Data Science for Startups

Ben G Weber

2018-05-22

Contents

1 Introduction **5**
- 1.1 Why Data Science? 5
- 1.2 Book Overview 6
- 1.3 Tooling 7

2 Tracking Data **9**
- 2.1 What to Record? 10
- 2.2 Tracking Specs 12
- 2.3 Client vs Server Tracking 13
- 2.4 Sending Tracking Events 14
- 2.5 Message Encoding 19
- 2.6 Building a Tracking API 20
- 2.7 Privacy 21
- 2.8 Conclusion 21

3 Data Pipelines **23**
- 3.1 Types of Data 25
- 3.2 The Evolution of Data Pipelines 25
- 3.3 A Scalable Pipeline 31
- 3.4 Conclusion 44

4 Business Intelligence **47**
- 4.1 KPIs 48
- 4.2 Reporting with R 49
- 4.3 ETLs 58
- 4.4 Reporting Tools 61
- 4.5 Conclusion 63

Chapter 1

Introduction

I recently changed industries and joined a startup company where I'm responsible for building up a data science discipline. While we already had a solid data pipeline in place when I joined, we didn't have processes in place for reproducible analysis, scaling up models, and performing experiments. The goal of this book is to provide an overview of how to build a data science platform from scratch for a startup, providing real examples using Google Cloud Platform (GCP) that readers can try out themselves.

This book is intended for data scientists and analysts that want to move beyond the model training stage, and build data pipelines and data products that can be impactful for an organization. However, it could also be useful for other disciplines that want a better understanding of how to work with data scientists to run experiments and build data products. It is intended for readers with programming experience, and will include code examples primarily in R and Java.

1.1 Why Data Science?

One of the first questions to ask when hiring a data scientist for your startup is how will data science improve our product? At Windfall Data, our product is data, and therefore the goal of data science aligns well with the goal of the company, to build the most accurate model for estimating net worth. At other organizations, such as a mobile gaming company, the answer may not be so direct, and data science may be more useful for understanding how to run the business rather than improve products. However, in these early stages it's usually beneficial to start collecting data about customer behavior, so that you can improve products in the future.

Some of the benefits of using data science at a start up are:

- Identifying key business metrics to track and forecast
- Building predictive models of customer behavior
- Running experiments to test product changes
- Building data products that enable new product features

Many organizations get stuck on the first two or three steps, and do not utilize the full potential of data science. A goal of this book is to show how managed services can be used for small teams to move beyond data pipelines for just calculating run-the-business metrics, and transition to an organization where data science provides key input for product development.

1.2 Book Overview

Here are the topics I am covered in this book. Many of these chapters are based on my blog posts on Medium[1].

- **Introduction:** This chapter provides motivation for using data science at a startup and provides an overview of the content covered in this book. Similar posts include functions of data science, scaling data science and my FinTech journey.
- **Tracking Events:** Discusses the motivation for capturing data from applications and web pages, proposes different methods for collecting tracking data, introduces concerns such as privacy and fraud, and presents an example with Google PubSub.
- **Data pipelines:** Presents different approaches for collecting data for use by an analytics and data science team, discusses approaches with flat files, databases, and data lakes, and presents an implementation using PubSub, DataFlow, and BigQuery. Similar posts include a scalable analytics pipeline and the evolution of game analytics platforms.

[1] https://medium.com/@bgweber

- **Business Intelligence:** Identifies common practices for ETLs, automated reports/dashboards and calculating run-the-business metrics and KPIs. Presents an example with R Shiny and Data Studio.
- **Exploratory Analysis:** Covers common analyses used for digging into data such as building histograms and cumulative distribution functions, correlation analysis, and feature importance for linear models. Presents an example analysis with the Natality public data set. Similar posts include clustering the top 1% and 10 years of data science visualizations.
- **Predictive Modeling:** Discusses approaches for supervised and unsupervised learning, and presents churn and cross-promotion predictive models, and methods for evaluating offline model performance.
- **Model Production:** Shows how to scale up offline models to score millions of records, and discusses batch and online approaches for model deployment. Similar posts include Productizing Data Science at Twitch, and Producizting Models with DataFlow.
- **Experimentation:** Provides an introduction to A/B testing for products, discusses how to set up an experimentation framework for running experiments, and presents an example analysis with R and bootstrapping. Similar posts include A/B testing with staged rollouts.
- **Recommendation Systems:** Introduces the basics of recommendation systems and provides an example of scaling up a recommender for a production system. Similar posts include prototyping a recommender.
- **Deep Learning:** Provides a light introduction to data science problems that are best addressed with deep learning, such as flagging chat messages as offensive. Provides examples of prototyping models with the R interface to Keras, and productizing with the R interface to CloudML.

1.3 Tooling

Throughout the book, I'll be presenting code examples built on Google Cloud Platform. I choose this cloud option, because GCP provides a number of managed services that make it possible for small teams to build data pipelines, productize predictive models, and utilize deep learning. It's also possible to sign up for a free trial with GCP and get $300 in credits. This should cover most of the topics presented in this book, but it will quickly expire if your goal is to dive into deep learning on the cloud.

For programming languages, I'll be using R for scripting and Java for production, as well as SQL for working with data in BigQuery. I'll also present other tools such as R Shiny. Some experience with R and Java is recommended, since I won't be covering the basics of these languages.

This book is based on my blog series "Data Science for Startups"[2]. I incorporated feedback from these posts into book chapters, and authored the book using the excellent bookdown package (Xie, 2018).

[2]https://medium.com/p/80d022a18aec

Chapter 2

Tracking Data

In order to make data-driven decisions at a startup, you need to collect data about how your products are being used. You also need to be able to measure the impact of making changes to your product and the efficacy of running campaigns, such as deploying a custom audience for marketing on Facebook. Again, collecting data is necessary for accomplishing these goals.

Usually data is generated directly by the product. For example, a mobile game can generate data points about launching the game, starting additional sessions, and leveling up. But data can also come from other sources, such as an email vendor that provides response data about which users read and click on links within an email. This chapter focuses on the first type of data, where tracking events are being generated by the product.

Why record data about product usage?

- **Track metrics:** You may want to record performance metrics for tracking product health or other metrics useful for running the business.
- **Enable experimentation:** To determine if changes to a product are beneficial, you need to be able to measure results.
- **Build data products:** In order to make something like a recommendation system, you need to know which items users are interacting with.

It's been said that data is the new oil, and there's a wide variety of reasons to collect data from products. When I first started in the gaming industry, data tracked from products was referred to as telemetry. Now, data collected from products is frequently called tracking.

This chapter discusses what type of data to collect about product usage, how to send data to a server for analysis, issues when building a tracking API, and some concerns to consider when tracking user behavior.

2.1 What to Record?

One of the questions to answer when deploying a new product is:

- What data should we collect about user behavior?

The answer is that it depends on your product and intended use cases, but there are some general guidelines about what types of data to collect across most web, mobile, and native applications.

- **Installs:** How big is the user base?
- **Sessions:** How engaged is the user base?
- **Monetization:** How much are users spending?

For these three types of events, the data may actually be generated from three different systems. Installation data might come from a third party, such as Google Play or the App Store, a session start event will be generated from the client application, and spending money in an application, or viewing ads, may be tracked by a different server. As long as you own the service that is generating the data points, you can use the same infrastructure to collect data about different types of events.

Collecting data about how many users launch and log into a application will enable you to answer basic questions about the size of your base, and enable you to track business metrics such as DAU, MAU, ARPDAU, and D-7 retention. However, it doesn't provide much insight into what users are doing within an application, and it doesn't provide many data points that are useful for building data products. In order to better understand user engagement, it's necessary to track data points that are domain or product specific. For example, you might want to track the following types of events in a multiplayer shooter game for consoles:

- **GameStarted:** tracks when the player starts a single or multiplayer game.
- **PlayerSpawn:** tracks when the player spawns into the game world and tracks the class that the user is playing, such as combat medic.
- **PlayerDeath:** tracks where players are dying and getting stuck and enables calculating metrics such as KDR (kill/death ratio).
- **RankUp:** tracks when the player levels up or unlocks a new rank.

Most of these events translate well to other shooter games and other genres such as action/adventure. For a specific game, such as FIFA, you may want to record game specific events, such as:

- **GoalScored:** tracks when a point is scored by the player or opponent.
- **PlayerSubstitution:** tracks when a player is substituted.
- **RedCardReceived:** when the player receives a red card.

Like the prior events, many of these game-specific events can actually be generalized to sports games. If you're a company like EA with a portfolio of different sports titles, it's useful to track all of these events across all of your sports titles (the red card event can be generalized to a penalty event).

If we're able to collect these types of events about players, we can start to answer useful questions about the player base, such as:

- Are users that receive more red cards more likely to quit?
- Do online focused players play more than single-player focused players?
- Do users play the new career mode that was released?

A majority of tracking events are focused on collecting data points about released titles, but it's also possible to collect data during development. At Microsoft Studios, I worked with the user research team to get tracking in place for playtesting. As a result, we could generate visualizations that were useful for conveying to game teams where players were getting stuck. Incorporating these visualizations into the playtesting results resulted in a much better reception from game teams.

When you first add tracking to a product, you won't know of every event and attribute that will be useful to record, but you can make

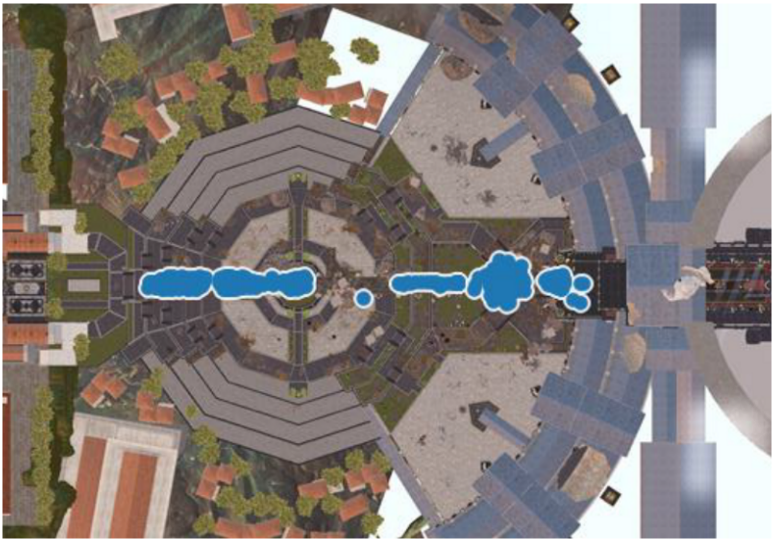

Figure 2.1: Ryse: Son of Rome Playtesting - Microsoft Studios User Research

a good guess by asking team members what types of questions they intend to ask about user behavior and by implementing events that are able to answer these questions. Even with good tracking data, you won't be able to answer every question, but if you have good coverage you can start to improve your products.

