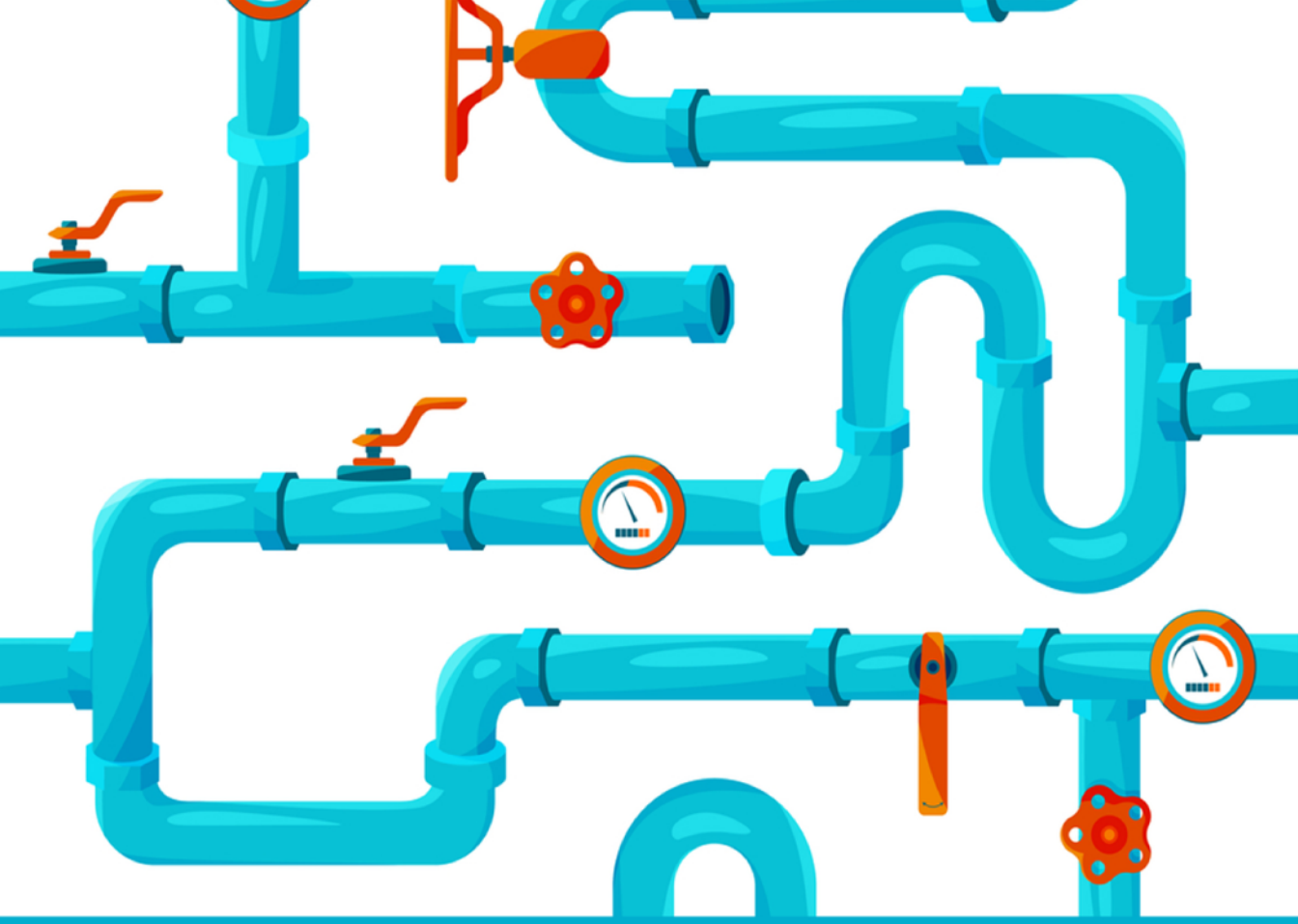


REAL-TIME CRYPTO

a big data approach to analyzing & automating cryptocurrency trading

Brandon Rose

Real-Time Crypto

A big data approach to analyzing & automating cryptocurrency trading

Brandon Rose

This book is for sale at http://leanpub.com/realtimecrypto

This version was published on 2018-05-22

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

This book is dedicated to the people who have taught me the most, including how to write: my parents.

Contents

About the Author

Brandon Rose

Brandon Rose is passionate about sharing his knowledge of big data, machine learning and cryptocurrency. Brandon is a technologist with experience deploying big data applications in both the public and private sectors.

He has used Python to wrangle massive datasets and to build data pipelines with tools like Spark, Kafka, and Elasticsearch. Brandon is passionate about using Python for natural language processing and finding meaning in huge quantities of unstructured data. He is also interested in cryptocurrencies and has used Python to analyze the entire Bitcoin blockchain, revealing inefficient practices among the major players in the space.

He currently spends his time on a big data startup which is using geospatial data to build a product in the cybersecurity and physical security space.

Contributors

Robert Dempsey

This book contains significant contributions from Robert Dempsey. Robert is a tested leader and technology professional delivering solutions and products that solve tough business challenges. His experience forming and leading agile teams combined with more than 17 years of technology experience enables him to solve complex problems while always keeping the bottom line in mind.

He founded and built three startups in tech and marketing, developed and sold online applications, consulted to Fortune 500 and Inc. 500 companies, authored the Python Business Intelligence Cookbook, and has spoken nationally and internationally on software development and agile project management. He has expertise in from-the-front leadership and mentoring, microservices architectures and API development, cloud services particularly Amazon Web Services, and distributed data gathering and processing systems.

Disclaimer

The contents of this book are delivered as is. The examples in this book are meant to be illustrative. The author fully expects that the code will, in the future, become outdated as technologies naturally evolve. The author bears no responsibility for the outcome of real financial trades which readers of the book may undertake.

Introduction

Have you ever wondered how to build a cryptocurrency trading bot? Have you wondered how you can apply machine learning to the financial markets? Have you wondered how you can get out of Excel and into the world of big data? You have come to the right place! Welcome to *Real-Time Crypto*. In this book, we will build on your basic Python and data skills and turn you into a bona fide big data engineer. Along the way, you will learn how to trade cryptocurrency in real-time, using cutting edge big data technology and machine learning.

What is Real-Time Crypto?

Let's start with a basic definition of what we mean by *Real-Time Crypto* and what we will build in this book:

> A big data pipeline which receives streaming data from a cryptocurrency exchange, processes the data, applies a predictive algorithm, executes a trade, and stores the results for future usage.

To accomplish this we are going to have to:

1. Process large volumes of data
2. Combine multiple data sources
3. Apply a machine learning model

Processing Data

In this book, you will gain exposure to the fundamentals of data processing. These include:

- normalizing data fields: putting like with like
- information extraction: finding critical information buried within the data
- data enrichment: adding relevant information to improve the quality of the data.

Combining Sources

We will need to combine multiple data sources to accurately inform our trading decisions. This is a common task for a data scientist or data engineer so we'll get hands-on by combining historic pricing data and real-time data from an exchange.

Applying Models

We are going to train a machine learning model on a historic, static data set. In our real-time pipeline we will learn how to apply this model to data on the fly.

> Cryptocurrency trading pipelines don't have to be complicated, and they shouldn't be. Many times, the simplest solution is the best solution. The complexity of your pipeline's architecture should depend on the volume of your data and sophistication of your strategy. Remember though, simple can be powerful.

Goals of the Book

By the time you finish this book you'll have a foundational understanding of cryptocurrency analysis and automation. You'll have a solid understanding of machine learning-based cryptocurrency trading pipelines-how they work and how to build one-and you'll have a complete pipeline running on your computer! You will feel confident in your ability to take what you've learned and apply it to the creation of production-level pipelines for processing whatever data may come your way.

Real-World Use Case

Rather than work with a toy dataset that doesn't look anything like the stuff you typically deal with, you'll be implementing a real-world use case with real world data: predicting how Bitcoin prices will change in order to time trades with your very own trading bot.

