a separate channel. This helps with the processes of team collaboration and peer review, and can also help prevent feedback loops.
- Consider a distributed and coordinated service if your scale goes beyond what a single server can handle. In some instances, having multiple clients connected and getting different results at different times causes confusion (and worse).
- Use technologies that abstract the specific streaming protocol and allow multiple protocols to be used easily and interchangeably.

Going from not streaming data to streaming data might be the largest obstacle in producing a working solution that you can show to others to get feedback. It will require some access to data or systems that are not typical. If you can get a plug-in or official feature to stream what you need, that’s always your best bet. If not, you can still connect at the points mentioned here.

Chapter 5. Processing Streaming Data for Visualization



Processing data is the most common operation mentioned in this book. There are specific considerations to bear in mind when processing streaming data to be

visualized.

Batch Processing

Batch processing is the most common approach for handling high volumes of data. The process of batching means that data will be cached somewhere to be processed at intervals. The processing interval is chosen according to the data's significance and the ability to take actions on it. Processing daily batches overnight is by far the most common approach, but daily batch processing falls short when there are significant events that may have occurred almost 24 hours earlier by the time the report is reviewed by a person. An indicator that your brand has been mimicked publicly for malicious purposes would be an instance where every minute counts. In order to deal with this, hourly batch processing is often used. Most applications will not process batches more often than hourly because of perceived limitations in being able to act on the data any faster. Another reason for not processing batches too often is that it's a complex process and has the potential to not finish before processing of the next batch begins, causing a backlog.

The process that runs at the chosen interval will query the data from where it's stored in order to create the aggregate statistics predetermined to be significant. These statistics will be saved for later visualization in an application. The application that shows the aggregate data in a dashboard is usually the first view someone has of the data. The high-level view is used to determine what is worth drilling down into and seeing in more detail. When record details are requested, they are retrieved to be shown in addition to the original information.

Inline Processing

Inline processing is less common than batch processing because its immediate event-based approach is more complex and it's susceptible to issues during spikes of activity that are beyond its capability to keep up with. With this method, each event will kick off a chain of decisions and processing before storing a result or displaying it to a person. This approach lends itself well to visualizing streaming data because it shows the required information rapidly and can at least show the amount of data at a frequency that a person can understand.

Any process that is complex or mature enough will require both inline and batch processing models. Visualizing streaming data effectively requires both approaches to complement each other. Microbatches are required to keep up with the data volume, while inline processing will get secondary information on a selective set of filtered, throttled, and vetted results of the microbatches.

Processing Patterns

Many of the frameworks for streaming data mentioned in this book offer an easy way to process the data inline. You may need to do something more custom to achieve what you are after, though. If you are streaming data for the purpose of analytics, the more you can process inline and show to the analyst to help them understand a pattern or make a decision, the more effective their analysis will be. There are a few principles and patterns to keep in mind when programming for streaming data:

- Use microservices for calling information inline.
- Assume information returned will be asynchronous and nonblocking.
- Assume everything is a batch of data, with batch sizes of whatever occurs in ~1–60 seconds.
- Not all processes need to occur for all events.
- Stream-processing ecosystems can be heterogeneous.
- Separate data dispatch from data processing for scalability.
- Caching can help reduce redundant processing.
- Store the original event before inline processing for visualization.
- Route events to their respective processing pipelines as early as possible.
- Processing flows can be cyclical.
- Each step and component can be displayed if needed.
- Consider using the pub/sub pattern.

In a microservices architecture, functions are broken into small atomic components that can be called separately. Each one is ignorant of the larger environment and can run independently. This approach is a foundation component for a complex, flexible ecosystem that is highly redundant and scalable without being difficult to monitor and maintain.

For example, if you have a set of URLs that you want to

For example, if you have a stream of URLs that you need to process, you may need to do some processing, parsing, and gathering of statistics. All of this can be done in separate components. You may also need to crawl each URL, grab some screenshots, and look up some third-party information about the history, reputation, and risk of visiting the site. After gathering all this information, you might want to visualize it quickly in order to make a decision about each URL. If you were to perform all of these tasks inline, you would have a process that takes the maximum amount of time because nothing would be done in parallel. If something failed, everything that had successfully completed would be lost (unless cached). In a single all-encompassing process, you also have a mix of resource requirements and have difficulty knowing if any components are down until they fail.

The microservices architecture is commonly used to avoid these issues, and is used today in many processes that don't involve streaming data as well. Breaking out each component into something that can be called independently makes it easier to constantly monitor each of them. You can also call them from multiple events or several times in the same event without needing to repeat the code. You can scale the components separately and with different resources. In our example, the screenshots and crawling may take a while to return, while the other components could return very quickly. With the microservices architecture, you can have several screenshot and crawl components in a pool ready to be called, but only enough of the other components to be redundant. You can call all of these components at the same time and wait for all of them to return to be displayed, or show information as it's available. Whether you wait to display everything at once or show results as they're completed should depend on how much is required by an interactive analyst to make a decision.

Figure 5-1 illustrates an example microservices architecture, showing independent scaling of services, monitoring, and streaming through modularity and distribution.

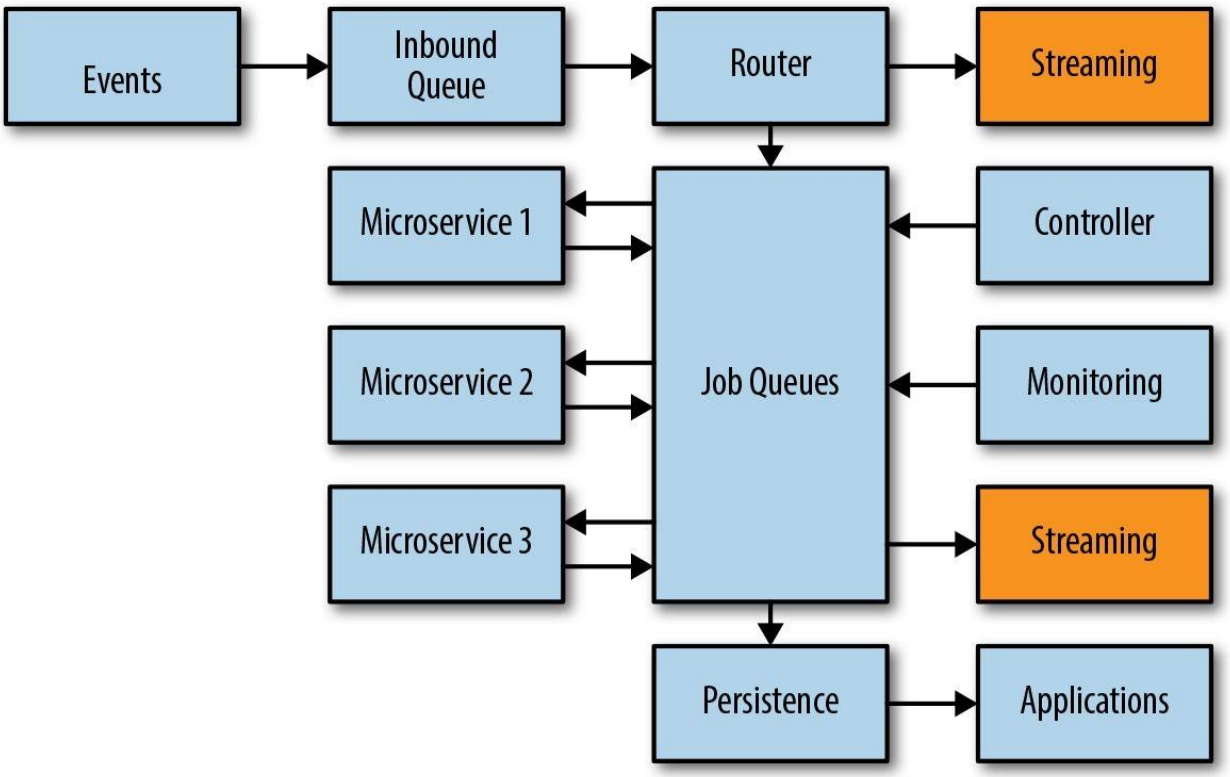


Figure 5-1. Example microservices architecture

Streaming data is often assumed to be completely event-based. This is true up to a certain volume and scale, but once the number of events to be processed reaches a certain level, or if a significant overhead can be saved by processing multiple events at once, there is a need to create small batches defined by practical limits. Example 5-1 shows how this can be done.

Example 5-1. Processing in batches of one second

```
var arrBatch=[];
var intTimeStart=Date.now();
var intLimit=1000;
var fnOnMessage=function(objMsg){
  var intNow=Date.now();
  arrBatch.unshift(objMsg);
  if(intNow>intTimeStart+intLimit){
    intTimeStart=intNow;
    fnProcessBatch();
  }
}
var fnProcessBatch=function(){
  for(var i=0;i<arrBatch.length;i++){
    // process each record
  }
```

```
}
```

For visualizing streaming data, the practical limits are somewhere between one second and one minute per batch—this is the approximate amount of time that someone watching the information needs to understand and get any intuitive insight from it. People can definitely make subsecond decisions, but when there are dozens of records per second or more, it's a pretty safe assumption that this is beyond the upper limit of human speed. Find a batch size/time that works best for your goals.

Even with a nice scalable microservices architecture, it's easy to have more events than you can or need to process occurring across all of your components. In our URLs example, if you do the fast lookup for history and other third-party data, you may be able to make a determination without a screenshot or crawling. And you may be able to bypass a decision altogether based on some data or the time elapsed since a previous related decision. Anything you can do to quickly account for events and then filter them out will gain efficiency in both processing and analyst time.

When you are gathering your components to be used in stream processing, they need to be compatible only to the point that their inputs and outputs can be easily normalized to work with each other. The language they are written in, the data store behind them, licensing, and many of the other obstacles in software development can be avoided by using the components as modules.

In order to coordinate all of the processing that needs to occur for all of the events and what to do when they return, you'll need a controller of some sort. It may be necessary to create a separate service for dispatching. The dispatcher will see all events and act as a load balancer between controllers. A lot of common libraries can handle this capability at scale, including message queue systems, like RabbitMQ and ZeroMQ.

In most data processes, there is a lot of redundancy. In the URL example, you may get a lot of the same domains with some slight URL variations, if you aren't seeing complete duplicates. If each of the components is able to keep some cache of its responses, they can return results quickly and move on to the next thing that needs to be processed. If you chart the unique versus duplicate requests to each component over a month, you will see a clear point of diminishing returns in terms of caching. This exercise is recommended for the

most commonly called components of the highest resource intensity. Common caching options include a fast database or an in-memory key/value store like Memcached. Checking the cache will add some overhead to your components' processing time, and for this reason it's recommended to not have a larger cache than can be quickly searched in memory or at memory speeds. It's essential to ensure that all cache mechanisms have an intentional method to prevent them from expanding beyond their useful size and lifetime.

There is most likely some redundancy between the streaming data processing you need to do and the normal batch processing that you'd need to do for any longer-term reporting. This can create the temptation to wait until all the information is gathered and store it later. The events may not reoccur, though, so it's important to record them however you need to first—even if it just means accounting for some benign and common event before throwing it out. Any of the inline processing components that are called can add data to later be referenced by any longer-term process that occurs.

If you have multiple processing flows (a set of components to be used in a specific sequence), the most efficient thing to do is to separate them as soon as possible. If possible, separate them at an architecture level: have different events point to different dispatching queues. If a single data source creates multiple types of events that require processing, you will need to include that logic after they are all in the queue.