2.2 Tracking Specs

Some teams write tracking specifications to in order to define which tracking events need to be implemented in a product. Other teams don't have any documentation and simply take a best guess approach at determining what to record. I highly recommend writing tracking specifications as a best practice. For each event, the spec should identify the conditions for firing an event, the attributes to send, and definitions for any event-specific attributes. For example, a session start event for a web app might have the following form:

- **Condition:** fired when the user first browses to the domain. The event should not be fired when the user clicks on new

pages or uses the back button, but should fire it the user browses to a new domain and then back.
- **Properties:** web browser and version, userID, landing page, referring URL, client timestamp
- **Definitions:** referring URL should list the URL of the page that referred the user to this domain, or the application that referred the user to the web page (e.g. Facebook or Twitter).

Tracking specs are a highly useful piece of documentation. Small teams might be able to get away without having an official process for writing tracking specs, but a number of scenarios can make the documentation critical, such as implementing events on a new platform, re-implementing events for a new backend service, or having engineers leave the team. In order for specs to be useful, it's necessary to answer the following questions:

- Who is responsible for writing the spec?
- Who is responsible for implementing the spec?
- Who is responsible for testing the implementation?

In small organizations, a data scientist might be responsible for all of the aspects of tracking. For a larger organization, it's common for the owners to be a product manager, engineering team, and testing group.

2.3 Client vs Server Tracking

Another consideration when setting up tracking for a product is determining whether to send events from a client application or a backend service. For example, a video-streaming web site can send data about which video a user is watching directly from the web browser, or from the backend service that is serving the video. While there are pros and cons to both approaches, I prefer setting up tracking for backend services rather than client applications if possible. Some of the benefits of server-side tracking are:

- **Trusted Source:** You don't need to expose an endpoint on the web, and you know that events are being generated from your services rather than bots. This helps avoid fraud and DDoS attacks.
- **Avoid Ad Blocking:** If you send data from a client application to an endpoint exposed on the web, some users may

block access to the endpoint, which impacts business metrics.
- **Versioning:** You might need to make changes to an event. You can update your web servers as needed, but often cannot require users to update a client application.

Generating tracking from servers rather than client applications helps avoid issues around fraud, security, and versioning. However, there are some drawbacks to server-side tracking:

- **Testing:** You might need to add new events or modify existing tracking events for testing purposes. This is often easier to do by making changes on the client side.
- **Data availability:** Some of the events that you might want to track do not make calls to a web server. For example, a console game might not connect to any web services during a session start, and instead want until a multiplayer match starts. Also, attributes such as the referring URL may only be available for the client application and not the backend service.

A general guideline is to not trust anything sent by a client application, because often endpoints are not secured and there is no way to verify that the data was generated by your application. But client data is very useful, so it's best to combine both client and server side tracking and to secure endpoints used for collecting tracking from clients.

2.4 Sending Tracking Events

The goal of sending data to a server is to make the data available for analysis and data products. There's a number of different approaches that can be used based on your use case. This section introduces three different ways of sending events to an endpoint on the web and saving the events to local storage. The samples below are not intended to be production code, but instead simple proofs of concept. The next chapter covers building a pipeline for processing events. Code for the samples is available on Github[1].

[1] https://github.com/bgweber/StartupDataScience/

2.4.1 Web Call

The easiest way to set up a tracking service is by making web calls with the event data to a web site. This can be implemented with a lightweight PHP script, which is shown in the code block below.

```
<?php
    $message = $_GET['message'];
    if ($message != '') {
        $dataFile = fopen("telemetry.log", "a");
        fwrite($dataFile, "$message\n");
        fflush($dataFile);
        fclose($dataFile);
    }
?>
```

This php script reads the message parameter from the URL and appends the message to a local file. The script can be invoked by making a web call:

```
http://.../tracking.php?message=Hello_World
```

The call can be made from client or server using the following code:

```
// endpoint
String endPoint = "http://.../tracking.php";

// send the message
String message = "Hello_World";
URL url = new URL(endPoint + "?message=" + message);
URLConnection con = url.openConnection();
BufferedReader in = new BufferedReader(new
    InputStreamReader(con.getInputStream()));

// process the response
while (in.readLine() != null) {}
in.close();
```

This is one of the easiest ways to start collecting tracking data, but it doesn't scale and it's not secure. It's useful for testing, but should be avoided for anything customer facing. I did use this approach in the past to collect data about players for a Mario level generator experiment[2].

2.4.2 Web Server

Another approach you can use is setting up a web service to collect tracking events. The code below shows how to use Jetty to set up a lightweight service for collecting data. In order to compile and run the example, you'll need to include the following pom file. The first step is to start a web service that will handle tracking requests:

```
public class TrackingServer extends AbstractHandler {
  public static void main(String[] args) {
    Server server = new Server(8080);
    server.setHandler(new TrackingServer());
    server.start();
    server.join();
  }

  public void handle(String target,
    Request baseRequest, HttpServletRequest request,
    HttpServletResponse response)
    throws IOException, ServletException {

    // Process Request
  }
}
```

In order to process events, the application reads the message parameter from the web request, appends the message to a local file, and then responds to the web request.

```
// append the event data to a local file
String message = baseRequest.getParameter("message");
if (message != null) {
  BufferedWriter writer = new BufferedWriter(
      new FileWriter("tracking.log", true));
```

[2] http://www.gamasutra.com/blogs/BenWeber/20131228/207819/

```
  writer.write(message + "\n");
  writer.close();
}

// service the web request
response.setStatus(HttpServletResponse.SC_OK);
```

In order to call the endpoint with Java, we'll need to modify the URL:

```
URL url = new URL(
    "http://localhost:8080/?message=" + message);
```

This approach can scale a bit more than the PHP approach, but is still insecure and not the best approach for building a production system. My advice for building a production ready tracking service is to use a stream processing system such as Kafka, Amazon Kinesis, or Google's PubSub.

2.4.3 Subscription Service

Using messaging services such as PubSub enables systems to collect massive amounts of tracking data, and forward the data to a number of different consumers. Some systems such as Kafka require setting up and maintaining servers, while other approaches like PubSub are managed services that are serverless. Managed services are great for startups, because they reduce the amount of DevOps support needed. But the tradeoff is cost, and it's pricer to use managed services for massive data collection.

The code below shows how to use Java to post a message to a topic on PubSub. In order to run this example, you'll need to set up a free google cloud project, and enable PubSub. More details on setting up GCP and PubSub are available online[3].

```
// Set up a publisher
TopicName topicName =
    TopicName.of("projectID", "raw-events");
```

[3]https://medium.com/p/4087b66952a1

```
Publisher publisher = Publisher
    .newBuilder(topicName).build();

//schedule a message to be published
String message = "Hello World!";
PubsubMessage msg = PubsubMessage
    .newBuilder().setData(ByteString
    .copyFromUtf8(message)).build();

// publish the message, add a callback listener
ApiFuture<String> future = publisher.publish(msg);
ApiFutures.addCallback(future,
    new ApiFutureCallback<String>() {

  public void onFailure(Throwable arg0) {}
  public void onSuccess(String arg0) {}
});

publisher.shutdown();
```

This code example shows how to send a single message to PubSub for recording a tracking event. For a production system, you'll want to implement the onFailure method in order to deal with failed deliveries. The code above shows how to send a message with Java, while other languages are supported including Go, Python, C#, and PHP. It's also possible to interface with other stream processing systems such as Kafka.

The next code segment shows how to read a message from PubSub and append the message to a local file.In the next chapter I'll show how to consume messages using DataFlow.

```
// set up a message handler
MessageReceiver receiver = new MessageReceiver() {
  public void receiveMessage(PubsubMessage message,
    AckReplyConsumer consumer) {
    try {
      BufferedWriter writer = new BufferedWriter(new
        FileWriter("tracking.log", true));
      writer.write(
          message.getData().toStringUtf8() + "\n");
```

```
      writer.close();
      consumer.ack();
    }
    catch (Exception e) {}
}};

// start the listener for 1 minute
SubscriptionName subscriptionName =
    SubscriptionName.of("projectId", "raw-events");
Subscriber subscriber = Subscriber.newBuilder(
    subscriptionName, receiver).build();

subscriber.startAsync();
Thread.sleep(60000);
subscriber.stopAsync();
```

We now have a way of getting data from client applications and backend services to a central location for analysis. The last approach shown is a scalable and secure method for collecting tracking data, and is a managed service making it a good fit for startups with small data teams.

2.5 Message Encoding

One of the decisions to make when sending data to an endpoint for collection is how to encode the messages being sent, since all events that are sent from an application to an endpoint need to be serialized. When sending data over the internet, it's good to avoid language specific encodings, such as Java serialization, because the application and backend services are likely implemented in different languages. There's also versioning issues that can arise when using a language-specific serialization approach.

Some common ways of encoding tracking events are using the JSON format and Google's protocol buffers. JSON has the benefit of being human readable and supported by a wide variety of languages, while buffers provide better compression and may better suited for certain data structures. One of the benefits of using these approaches is that a schema does not need to be defined before you can send events, since metadata about the event is included in the message. You can add new attributes as needed, and even change data types, but this may impact downstream event processing.

When getting started with building a data pipeline, I'd recommended using JSON to get started, since it's human readable and supported by a wide variety of languages. It's also good to avoid encodings such as pipe-delimited formats, because you many need to support more complex data structures, such as lists or maps, when you update your tracking events. Here's an example of what a message might look like:

```
# JSON
{"Type":"Session","Version":1.0,"UserID":"12345"
    ,"Platform":"iOS"}

# Pipe delimited
Session|1.0|12345|iOS
```

What about XML? No!

2.6 Building a Tracking API

To build a production system, you'll need to add a bit more sophistication to your tracking code. A production system should handle the following issues:

- **Delivery Failures:** if a message delivery fails, the system should retry sending the message, and have a backoff mechanism.
- **Queueing:** if the endpoint is not available, such as a phone without a signal, the trackling library should be able to store events for later transmission, such as when wifi is available.
- **Batching:** instead of sending a large number of small requests, it's often useful to send batches of tracking events.
- **Prioritization:** some messages are more important to track than others, such as preferring monetization events over click events. A tracking library should be able to prioritize more critical events.

It's also useful to have a process in place for disabling tracking events. I've seen data pipelines explode from client applications sending way too much data, and there was no way of disabling the clients from sending the problematic event without turning off all tracking.

Ideally, a production level system should have some sort of auditing in place, in order to validate that the endpoints are receiving all of the data being sent. One approach is to send data to a different endpoint built on a different infrastructure and tracking library, but that much redundancy is usually overkill. A more lightweight approach is to add a sequential counting attribute to all events, so if a client sends 100 messages, the backend can use this attribute to know how many events the client attempted to send and validate the result.

2.7 Privacy

There's privacy concerns to consider when storing user data. When data is being made available to analytics and data science teams, all personally identifiable information (PII) should be stripped from events, which can include names, addresses, and phone numbers. In some instances, user names, such as a player's gamertag on Steam, may be considered PII as well. It's also good to strip IP addresses from any data being collected, to limit privacy concerns. The general recommendation is to collect as much behavioral data as needed to answer questions about product usage, while avoiding the need to collect sensitive information, such as gender and age. If you're building a product based on sensitive information, you should have strong user access controls in place to limit access to sensitive data. Policies such GDPR are setting new regulations for collecting and processing data, and GDPR should be reviewed before shipping a product with tracking.