First, we will process historical data and train a predictive machine learning model. We'll be using the past few years of minute-by-minute Bitcoin prices. Our time series forecasting model will predict price swings in Bitcoin so that we know *when* to trade.

Next, we will incorporate this algorithm into a streaming data pipeline. This pipeline will take real-time data from Gemini, a major cryptocurrency exchange, and use our algorithm to determine *whether* to execute trades. Our pipeline's architecture will include websockets, Kafka, Spark, and Elasticsearch.

What this Book Assumes

This book assumes you have a basic to intermediate familiarity with Python. This book assumes that you are capable of following installation instructions so that you can run the examples.

What this Book does not Assume

This book does not assume that you have a deep knowledge of statistics, machine learning, math or cryptocurrency. That said, by the end of this book you will enhance your knowledge on all the above.

Conventions

Here are a few notes on the conventions you'll see in the book.

Tone

I've written the book in a conversational and casual tone. I want you to have fun reading and working through this book so I've kept it light. But don't mistake this tone for a lack of seriousness. I'm very serious. Seriously.

I've kept the writing terse. I doubt you have all day to read books like this one, though if you do I might want your job!

Terminology

There are certain terms I'll write out once and then use an abbreviated form afterwards. Examples:

- Machine Learning ⇒ ML
- Elasticsearch ⇒ ES
- Command-Line ⇒ CLI

When you need to run a command at the command line, it'll look like this:

```
python hello_world.py
```

Code Blocks

Blocks of code will be formatted like this:

```
print("hello world")
```

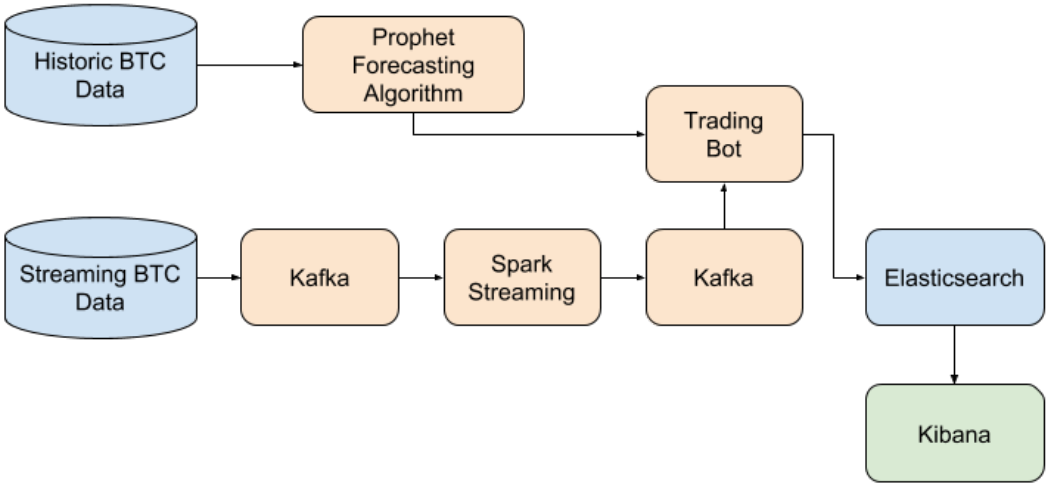
System Requirements

To take advantage of the code examples you'll need:

1. A Mac or Ubuntu computer (this is preferred; Windows or other Linux flavor will work too)
2. Python (Anaconda preferred)
3. Text editor or IDE
4. Git source control

A Roadmap for what we're Building

Let's briefly talk architecture. In this chapter, you will get an overview of the real-time crypto trading pipeline we're going to build and the technologies involved.



Real-Time Crypto Architecture

This may look like there are a lot of moving pieces, but not to worry–it's actually pretty straightforward.

Developing a forecast

First, we'll use historic Bitcoin (BTC) data to feed a time series forecasting algorithm developed at Facebook called Prophet (https://research.fb.com/prophet-forecasting-at-scale/). We'll feed this machine learning model's output directly to our trading bot.

Streaming exchange data

Next, we'll ingest streaming BTC order data from the Gemini exchange. We'll use Kafka as a message broker. From Kafka, we will use Spark Streaming to calculate market liquidity. We'll then pass the market liquidity information back to Kafka.

Trading

Our trading bot will read in our price forecasts from Prophet while streaming liquidity data from Kafka. It will use the forecasts to determine when to execute trades as long as liquidity levels do not indicate massive volatility. The bot will interact directly with the Gemini Exchange. As the bot trades, it will store all the data it processes in Elasticsearch. From there we can use Kibana, a visualization tool, to analyze our results.

Summary

This type of pipeline is used by many organizations and will serve you well, whether you choose to execute cryptocurrency trades in real-time, analyze streaming social media data, or process corporate transactional data. Get ready to use cutting-edge technologies like Kafka, Spark, and Elasticsearch to apply an advanced algorithm like Prophet in a big data machine learning pipeline.

Setting up your Environment

In this chapter, I will walk you through how to set up your computing environment. We're going to cover how you can go from zero to having a full stack data engineering and data science environment in no time. You'll learn about Anaconda and Python package management, Docker and containerized services, and how to use Github.

Github

Git is a version control and collaboration software. When you make changes to your project (or in Git terminology, your **repo**) you make a **commit**. **Commits** are like snapshots in time. They compare the current state of your **repo** with the last **commit** and include the new additions and subtractions along with a **commit message**. This lets you keep a running history of the work you've done along with notes about the changes you made. Realize that something is amiss? Git lets you roll back your changes to a prior commit where things worked smoothly.

Github is a social network and cloud hosting solution for Git repositories. Github gained popularity due to its willingness to host open source repositories for free. Now, it is used by companies around the world to enable collaboration among teams working on the same code base. And, it's used by us to host a repository for this book! So, let's get Git installed. If you don't have a Github account yet get over to http://github.com/ and sign up.

Installation

First, install the latest version of Git for your operating system. The downloads are available at https://git-scm.com/downloads.

Now, in a terminal, you should set your Git username and commit email address:

```
1  git config --global user.name "MyGithubUsername"
2  git config --global user.email "MyEmail@coolmail.com"
```

You are now ready to download (or **clone**) the repository for this book. This repository will contain all the code examples and Dockerfiles you'll need to follow along with us, so it is

critical that you **clone** it. First, navigate to the directory where you wish to store the repo. Many of our command line examples assume this is a directory called repos which resides in your home directory. You can create this directory with:

```
cd ~/
mkdir repos
cd repos
```

Now you are in your `repos` directory and are ready to clone the book's repo with:

```
git clone https://github.com/brandomr/realtimecrypto.git
```

With these code examples at hand, you're ready to hit the ground running!

Anaconda (Python)

Anaconda is the go to distribution of Python for data scientists. It gets you up and running quickly and includes some of the most fundamental data science packages for Python, including Pandas and Sci-kit Learn. It is a free and open source (FOSS) software created and maintained by Continuum Analytics.

Installation

To get started, go to the MacOS installer download page here: https://www.anaconda.com/download/#mac Go ahead and download the installer for Python 3.6, not Python 2.7. If Anaconda is at a higher version than 3.6 that is fine, just choose the installer for 3+ as Python 2 is no longer being actively supported by the Python community.

Once the installer is downloaded, open it up. Hint: the downloaded file will look something like `Anaconda3-4.4.0-MacOSX-x86_64.pkg`. Follow the instructions through the graphical installer to get Anaconda set up on your computer.

Once the installation process is complete, let's test that everything works as expected. Go ahead and open up a terminal window. At the command line interface (CLI) try:

```
conda -V
```

You should see "conda" echoed, with a version number such as:

```
conda 3.16.0
```

If so, let's go ahead and activate the root conda environment. This environment has tons of core data science packages preloaded for you.

```
source activate root
```

Try listing the Python packages that are included with:

```
pip list
```

You should see nearly 200 packages including Pandas, Sci-kit Learn, Scipy, Numpy, and Jupyter. These are absolutely critical pieces of software for any working data scientist and can be a bit tricky to install on your own.