Some components or processes can also be event sources. In the URLs example, after crawling and discovering referenced or related URLs, those may also be submitted into the queue to be processed. This is another time that a cache is important. If the crawler keeps some basic data on what it has processed recently, it can prevent an infinite loop.

Even if a number of processes are required before an analyst can make a decision, there are a few reasons you may want to present information in the middle of the processing flow. It can offer insight into what's being done with the data, and it can let you see gaps or errors in the processing logic. It can also be significant for knowing when components need to scale or be tuned based on their frequency of use and time to return results.

Lookups

Streaming data shouldn't exist in isolation. Certain events will justify a data lookup from another source, and sometimes it's even worth creating your own lookup source, such as a screenshot of a website. Once you have processed, categorized, and prioritized the data, you should be able to use those results to determine what to investigate next. Any time you require more information to make a timely decision, making a secondary lookup for that information may be worthwhile.

This is especially useful when you need to display something to an analyst to enable them to make an intuitive call. A good example of this is phishing scams. *Phishing* is a term used for a type of attack where someone does whatever they can to convince you to click a malicious link. The attacker will typically impersonate a legitimate service such as a bank or online service and give you a compelling reason to pay attention. Many characteristics of phishing can be automatically determined, such as use of a domain name that is visually similar to but not exactly the same as that of a popular service. For example, *Bank0fAmerica.com* looks close enough to *BankOfAmerica.com* to not stand out at first glance (the O is replaced with a zero); if there is a domain like this available, an attacker can register it to deceive people. If you were streaming data for the purpose of sifting through potentially malicious links, something like this should get a very high priority.

It can be hard to make a decision by looking at the link alone, though. The next step would be to go to the domain to take a look, but this is not always possible or advisable when looking at potentially malicious domains, and it can take a lot of accumulated time and effort when you're looking at hundreds per day. This is where a data lookup comes in handy. You can set up a safe system to get a screenshot automatically when the automatically collected data justifies it, and display it to an analyst for the intuitive call. Whatever decision the analyst comes to can be saved with the data and used in further processing or training of supervised machine learning models. This workflow of processing and doing lookups may go through several iterations before anything is displayed to an analyst, but the concept remains the same.

Lookup Types

There are three types of lookups: API, database, and application.

API lookups are the most frequently implemented. APIs are the most common interface for data exchange, whether internal or external to the organization. Once you know what you are looking to get more information on, you send it to the appropriate API and handle the response. API calls take a while. They likely won't be able to keep up with the streaming data if used too frequently. This is why it makes sense to filter and a process a stream before making an API call, so that it can be narrowed down to whatever justifies it. For example, if looking at Wikimedia edits, you may be interested in looking up only more information about any edits that were blocked and why.

The following example shows how to do an API lookup for IP data related to the IP address we get in a record. ThreatCrowd is a popular place to look up IP data for free and has an easy-to-use API. Gathering this type of information before a person or system needs to look at it helps a lot—it saves them time looking it up and immediately gives them some context to go with it. The code in Example 5-2 is specific to AngularJS.

Example 5-2. Sample API lookup

```
var strIp = objMsg.ip;
var objConfig={
  'url':
  "https://www.threatcrowd.org/searchApi/v2/ip/report/?ip="+strIp
};
if(strIp!=='127.0.0.1'){
  $http(objConfig).then(
    function fnSuccess(objResponse){
    // add the API response to the data
    objMsg.api={objResponse};
  },
    function fnError(objResponse){}
  );
}
```

Database lookups are required when you have previously stored information that is relevant to a current decision. This is usually related to history. If you need to decide whether something is typical, you need to keep history and compare to it. You can connect your app directly to a database or put a small API in front of it

and do something similar to the previous example.

Application lookups are quite varied, depending on the purpose and capability of the application. The difference between an application and an API lookup is that the application will not return an immediate result. This means that it either needs to be polled for information or needs to create its own data stream for the results.

Normalizing Events

When data is being analyzed for outliers, it helps to have it be as consistent as possible during normal operation. It might be normal not to get a field when it's blank. Instead of not showing that field at all, assigning a default value can make things align better downstream. Example 5-3 shows one approach for this.

Example 5-3. Normalize the data structure, add defaults when missing

```
var objTemplate={ip:'127.0.0.7',system:'unknown'};
var arrTemplateKeys=Object.keys(objTemplate);
var fnOnMessage=function(objMsg){
  for(var i=0;i<arrTemplateKeys.length;i++){
    if(typeof objMsg[arrTemplateKeys[i]] === 'undefined'){
      // this expected field doesn't exist, add it with default val
      objMsg[arrTemplateKeys[i]]=objTemplate[arrTemplateKeys[i]];
    }
  }
}
```

Another type of normalization could be in the values that need to be compared. The most common example is time. It's a common practice to store and transmit all data with timestamps in UTC. If this isn't done for some reason, it's a good idea to translate the timestamps before the data is used for statistics and visualizations.

Extracting Value

You can have data streaming and being visualized and still not make very good use of it. It can be an overwhelming blast of incoherent information that makes you question the wisdom of streaming data to look at. This is a hurdle that often discourages people. Although visualization of streaming data has been avoided for so long, you can gain a lot of insight by performing some simple exercises. Within a few steps, you can make the data easier to digest and have a better understanding of its possibilities. These can be considered ingredients to be used in the processing components discussed in the preceding chapter. A lot of libraries do this type of thing. In the next section, we will use an open source one that will be easy to show examples from that can be found online. its purpose is to define these simple transformations in a set of easily interpreted JSON definitions. You can pass batches of records in as an array of JSON records along with a small config, and it will return the modified collection of records. One advantage of using a library like this is you can embed it into your applications and exchange portable configurations.

The processing examples shown next are logically straightforward. They allow data to be added, removed, or transformed based on easily defined conditions. This library (and easily defined logic for processing JSON data) can be used to accomplish a lot, but you may want to add other things to your processing that are more complex. You can use machine learning–trained classifiers or even developed algorithms to score and classify the data before showing it to an analyst. If these are things you would do anyway, it helps to show the results of the automated decisions along with the other data to the analyst to make decisions and find issues with the classification and scores. These new values can also be used with the processing rules mentioned. For example, you may have a classifier look at a URL string and make a decision about how suspicious it looks based on its various characteristics, and a guess as to what type of content will be behind it. We will talk about machine learning more in Chapter 12.

Besides just updating the data, a separate score that is unique to presenting the data can be added. This score might indicate how interesting the data is for an analyst to look at. This will allow you to do some client-side sorting by interest

score and keep records in a queue for an analyst to address.

The JSON Collection Decorator

The JSON Collection Decorator is a free open source library with a particularly utilitarian name:

- *JSON* is a common data format. It's lightweight, allows hierarchy, and works well with JavaScript.
- A *collection* is a group of objects, like a batch of records.
- *Decoration* is a term used for taking an object such as a record and passing it through some functions to add, remove, or modify its properties.

A lot of similar libraries are available on Git, but this one can serve as a working example of how you can process small batches of streaming data inline. If you choose another library for this function, look for the following qualities:

- Able to take a batch of records at once.
- Records need to be looped through only once, even if there are multiple conditions and actions.
- Filtered records are removed before the rest of the processing, so they aren't decorated unnecessarily.
- Able to match multiple conditions before performing an action.
- Actions can be performed on different properties than the conditions.
- Multiple actions can be performed per condition.

The difficult balance to find with this set of functions is between performance, capability, and flexibility.

Example 5-4 illustrates how you can use the JSON Collection Decorator to process a batch of records.

Example 5-4. Use of JSON Collection Decorator

```
var objConfig={
   filters:[
     { path:"path.to.key",op:"eq",val:"value to match" }
   ]
```

```
   ⌟
,decorate:[
    {
      find:[{ path:"path.to.key",op:"eq",val:"value to match" }]
      ,do:[{ path:"path.to.key",act:"set",val:"value to set" }]
    },{
      ,find:{ path:"path.to.key",op:"eq",val:"value to match" }
      ,do:{path:"path.to.key",act:"stack"
        ,val:"value to add to array" }
    }
]
}

arrResults = decorate(objConfig,arrCollection);
```

Processing Checklist

Here are some common tasks in the processing of a data stream:

1. Count anything that you want to account for later before filtering.
2. Filter out uninteresting records.
3. Filter out unimportant or consistent fields.
4. Parse the data types found.
5. Extract data.
6. Categorize based on content.

Let's go through these step by step.

Getting counts for things sounds like a no-brainer—it's the simplest possible statistic. However, it's often forgotten prior to filtering or throttling. It's important to know what your totals are so that you can reference them when you have more granular counts of things later that you kept.

Next, filter out records that don't offer you any insights: `{filters:[ { path:"path.to.key",op:"eq",val:"value to filter by" } ]}`

Now that you have counts of these records, it makes sense to display the counts instead of the records. All you need to do is assign them a label that accurately describes what the counts will represent.

Now, filter fields that have no variance or value: `{decorate:[{ find:[{ path:"path.to.key",op:"eq",val:"value to match" }] ,do:[{ path:"path.to.key",act:"remove" }] }]}`

Records often contain so much information that it can be difficult to even see one record on a screen at a time. This is when filtering fields makes a big difference. After you filter out the fields you don't need at the moment, you can digest the rest of the record more easily. When filtering fields, be careful not to filter out anomalies. If a field is unused but always listed, it's safe to filter, but if it just commonly has the same value, make the filter conditional on it being that value so you can still see when it's different.

value so you can still see when it's different.

Next, parse data types that can benefit from it. A lot of data has subcomponents that can be valuable as statistics if separated. For example, several components of a URL can be significant separately. A URL contains a protocol, domain or host, path, and query string, which itself has key/value pairs. Some URLs will also contain authentication credentials. You can parse URLs with the JSON Collection Decorator as follows: `{decorate:[{ find:[{ path:"path.to.key",op:"eq",val:"value to match" }] ,do:[{ path:"path.to.key",act:"parseUrl" }] }]}`

Anything that can quickly be parsed for valuable subcomponents is worthy of the effort for later use.

Then extract data from any files that are referenced. The metadata record you get for a file is rarely a complete set of all the valuable data available. There is a lot of information within files that should be contained within a record if useful, and if it isn't, you'll need to extract it. For example, images often have geographic coordinates embedded in them. You'll want to know what kind of file the operating system sees the file as, and if there are any certificates or origin information. If the file is a package, you'll need to extract it and create records for the files it contains. If you are looking at files for security purposes, you'll need to get hashes for the files.

Finally, content categorization is useful for knowing what processing workflows to apply to your data. There are a lot of methods for content categorization, from regular expressions (regexes) to machine learning. The more unstructured data there is, the more sophisticated the process will be to categorize it. Content categories can be things like "support, bug, question, complaint, comment" or "simple, complex, ludicrous." It all depends on the context of the data and what useful categorizations you can derive from it. Here's an example: `{decorate:[{ find:[ { path:"path.to.key",op:"eq",val:"value to match" } ,{ path:"path.to.key",op:"eq",val:"value to match" } ] ,do:[{ path:"status",act:"set",val:"ludicrous" }] }]}`

Streaming Statistics

An important and distinct consideration when keeping statistics for streaming data is to try to keep the statistics in a cumulative fashion as much as possible. If every time you add an item to an array you need to evaluate the whole array to recalculate the statistics you need, as in Example 5-5, you will hit scalability limits much faster than if you keep the information as you go.