2.8 Conclusion

Tracking data enables teams to answer a variety of questions about product usage, enables teams to track the performance and health of products, and can be used to build data products. This chapter discussed some of the issues involved in collecting data about user behavior, and provided examples for how to send data from a client application to an endpoint for later analysis. Here are the key takeaways to from this chapter:

- Use server-side tracking if possible. It helps avoid a wide variety of issues.
- QA/test your tracking events. If you're sending bad data, you may be drawing incorrect conclusions from your data.
- Have a versioning system in place. You'll need to add new events and modify existing events, and this should be a simple process.
- Use JSON for sending events. It's human readable, extensible, and supported by a wide variety of languages
- Use managed services for collecting data. You won't need to spin up servers and can collect huge amounts of data.

As you ship more products and scale up your user base, you may need to change to a different data collection platform, but this advice is a good starting point for shipping products with tracking.

Chapter 3

Data Pipelines

Building data pipelines is a core component of data science at a startup. In order to build data products, you need to be able to collect data points from millions of users and process the results in near real-time. While the previous chapter discussed what type of data to collect and how to send data to an endpoint, this chapter will discuss how to process data that has been collected, enabling data scientists to work with the data. The chapter on model production will discuss how to deploy models on this data platform.

Typically, the destination for a data pipeline is a data lake, such as Hadoop or parquet files on S3, or a relational database, such as Redshift. An data pipeline should have the following properties:

- **Low Event Latency:** Data scientists should be able to query recent event data in the pipeline, within minutes or seconds of the event being sent to the data collection endpoint. This is useful for testing purposes and for building data products that need to update in near real-time.
- **Scalability:** A data pipeline should be able to scale to billions of data points, and potentially trillions as a product scales. A high performing system should not only be able to store this data, but make the complete data set available for querying.
- **Interactive Querying:** A high functioning data pipeline should support both long-running batch queries and smaller interactive queries that enable data scientists to explore tables and understand the schema without having to wait min-

utes or hours when sampling data.
- **Versioning:** You should be able to make changes to your data pipeline and event definitions without bringing down the pipeline and suffering data loss. This chapter will discuss how to build a pipeline that supports different event definitions, in the case of changing an event schema.
- **Monitoring:** If an event is no longer being received, or tracking data is no longer being received for a particular region, then the data pipeline should generate alerts through tools such as PagerDuty.
- **Testing:** You should be able to test your data pipeline with test events that do not end up in your data lake or database, but that do test components in the pipeline.

There's a number of other useful properties that a data pipeline should have, but this is a good starting point for a startup. As you start to build additional components that depend on your data pipeline, you'll want to set up tooling for fault tolerance and automating tasks.

This chapter will show how to set up a scalable data pipeline that sends tracking data to a data lake, database, and subscription service for use in data products. I'll discuss the different types of data in a pipeline, the evolution of data pipelines, and walk through an example pipeline implemented on GCP.

Before deploying a data pipeline, you'll want to answer the following questions, which resemble our questions about tracking specs:

- Who owns the data pipeline?
- Which teams will be consuming data?
- Who will QA the pipeline?

In a small organization, a data scientist may be responsible for the pipeline, while larger organizations usually have an infrastructure team that is responsible for keeping the pipeline operational. It's also useful to know which teams will be consuming the data, so that you can stream data to the appropriate teams. For example, marketing may need real-time data of landing page visits to perform attribution for marketing campaigns. And finally, the data quality of events passed to the pipeline should be thoroughly inspected on a regular basis. Sometimes a product update will cause a tracking event to drop relevant data, and a process should be set up to capture these types of changes in data.

3.1 Types of Data

Data in a pipeline is often referred to by different names based on the amount of modification that has been performed. Data is typically classified with the following labels:

- **Raw:** Is tracking data with no processing applied. This is data stored in the message encoding format used to send tracking events, such as JSON. Raw data does not yet have a schema applied. It's common to send all tracking events as raw events, because all events can be sent to a single endpoint and schemas can be applied later on in the pipeline.
- **Processed:** Processed data is raw data that has been decoded into event specific formats, with a schema applied. For example, JSON tracking events that have been translated into a session start events with a fixed schema are considered processed data. Processed events are usually stored in different event tables/destinations in a data pipeline.
- **Cooked:** Processed data that has been aggregated or summarized is referred to as cooked data. For example, processed data could include session start and session end events, and be used as input to cooked data that summarizes daily activity for a user, such as number of sessions and total time on site for a web page.

Data scientists will typically work with processed data, and use tools to create cooked data for other teams. This chapter discusses how to build a data pipeline that outputs processed data, while the Business Intelligence chapter will discuss how to add cooked data to your pipeline.

3.2 The Evolution of Data Pipelines

Over the past two decades the landscape for collecting and analyzing data has changed significantly. Rather than storing data locally via log files, modern systems can track activity and apply machine learning in near real-time. Startups might want to use one of the earlier approaches for initial testing, but should really look to more recent approaches for building data pipelines. Based on my experience, I've noticed four different approaches to pipelines:

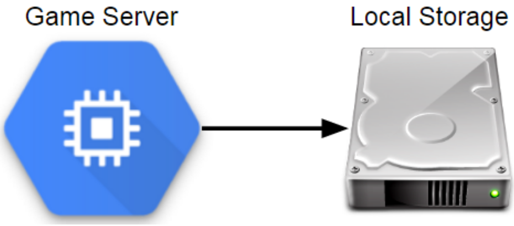

Figure 3.1: Components in a pre-database Analytics Architecture

- **Flat File Era:** Data is saved locally on game servers
- **Database Era:** Data is staged in flat files and then loaded into a database
- **Data Lake Era:** Data is stored in Hadoop/S3 and then loaded into a DB
- **Serverless Era:** Managed services are used for storage and querying

Each of the steps in this evolution support the collection of larger data sets, but may introduce additional operational complexity. For a startup, the goal is to be able to scale data collection without scaling operational resources, and the progression to managed services provides a nice solution for growth.

The data pipeline that we'll walk through in the next section of this chapter is based on the most recent era of data pipelines, but it's useful to walk through different approaches because the requirements for different companies may fit better with different architectures.

3.2.1 Flat File Era

I got started in data science at Electronic Arts in 2010, before EA had an organization built around data. While many game companies were already collecting massive amounts of data about gameplay, most telemetry was stored in the form of log files or other flat file formats that were stored locally on the game servers. Nothing could be queried directly, and calculating basic metrics such as monthly active users (MAU) took substantial effort.

At Electronic Arts, a replay feature was built into Madden NFL 11 which provided an unexpected source of game telemetry. After every game, a game summary in an XML format was sent to a game server that listed each play called, moves taken during the play, and the result of the down. This resulted in millions of files that could be analyzed to learn more about how players interacted with Madden football in the wild.

Storing data locally is by far the easiest approach to take when collecting gameplay data. For example, the PHP approach presented in the last chapter is useful for setting up a lightweight analytics endpoint. But this approach does have significant drawbacks.

This approach is simple and enables teams to save data in whatever format is needed, but has no fault tolerance, does not store data in a central location, has significant latency in data availability, and has standard tooling for building an ecosystem for analysis. Flat files can work fine if you only have a few servers, but it's not really an analytics pipeline unless you move the files to a central location. You can write scripts to pull data from log servers to a central location, but it's not generally a scalable approach.

3.2.2 Database Era

While I was at Sony Online Entertainment, we had game servers save event files to a central file server every couple of minutes. The file server then ran an ETL process about once an hour that fast loaded these event files into our analytics database, which was Vertica at the time. This process had a reasonable latency, about one hour from a game client sending an event to the data being queryable in our analytics database. It also scaled to a large volume of data, but required using a fixed schema for event data.

When I was a Twitch, we used a similar process for one of our analytics databases. The main difference from the approach at SOE was that instead of having game servers scp files to a central location, we used Amazon Kinesis to stream events from servers to a staging area on S3. We then used an ETL process to fast load data into Redshift for analysis. Since then, Twitch has shifted to a data lake approach, in order to scale to a larger volume of data and to provide more options for querying the datasets.

The databases used at SOE and Twitch were immensely valuable for both of the companies, but we did run into challenges as we scaled the amount of data stored. As we collected more detailed

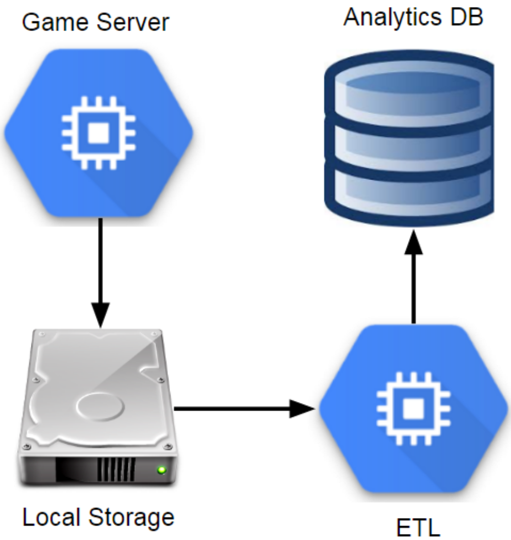

Figure 3.2: Components in an ETL-based Analytics Architecture

information about gameplay, we could no longer keep complete event history in our tables and needed to truncate data older than a few months. This is fine if you can set up summary tables that maintain the most important details about these events, but it's not an ideal situation.

One of the issues with this approach is that the staging server becomes a central point of failure. It's also possible for bottlenecks to arise where one game sends way too many events, causing events to be dropped across all of the titles. Another issue is query performance as you scale up the number of analysts working with the database. A team of a few analysts working with a few months of gameplay data may work fine, but after collecting years of data and growing the number of analysts, query performance can be a significant problem, causing some queries to take hours to complete.

The main benefits of this approach are that all event data is available in a single location queryable with SQL and great tooling is available, such as Tableau and DataGrip, for working with relational databases. The drawbacks are that it's expensive to keep all data loaded into a database like Vertica or Redshift, events needs to have a fixed schema, and truncating tables may be necessary.

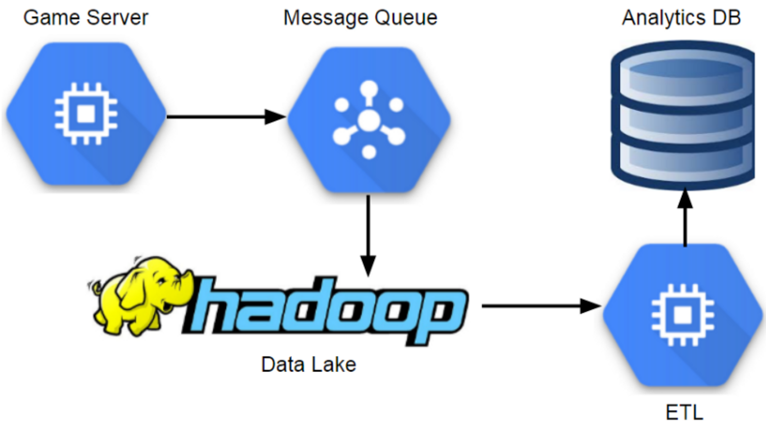

Figure 3.3: Components in a Data Lake Analytics Architecture

Another issue with using a database as the main interface for data is that machine learning tools such as Spark's MLlib cannot be used effectively, since the relevant data needs to be unloaded from the database before it can be operated on. One of the ways of overcoming this limitation is to store gameplay data in a format and storage layer that works well with Big Data tools, such as saving events as Parquet files on S3. This type of configuration became more population in the next era, and gets around the limitations of needing to truncate tables and the reduces the cost of keeping all data.