Environments

Conda can create separate environments for your various projects. This allows you to run different versions of Pandas for different projects. For example, let's say you created a project a while ago using Pandas version 0.16.2. Yesterday, you started a new project and wanted to update your version of Pandas to the latest (0.20.3). No problem right? Wrong. There were significant changes between the 0.1 and 0.2 major releases, including deprecations of functionality that could cause your code to break.

However, if we use conda environments we don't need to worry about this. Why? Because we can have one environment for our earlier project that has 0.16.2 and another environment for 0.20.3. For now, let's create an environment that we'll be using for the rest of this book. Let's call it `rtcenv`.

```
conda create --name rtcenv
```

Now we can activate that environment with

```
source activate rtcenv
```

We've now activated our `rtcenv` environment. We can install whatever we need in this environment without having to impact other projects or change any underlying Python dependencies that your computer may have. To do that, navigate to the code repository for this book. At the top-level, you'll find a file called `requirements.txt`. You can install these requirements with:

```
pip install -r requirements.txt
```

Docker

Docker is the leading container platform on the market today. It is freely available to use unless you need the Docker Enterprise edition. So, we're going to be using Docker to make our lives much, much easier.

Docker runs containers, which are basically pieces of software that are bundled up with all their various dependencies. The beauty of Docker containers is that they are extremely lightweight relative to running a VM as they don't install the entire operating system. The real magic of Docker containers is that they just work. Installing the big data stack can be a challenge due to the need to orchestrate software configuration across a computing cluster; for our purposes, we can use Docker and avoid all that headache.

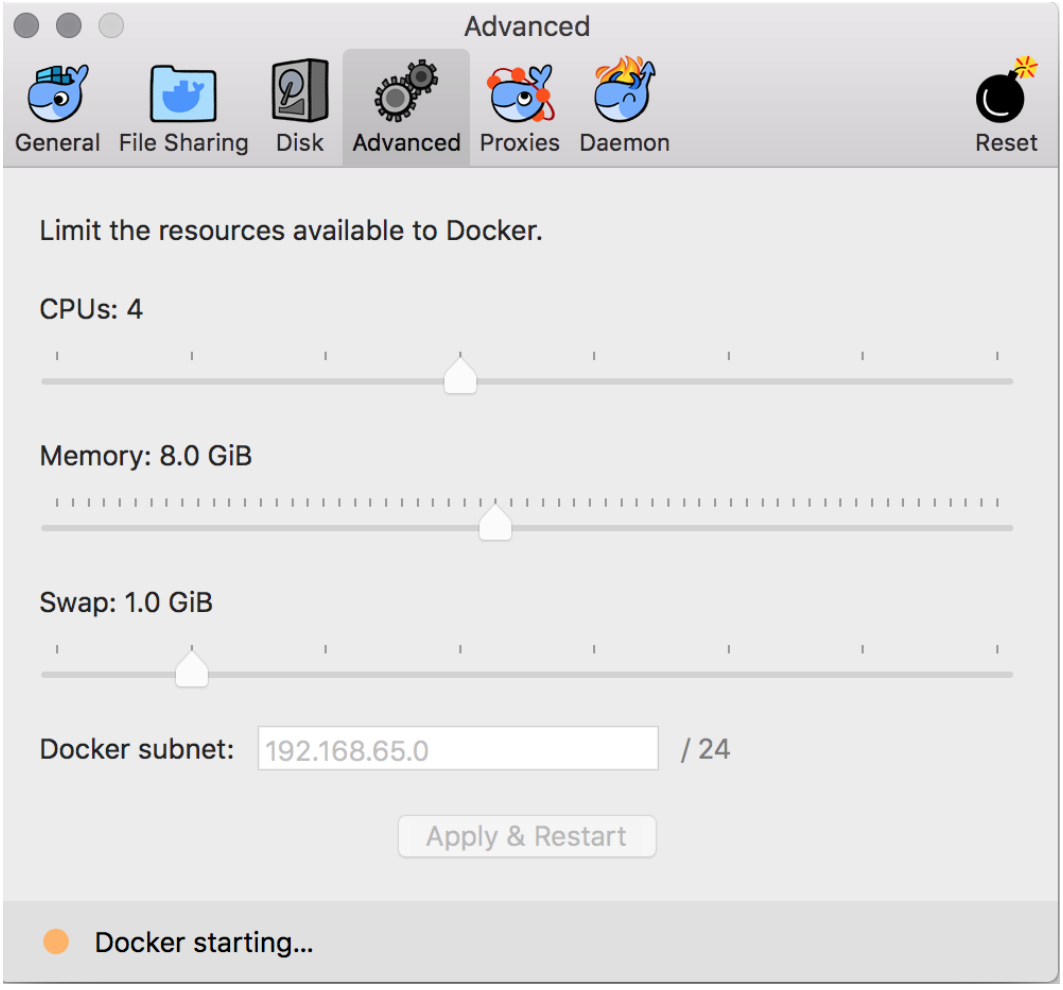
Installation

To install Docker on a Mac, follow their instruction guide here: https://docs.docker.com/docker-for-mac/install/. If you're on a Windows machine follow their Windows installation guide here: https://docs.docker.com/docker-for-windows/install/. To test that your installation worked you should try running in your CLI:

```
docker version
```

You should see an output which includes information about the version of Docker you've installed.

Once Docker is running, open up your Docker preferences and increase the amount of memory allocated to it. Hey, running a big data machine learning pipeline takes a lot of resources! Let's try 8 gb of memory. If your computer is smaller try 4 or more gb of memory.



"Docker settings"

Atom and SublimeText

One final thing before we get started though; you need a text editor. I like Atom or SublimeText. Both are extremely powerful and are industry standards for writing code. Code, after all, is just text. These things are the Swiss Army Knives of the text editing world and if you get to learn some of their shortcuts you'll find that your productivity goes through the roof.

Installation

You can download and install Atom at https://atom.io/ and SublimeText at https://www.sublimetext.com/.
Happy text editing!

Cryptocurrency and Blockchain: a Primer

This chapter provides a primer on cryptocurrencies, how they work, and how we will use them in this book. Feel free to skip these sections if you are already familiar with cryptocurrency and/or how financial exchanges work.

Why Cryptocurrency?

What is the value of cryptocurrency? Why was it created? What practical purpose does it serve? Cryptocurrency has been around for many years but only entered the mainstream in 2017. There are several features of cryptocurrency that make it an attractive mechanism to exchange value between individuals or organizations:

- Transactions can be trustless
- Transactions can be irreversible
- Currency can be resistant to government censorship and interference

Trustless Transactions

Cryptocurrencies enable transactions to occur between individuals in a way that allows each party to know that they are not being cheated. Take a common scenario: a credit card transaction. Susan pays Bob with her Visa card and receives 10 widgets in exchange. If Susan calls Visa and disputes the charge, claiming that Bob never sent the requisite widgets, she may very well be able to convince Visa to reverse the charge. To protect himself from such a scenario, Bob must get extra information from Susan at the time of her purchase so that he can, if necessary, seek a legal resolution to such fraud. In this case, there must be trust between Susan and Bob for a transaction to occur.

Had Susan paid Bob in cash this would not be an issue since Bob would not need to trust that Susan delivers payment; she would have already done so in a non-reversible way. Cryptocurrency, like cash, enables trustless transactions.

Irreversible Transactions

The issue we just outlined-that trust is required in the modern financial system-is eliminated with cryptocurrency. This is primarily accomplished by the fact that transactions are irreversible. Once Susan pays Bob with Bitcoin, Susan can never undo that transaction.

Censorship Resistance

In many countries, there are strong capital controls. During Greece's financial crisis in 2015, the Greek government put limits on the daily amount of money its citizens could withdraw from banks and restricted the ability to transfer money out of the country. The goal was to prevent a run on the bank and to prevent capital flight, the phenomenon where assets or money move rapidly out of a country. This is usually caused by (and further exacerbates) a financial crisis. Chinese citizens can only withdraw about $15,000 a year in foreign currency. In other countries, such as Venezuela, the government inflates the currency to keep the economy moving at the expense of cash holders.