Example 5-5. Typical stats processing in JavaScript

```
// define what's being pushed into the array
var intData=42;
// add an element to the array
arrData.push(intData);
// then use a function to get the stats you need; this uses loDash
intMax = _.max(arrData);
intMin = _.min(arrData);
intSum = _.sum(arrData);
intMean = _.mean(arrData);
```

When you are visualizing streaming data, you often need to show the current values as well as how those values relate to a larger context. This means you need to keep basic statistics up-to-date for presentation. If you use the typical process just shown for each record added, you will be looping through your entire array of items for every record, times the number of stats you are getting. That's a lot of processing for a simple task. If it needs to be done hundreds or thousands of times per second, the system processing the data might not be able to keep up. Consider the approach shown in Example 5-6 as an alternative.

Example 5-6. Cumulative stats in JavaScript

```
// define what's being pushed into the array
var intData=42;
// update the stats
intCount++;
intSum=intSum+intData;
if(intData > intMax){ intMax=intData; }
if(intData < intMin){ intMin=intData; }
intMean = intSum/intCount;
```

This will produce the same statistics but not require any looping through the array in order to accomplish it. These types of small savings add up rapidly when processing needs to be done per record and the arrays of information

stored are large. This approach also allows you to keep stats that outlive your array in case you need to keep that at a fixed size or within limits. If you have a lot of these processes running in parallel, you can have them occasionally reconcile with each other and decide who has the max, the min, etc., all without the costly processing of looping through all of the data.

Types of Statistics

These are some basic statistics to consider keeping track of as data streams rather than calculating them on demand:

- Basic statistics (min, max, average, quartiles)
- Comparison of values
- Frequency of values
- Cardinality of values
- Co-occurrence of values
- Statistics within time windows

Basic statistics are fast and easy to get. They are useful for so many unexpected things that it's a good idea to keep track of them whenever possible. They are also nice to display on their own. When you see individual values, it helps a lot to know how those compare to the rest.

Comparing values within a record is a good way to quickly get some insights. With the Wikimedia data, it's pretty significant if an edit of a page is larger than the original page. This means that the edit represents a large overhaul or infusion of content. This deserves some recognition over other edits that might affect a fraction of a percent of the content.

Frequency of values can quickly tell you how common a given value is. A more complex iteration on frequency would be some time pattern analysis. This could help you understand that while a value is common overall, it is not for the time period seen.

Cardinality of values helps you to understand how varied the data is. It makes a big difference in deciding how to present data. If you have 3 to 12 unique values, it might be a good idea to compare them in a bar chart; if you have thousands, a bar chart will not work very well.

Co-occurrence of values is a useful and simple statistic to show the intersection of values. If you are getting source and destination information in your records,

simply listing the top sources and the top destinations is OK, but listing the top source-to-destination combinations is much more valuable and can give a very different result.

Statistics in time windows will help with time pattern analysis. Streaming data is about what's happening now. Statistically, what represents "now" is a matter of scope. The current year is "now," but that's probably too large of a scope to be relevant. A useful time window for streaming data is usually less than one day, maybe hourly. Time windows can also be event-based such as the beginning and end of a show. Statistics within these time windows can be used for a historical comparison of the current window to previous ones.

Record Context Checklist

Some broad things to consider in providing context for your record data are as follows:

- Status
- Priority
- Comparison of values to statistics

Determining a status for the record is useful for a complex workflow. This especially applies when the status is not something that can always be determined from the data originally provided but is derived from data that's added and interpreted. The status can determine where a record goes next in a workflow.

Priority can be based on a number of things, such as severity, confidence, and status. It is usually used for ordering in a queue, whether automated or interactive. Priority can be used to make sure the most important items are processed before any that can wait. This is the primary method to override a purely time-based first come, first served ordering of processing.

Comparing values to statistics will give you additional information as to where a record fits within a larger context. It can be good to know how close a value is to a minimum, maximum, or average for the existing time frame.

Scaling Data Streams

After you have applied the necessary data manipulations, you'll have reduced the data down to what is worth displaying; you have a priority to order it in, and enough information to give fast visual cues to quickly digest. These are the things that start to make the streaming data consumable and useful by someone monitoring it. If after all of this you still have too much data, however, you may need to apply extra methods to scale it to something manageable.

Scaling the data should be looked at only after all intelligent options have been exhausted. It's always better to intelligently decide which records you view and where they are in the larger context. If the scale of the data makes that too impractical, consider the following comparison of methods to keep up with processing large volumes of data as they are generated (Table 5-1).

Sampling is the process of randomly grabbing every *n*th record (see Example 5-7). This can allow you to work with a more manageable scale and know what that scale is. The more records you skip in sampling, the less relevant the retained data is.

Table 5-1. Comparison table of methods to process subsets of data

	Known scale	Relevance
Sampling	X	Depends on sample rate
Throttling		Depends on whether caps are reached
Hardware-bound		Depends on whether caps are reached
Dynamic	X	Best possible

Example 5-7. Sampling logic

```
var intSample=10;
var intCount=0;
// call the sampling function every time
var fnSample=function(objMsg){
  intCount++;
  if( intCount%intSample===0){
    // the nth count, fire the function to process
```

```
    fnOnMessage(objMsg);
  }
};
```

Throttling (Example 5-8) puts a cap on the amount of data that is digested in a given time frame—it might be something like 1,000 records per second. This is less preferred than sampling because you don't know how many records there are in total. You cannot multiply the number of records kept to get an estimate of the total. You may be able to be more selective about what you keep and what you don't, though; you can keep a certain number of each type of record, or be selective about what records you keep after the cap is reached.

Example 5-8. Throttle logic

```
var intCap=1000;
var intCount=0;
var intStart=Date.now();
// call the throttling function every time
var fnThrottle=function(objMsg){
  var intNow=Date.now();
  // see if it's in a new time increment
  if(intNow>intStart+1000){ intStart=intNow; intCount=0; }
  else{ intCount++; }
  if( intCount < intCap){
    // only run if under the cap per time period
    fnOnMessage(objMsg);
  }
};
```

Hardware-bound scaling means that the limits depend on how much the hardware you are running on can handle. Without any other limits set, this is how processes run. If the applications and hardware are able to process all the data, you get everything. If the hardware cannot keep up with the data volume, you get an unpredictable percentage of the total data. The worst part about this is not knowing when the complete data was received and, when you get only a portion, how much of the total that portion represents.

Finally, *dynamic scaling* can be used to get the most data at the most predictable scale. This approach adjusts the throttling or sampling rate based on variables such as how many records are waiting to be processed in a queue. The key thing is to know what method is being used when, in order to make predictions and appropriate comparisons where possible. It's difficult to compare data from a throttled data source where caps are reached to a complete source. A lot of

assumptions need to be made that are more appropriately explored in a deeper study of statistics.

Here are a few guidelines to keep in mind if you need to scale your data:

- First, you need to know the limits of your hardware for processing the records.
- Set the throttled amount to a comfortable value under that limit.
- When that limit is reached, switch to sampling. Sample at the lowest rate possible, keeping more records than you toss, and adjust as required, always keeping track of the sample rate.
- Set a throttle cap only as a safety stop to prevent the system from becoming unresponsive. Hopefully, this cap will never be reached, or reached only while the dynamic sampling rate is being adjusted.
- Allow the sampling rate to be reduced as well.
- Keep track of the sampling rates and of whether caps were ever reached (and when).

Presenting Processing

The preparation of data for use in downstream systems and for presentation is a significant part of business. So far, we have talked about the processing required in order to present data to an analyst with as much information as possible to enable them to quickly make a decision. A few things specific to processing deserve consideration as well:

- Presenting where data being shown is situated within the larger processing pipeline
- Presenting any significant routes taken due to processing decisions
- Having a streaming data visualization for a complex processing ecosystem

The moment at which you see a record may not be the end of its trip through a processing pipeline. It can be useful to stream data to a client from multiple points within a larger ecosystem. When doing so, it's good to know where that data is within that context. You can either display the relevant information with the data, or associate it by its position within a layout (see Figure 5-2; layouts are discussed later in the book). Both may be required when there is a very complex pipeline. The layout will be a linear association, and any more detailed context would need to be in the data itself.

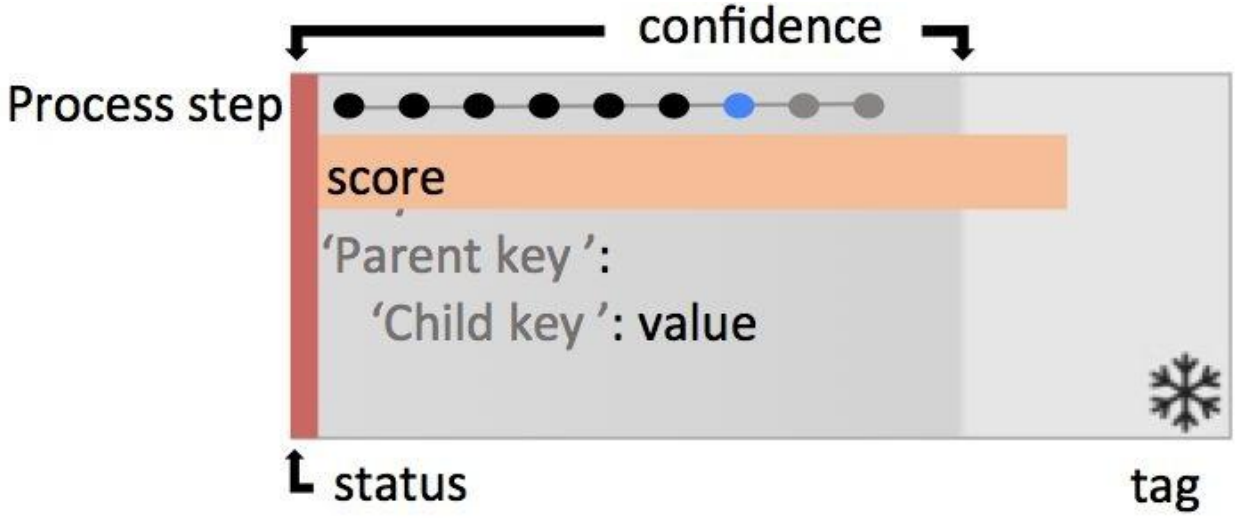


Figure 5-2. Record displayed with a reference to its processing location

A lot of decisions are made within a processing pipeline. Values, history, and other factors can influence what needs to be done with the data. When these decisions are made, it's good to present these decision forks to the analyst. The person looking at the data might not immediately know that certain records are processed by a different set of rules when they have values that meet certain criteria. Adding this information can help a lot in a critical situation. The indicator can be simple, such as "Bypassed for download, in cache," or even shorter than that as long as the analyst has something to quickly reference that reveals more detail. Providing this information could allow an analyst to quickly resubmit something to go down a different processing path, or prompt them to adjust future processing rules to have a new route on a new condition. Without having that context displayed, they wouldn't have the opportunity.

The processing pipeline is essential to streaming data visualization in many ways, but it can also help you to understand the status of a complex ecosystem. It's a good fit because it has information that you need to see and act on as soon as possible. It's also complex and needs to give you as much information as possible without trying to lead you to predetermined conclusions.