3.2.3 Data Lake Era

The data storage pattern that was most common while I was working as a data scientist in the games industry was a data lake. The general pattern is to store semi-structured data in a distributed database, and run ETL processes to extract the most relevant data to analytics databases. A number of different tools can be used for the distributed database: at Electronic Arts we used Hadoop, at Microsoft Studios we used Cosmos, and at Twitch we used S3.

This approach enables teams to scale to massive volumes of data, and provides additional fault tolerance. The main downside is that it introduces additional complexity, and can result in analysts having access to less data than if a traditional database approach was used, due to lack of tooling or access policies. Most analysts will interact with data in the same way in this model, using an analytics database populated from data lake ETLs.

One of the benefits of this approach is that it supports a variety of different event schemas, and you can change the attributes of an event without impacting the analytics database. Another advantage is that analytics teams can use tools such as Spark SQL to work with the data lake directly. However, most places I worked at restricted access to the data lake, eliminating many of the benefits of this model.

This approach scales to a massive amount of data, supports flexible event schemas, and provides a good solution for long-running batch queries. The down sides are that it may involve significant operational overhead, may introduce large event latencies, and may lack mature tooling for the end users of the data lake. An additional drawback with this approach is that usually a whole team is needed just to keep the system operational. This makes sense for large organizations, but may be overkill for smaller companies. One of the ways of taking advantage of using a data lake without the cost of operational overhead is by using managed services.

3.2.4 Serverless Era

In the current era, analytics platforms incorporate a number of managed services, which enable teams to work with data in near real-time, scale up systems as necessary, and reduce the overhead of maintaining servers. I never experienced this era while I was working in the game industry, but saw signs of this transition happening. Riot Games is using Spark[1] for ETL processes and machine learning, and needed to spin up infrastructure on demand. Some game teams are using elastic computing methods for game services, and it makes sense to utilize this approach for analytics as well.

This approach has many of the same benefits as using a data lake, autoscales based on query and storage needs, and has minimal operational overhead. The main drawbacks are that managed services can be expensive, and taking this approach will likely result in using platform specific tools that are not portable to other cloud providers.

[1] https://databricks.com/customers/riot-games

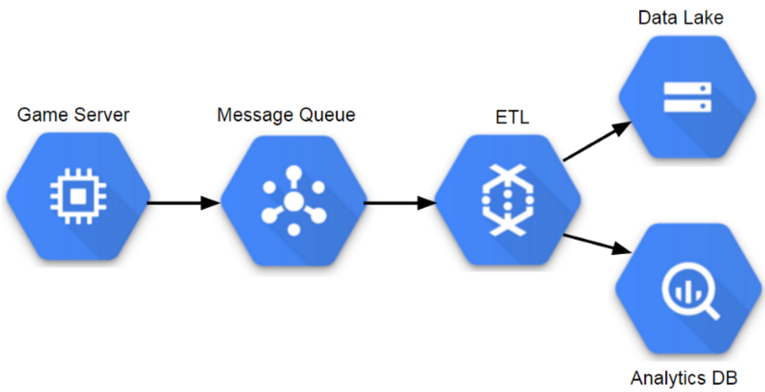

Figure 3.4: Components in a managed Analytics Architecture

In my career I had the most success working with the database era approach, since it provided the analytics team with access to all of the relevant data. However, it wasn't a setup that would continue to scale and most teams that I worked on have since moved to data lake environments. In order for a data lake environment to be successful, analytics teams need access to the underlying data, and mature tooling to support their processes. For a startup, the serverless approach is usually the best way to start building a data pipeline, because it can scale to match demand and requires minimal staff to maintain the data pipeline. The next section will walk through building a sample pipeline with managed services.

3.3 A Scalable Pipeline

We'll build a data pipeline that receives events using Google's PuSub as an endpoint, and save the events to a data lake and database. The approach presented here will save the events as raw data, but I'll also discuss ways of transforming the events into processed data.

The data pipeline that performs all of this functionality is relatively simple. The pipeline reads messages from PubSub and then transforms the events for persistence: the BigQuery portion of the pipeline converts messages to TableRow objects and streams directly to BigQuery, while the AVRO portion of the pipeline batches

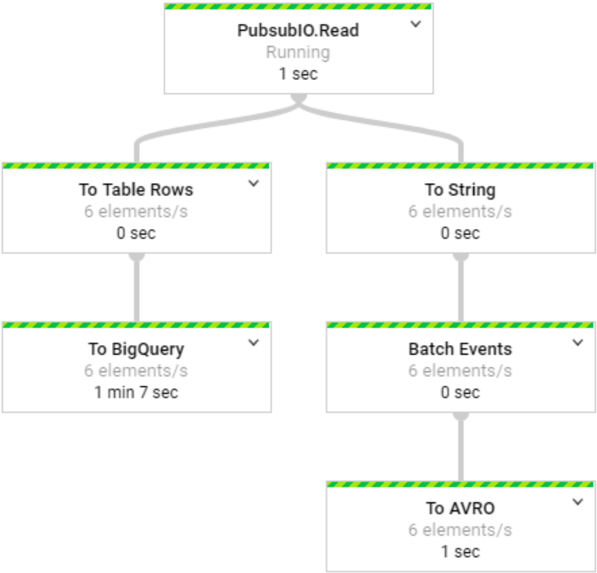

Figure 3.5: The streaming pipeline deployed to Google Cloud

events into discrete windows and then saves the events to Google Storage. The graph of operations is shown in the figure above.

3.3.1 Setting up the Environment

The first step in building a data pipeline is setting up the dependencies necessary to compile and deploy the project. I used the following maven dependencies to set up environments for the tracking API that sends events to the pipeline, and the data pipeline that processes events.

```
<!-- Dependencies for the Tracking API ->
<dependency>
  <groupId>com.google.cloud</groupId>
  <artifactId>google-cloud-pubsub</artifactId>
  <version>0.32.0-beta</version>
  </dependency>
</dependencies>
```

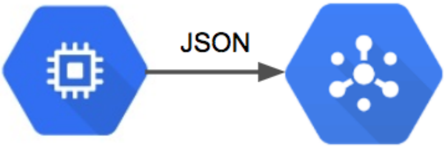

Figure 3.6: Sending events from a server to a PubSub topic

```
<!-- Dependencies for the data pipeline ->
<dependency>
  <groupId>com.google.cloud.dataflow</groupId>
  <artifactId>google-cloud-dataflow-java-sdk-all
      </artifactId>
  <version>2.2.0</version>
</dependency>
```

I used Eclipse to author and compile the code for this tutorial, since it is open source. However, other IDEs such as IntelliJ provide additional features for deploying and monitoring DataFlow tasks. Before you can deploy jobs to Google Cloud, you'll need to set up a service account for both PubSub and DataFlow. Setting up these credentials is outside the scope of this book, and more details are available in the Google documentation[2].

An additional prerequisite for running this data pipeline is setting up a PubSub topic on GCP. I defined a raw-events topic that is used for publishing and consuming messages for the data pipeline. Additional details on creating a PubSub topic are available online[3].

To deploy this data pipeline, you'll need to set up a java environment with the maven dependencies listed above, set up a Google Cloud project and enable billing, enable billing on the storage and BigQuery services, and create a PubSub topic for sending and receiving messages. All of these managed services do cost money, but there is a free tier that can be used for prototyping a data pipeline.

[2]https://cloud.google.com/bigquery/docs/authentication
[3]https://cloud.google.com/pubsub/docs/quickstart-console

3.3.2 Publishing Events

In order to build a usable data pipeline, it's useful to build APIs that encapsulate the details of sending event data. The Tracking API class provides this functionality, and can be used to send generated event data to the data pipeline. The code below shows the method signature for sending events, and shows how to generate sample data.

```
// send a batch of events
for (int i=0; i<10000; i++) {

  // generate event names
  String type = Math.random() < 0.5 ? "Session"
   (Math.random() < 0.5 ? "Login" : "MatchStart");

  // create attributes to send
  HashMap attributes = new HashMap();
  attributes.put("userID", (int)(Math.random()*100));
  attributes.put("deviceType", Math.random() < 0.5 ?
    "Android" : (Math.random() < 0.5 ? "iOS" : "Web"));

  // send the event
  tracking.sendEvent(type, "V1", attributes);
}
```

The tracking API establishes a connection to a PubSub topic, passes events as a JSON format, and implements a callback for notification of delivery failures. The code used to send events is provided below, and is based on Google's PubSub example.

```
// Setup a PubSub connection
TopicName topicName = TopicName.of(projectID, topicID);
Publisher publisher = Publisher
    .newBuilder(topicName).build();

// Specify an event to send
String event =
    {\"type\":\"session\",\"version\":\"1\"}";
```

Figure 3.7: Streaming event data from PubSub to DataFlow

```
// Convert the event to bytes
ByteString data = ByteString
    .copyFromUtf8(event.toString());

//schedule a message to be published
PubsubMessage msg =
  PubsubMessage.newBuilder().setData(data).build();

// publish the message
ApiFuture<String> future = publisher.publish(msg);
ApiFutures.addCallback(future, this);
```

The code above enables apps to send events to a PubSub topic. The next step is to process this events in a fully-managed environment that can scale as necessary to meet demand.

3.3.3 Storing Events

One of the key functions of a data pipeline is to make instrumented events available to data science and analytics teams for analysis. The data sources used as endpoints should have low latency and be able to scale up to a massive volume of events. The data pipeline defined in this tutorial shows how to output events to both BigQuery and a data lake that can be used to support a large number of analytics business users.

The first step in this data pipeline is reading events from a PubSub topic and passing ingested messages to the DataFlow process. DataFlow provides a PubSub connector that enables streaming of PubSub messages to other DataFlow components. The code below shows how to instantiate the data pipeline, specify streaming

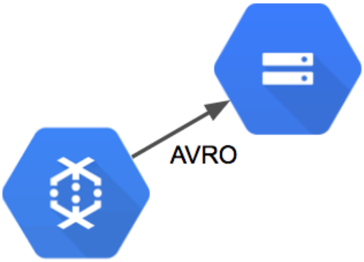

Figure 3.8: Batching events to AVRO format and saving to GS.

mode, and to consume messages from a specific PubSub topic. The output of this process is a collection of PubSub messages that can be stored for later analysis.

```
// set up pipeline options
Options options = PipelineOptionsFactory.fromArgs(args)
  .withValidation().as(Options.class);
options.setStreaming(true);
Pipeline pipeline = Pipeline.create(options);

// read game events from PubSub
PCollection<PubsubMessage> events = pipeline
  .apply(PubsubIO.readMessages().fromTopic(topic));
```

The first way we want to store events is in a columnar format that can be used to build a data lake. While this chapter doesn't show how to utilize these files in downstream ETLs, having a data lake is a great way to maintain a copy of your data set in case you need to make changes to your database. The data lake provides a way to backload your data if necessary due to changes in schemas or data ingestion issues. The portion of the data pipeline allocated to this process is shown below.