Control of cryptocurrencies can be totally distributed so that no individual or group can change the fundamental dynamics of the currency system. The network supporting Bitcoin is run on thousands of servers around the world. No single group controls enough of them to force through changes to the system that would destabilize it.

As a result of this, holders of cryptocurrency can transact freely among themselves. There are no limits to how they can transact and with whom. If you live in a country with strict capital controls or hyperinflation this is very attractive.

How Bitcoin Works

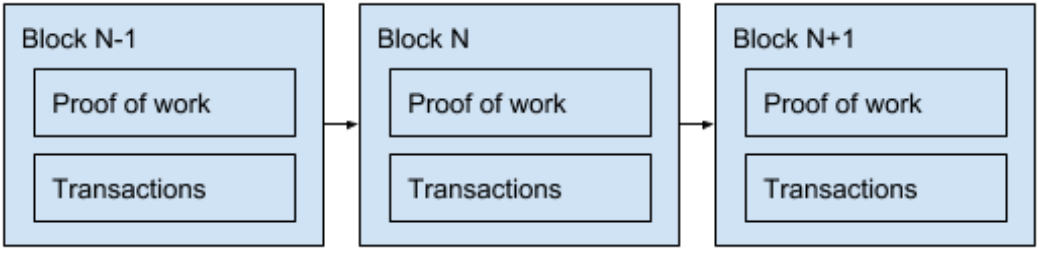
There are a few key components that enable Bitcoin to work. The first is the concept of a blockchain which is a type of public ledger. The next is the notion of cryptographic difficulty and proof of work. Finally, I'll describe the user experience of making transactions through wallets.

Blockchain

2017 could be called the year of the blockchain. The rapid rise in the price of Bitcoin led everyone, from politicians to pundits, to discuss Bitcoin. And no discussion of Bitcoin is complete without mention of blockchain. So, what is blockchain? The blockchain is the

underlying construct that enables Bitcoin to exist. Each transaction is encoded into a block on the blockchain. Each block builds on the prior block. And the entire blockchain is public.

Each node on the Bitcoin network contains a copy of the blockchain. Since each transaction on the Bitcoin network is encoded in a block, the entire transaction history of all Bitcoins for all time is publicly available.



Blockchain

Proof of work

Each block is connected to the next via Bitcoin's proof of work mechanism. This ensures that transactions cannot be replayed and that money cannot be double spent. You may have heard of Bitcoin mining. What is that exactly and how does it relate to the blockchain?

Computers on the Bitcoin network running the Bitcoin software are in a race. Each of them, let's call them "the miners", is looking for an input value that will solve a puzzle and whoever solves that puzzle first can claim the next block. Since finding a block comes with a reward, in the form of Bitcoin, miners are highly incentivized to find new blocks. The puzzle here is Bitcoin's proof of work algorithm. The output of the puzzle is the block's hash. A hash is a mechanism in cryptography of verifying that a certain input matches a specified output. Popular hashing algorithms are the MD5 and SHA-256 algorithms; Bitcoin uses SHA-256.

Bitcoin's proof of work requires that each miner attempts to generate a hash with a certain number of 0 digits at the front. Let's take the SHA-256 hash of the name of my dog:

```
from hashlib import sha256
dog = "Moose"
dog_encoded = dog.encode("utf-8")
sha256(dog_encoded).hexdigest()
```

This returns the hash `63af1f5974e26c12572178af017bdc3a6a4c648af3f1da79e5b5105011eae54d`

Note that every time we SHA-256 hash the name "Moose" we will get this value, without fail. You can go to this online hash generator https://passwordsgenerator.net/sha256-hash-generator/ to give it a shot yourself if you're too lazy to open up the notebook for this chapter!

Let's complicate things a bit. Let's add a "nonce" to our string to be hashed. A nonce is a value in cryptography which can be used only one time. In the Bitcoin proof of work algorithm, each miner takes the hash of the previously mined block and adds a nonce. They then hash this to try to match the desired output. In their case, they need to find a hash that has a certain number of leading 0's. Let's add a nonce to "Moose" and see how many tries it takes until we find a hash that starts with one 0.

```
from hashlib import sha256

def find_hash(num_zeros):
    nonce = 0
    dog = "Moose"

    while True:
        dog_nonce = dog + str(nonce)
        dog_nonce_encoded = dog_nonce.encode("utf-8")
        _hash_ = sha256(dog_nonce_encoded).hexdigest()
        if _hash_.startswith(num_zeros * "0"):
            return nonce, _hash_
        else:
            nonce += 1
```

This function takes in the number of preceding zeros we will require from the hash. With one preceding 0 the function takes about 50 microseconds to run on a laptop. We try:

```
%time nonce, _hash_ = find_hash(1)
print("Nonce: {}".format(nonce))
print("Hash: {}".format(_hash_))
```

and we get:

```
CPU times: user 46 µs, sys: 0 ns, total: 46 µs
Wall time: 50.8 µs
Nonce: 15
Hash: 0474e4ac2493bf794c74ebe538d1653259d979cdf43c445427df8b3efbe1a652
```

Note that just 15 tries were required to achieve a hash that begins with one 0. I know this because I started with a nonce of 1 and an increment of +1 for each try. You can check this by hashing "Moose15" with https://passwordsgenerator.net/sha256-hash-generator/.

> `%time` is an IPython "magic" function to calculate computation time. It is very handy, but will only work within a Jupyter Notebook or in IPython.

When we try to find a hash that starts with two 0's, things get much harder. Actually, the difficulty increases exponentially:

Hash	Nonce (number of tries)	Time Required
0474e4ac2493bf794c74ebe538d1653259d979cdf43c445427df8b3efbe1a652	15	50 microsecond
003486f2a623f900d13c02202a680fe92d4aff6bb46b7fbbce14fc072940e694	208	489 microseconds
000e4846f705fef51e3969aa20cc649d3c3b8c59eecded4880f97bc37801e6d9	4011	10 milliseconds
00006e0f59de96e389d813d004e261ea5be10d695287b0575bad5bc5d9ec59fa	254911	521 milliseconds
00000a31389788ab165e55565c7aa5a23007576dce1e3c610649e6b0a887d4bf	1021647	2.3 seconds
0000005086508716827a2e83edbab838a371b27137b11f58446309e28aee652e	42717565	1 minute 33 seconds

Hashing Difficulty

The more leading 0's the more difficult it is to solve Bitcoin's proof of work. Now, consider that the hash for the latest Bitcoin block was

```
000000000000000000054b27b0d27160729dfbb9be3ab3e1b1782b04cd8a70402
```

Instead of using a "Moose" as a seed, the prior block's hash is used, but otherwise, the principal is the same. The benefit of this proof of work mechanism is that in order to rewrite history and change the blockchain, you'd have to start with the first block and work your way forward. This would take an incredible amount of computing power–actually, it would take the same amount of computing power that has been required to get the Bitcoin blockchain to where it is today. Seeing as how Bitcoin miners are highly sophisticated and located in places where electricity is cheap, such as Iceland (due to its geothermal energy), it is not likely that anyone will be able to execute such an attack on the blockchain. Proof of work is power!

Wallets and transactions

How do you actually use Bitcoin? For that, you'll need a wallet. The easiest place to get a wallet is through Coinbase (http://coinbase.com/), which is a web-based wallet provider. But, there are also hardware wallets such as the Ledger (http://ledgerwallet.com/) and mobile wallets such as Airbitz (https://airbitz.co/). You can even create a paper wallet which is just a combination of a public key and a private key.

Bitcoin uses public key cryptography. You have a private key which you use to "sign" your transactions. When you send Bitcoin from your wallet to another person's wallet you use your public key and private key to sign the transaction. Since only the holder of your private key (this should be you!) can sign the transaction, then the transaction was verifiably yours. When you sign a transaction, that transaction is publicized by your wallet to the Bitcoin network so that it can be included in the next block. Once it is "confirmed" in the next block, it is written in history and can't be undone or reversed.