> *It's not as important that the information presented is accurate as much as giving the person the idea that they need to make a decision.*
>
> —Casey Rosenthal, manager at Netflix Chaos and Intuitive Engineering and one of the creators of Vizceral

Figure 5-3 shows an example of a streaming visualization. The open circles represent microservices. The connections between them in a process are the connecting lines. The dots on the lines represent traffic: an increase in volume is shown by more dots. This can quickly give someone an idea of the health and load of the system. The dots that represent traffic are not to scale, and they don't need to be in order to elicit an intuitive read by the observer.

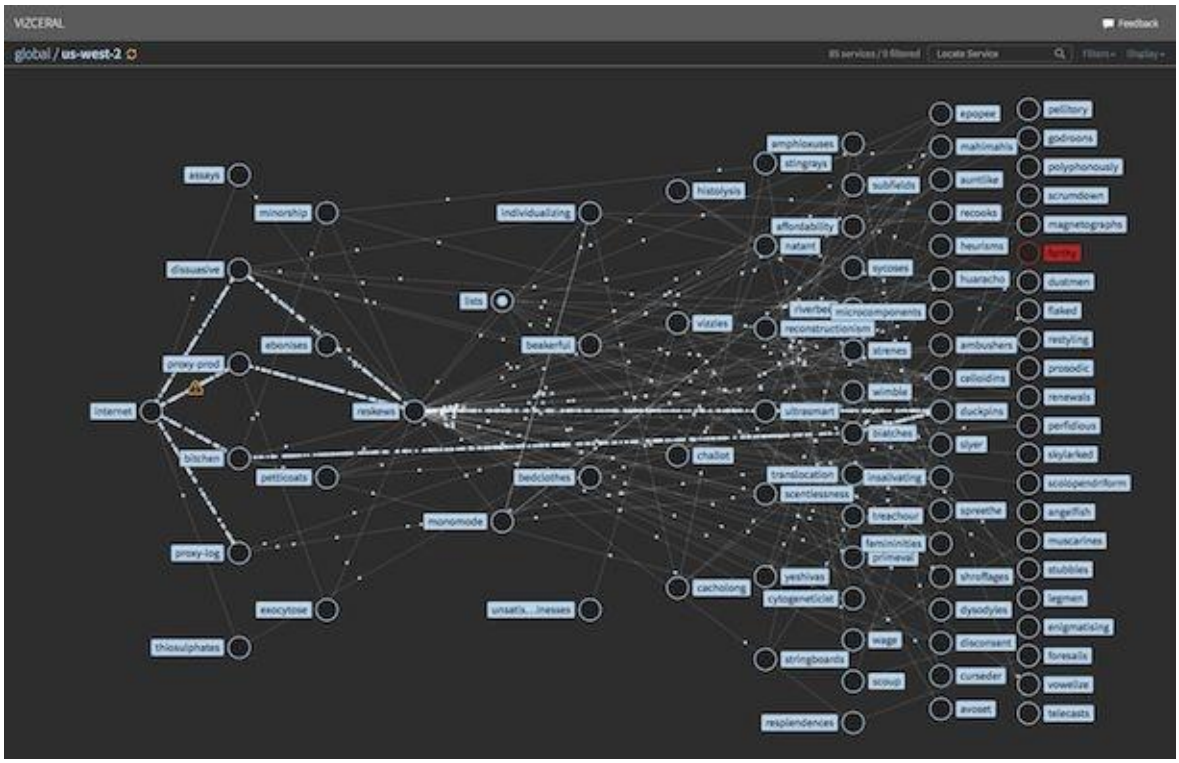


Figure 5-3. A streaming visualization at Netflix (source: http://bit.ly/2LKKxkj)

Processing for visualization and visualizing processing are both niches in much larger fields. Stream processing for visualizations is most likely going to be something new to you. Visualizing processing is helpful in understanding what's going on. Working together, these techniques can accomplish a lot.

Chapter 6. Developing a Client

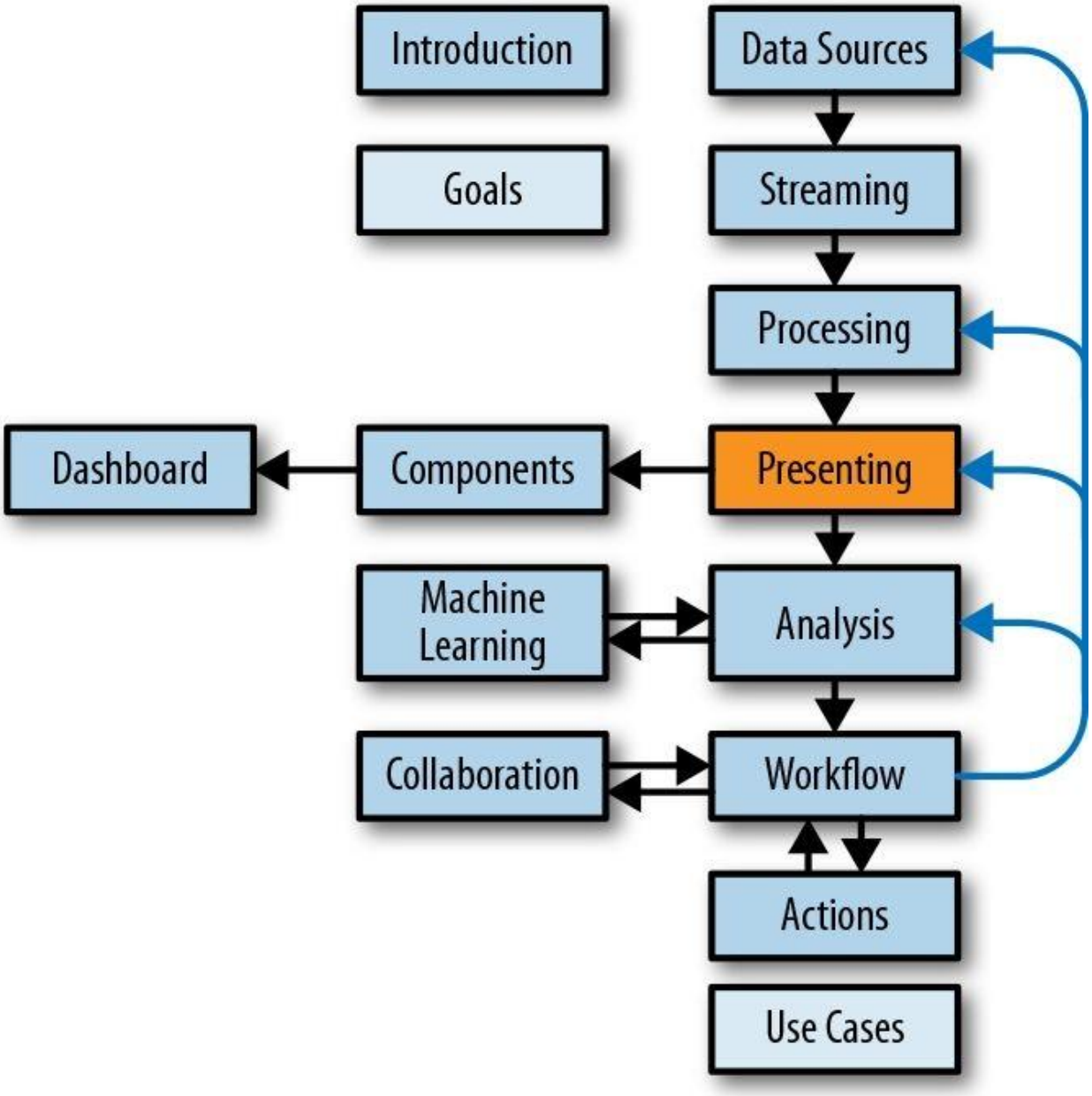
> **NOTE**
>
> The workflow diagram at the beginning of the other chapters was intentionally left out of this one, as it doesn't have a place at the same level as the rest of the topics. The client is where the other subjects are applied. Which subjects are applied in the client (as opposed to other systems) depends on scale and complexity.

After you've identified and figured out how to process your streaming data, you need a way for people to be able to view and interact with it. The interface for presenting streaming data is known as a *client application*. The client can be in a browser, run on a device like a phone or desktop, or be firmware embedded in a device. In order to meet your needs for presenting streaming data in a useful manner, you may need to purpose-build a client for it.

One of the first things to consider is how much of the workload will be in the client. This choice is not as simple as delegating as much as possible to server components—the client will have a faster response time and be more interactive. Allowing the client to handle a portion of the load can also be significant for widely distributed applications. Anything that is consistent for all clients can be done on the server. Anything that needs to be customized per individual client makes sense to be done on the client.

We'll look at some of the other major considerations next, and go over the basics of creating a client application. There is one provided to help get you started quickly. You could also choose to apply the concepts introduced here to a different application or language if the provided client app isn't a starting point that works well for you.

Chapter 7. Presenting Streaming Data



This chapter introduces some topics and elements to consider with regard to presentation. It can be used like an idea generator that you revisit as needed to

see if any of the mentioned elements make sense to incorporate.

Staring at thousands or millions of raw records is not very efficient or sustainable for any goal. It’s important to know what the raw data looks like, but only in order to understand what would be interesting to summarize in a more friendly format. Visualizations are that friendlier format. They engage a different part of the brain that understands color, spatial references, and patterns. Visualizations have such an immediate impact on the viewer that they are often used to imply a conclusion from data. This is why there is a natural division in data visualization goals among sales, reporting, and analysis.

Visualizations can be employed to more effectively achieve all of these goals. They can take overwhelming amounts of information and present it in a way that can be easily understood. A “sparkline” can show a pattern of values over time, including the minimum, maximum, average, first, and last values, all within a space that is typically reserved for just a couple of values. This represents a huge increase in data density. Visualizations can also be used to tap into a common lexicon of meaningful images and styles. Translating text into recognizable icons can increase the consumable data density without any explanation needed. A common use of this is displaying male and female icons when showing statistics for demographics. Of course, there is always a proportional trade-off between gaining data density and losing context and nuance. The higher the data density, the more patterns, anomalies, and unasked questions will be abstracted away. This is why the various levels of visualizations work well in an iterative process.

Showing Streaming Data

All streaming data can be presented. Streaming records to the console as they are generated is the original streaming data visualization. In a streaming data client, you can apply a lot of enhancements to that. It's definitely recommended to stream some data in the client as opposed to having only stats and visualizations, to provide context and supporting evidence. The records can be organized in various ways and then have visual cues and styles added to them to help the viewer understand what is flying by and how it relates to everything else being presented.

Figure 7-1 is a modern approach that emulates streaming data in a console so that it can add some functionality that a console cannot do.