For AVRO, we can't use a direct streaming approach. We need to group events into batches before we can save to flat files. The way this can be accomplished in DataFlow is by applying a windowing function that groups events into fixed batches. The code below applies transformations that convert the PubSub messages into String objects, group the messages into 5 minute intervals, and output the resulting batches to AVRO files on Google Storage. To summarize, the code batches events into 5 minute windows and then exports the events to AVRO files on Google Storage.

```
// AVRO output portion of the pipeline
events.apply("To String",
    ParDo.of(new DoFn<PubsubMessage, String>() {

  @ProcessElement
  public void processElement(ProcessContext c)  {
    String msg = new String(c.element().getPayload());
    c.output(msg);
  }
}))

// Batch events into 5 minute windows
.apply("Batch Events", Window.<String>into(
    FixedWindows.of(Duration.standardMinutes(5)))
  .triggering(AfterWatermark.pastEndOfWindow())
  .discardingFiredPanes()
  .withAllowedLateness(Duration.standardMinutes(5)))

// Save the events in ARVO format
.apply("To AVRO", AvroIO.write(String.class)
  .to("gs://your_gs_bucket/avro/raw-events.avro")
  .withWindowedWrites()
  .withSuffix(".avro"));
```

The result of this portion of the data pipeline is a collection of AVRO files on google storage that can be used to build a data lake. A new AVRO output is generated every 5 minutes, and downstream ETLs can parse the raw events into processed event-specific table schemas. The image below shows a sample output of AVRO files.

In addition to creating a data lake, we want the events to be immediately accessible in a query environment. DataFlow provides a BigQuery connector which serves this functionality, and data streamed to this endpoint is available for analysis after a short duration. This portion of the data pipeline is shown below.

The data pipeline converts the PubSub messages into TableRow objects, which can be directly inserted into BigQuery. The code

	Name	Size	Type
	raw-events.avro2018-03-27T04:02:00.000Z-2018-03-27...	2.46 KB	application/octet-stream
	raw-events.avro2018-03-27T04:02:00.000Z-2018-03-27...	2.43 KB	application/octet-stream
	raw-events.avro2018-03-27T04:03:00.000Z-2018-03-27...	2.5 KB	application/octet-stream
	raw-events.avro2018-03-27T04:03:00.000Z-2018-03-27...	2.52 KB	application/octet-stream
	raw-events.avro2018-03-27T04:04:00.000Z-2018-03-27...	2.96 KB	application/octet-stream
	raw-events.avro2018-03-27T04:04:00.000Z-2018-03-27...	2.99 KB	application/octet-stream
	raw-events.avro2018-03-27T04:05:00.000Z-2018-03-27...	2.41 KB	application/octet-stream
	raw-events.avro2018-03-27T04:05:00.000Z-2018-03-27...	2.38 KB	application/octet-stream

Figure 3.9: AVRO files saved to Google Storage

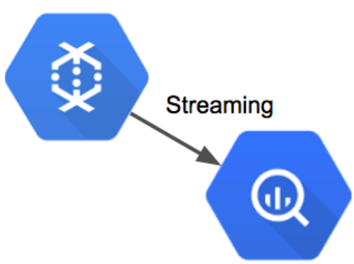

Figure 3.10: Streaming events from DataFlow to BigQuery

below consists of two apply methods: a data transformation and a IO writer. The transform step reads the message payloads from PubSub, parses the message as a JSON object, extracts the eventType and eventVersion attributes, and creates a TableRow object with these attributes in addition to a timestamp and the message payload. The second apply method tells the pipeline to write the records to BigQuery and to append the events to an existing table.

```
// parse the PubSub events and create rows
events.apply("To Table Rows", ParDo.of(new
          DoFn<PubsubMessage, TableRow>() {

 @ProcessElement
 public void processElement(ProcessContext c) {
   String msg = new String(c.element().getPayload());

   // parse the json message for attributes
   JsonObject jsonObject =
      new JsonParser().parse(msg).getAsJsonObject();
   String type = jsonObject.get("type").getAsString();
   String eventVersion = jsonObject.
       get("eventVersion").getAsString();
   String serverTime = dateFormat.format(new Date());

   // create and output the table row
   TableRow record = new TableRow();
   record.set("eventType", type);
   record.set("eventVersion", eventVersion);
   record.set("serverTime", serverTime);
   record.set("message", message);
   c.output(record);
  }
}))
//stream the events to Big Query
.apply("To BigQuery",BigQueryIO.writeTableRows()
  .to(table)
  .withSchema(schema)
  .withCreateDisposition(
    CreateDisposition.CREATE_IF_NEEDED)
  .withWriteDisposition(WriteDisposition.WRITE_APPEND))
```

Row	eventType	eventVersion	server_time	message
1	Login	V1	2018-03-25 19:18:56.720	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"Web"}
2	Login	V1	2018-03-25 19:19:27.131	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"Web"}
3	Login	V1	2018-03-25 20:35:27.784	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"Web"}
4	Login	V1	2018-03-25 20:36:35.106	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"Web"}
5	Login	V1	2018-03-25 20:40:49.629	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"Web"}
6	Login	V1	2018-03-25 20:35:21.833	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"Web"}
7	Login	V1	2018-03-25 20:35:25.961	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"iOS"}
8	Login	V1	2018-03-25 20:36:20.849	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"iOS"}
9	Login	V1	2018-03-25 20:36:41.328	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"iOS"}
10	Login	V1	2018-03-25 20:36:24.757	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"iOS"}

Figure 3.11: Game event records queried from the raw-events table in BigQuery

Each message that is consumed from PubSub is converted into a TableRow object with a timestamp and then streamed to BigQuery for storage. The result of this portion of the data pipeline is that events will be streamed to BigQuery and will be available for analysis in the output table specified by the DataFlow task. In order to effectively use these events for queries, you'll need to build additional ETLs for creating processed event tables with schematized records, but you now have a data collection mechanism in place for storing tracking events.

3.3.4 Deploying and Auto Scaling

With DataFlow you can test the data pipeline locally or deploy to the cloud. If you run the code samples without specifying additional attributes, then the data pipeline will execute on your local machine. In order to deploy to the cloud and take advantage of the auto scaling capabilities of this data pipeline, you need to specify a new runner class as part of your runtime arguments. In order to run the data pipeline, I used the following runtime arguments:

```
--runner=org.apache.beam.runners.dataflow.DataflowRunner
--jobName=game-analytics
--project=your_project_id
--tempLocation=gs://temp-bucket
```

Once the job is deployed, you should see a message that the job has been submitted. You can then click on the DataFlow console

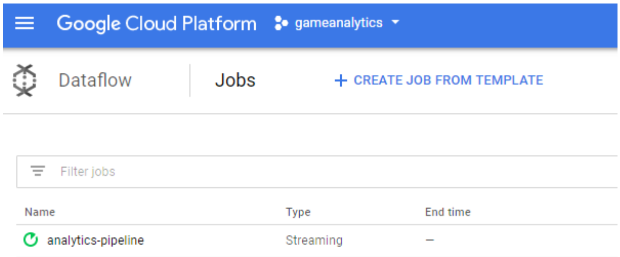

Figure 3.12: The steaming data pipeline running on Google Cloud

Figure 3.13: An example of Dataflow auto scaling.

to see the task. The runtime configuration specified above will not default to an auto scaling configuration. In order to deploy a job that scales up based on demand, you'll need to specify additional attributes:

```
--autoscalingAlgorithm=THROUGHPUT_BASED
--maxNumWorkers=30
```

Additional details on setting up a DataFlow task to scale to heavy workload conditions are available from Spotify[4]. The image below shows how DataFlow can scale up to meet demand as necessary.

3.3.5 Raw to Processed Events

The pipeline presented so far saves tracking events as raw data. To translate these events to processed data, we'll need to apply event

[4]https://labs.spotify.com/2016/03/10/

specific schemas. There's a few different approaches we can take with this pipeline:

- Apply the schemas in the current DataFlow pipeline and save to BigQuery
- Apply the schemas in the pipeline and send to a new PubSub
- Apply additional attributes to the raw events and send to a new PubSub
- Use downstream ETLs to apply schemas

The first approach is the simplest, but it doesn't provide a good solution for updating the event definitions if needed. This approach can be implemented as shown in the code below, which shows how to filter and parse MatchStart events for entry into BigQuery.

```
events.apply("To MatchStart Events", ParDo.of(
    new DoFn<PubsubMessage, TableRow>() {

@ProcessElement
public void processElement(ProcessContext c) {
  String msg = new String(c.element().getPayload());
  JsonObject jsonObject = new
      JsonParser().parse(msg).getAsJsonObject();
  String eventType = jsonObject.get("type");
  String version = jsonObject.get("eventVersion");
  String serverTime = dateFormat.format(new Date());

  // Filter for MatchStart events
  if (eventType.equals("MatchStart")) {

    TableRow record = new TableRow();
    record.set("eventType", eventType);
    record.set("eventVersion", version);
    record.set("server_time", serverTime);

    // event specifc attributes
    record.set("userID", jsonObject.get("userID"));
    record.set("type", jsonObject.get("deviceType"));
    c.output(record);
  }
}}))
.apply("To BigQuery",BigQueryIO.writeTableRows()
```

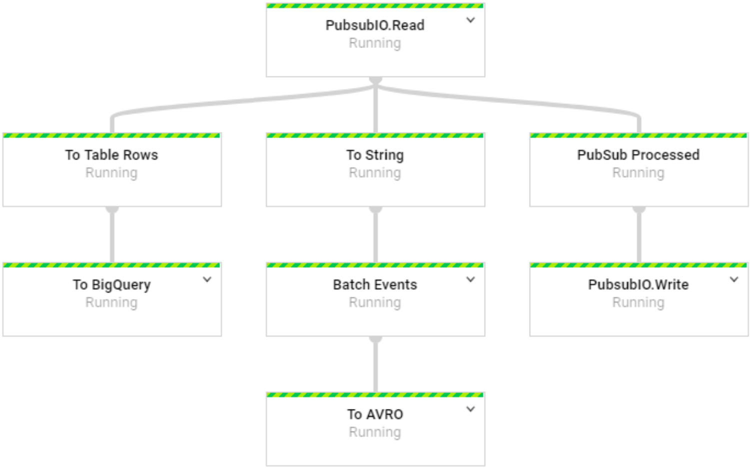

Figure 3.14: The streaming pipeline with an additional output,

In order to implement this approach, you'd need to create a new DoFn implementation for each type of event. The second approach is similar to the first, but instead of passing the parsed events to BigQuery, they are passed to a new PubSub topic. It's possible to send multiple types of events to a single topic or create a topic per event. The drawback of using the first two approaches is that the message parsing logic is part of the raw event pipeline. This means that changing event definitions involves restarting the pipeline.