Cryptocurrency Exchanges

Cryptocurrency exchanges basically function like other currency markets. Traders buy and sell the cryptocurrencies as though they are a stock. Some are trying to buy cryptocurrency and others are trying to sell it. As buyers make *bids* to purchase cryptocurrency and sellers make *asks* trying to sell their crypto, the market reaches an equilibrium price. Each exchange has an order book which lists the *buyers' bids* and the *sellers' asks*. We'll use the Gemini exchange because they have a robust API for programmatic trading which we'll need to build our pipeline.

If you haven't already, head on over to http://gemini.com/ and sign up for a trading account. It may take a day or two for them to approve your account. Once you do, get a set of API keys so that you can follow along and make your own trades!

Orders and the Order Book

Let's stream the Gemini order book so that we can learn a bit more about what it contains. First, we need to establish a websocket:

```
import websocket
import json

ws = websocket.WebSocketApp("wss://api.gemini.com/v1/marketdata/BTCUSD", on_message=o\
n_message)
```

Next we need to define an on_message function-what we want to do once the websocket receives a message. Basically, if the event is an update to the order book, we want to print it out.

```
def on_message(ws, message):
    message = json.loads(message)
    if message["type"] == "update":
        for i in message["events"]:
            if "side" in i:
                payload = {"side": i["side"], "price": i["price"], "remaining": i["re\
maining"]}
                print(payload)
```

We can then run the websocket with:

```
ws.run_forever()
```

We should see orders streaming in live from the Gemini order book. They will look like:

```
{"side": "bid", "price": "11264.17", "remaining": "0.75"}
{"side": "bid", "price": "11266.24", "remaining": "3.18598"}
{"side": "ask", "price": "11266.25", "remaining": "0.00725886"}
{"side": "ask", "price": "11273.50", "remaining": "0.4"}
```

Each order is a dictionary which includes the following keys:

- Side: either bid (buy) or ask (sell)

- Price: the price being offered
- Remaining: the amount (in BTC) remaining for that specific order

So, the first row is a bid placed by a trader looking to buy 0.75 BTC at a price of $11,264.17 per BTC. Note that the ask orders are priced above the bids. The difference between the price at which buyers are willing to buy and sellers are willing to sell is called the bid-ask spread. Both buyers and sellers can "take" any order that is on the book immediately and execute a trade. If a trader has 0.75 BTC that she is willing to sell at $11,264.17, she may do so immediately and that bid order will drop off the order book. This removes liquidity from the order book because there is one less buyer available on the market. Orders that add liquidity to the market are called market "making" orders; these orders are not immediately fulfilled. Orders that remove liquidity are called market "taking" orders since they take the best price they can from the order book.

The Pieces of the Pipeline

In this chapter, I will provide an overview of each of the pieces of the pipeline which we will ultimately build: Kafka, Spark, Elasticsearch, and Kibana. We'll learn about each of these components individually so that you will be able to understand how each component can stand alone before tying them all together.

Kafka

Kafka is a distributed streaming platform which was originally developed at LinkedIn and later open sourced. Its development is now managed by the Apache Software Foundation and it is actively used by LinkedIn, Netflix, Spotify, Uber and many other major tech companies. In fact, the Docker container that we will use for Kafka is actually the one used and maintained by Spotify.

So, what exactly is a distributed streaming platform? First, let's focus on the streaming platform component. In the example use case we will pursue later in the book, we deal with a real-time price feed for Bitcoin. This feed produces data continuously, tracking the bid/ask pricing for Bitcoin on a major exchange (Gemini). We don't want to store this raw data, we want to place it in a queue to be processed downstream by Spark. This is where Kafka comes in. Kafka lets us create a "topic" which you can think of as a feed, much like an RSS feed, or even an application log. This topic supports two primary actions:

- **Produce**: add data to the topic
- **Consume**: read data from the topic

Much like articles in an RSS feed, the data in a topic is ordered based on the time it was added to the topic. When an application consumes from a topic it receives data in the exact order that the data was produced to the topic. In our use case, this makes sense as we want to process Bitcoin prices in chronological order.

The other key element of Kafka is that it is distributed. This means that a single Kafka instance can be deployed in a linearly scalable fashion across a cluster of servers. The elegance of the Kafka application programming interface (API) is that when we build our pipeline the

distributed nature of Kafka is abstracted away for us. So, we only need to focus on the topics, producers, and consumers. Behind the scenes, a given topic is partitioned across the Kafka cluster and significant orchestration is required in order to ensure delivery of the correct data to the consumer. But let's not get bogged down in that. Let's kick the tires on Kafka!

In the book's code repository, navigate to Chapter 5. Within the Chapter 5 directory, there is a folder for Kafka. In your CLI navigate there and run `docker-compose`:

```
cd ~/repos/realtimecrypto/chapter-5/kafka/
docker-compose up
```

You should see the Kafka service starting. The `docker-compose.yml` file in that directory specifies that Kafka's default port (9092) is made available to your computer. We can open another terminal window and navigate to the same directory. Make sure you've activated your conda environment, then go ahead and open a Jupyter Notebook and open `Introduction-to-Kafka.ipynb`.

Next, let's create a Kafka topic on the fly and produce some messages to it:

```
from kafka import KafkaProducer

producer = KafkaProducer(bootstrap_servers=['localhost:9092'])

msg_count = 20

for i in range(0,msg_count):
    msg = {'id': i, 'payload': 'Here is test message {}'.format(i)}
    sent = producer.send('test-topic', bytes(json.dumps(msg), 'utf-8'))
```

Note that here we've created a producer which instantiates a connection to Kafka, now available at `localhost:9092`.

We created 20 messages, each of which is a JSON object, and produce them to the Kafka topic we have called `test-topic`. The first argument for `send` is the name of the topic we wish to produce to. If that topic does not exist, it is created using default settings.

Now that we've created the topic, we can try to actually consume the messages we've produced.

```
from kafka import KafkaConsumer

consumer = KafkaConsumer('test-topic',
                         bootstrap_servers=['localhost:9092'],
                        auto_offset_reset='earliest')

for message in consumer:
    print ("%s:%d:%d: key=%s value=%s" % (message.topic, message.partition,   message\
.offset, message.key, message.value))
```

First, we create a consumer for `test-topic`. We instruct the consumer to have the earliest offset reset which instructs the consumer to begin reading data from the earliest possible position in the Kafka topic. Otherwise, the consumer will only read new data produced to the topic. Since the `consumer` object is iterable, we can simply begin consuming each message in it until we wish to stop the process. You should see messages flowing into the notebook like these:

```
test-topic:0:1: key=None value=b'{"id": 0, "payload": "Here is test message"}'
test-topic:0:2: key=None value=b'{"id": 1, "payload": "Here is test message"}'
test-topic:0:3: key=None value=b'{"id": 2, "payload": "Here is test message"}'
test-topic:0:4: key=None value=b'{"id": 3, "payload": "Here is test message"}'
test-topic:0:5: key=None value=b'{"id": 4, "payload": "Here is test message"}'
test-topic:0:6: key=None value=b'{"id": 5, "payload": "Here is test message"}'
```

We can grab the last message and actually load it as JSON to inspect it. Note that we'll access items in the message using dot notation as it is a Python object.

```
import json
msg = json.loads(message.value)
msg
```