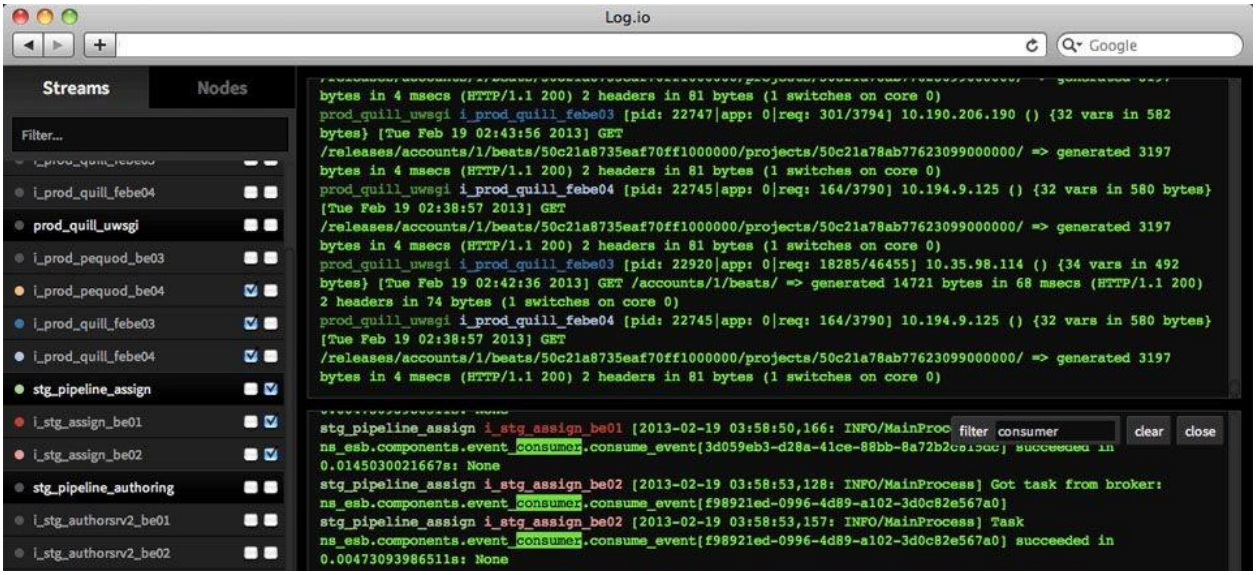


Figure 7-1. Example from log.io of streaming records in a client

Events

Event-based data is what we call data that is the result of something occurring. This data is not simply polled or collected or queried. Events naturally fit well into streaming data visualizations because the time when they occur in relation to each other can have meaning and show prominently. Some technical considerations of streaming event-based data are as follows:

- Events can be acted upon immediately only up to a performance threshold, where they start to get backlogged. At this point, you need to batch, trim, or distribute the workload.
- If a timestamp is not already added to the events, create one. If there are several steps in the lifetime of an event, keep a separate timestamp for each; they can be significant when distinct. You can always keep a `last_updated` timestamp additionally if needed.
- If you treat all events as batches within time limits, you will save yourself some headaches.

Showing the differences in data as they occur works only up to a certain threshold. Once that is greater than one change per second, too much overhead is required to visually update the information, and anyone watching it can't absorb it. To avoid those issues, you can keep batches of data to process every second and update the display. Example 7-1 shows how this works.

Example 7-1. Batch everything in a time frame, and then process

```
// a time reference is needed to keep track of batches
var intBatchStartTime=Date.now();
// an array to hold the batches
var arrBatch=[];
// a function to be called on events
var fnOnEvent=function(objEvent){
  var intNow=Date.now();
  // always add the event to the batch
  arrBatch.push(objEvent);
  if(intBatchStartTime+1000<intNow){
    // time to process call the function that processes
    // note that if the array isn't copied, it will have a race condition
    fnProcessBatch(arrBatch);
    // reset the batch
```

```
    arrBatch=[];
    intBatchStartTime=intNow;
  }
}
```

Logs

Logs are records of events that are commonly saved to a file. By far the most common streaming data watched is updates to a log file in a console. This is typically done with the Unix command `tail -f <filename>`. You can do this with error logs and watch the servers, behind-the-scenes reactions to your interactions. The console is pretty limited, though. It can be hard to pick out significant information or do anything with it. In order to have more options, you can bring the same log files into a streaming data client with a library like `nodetail`. Example 7-2 illustrates its use.

Example 7-2. Tailing a log file with nodetail

```
// watch a file
tail = new Tail("fileToTail");
// fire every time a new line comes in
tail.on("line", function(strLine) {
  // call your defined function for the event
  // note: in most cases you'll need to parse the string
  fnOnEvent(strLine);
});
```

Records

Records are what is returned when querying a database. In most cases, the records need to be seen all at once and in a grid ordered on significant fields that aren't chronological. This is a good fit for a streaming visualization only if the records have timestamps and if there's some value in showing them chronologically. For example, streaming records at an interval can be used as a playback of events from a point in time. When working with records, you are completely outside an event-based pattern. You can either show a number of records per second or play them back as events with a relative or accelerated time scale. Example 7-3 shows how to stream a record per `intDelay`, where `intDelay` is in milliseconds.

Example 7-3. Records per second

```
// use set interval for timing, this var will be a reference to it
var objThrottle={};
// assuming arrRecords is a collection of records
objThrottle=setInterval(function(){
  // assuming fnSendRecord is the function to run for each record
  fnSendRecord(arrRecords[intIndex]);
  if(intIndex===intRecords-1){ clearInterval(objThrottle); }
  else{ intIndex++; }
}, intDelay);
```

Dashboards

A streaming data dashboard will bring together several elements that share context. A typical dashboard has some common rules. According to *Information Dashboard Design* by Stephen Few (O'Reilly), it should provide information at a glance and be displayed on a single page. Figure 7-2 shows an example of such a dashboard.

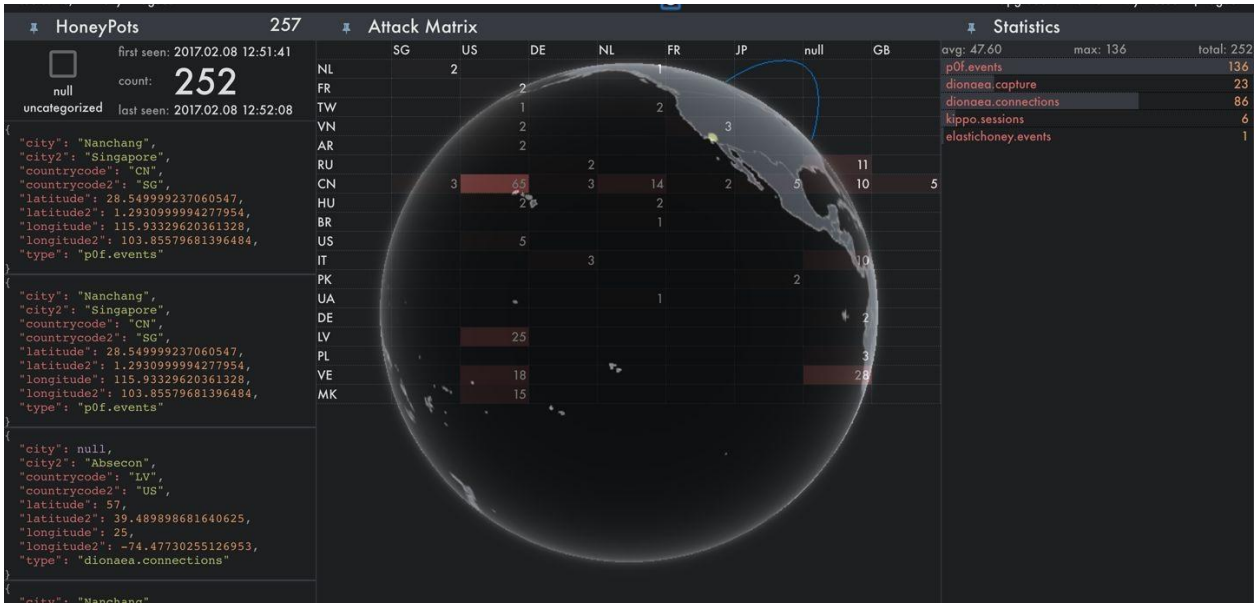


Figure 7-2. Multiple elements in a streaming data dashboard (source: https://ohm.ai)

We will go over the options for assembling a streaming data dashboard in Chapter 11. For now, it's good to keep this end goal in mind while reviewing the elements that will compose it.

Visual Elements and Properties

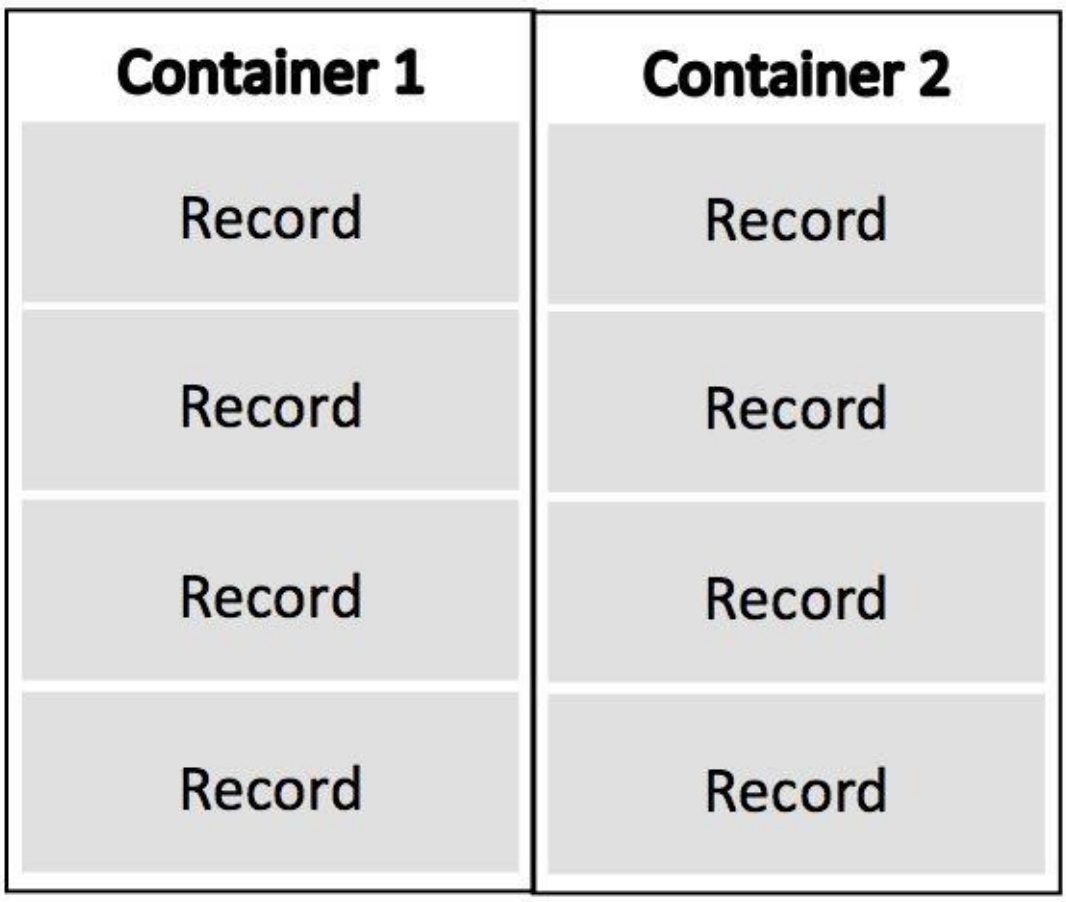
A lot of visual elements and properties can be used in visualizations. The choice can be daunting. The following list shows some examples of when using them makes good sense. All of the examples can be found in the file *elements.htm* in the accompanying code repository.

Containers

A container is a grouping of elements with a border. The border doesn't need to be a line; it can also be spacing. Containers can take many shapes that have their own terms.

When to use it:

When a distinct separation between groups of items is needed.