A third approach that can be used is sending raw events with additional attributes to another PubSub topic. A second DataFlow job can then be set up to parse events as needed. The code below shows how to parse raw events, add additional attributes to the PubSub message for filtering, and publish the events to a second topic. This approach enables event definitions to be changed without restarting the raw event pipeline.

```
// topic for raw events with additional attributes
private static String processed =
  "projects/your_project_id/topics/processed-events";

events.apply("PubSub Processed",
```

```
  ParDo.of(new DoFn<PubsubMessage, PubsubMessage>() {

  @ProcessElement
  public void processElement(ProcessContext c)  {
    String msg = new String(c.element().getPayload());

    // parse the JSON message for attributes
    JsonObject jsonObject = new
        JsonParser().parse(msg).getAsJsonObject();
    String eventType = jsonObject.get("eventType");

    // Add additional attributes for filtering
    HashMap<String, String> atts = new HashMap();
    atts.put("EventType", eventType);
    PubsubMessage out = new PubsubMessage(
        msg.getBytes(), atts);
    c.output(out);
  }
}))
.apply(PubsubIO.writeMessages().to(processed));
```

A fourth approach that can be used is having downstream ETLs processes apply schemas to the raw events and break apart the raw events table into event specific tables. We'll cover this approach in the next chapter.

3.4 Conclusion

This chapter has provided an introduction to building a data pipeline for a startup. We covered the types of data in a pipeline, desired properties of a high functioning data pipeline, the evolution of data pipelines, and a sample pipeline built on GCP. The full source code for this sample pipeline is available on Github[5].

There is now a variety of tools available that make it possible to set up an analytics pipeline for an application with minimal effort. Using managed resources enables small teams to take advantage of serverless and autoscaling infrastructure to scale up to massive event volumes with minimal infrastructure management. Rather

[5] https://github.com/bgweber/GameAnalytics

than using a data vendor's off-the-shelf solution for collecting data, you can record all relevant data for your app. While the approach presented here isn't directly portable to other clouds, the Apache Beam library used to implement the core functionality of this data pipeline is portable and similar tools can be leveraged to build scalable data pipelines on other cloud providers.

Chapter 4

Business Intelligence

A lot of the heavy lifting involved in setting up data science at a startup is convincing the product team to instrument and care about data. If you're able to achieve this goal, the next step is being able to answer all sorts of questions about product health within your organization. A novice data scientist might think that this type of work is outside the role of a data scientist, but identifying key metrics for product health is one of the core facets of the role.

I've titled this chapter as business intelligence, because once you've set up a data pipeline, a data scientist in a startup is expected to answer every question about data. This is not surprising given the new flood of data, but also a time for a data scientist to set expectations for the rest of the organization. As a data scientist in a startup, your function is not to answer data questions, but to inform the leadership about what metrics should be important.

This chapter covers the basics of how to turn raw data into cooked data that can summarize the health of a product. I'll discuss a few different approaches to take when working with raw data, including SQL queries, R markdown, and vendor tools. The general takeaway is to show that several options are available for processing data sets, and you should choose a solution that fits the goals of your team. I'll discuss past experiences with tools such as Tableau, and provide recommendations for scaling automated reporting across a team.

We'll use two data sources for this chapter. The first is a public data set that we'll aggregate and summarize with key metrics. The second is data generated by the tracking API in the second chapter of this series. We'll focus on the second data set for transforming raw to processed data, and the first data set for processed to cooked data.

4.1 KPIs

Key Performance Indicators (KPIs) are used to track the health of a startup. It's important to track metrics that capture engagement, retention, and growth, in order to determine if changes made to the product are beneficial. As the data scientist at a startup, your role has the responsibility of identifying which metrics are important.This function aligns which the data science competency of domain knowledge, and is one of the areas where a data scientists can be highly influential.

KPIs that are established by an early data scientist can have have a resounding impact. For example, many of the past companies I worked at had company goals based on past analyses of data scientists. At Electronic Arts we were focused on improving session metrics, at Twitch we wanted to maximize the amount of content watched, and at Sony Online Entertainment we wanted to improve retention metrics for free-to-play titles. These were game industry metrics, but there are more general metrics such as engagement, growth, and monetization that are important to track when building a company.

It's important when building a data science discipline at a startup to make sure that your team is working on high impact work. One of the problems I've seen at past companies is data scientists getting pulled into data engineering and analytics type of work. This is expected when there's only one data person at the company, but you don't want to support too many manual data processes that won't scale. This is why setting up reproducible approaches for reporting and analysis is important. It should be trivial to rerun an analysis months down the road, and it should be possible for another team member to do so with minimal direction.

My main advice for new data scientists to prevent getting overwhelmed with requests from product managers and other teams is to set up an interface to the data science team that buffers direct requests. Instead of having anyone at the company being able to ask the data science team how things are performing, a baseline set of dashboards should be set up to track product performance. Given that a data scientists may be one of the first data roles at a startup, this responsibility will initially lie with the data scientist and it's important to be familiar with a number of different tools in order to support this function at a startup.

4.2 Reporting with R

One of the key transitions that you can make at a startup as a data scientist is migrating from manual reporting processes to reproducible reports. R is a powerful programming language for this type of work, and can be used in a number of different ways to provide automated reporting capabilities. This section discusses how to use R for creating plots, generating reports, and building interactive web applications. While many of these capabilities are also provided by Python and the Jupyter suite, the focus on automation is more important than the language used to achieve this goal.

It's possible to achieve some of this type of functionality with Excel or Google Sheets, but I would advise against this approach for a startup. These tools are great for creating charts for presentations, but not suitable for automated reporting. It's not sustainable for a data scientists to support a startup based on these types of reports, because so many manual steps may be necessary. Connectors like ODBC in Excel may seem useful for automation, but likely won't work when trying to run reports on another machine.

This section covers three approaches to building reports with R: using R directly to create plots, using R Markdown to generate reports, and using Shiny to create interactive visualizations. All of the code listed in this section is available on Github[1].

4.2.1 Base R

Consider a scenario where you are part of a NYC startup in the transportation sector, and you want to determine what type of

[1] https://github.com/bgweber/StartupDataScience/tree/master/BusinessIntelligence

payment system to use to maximize the potential of growing your user base. Luckily, there's a public data set that can help with answering this type of question: BigQuery's NYC Taxi and Limousine Trips public data set[2]. This collection of trip data includes information on payments that you can use to trend the usage of payment types over time.

The first approach we'll use to answer this question is using a plotting library in R to create a plot. I recommend using the RStudio IDE when taking this approach. Also, this approach is not actually "Base R", because I am using two additional libraries to accomplish the goal of summarizing this data set and plotting the results. I'm referring to this section as Base R, because I am using the built-in visualization capabilities of R.

One of the great aspects of R is that there's a variety of different libraries available for working with different types of databases. The bigrquery library provides a useful connector to BigQuery that can be used to pull data from the public data set within an R script. The code for summarizing the payment history over time and plotting the results as a chart are shown below.

```
library(bigrquery)
library(plotly)
project <- "your_project_id"
sql <- "SELECT
  substr(cast(pickup_datetime as String), 1, 7) as date
  ,payment_type as type
  ,sum(total_amount) as amount
FROM `nyc-tlc.yellow.trips`
group by 1, 2"
df <- query_exec(sql, project = project,
          use_legacy_sql = FALSE)
plot_ly(df, x = ~date, y = ~amount,
          color = ~type) %>% add_lines()
```

The first part of this script, which includes everything except for the last line, is responsible for pulling the data from BigQuery. It loads the necessary libraries, states a query to run, and uses bigrquery to fetch the result set. Once the data has been pulled into a data frame, the second part of the script uses the plotly library to display

[2]https://cloud.google.com/bigquery/public-data/nyc-tlc-trips

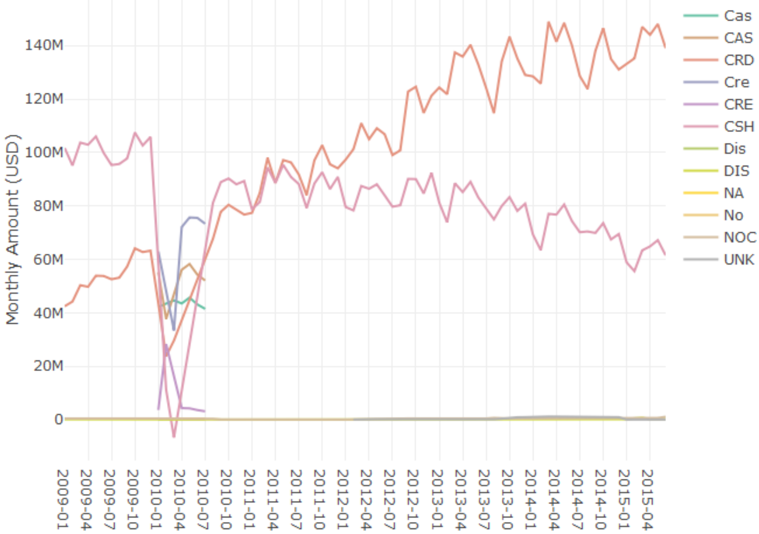

Figure 4.1: Monthly Spending by Payment

the results as a line chart. Some additional formatting steps have been excluded from the script, and the full code listing is available on Github. In RStudio, the chart will show up as an interactive plot in the IDE, and Jupyter provides similar functionality. The result of this code snippet is shown in the chart below.

TypeThe query calculates the total monthly spend by payment type for taxi trips in NYC, using data from 2009 to 2015. The results show that credit cards (CRD) are now the preferred payment method over cash (CSH). To answer the initial question about what type of payment system to implement, I'd recommend starting with a system that accepts credit cards.

One topic worth bringing up at this point is data quality, since the chart has a number of different labels that seem to represent the same values. For example CAS and CSH both likely refer to cash payments and should be grouped together to get an accurate total of cash payments. Dealing with these types of issues is outside the scope of this approach, but there are a few methods that can be used for this type of scenario. The easiest but least scalable approach is to write queries that account for these different types:

```
,sum(case when payment_type in ('CSH', 'CAS')
       then amount else 0 end) as cash_payments
```

A different approach that can be used is creating a dimension table that maps all of the raw payment_type values to sanitized type values. This process is often called attribute enrichment, and is useful when building out cooked data sets from raw or processed data.

We've answered the first question about determining the most popular payment method, but what if we have a second question about whether or not the transportation market in NYC is growing? We can easily plot data to answer this question using the existing data:

```
total <- aggregate(df$Amount,
             by=list(Category=df$Date), FUN=sum)
plot_ly(total, x = ~Category, y = ~x) %>% add_lines()
```

This code computes the total monthly payments across all of the different payment types, and plots the aggregate value as a single line chart. The results are shown in the figure below. Based on the initial observation of this data, the answer to the second question is unclear. There's been a steady increase in taxi spending in NYC from 2009 to 2013, with seasonal fluctuations, but spending peaked in sum mer of 2014. It's possible that Uber and Lyft account for this trend, but further analysis is needed to draw a firm conclusion.