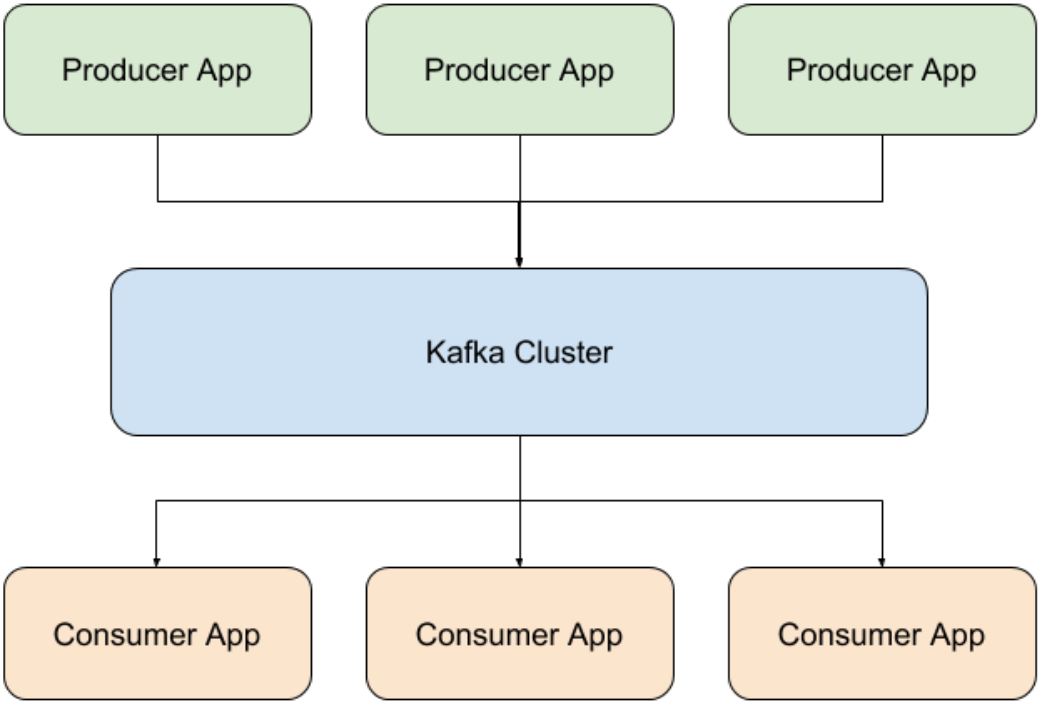
which returns:

```
{"id": 3, "payload": "Here is test message 3"}
```

For our real-time crypto pipeline, we are going to be producing to Kafka from a Python application and consuming the topic in Spark for stream processing.

Kafka Under the Hood

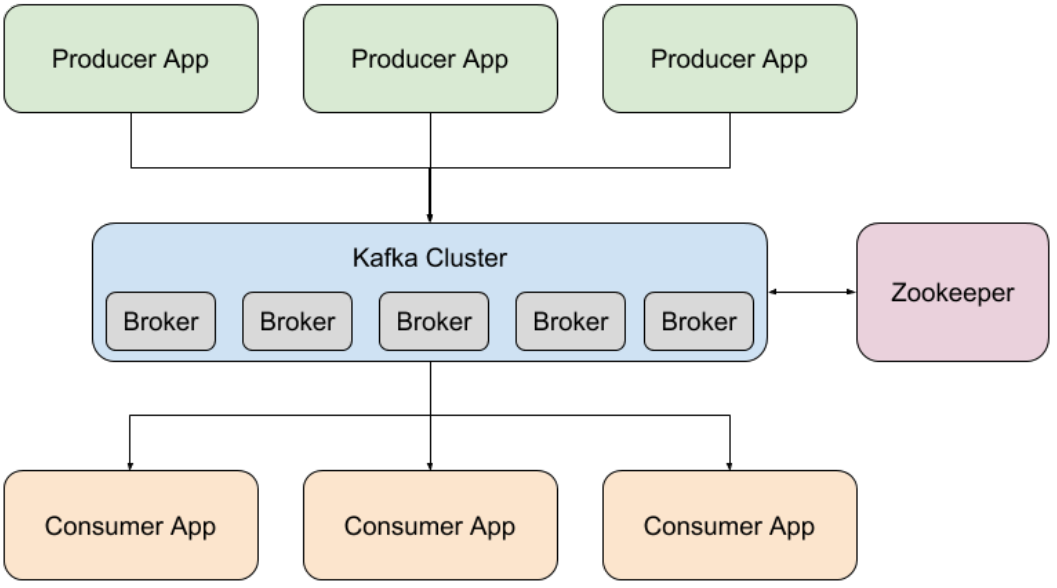
So how does Kafka actually work? For most data scientists and data engineers, the inner workings of Kafka can be somewhat mysterious and arcane. In most cases, you won't need to know too much about what's going on beneath the surface, but in case you do I want you to be prepared!



Kafka Architecture

One or more applications produce data to the Kafka cluster. This data can be consumed by one or more consumer applications. What makes Kafka so powerful is that multiple producers can write to the same topic and multiple consumers can consume that topic. This enables a huge amount of data throughput. More on this later.

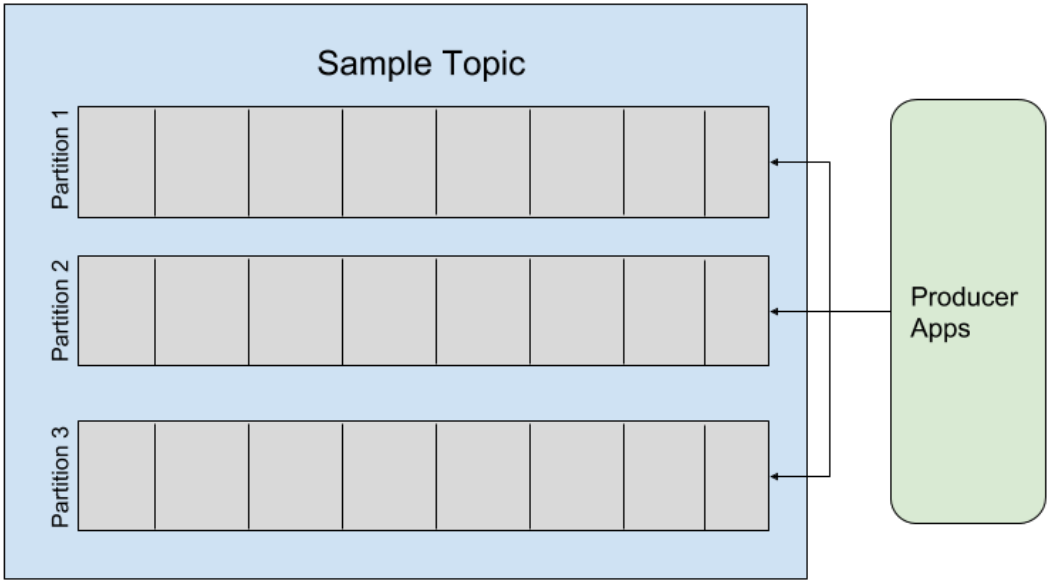
Each Kafka Cluster is comprised of one or more Kafka Brokers. A Kafka Broker takes in data sent to it by producers and returns that data to consumers seeking data from a topic. Coordination between Brokers is managed by Apache Zookeeper (https://zookeeper.apache.org/). Zookeeper offers a centralized service for maintaining data on the Kafka cluster, including which Brokers hold which topics and information about consumer groups.



Kafka Brokers

Kafka Topics and Partitions

Each topic in Kafka is split into a set of partitions. These partitions may be split across one or more brokers in the Kafka cluster. Each topic can be replicated a configurable number of times to ensure resiliency in case of broker failure.