Visibility

Visibility

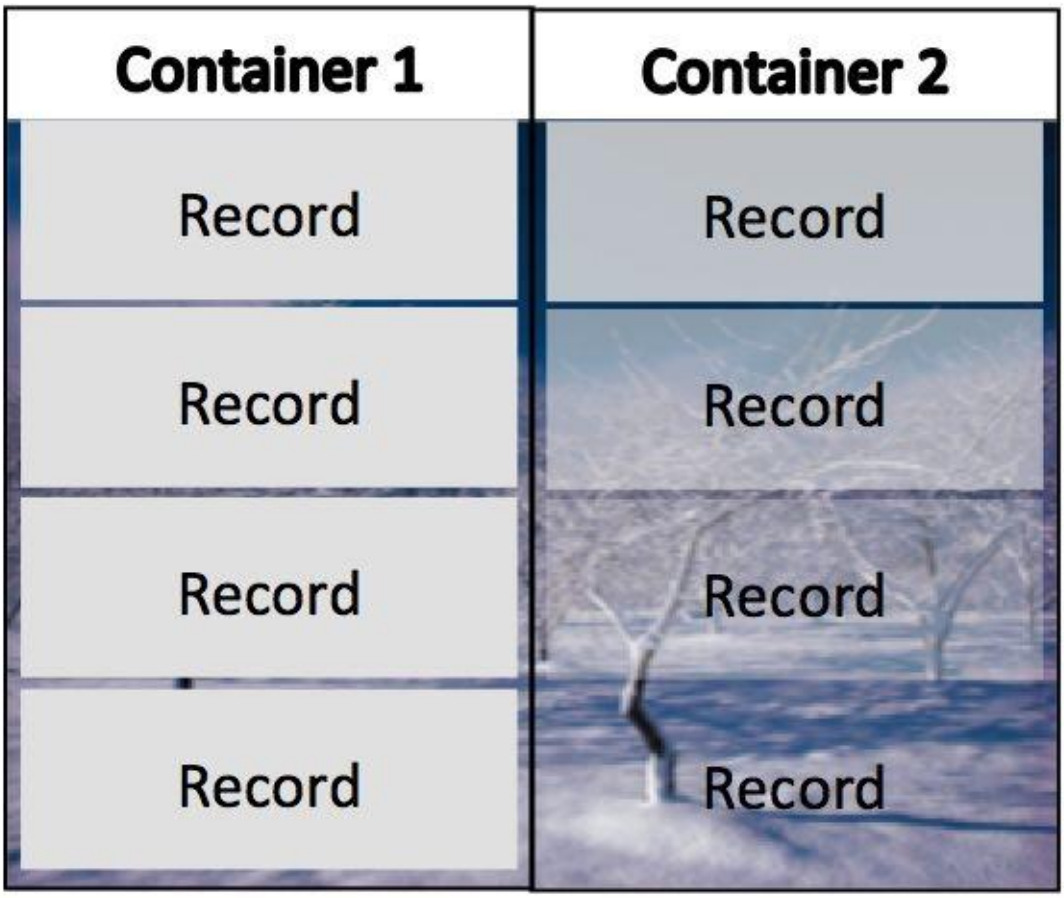
Elements can be hidden, shown, or have varying levels of transparency.

When to use it:

Show what's significant at the time and hide what is not. Use transparency to give a vague idea of what's going on behind.

Tip:

Use the CSS `opacity` property for transparency.



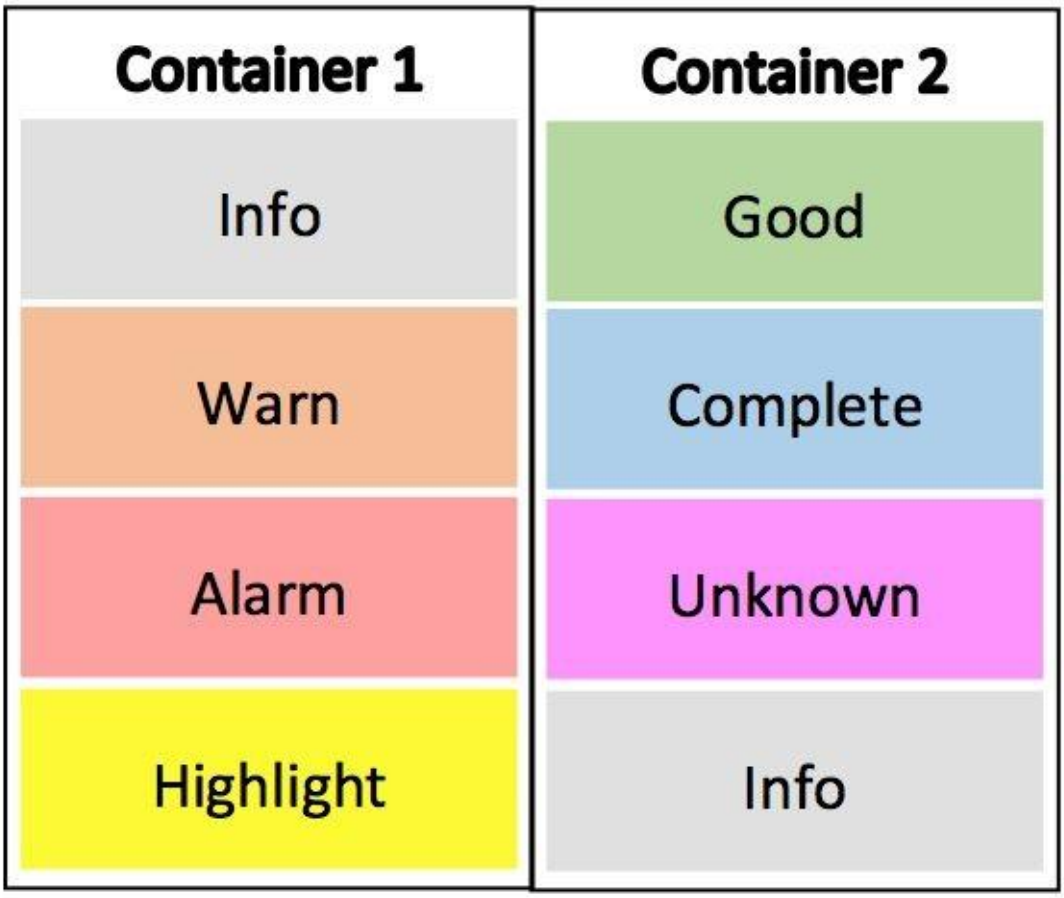
Color

Color can be modified on nearly all elements to represent something significant. Color requires some thought and consistency across elements to be most effective.

When to use it:

Only when the color has consistent meaning.

Only when the color has consistent meaning.



Background

Background fills can be used as an indicator for the entire record that stands out. A select few background colors should be chosen, ensuring that the overlying text and elements are still readable.

When to use it:

As a quick reference of meaning for the record as a whole, conveying status, priority, significance, or type.

Size

Size is an intuitive indicator of weight, importance, priority, or volume. Too much size variance can make things too difficult to read. Best practice is to use a few predefined sizes that have meaning.

When to use it:

To convey a higher or lower weight than normal. Use as an exception.

Borders

Outlines of a container that can vary in shape, style, color, thickness, and opacity.

When to use it:

To clearly and quickly show a state for everything within a container.

Tip:

Use CSS `border` and `border-radius` styles.

Alignment

Also known as *justification*. Alignment refers to the placement of elements and use of negative space within a container.

When to use it:

Use sparingly when the intuitive effect is beneficial, such as right-aligning numbers while text is left-aligned.

Fonts

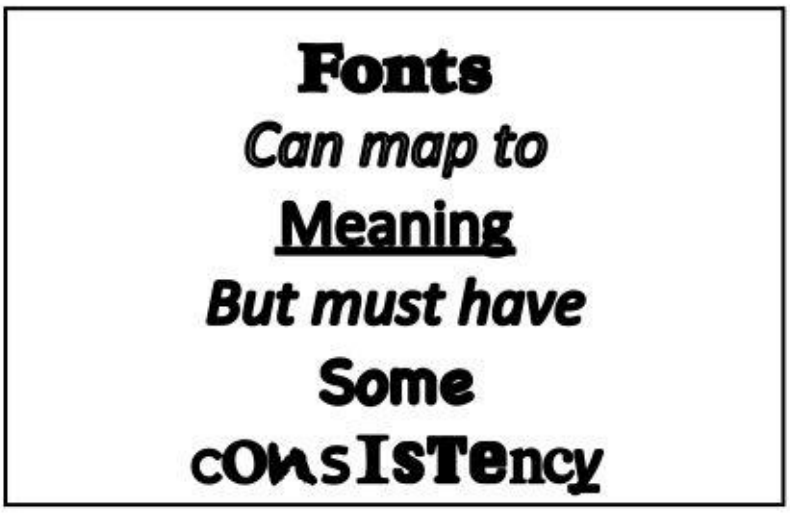
Fonts are a passionate subject in design. They can have a powerful impact, so when they are abused or unwittingly misused, it can be a big detractor. When used cautiously, deliberately, and with purpose, they can be a useful detail.

When to use it:

Syntax highlighting, title versus body, key versus value, label versus input (not all of these at once).

Tip:

Google Fonts is a great place to start.



Layout

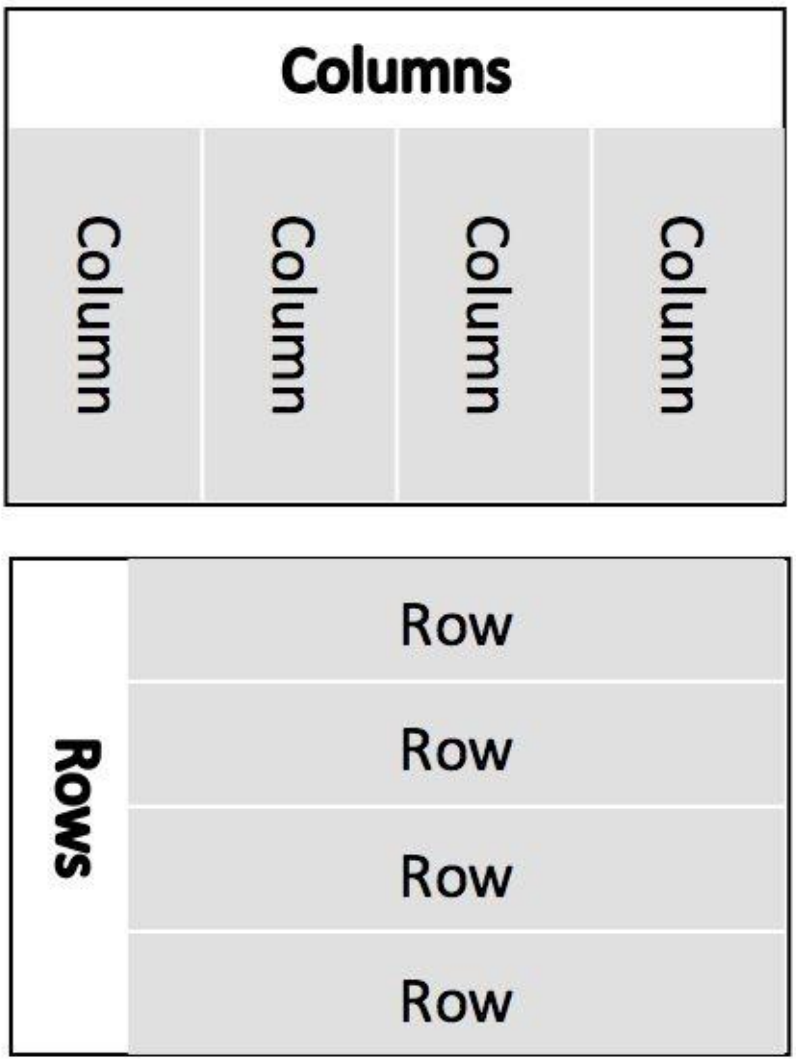
Layouts help create a logical flow of items that are shown.