This section has shown how to use R to generate plots from summarized data in BigQuery. While this sample used a fixed data set, the same approach could be used with a live data set that grows over time, and rerunning the script will include more recent data. This is not yet automated reporting, because it involves manually running the code in an IDE or notebook. One approach that could be used is outputting the plot to an image file, and running the script as part of a cron job. The result of this approach is an image of the plot that gets updated on a regular schedule. This is a good starting point, but there are more elegant solutions for automated reporting in R.

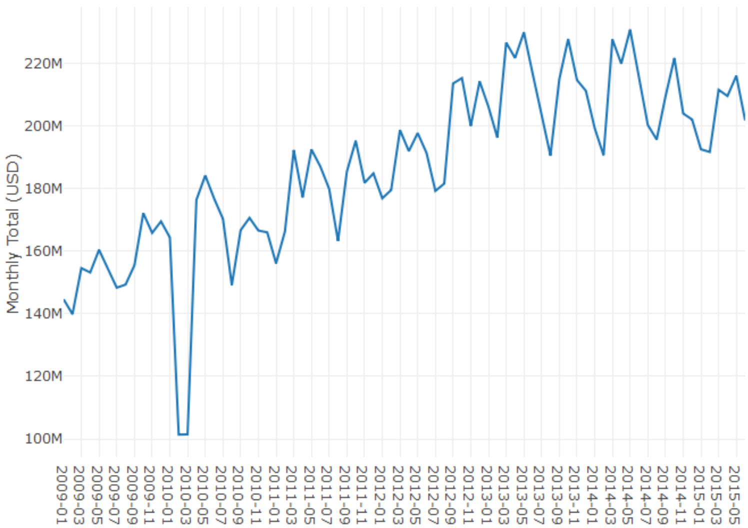

Figure 4.2: Total Monthly Spending

4.2.2 R Markdown

Let's say you want to perform the same analysis as before, but want to produce a report each time you run the script. R Markdown provides this capability, and can use R code to generate PDFs, word documents (DOCX), and web pages (HTML). You can even write books with R Markdown! R Markdown extends standard markdown to support inline R snippets that can be used to generate visualizations. The embedded R code can perform almost any standard R functionality, including using R libraries and making connections to databases. This means we can convert the code above into an R markdown file, and run the script regularly to build automated reporting.

The markdown snippet below is the previous R code now embedded in a report that will generate an HTML file as output. The first part of the file is metadata about the report, including the desired output. Next, markdown is used to add commentary to the report. And finally, a R code block is used to pull data from BigQuery and plot the results. The resulting plotly object is embedded into the document when running this report.

```
---
title: "Business Intelligence"
output: html_document
---
## Taxi Payments
R Markdown can outputs reports as PDF or HTML.

`` `{r echo=FALSE, message=FALSE, warning=FALSE}
library(bigrquery)
library(plotly)
project <- "your_project_id"
sql <- "SELECT
  substr(cast(pickup_datetime as String), 1, 7) as date
  ,payment_type as type
  ,sum(total_amount) as amount
FROM `nyc-tlc.yellow.trips`
group by 1, 2"
df <- query_exec(sql, project = project,
          use_legacy_sql = FALSE)
plot_ly(df, x = ~date, y = ~amount,
          color = ~type) %>% add_lines()
` ``
```

The resulting HTML document is shown in the figure below. It includes that same plot as before, as well as the markdown text listed before the code block. This output can be more useful than an image, because the plotly charts embedded in the file are interactive, rather than rendered images. It's also useful for creating reports with a variety of different charts and metrics.

To automate creating this report, you can again set up a cron job. The command for converting the Rmd file to a report is:

```
Rscript -e "rmarkdown::render('BI.Rmd')"
```

We now have a way of generating reports, and can use cron to start building an automated reporting solution. However, we don't yet have charts that provide filtering and drill-down functionality.

Business Intelligence

Ben Weber

May 21, 2018

Taxi Payments

R Markdown can outputs reports as PDF or HTML.

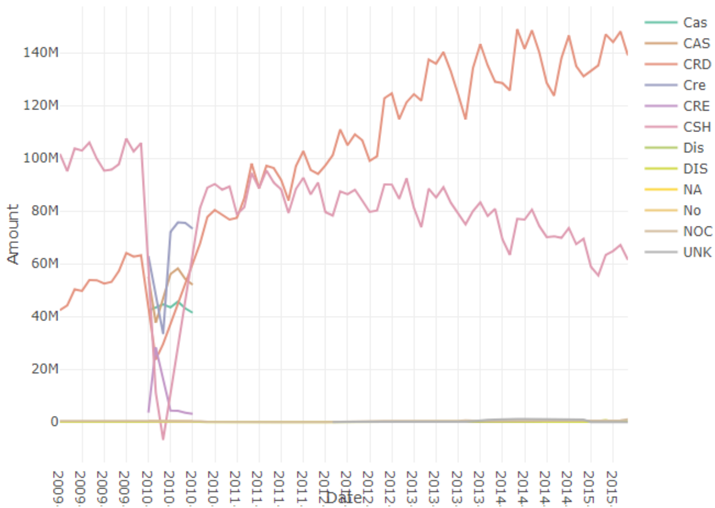

Figure 4.3: The report generated from the R Markdown file.

4.2.3 R Shiny

Shiny is a solution for building dashboards directly in R. It provides functionality for building reports with filtering and drill-down capabilities, and can be used as an alternative to tools such as Tableau. When using Shiny, you specify the UI components to include in the report and the behaviors for different components in a report, such as applying a filter based on changes to a slider component. The result is an interactive web app can that runs your embedded R code.

I've created a sample Shiny application based on the same code as the above reports. The first part of the code is the same, we pull data from BigQuery to a dataframe, but we also include the shiny library. The second part of the code defines the behavior of different components (server), and the layout of different components (ui). These functions are passed to the shinyApp call to launch the dashboard.

```
library(shiny)
library(bigrquery)
library(plotly)
project <- "your_project_id"
sql <- "SELECT
substr(cast(pickup_datetime as String), 1, 7) as date
,payment_type as type
,sum(total_amount) as amount
FROM `nyc-tlc.yellow.trips`
group by 1, 2"
df <- query_exec(sql, project = project,
                 use_legacy_sql = FALSE)
server <- function(input, output) {
  output$plot <-  renderPlotly({
    plot_ly(df[df$date >= input$year, ], x = ~date,
      y = ~amount, color = ~type) %>% add_lines()
  })
}
ui <- shinyUI(fluidPage(
  sidebarLayout(
    sidebarPanel(
      sliderInput("year", "Start Year:",
```

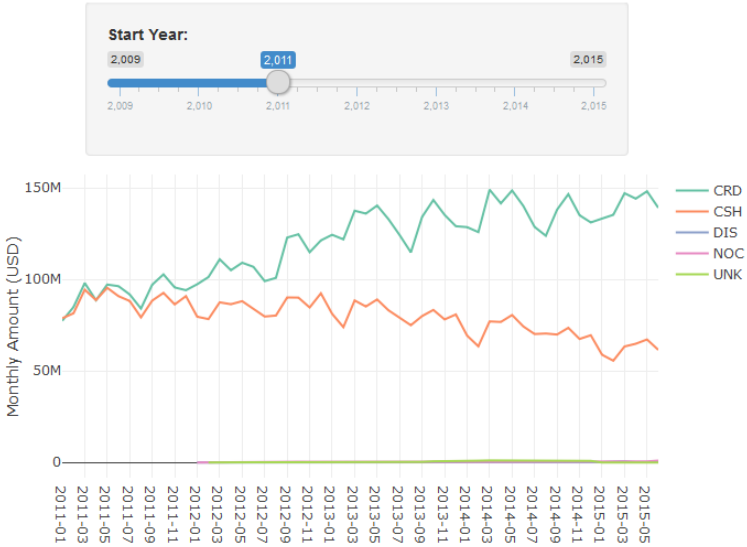

Figure 4.4: An interactive Chart in R Shiny.

```
        min = 2009, max = 2015, value = 2012)
    ),
    mainPanel(plotlyOutput("plot"))
  )
))
shinyApp(ui = ui, server = server)
```

The UI function specifies how to lay out the components in the dashboard. I started with the Hello Shiny example, which includes a slider and histogram, and modified the layout to use a plotlyOutput object instead of a plotOutput. The slider specifies the years to allow for selection, and sets a default value. The behavior function specifies how to respond to changes in UI components. The plot is the same as behavior, with one modification, it now filters on the starting data when using the data frame df$date >= input$year. The result is the interactive dashboard shown below. Moving the slider will now filter the years that are included in the chart.

I've now shown three different ways to generate reports using R. If you need interactive dashboards, then Shiny is a great tool to explore, while if you're looking to build static reports, then R Markdown is a great solution. One of the key benefits of both of these approaches is that you can embed complex R logic within your charts, such as using Facebook's prophet library to add forecasted values to your charts.

4.3 ETLs

In the chapter on data pipelines, I discussed using raw, processed, and cooked data. Most reports used for business intelligence should be based on cooked data, where data is aggregated, enriched, and sanitized. If you use processed or raw data instead of cooked data when building reports, you'll quickly hit performance issues in your reporting pipeline. For example, instead of using the nyc-tlc.yellow.trips table directly in the R section above, I could have created a table with the aggregate values precomputed.

ETL is an abbreviation of Extract-Transform-Load. One of the main uses of these types of processes is to transform raw data into processed data or processed data into cooked data, such as aggregation tables. One of the key challenges in setting up aggregates tables is keeping the tables updated and accurate. For example, if you started tracking cash payments using a new abbreviation (e.g. CAH), you would need to update the aggregation process that computes monthly cash payments to include this new payment type.

One of the outputs of the data pipeline is a raw events table, that includes data for all of the tracking events encoded as JSON. One of the types of ETL processes we can set up is a raw to processed data transformation. In BigQuery, this can be implemented for the login event as follows:

```
create table tracking.logins as (
  select eventVersion,server_time
    ,JSON_EXTRACT_SCALAR(message, '$.userID') userID
    ,JSON_EXTRACT_SCALAR(message, '$.deviceType') type
  from tracking.raw_events
  where eventType = 'Login'
)
```

This query filters on the login events in the raw events table, and uses the JSON extract scalar function to parse elements out of the JSON message. The result of running this DDL statement will be a new table in the tracking schema that includes all of the login data. We now have processed data for logins with userID and deviceType attributes that can be queried directly.