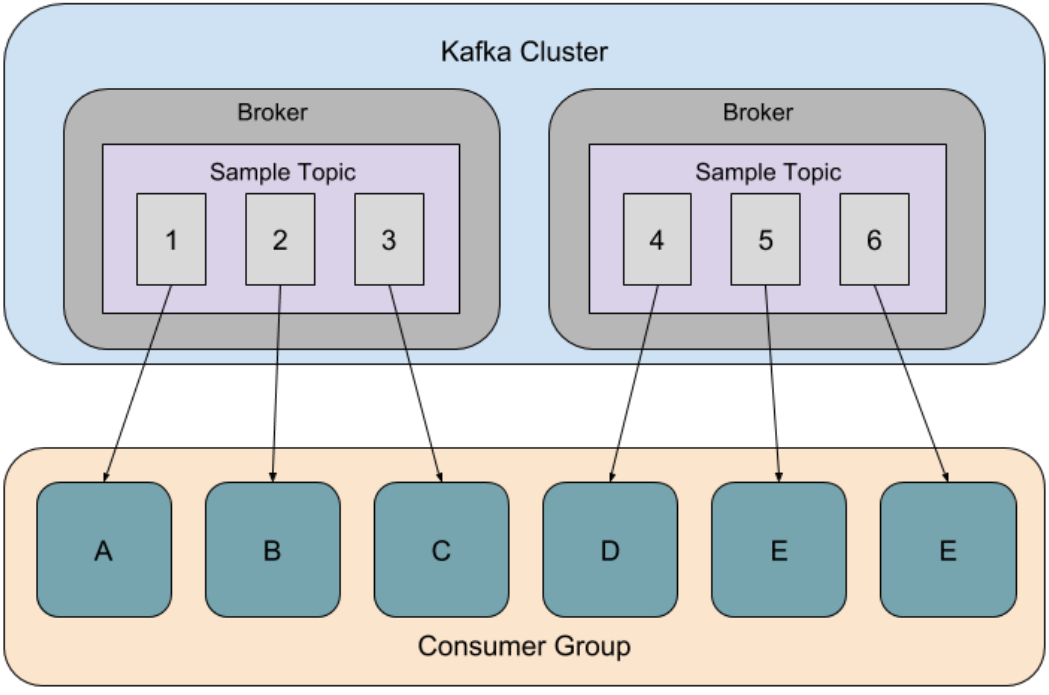


Kafka Partitions

When producers begin writing to a topic, they are provided metadata about the topic from the Broker; this metadata tells the producer which Brokers hold which topic partitions. The producer can then use a partitioning strategy to determine the partition it should append a given record to. Using a partitioning strategy ensures load balancing of writes across the Brokers which hold the specified topic.

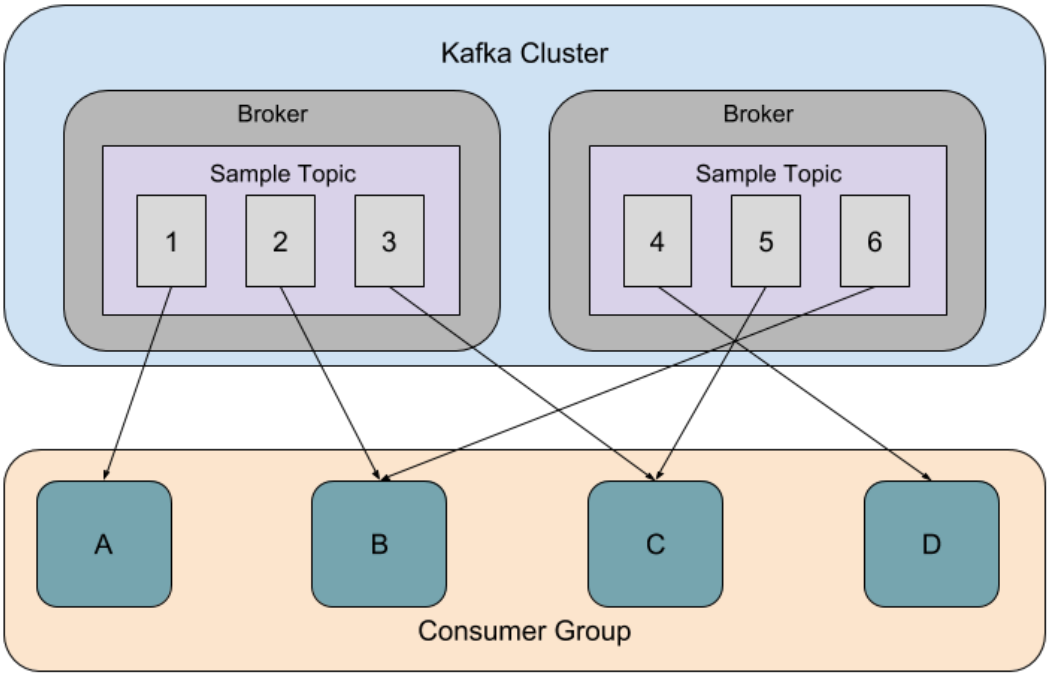
Each partition is basically a log written to disk. Producers append records to the partition (log) so that each record is stored at a given offset for that partition. These offsets are integer increments which indicate how many records have been written to the partition since the beginning of the partition. Consumers use offsets to track where they are within the topic; in case of failure, a consumer can pick back up at its last read offset and get itself back on track.

Consumer applications can be grouped together into consumer groups. These groups coordinate reading partitions so that each member of the group can consume a separate partition, enabling topic consumption in parallel. In the diagram below, a two node Kafka cluster has one topic ("Sample Topic") which has 6 partitions. The consumer group has 6 consumers so that each consumer can read directly from a partition. This maximizes data throughput.



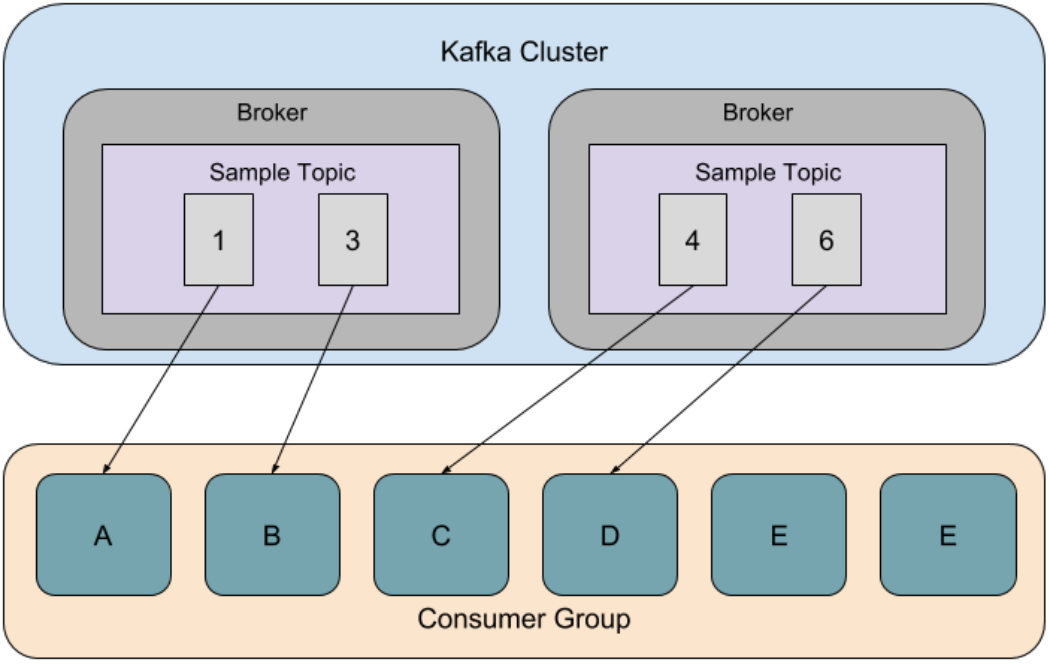
Kafka Topics

If there were fewer consumers in the consumer group than partitions in the topic, some consumers within the group will read from multiple partitions:



Fewer Consumers than Partitions

If, on the other hand, we have more consumers than partitions, some consumers will not read any data:



More Consumers than Partitions

In distributed streaming applications like the one we will build with Spark Streaming, this degree of understanding about how parallelism works in Kafka can be useful for conducting deep dives into performance tuning. For the most part (pun intended!) you won't need to stress over this.

Spark

Apache Spark is a free and open source cluster computing software. It can run on an arbitrary number of compute nodes and automatically scale your computing operations across the cluster, which is called parallelizing the operation. Spark is used to conduct massive computations by major tech companies including eBay, Amazon, Baidu, TripAdvisor, and even NASA.

Spark operates on data in memory. This means that once data is ingested by Spark it is held in shared stored memory across the cluster, as opposed to on the cluster's disk. The benefit this offers is Spark's ability to execute machine learning algorithms that require access to the entirety of the dataset, while iteratively finding an optima. One example of an algorithm that benefits from this approach is K-means clustering.