When to use it:

When more than one set of elements is presented.

Tip:

Use CSS Flexbox.



Thickness

Thickness can intuitively translate to weight or volume. Borders, connecting lines, dividing lines, edges, and other elements can have thickness mapped to a meaningful value.

When to use it:

To convey weight or volume.

Tip:

Use the CSS `border` properties.

Spacing

Spacing can hold meaning but is more commonly used as something necessary for readability. It’s used in code to show hierarchy and is used in data formats as well.

When to use it:

For readability.

Tip:

Use the CSS `padding` and `margin` properties.

Adjacency

Adjacency refers to elements being next to each other. This happens naturally, but carries intuitive meaning. When you use this feature intentionally, you can take advantage of this intuitive meaning.

When to use it:

Use adjacency when a relationship (or lack thereof) is relevant.

Position

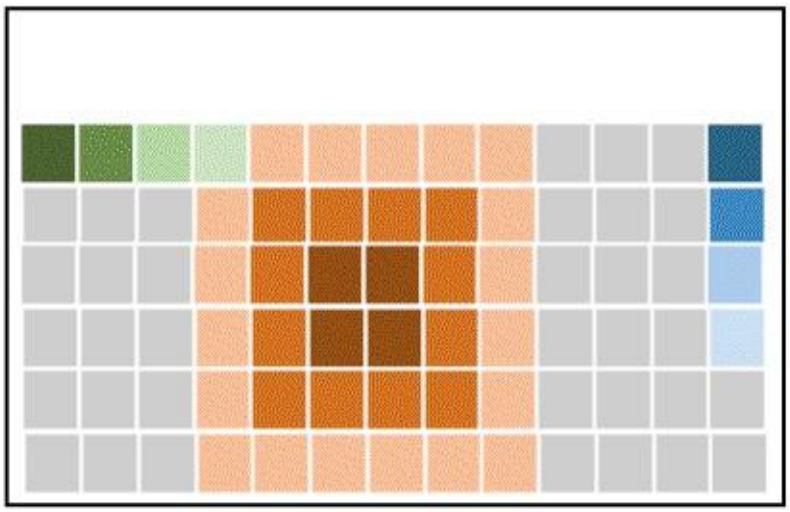
Element positions easily translate to their relationships with all of the other elements, indicating what's related, what's new and old, and what's in between.

When to use it:

Always. The elements will all end up somewhere, so put them somewhere deliberately.

Tip:

Use the CSS `top`, `bottom`, `left`, `right`, `float`, and `position` properties.

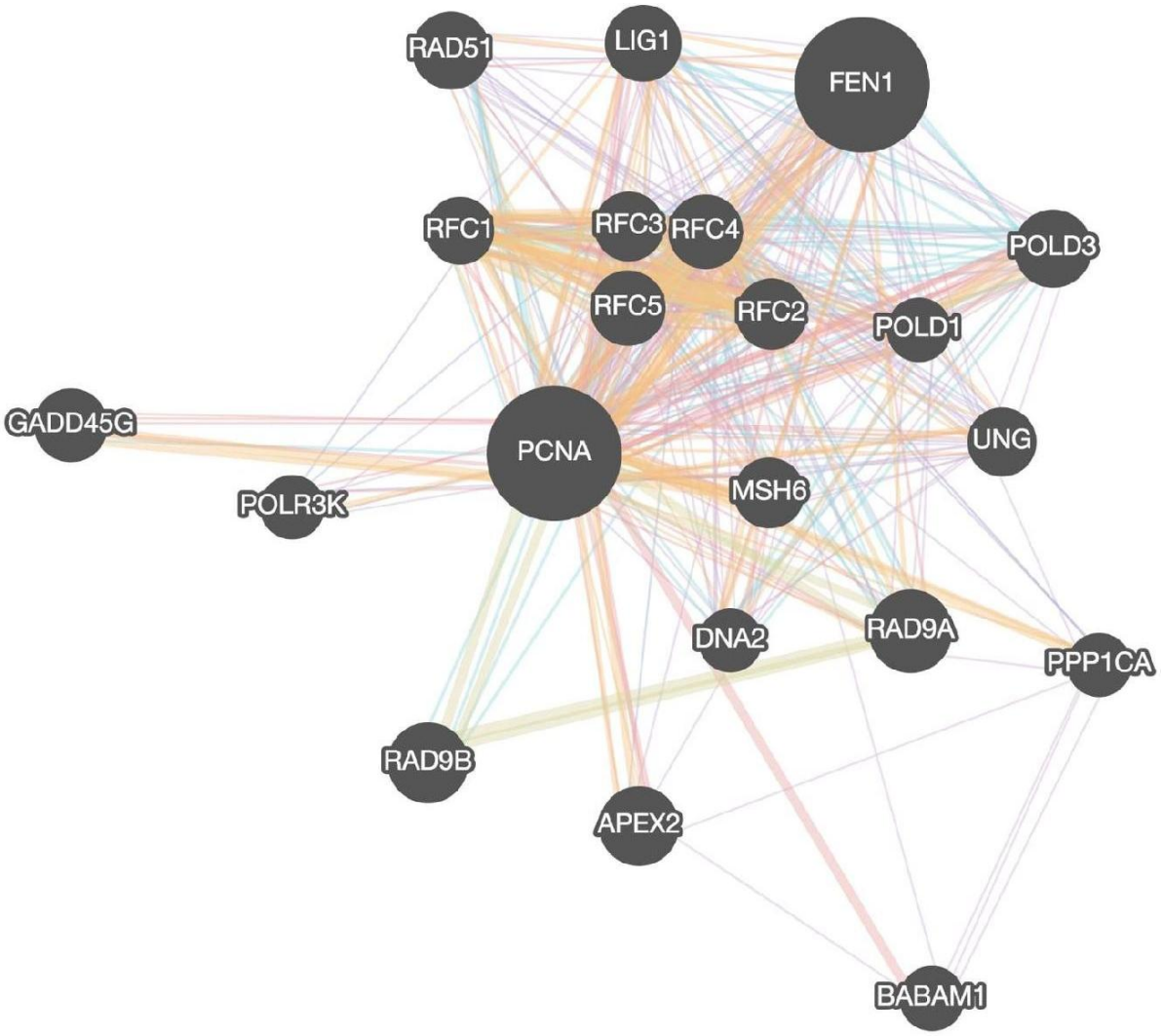


Connections, relationships

Connections are used to show relationships between elements. The connections can have meaning mapped to thickness, color, labels, and direction. This example was found at *http://js.cytoscape.org/*.

When to use it:

To indicate relationships.



Images

Images provided as content as well as images created based on content can add a lot of information quickly. They can also quickly take up a lot of precious screen real estate, so show them when they might be of most benefit and crop them to the most information-rich portion if possible.

When to use it:

When images are the subject of the record or an image can be used for a quick intuitive decision, like with a screenshot of a website.

Icons

Icons are especially useful for things like tags, where they can consistently represent common elements.

When to use it:

When you have something that can frequently and consistently be represented by an icon that is useful as a fast visual cue.

Tip:

Use Font Awesome.

icons


```
{
  "place": {
    "country_code": "BR"
  },
  "_tags": [
    "bell"
  ],
  "text": "Relendo estes RPGs qu
}
{
  "place": {
    "country_code": "PH"
  },
  "_tags": [
    "bomb"
  ],
  "text": "@kittyhalei24 ahahaha
}
{
  "place": {
    "country_code": "US"
  },
  "_tags": [
    "bug"
  ],
  "text": "ðŸ˜ª https://t.co/sRq
}
```

Shape

The shape of an element is usually a consistent style choice. A deliberate attempt at making something look edgy might have nonsquare corners. This doesn’t mean shape can’t be used effectively to convey meaning; that meaning is just less likely to be intuitive.

When to use it:

As a record-level indicator of something like status or type.

Tip:

Use the CSS `border-radius` property.

Movement

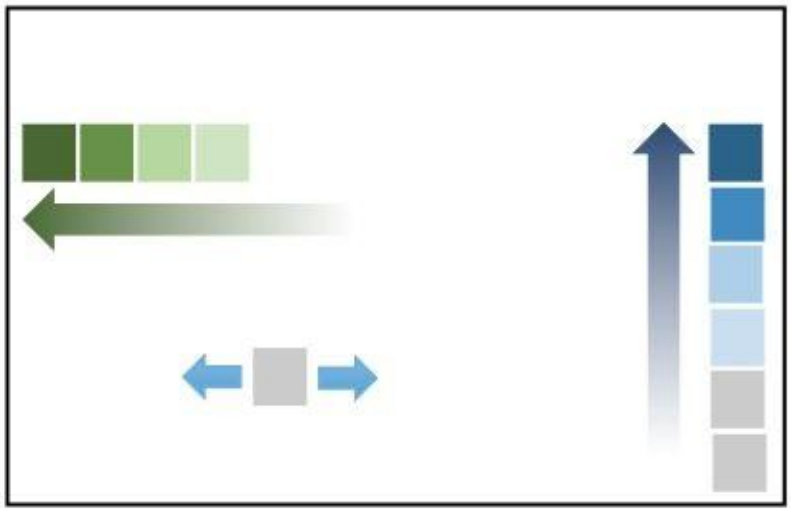
Movement draws a lot of attention initially. After that, there needs to be an instant and clear reason to keep the viewer’s attention. Almost everything can be moving at once as long as the movement has a pattern and purpose that can be intuitively understood. Movements can be between positions or within an existing position (shake).

When to use it:

For transitions and updates.

Tip:

Use CSS animations and transitions, and/or SVG animations.



Streaming Analysis Workflow

Analyzing streaming data has some specific added steps due to its fleeting, rapid-fire nature. It helps to start fresh as often as possible. Remove assumptions, abstractions, and filters, and then reapply them as needed. Doing this gives you a chance to see things that have been covered up, have changed, or have slipped through the cracks. Try to get a good overview of the stats, how those are changing over time, and what data is flowing through before deciding on a path to explore.

Your path may start out as simply looking for something that isn't what you are used to seeing all the time. In this case, you'll want to quickly filter out benign key/value combinations. Usually, a number of fields aren't useful and just get in the way of what you are trying to explore at the time. Hide them for now (you can look at them the next time around) so you can see more of the data you need at once. You might need to go through several iterations of collecting stats to get an idea of how frequent something is or in what range it occurs.

When you come across a theory, you'll need to do a number of things to validate it. This investigation will include things such as comparing it within the context of history, similar time frames, and other values. You'll likely need access to some data outside the original streaming source as well. Eventually, you end up with a report that states your findings with some confidence and lays out the actions recommended based on those findings. The entire workflow is summarized in Figure 9-5.