In practice, we'll want to build a table like this incrementally, transforming only new data that has arrived since the last time the ETL process ran. We can accomplish this functionality using the approach shown in the SQL code below. Instead of creating a new table, we are inserting into an existing table. With BigQuery, you need to specify the columns for an insert operation. Next, we find the last time when the login table was updated, represented as the updateTime value. And finally, we use this result to join on only login events that have occured since the last update. These raw events are parsed into processed events and added to the logins table.

```
insert into tracking.logins
    (eventVersion,server_time, userID, deviceType)
with lastUpdate as (
  select max(server_time) as updateTime
  from tracking.logins
)
select eventVersion,server_time
  ,JSON_EXTRACT_SCALAR(message, '$.userID') userID
  ,JSON_EXTRACT_SCALAR(message, '$.deviceType') type
from tracking.raw_events e
join lastUpdate l
  on e.server_time > updateTime
where eventType = 'Login'
```

A similar approach can be used to create cooked data from processed data. The result of the login ETL above is that we now can query against the userID and deviceType fields directly. This processed data makes it trivial to calculate useful metrics such as daily active users (DAU), by platform. An example of to to compute this metric in BigQuery is shown below.

Date	deviceType	DAU
2018-03-25	Android	9995
2018-03-25	Web	9814
2018-03-25	iOS	9825
2018-03-26	Android	2786
2018-03-26	Web	1553
2018-03-26	iOS	1493

Figure 4.5: Cooked Data: DAU by Platform.

```
create table metrics.dau as (
  select substr(server_time, 1, 10) as Date
    ,deviceType, count(distinct userID) as DAU
  from `tracking.logins`
  group by 1, 2
  order by 1, 2
)
```

The result of running this query is a new table with the DAU metric precomputed. A sample of this data is shown in the Cooked Data table. Similar to the previous ETL, in practice we'd want to build this metric table using an incremental approach, rather than rebuilding using the complete data set. A slightly different approach would need to be taken here, because DAU values for the current day would need to be updated multiple times if the ETL is ran multiple times throughout the day.

Once you have a set of ETLs to run for your data pipeline, you'll need to schedule them so that they run regularly. One approach you can take is using cron to set up tasks, such as:

```
bq query --flagfile=/etls/login_etl.sql
```

It's important to set up monitoring for processes like this, because a failure early on in a data pipeline can have significant downstream impacts. Tools such as Airflow can be used to build out complex data pipelines, and provide monitoring and alerting.

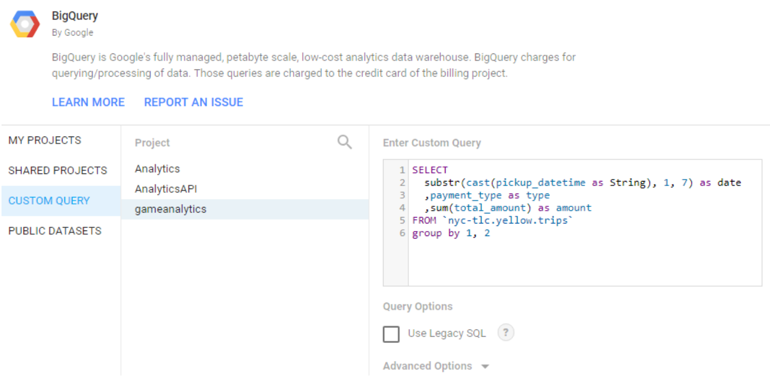

Figure 4.6: Setting up a Custom Data Source in Data Studio.

4.4 Reporting Tools

While R does provide useful tools for performing business intelligence tasks, it's not always the best tool for building automated reporting. This is common when reporting tools need to used by technical and non-technical users and vendor solutions for building dashboards are often useful for these types of scenarios. Here are a few of the different tools I've used in the past.

4.4.1 Google Data Studio

If you're already using GCP, then Google Data Studio is worth exploring for building dashboards to share within your organization. However, it is a bit clunkier than other tools, so it's best to hold off on building dashboards until you have a mostly complete spec of the reports to build.

The image above shows how to set up a custom query in Google Data Studio to pull the same data sets as used in the R reports. The same report as before, now implemented with Data Studio is shown below.

The main benefit of this tool is that it provides many of the collaboration features build into other tools, such as Google Docs and Google Sheets. It also refreshes reports as necessary to keep data from becoming stale, but has limited scheduling options available.

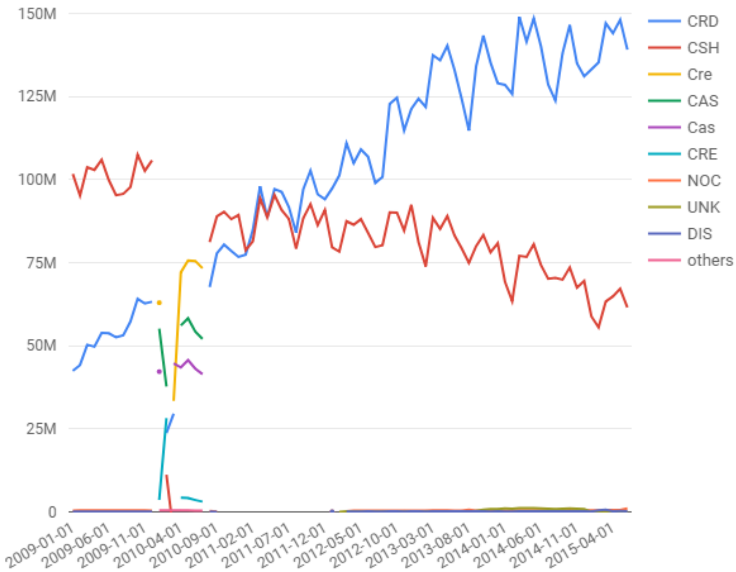

Figure 4.7: The Taxi Report recreated in Google Data Studio.

4.4.2 Tableau

One of the best visualization tools I've used is Tableau. It works well for the use case of building dashboards when you have a complete spec, and well as building interactive visualizations when performing exploratory analysis. The heatmap for DC Universe Online was built with Tableau, and is one of many different types of visualizations that can be built.

The main benefit of Tableau is ease-of-use in building visualizations and exploring new data sets. The main drawback is pricing for licenses, and a lack of ETL tooling, since it is focused on presentation rather than data pipelines.

4.4.3 Mode

At Twitch, we used a vendor tool called Mode Analytics. Mode made it simple to share queries with other analysts, but has a

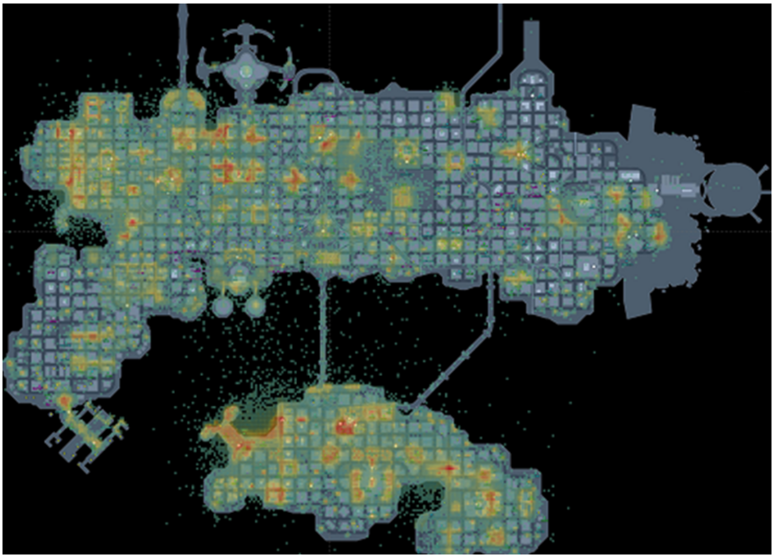

Figure 4.8: A heatmap in Tableau for the game DC Universe Online.

rather limited selection of visualization capabilities, and also was focused on only presentation and not ETL type tasks.

4.4.4 Custom Tooling

Another approach that can be used is creating custom visualizations using tools such as D3.js and Protovis. At Electronic Arts, D3 was used to create customer dashboards for game teams, such as the Data Cracker tool built by Ben Medler for visualizing playtesting data in Dead Space 2. Using custom tooling provides the most flexibility, but also requires maintaining a system, and is usually substantially more work to build.

4.5 Conclusion

One of the key roles of a data scientist at a startup is making sure that other teams can use your product data effectively. Usually this takes the form of providing dashboarding or other automated

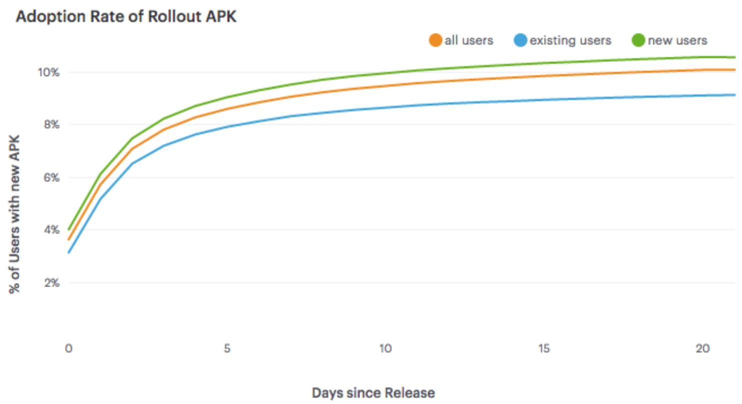

Figure 4.9: Line Charts in Mode Analytics.

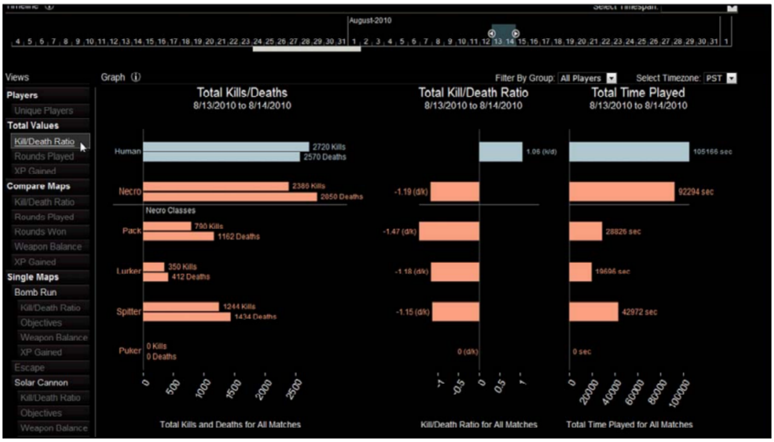

Figure 4.10: The Data Cracker Tool for Dead Space 2. Source: GDC Vault 2011.

reporting, in order to provide KPIs or other metrics to different teams. It also includes identifying which metrics are important for the company to measure.

This chapter has presented three different ways for setting up automated reporting in R, ranging from creating plots directly in R, using R Markdown to generate reports, and using Shiny to build dashboards. We also discussed how to write ETLs for transforming raw data to processed data and processed data to cooked data, so that it can be used for reporting purposes. And the last section discussed some different vendor solutions for reporting, along with tradeoffs.

After setting up tooling for business intelligence, most of the pieces are in place for digging deeper into data science type of work. We can move beyond retrospective types of questions, and move forward to forecasting, predictive modeling, and experimentation.

Bibliography

Xie, Y. (2018). *bookdown: Authoring Books and Technical Documents with R Markdown.* R package version 0.7.8.