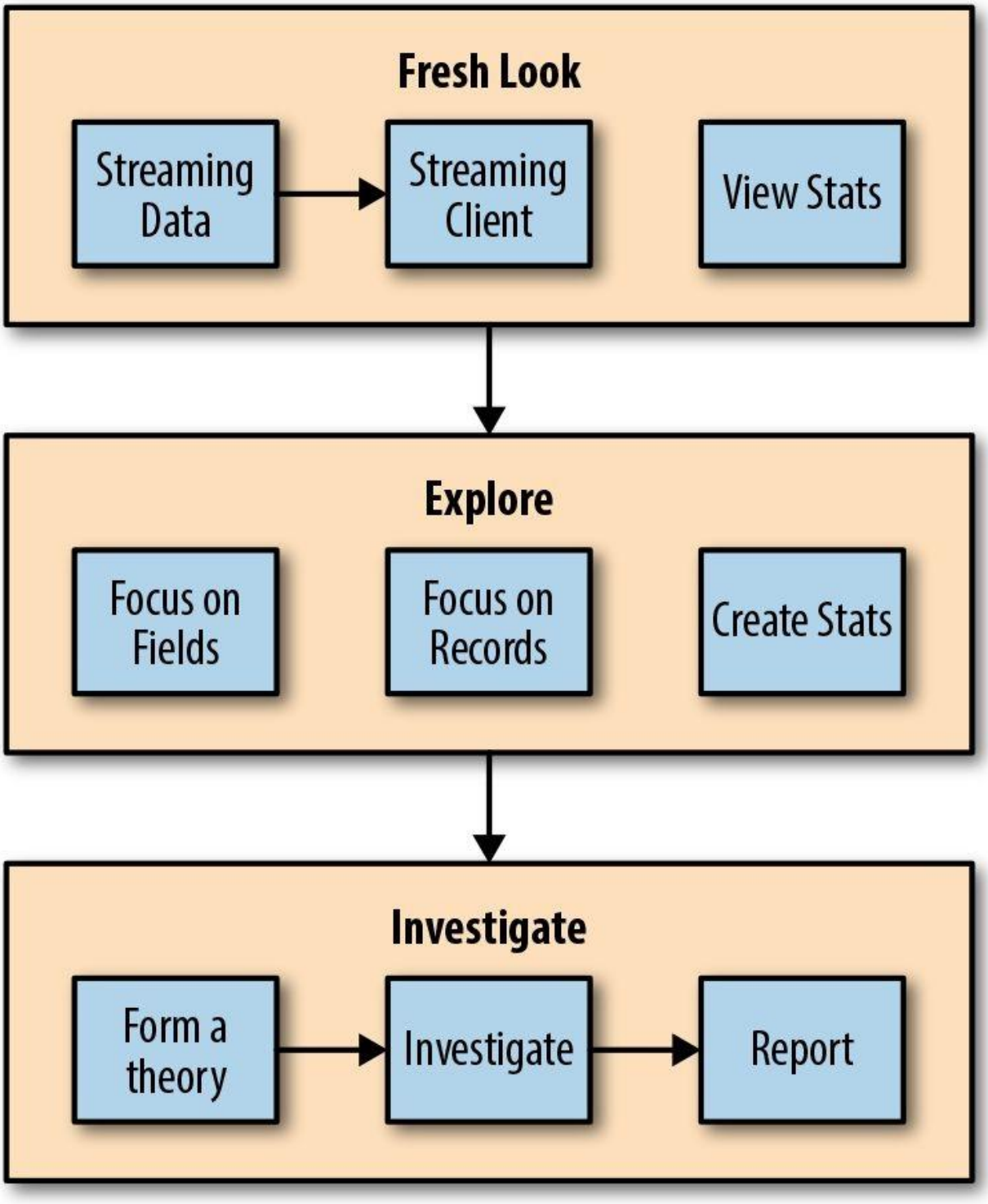


Figure 9-5. A streaming analysis workflow

Context Awareness

Context includes everything you need to know that is an important prerequisite to understanding the data. Add any context information to the visualization that is useful. Some context information will be difficult to bring into a display because it's too far out of scope for the data or hard to quantify (what world events are relevant, the rumor you heard from a different department, etc.). One of the best benefits of having a person analyze data is that they bring with them an understanding of many things that aren't machine-readable. They should be able to understand where this data fits in a larger process and what outside events will correlate with it. If you are looking at operational data for your data center and you can't see the major outage that occurred in the time frame being displayed, something is off. The timestamps might be wrong, or there might be too much cleaning of data for presentation, to the point where it's hiding problems.

Here are some examples of context to bring to analyzing your data:

- Where does this data fit in the larger process?
- What is the data supposed to represent?
- What is its scale compared to what's available and significant?
- What is going on outside the data that should be reflected in it?

Knowing where data fits in the larger process can help you know what to expect. If a lot of black-box processing is done before you see the data, it might prompt you to find answers closer to the source. If a lot of data points are downstream of the data presented, you may need to follow things further. As a developer of data visualizations, try to keep context in mind. It's a good rule to try to require as little manual investigation by the analyst as possible.

Next, what is the data supposed to represent? Does it reach that goal or is it missing elements required to accomplish it? Even if there's a lot missing in the larger context, it's important to know what whatever you're analyzing represents so that you can fill in the rest.

Scale is often an issue because of the technical complexity required to represent large amounts of data. As discussed in "Scaling Data Streams", a reduction method of some sort will be applied, intentional or otherwise. It's important to know what it is. Once it's identified, you might need to find different angles, see reports on the filtered data, or see the full data in smaller time windows or less data over a longer time frame. Scaling the data to be able to present it is OK, but losing sight of the fact that it's scaled and how is not.

Outside context will always be valuable to analysis. When things stand out in the data being presented, it helps to know if there are any known events that may explain them that can then be verified. It's not usually possible to automate bringing in this sort of information, because it can be difficult to define how different events relate to the data. Once identified, the events should be entered in the system and tagged somehow. Once these events are entered as data, they can be analyzed from a broader perspective.

Outliers Example

Outliers are a vague but significant factor to use in analysis of data. In order for them to be quickly and easily recognized by an analyst, it helps to automate as much of their detection as possible and then display the outliers in a prominent fashion to the analyst. Let's look at an example: detecting schema outliers.

The *schema* is the structure of the data. If you typically get a field called `ip` with a matching value in a format like "127.0.0.1," it's good to know if that field gets a new value of "2001:0db8:85a3:0000:0000:8a2e:0370:7334"—this lets you know that something is returning an IPv6 address where you usually get IPv4. It's highly likely that downstream applications receiving this data are designed to handle the data format that existed when it was first developed and will not be able to handle the new data format or detect what it is. One of the first things to check is the fields being sent, as shown in Example 9-1.

Example 9-1. Detecting unexpected data fields

```
var objTemplate={ ip:['ip'], md5:['md5'] }
var fnCompareSchema=function(objMsg){
  var arrKeys=Object.keys(objMsg);
  for(var i=0;i<arrKeys.length;i++){
    if(typeof objTemplate[arrKeys[i]] === 'undefined'){
      // this is a new field, do something about it here
    }
  }
};
```

The values can be a little more complex to detect variations in. In order to cover all of the common format possibilities, it helps to use a language called *regular expressions*. There are whole books dedicated to using regular expressions, and mastering the syntax takes practice. A lot of common patterns can be easily found in places like the Regular Expression Library.

I have compiled a few common patterns and given them names in a library that checks to see what patterns match any given value and returns the possibilities as an array (that's what `['ip']` and `['md5']` mean in Example 9-1). The library can be found on GitHub. Example 9-2 shows an example of running values through the data type regex library to see which ones match.

Example 9-2. Detecting new data type values

```
var objTemplate={ ip:['ip'], md5:['md5'] }
var arrDataTypes=objDataTester.test(objMsg.ip);
for(var i=0;i<arrDataTypes.length;i++){
  if(objTemplate.ip.indexOf(arrDataTypes[i]) === -1){
    // this data type isn't in the template
    console.log('new data type in msg',arrDataTypes[i]);
  }
}
```

Lastly, you may have few enough distinct values that it's worthwhile to detect a new value. Different techniques make sense at different scales of unique values. The code in Example 9-3 will work for thousands of unique values.

Example 9-3. Looking for unique values

```
var arrKnownValues=['127.0.0.1','75.75.75.75','8.8.8.8'];
if(arrKnownValues.indexOf(objMsg.ip) === -1){
  // this value is not in the known array of values
  console.log('new value:',objMsg.ip);
}
```

Don't expect a visualization to make any outliers stand out without some help. Something might be new since yesterday but in all of the data today, and it will look somewhat normal because it's prevalent. It helps to use a visual cue that stands out—like an icon, color, or border. This could be used while showing data or any charts and graphs the data is represented in (see Figure 9-6).

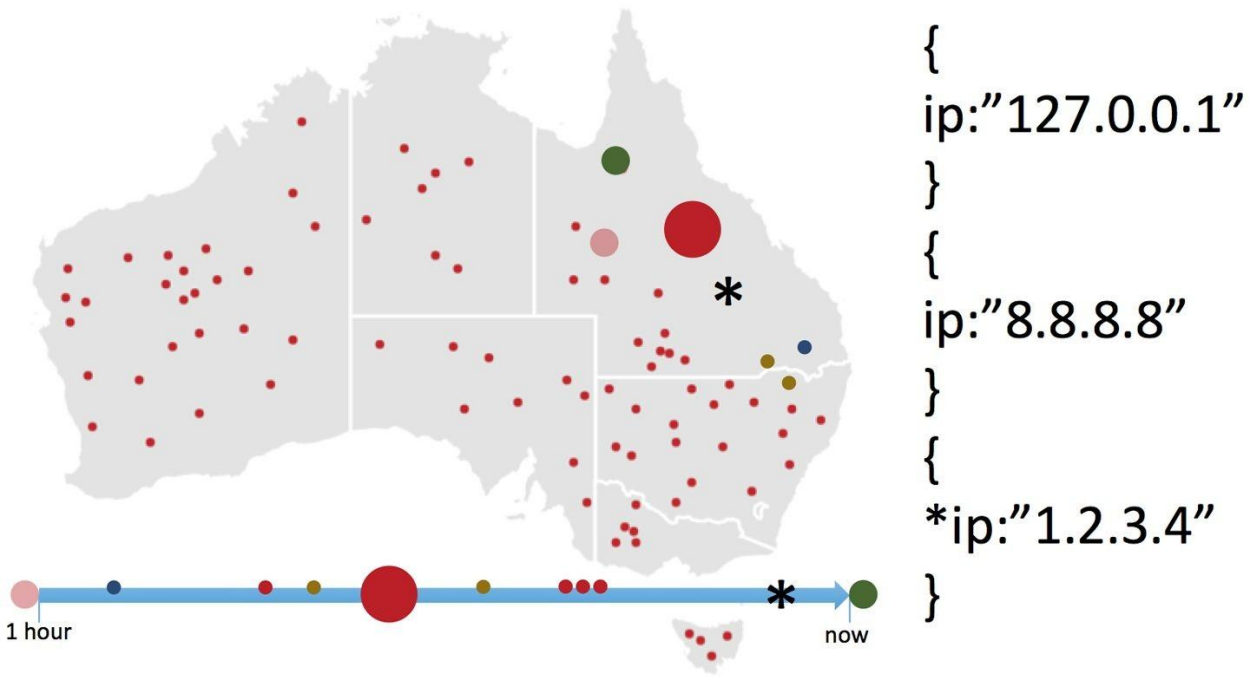


Figure 9-6. Outlier cues can be coordinated across visualization types

This chapter on analysis revolves around seeking answers to questions you may not have asked. It takes some discipline, patience, and an open mind to follow data where it leads you, particularly if it points in an unexpected (or undesirable) direction. Visualizing streaming data enables people to conduct open-ended analysis of data more easily.