At its core, Spark is comprised of a small number of key components:

- **Spark SQL**: this component is similar to Pandas in Python in that it enables operations on dataframes
- **Spark Streaming**: this component enables Spark to operate on streaming data
- **MLlib**: this component allows Spark to train and deploy machine learning algorithms
- **GraphX**: this component allows for graph processing with Spark

We are going to be using Spark Streaming extensively for the remainder of this book. But first, let's discuss what we mean by parallelization. Imagine you have a dataset with records that look like this:

Name	Age
Bilbo	28
Frodo	26
Gandalf	62
Samwise	30
Sauron	72
Aragorn	31

Now, imagine that this dataset had trillions of data points recording the name and age of individuals. Let's say we have terabytes of this data. What if we want to calculate the average age across the dataset? We have two problems:

1. This data won't fit on our laptops, which don't usually have terabytes of disk storage
2. Even if we have a massive laptop hard drive, the processing will be painfully slow

Fortunately, this problem is known as an "embarrassingly parallel" problem. What this means is that parallelizing the problem across a cluster is very straightforward. This is because the average across the entire dataset is the mathematical equivalent of computing the average in chunks. In other words, we can split the problem into multiple parts and compute the average of each part. Then we can take the average of the averages. In the context of Spark, we would split this massive dataset across our entire cluster, which could be thousands of nodes. We would **map** the average function to each node, then **reduce** the results into a single answer. This **map reduce** paradigm will basically occur under the hood but it is worth having at least a basic understanding of:

- **Map**: send an operation (such as average) across a cluster

- **Reduce**: produce the final output using outputs of the map tasks

Let's dive into using Spark. First, let's get the Spark Docker container up and running with:

```
cd ~/repos/realtimecrypto/chapter-5/spark/
docker-compose up
```

You should see logs that include a link to `localhost:8888`. It might look like:

```
spark_1  |     Copy/paste this URL into your browser when you connect for the first t\
ime,
spark_1  |     to login with a token:
spark_1  |         http://localhost:8888/?token=531149475d883699188a479e89a01a55e3883\
20a829ecc7d
```

Grab the link to `localhost`, including the token. This grants you access to the Jupyter Notebook that is running inside the Spark Docker container. We need to run the `Introduction-to-Spark.ipynb` notebook inside the container so that it can communicate with the Spark application.

Let's go ahead and create a Spark Dataset from our Lord of the Rings age data. Included in the Spark directory for this chapter is a file called `ages.json` which includes the age data in JSON lines format. It looks like:

```
{"Name": "Bilbo", "Age": 28}
{"Name": "Frodo", "Age": 26}
{"Name": "Gandalf", "Age": 62}
{"Name": "Samwise", "Age": 30}
{"Name": "Sauron", "Age": 72}
{"Name": "Aragorn", "Age": 31}
```

Using Spark, we are going to read in this data and calculate the average age. First, we need to initialize a `SparkSession`:

```
from pyspark.sql import SparkSession
spark = SparkSession \
    .builder \
    .appName("Spark Example") \
    .getOrCreate()
```

Now, we can read in `ages.json` as a Spark Dataset:

```
df = spark.read.json('ages.json')
```

Now we have a Dataset (also called `DataFrame` in accordance with Pandas) representing our data. We can leverage the Spark SQL API to calculate an aggregation over the dataset, which in our case is an average:

```
df.agg({"Age": "avg"}).collect()
```

We can also execute calculations at the row level. For example, let's calculate each of the character's age in dog years (age times 7):

```
df.withColumn('dog_years', df.Age*7).collect()
```

This returns:

```
[Row(Age=28, Name='Bilbo', dog_years=196),
 Row(Age=26, Name='Frodo', dog_years=182),
 Row(Age=62, Name='Gandalf', dog_years=434),
 Row(Age=30, Name='Samwise', dog_years=210),
 Row(Age=72, Name='Sauron', dog_years=504),
 Row(Age=31, Name='Aragorn', dog_years=217)]
```

Best of all, this calculation would have scaled automatically across our computing cluster if we had more than one node. Notice something at the end of each of the commands above? If you are thinking, what does `.collect()` do then you're onto something.

Spark executes code **lazily**. This means that **transformations** such as calculating the characters' age in dog years is only executed once an **action** is called. The `.withColumn()` command is a transformation while `.collect()` is the **action** which causes the **transformation** to be executed. Often, the action which causes execution of our **transformations** is writing the job's output to disk, HDFS, or S3.

Let's try to create a new Dataset which includes the characters' ages in dog years, then let's write this out to disk:

```
df_new = df.withColumn('dog_years', df.Age*7)
```

Now we have a new Dataset called `df_new`. Note that nothing has been calculated yet; we have simply mapped the function we want across the cluster so that when we call an action on df_new such as `.collect()` or try to write the output to disk, the transformation will be executed. We can write df_new to disk with the following:

```
df_new.write.mode('append').json("dog_years.json")
```

You'll notice that a directory was created called `dog_years.json`. If you open this directory you'll see that there is a `_SUCCESS` log and an output file named after the partition. This file contains the output we are interested in.

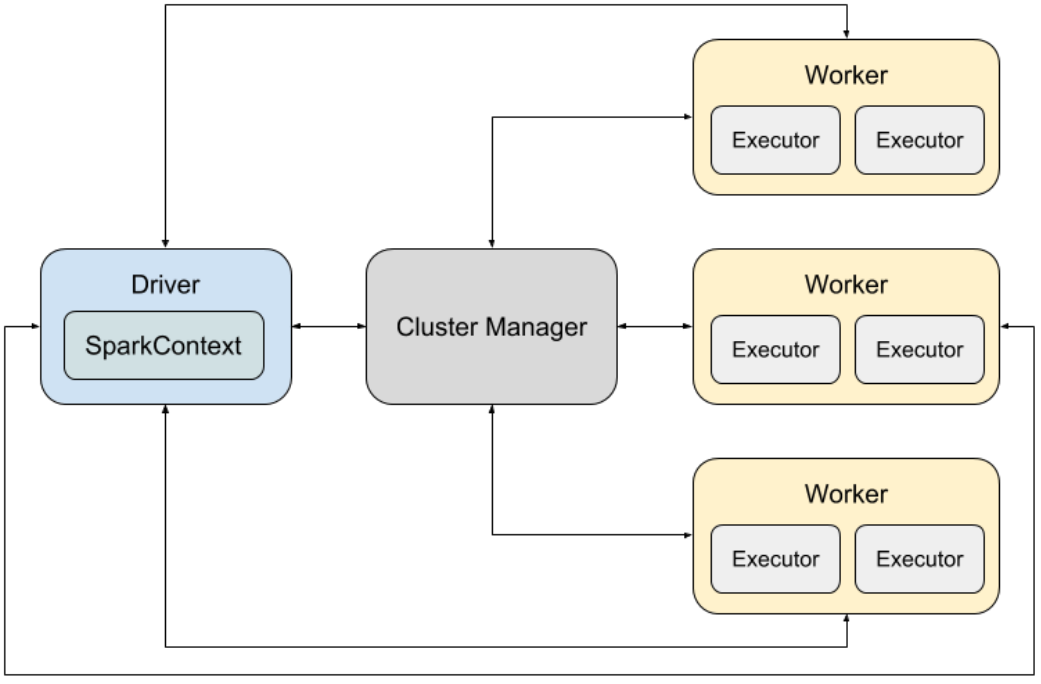
Under the hood, Spark splits your Dataset across the cluster using partitions. You generally want 2-4 partitions per core per node. So, if your cluster has 10 nodes with 4 cores each (a total of 40 cores) you would want between 80 and 160 partitions for your Dataset. Let's say you had 160 partitions. Each of the 40 cores would operate on 1 partition at a time. So, your cluster would progress through its tasks 40 at a time until all 160 partitions have been operated on. You don't worry about this too much, but it is something to be cognizant of when using Spark in a production setting with large clusters.

Spark Under the Hood

I sincerely hope that your time with Spark is spent writing code, not managing Spark deployments or fine-tuning and optimizing processing. However, if your Spark jobs are running slowly or you're dealing with other performance issues you might need to take a peek under the hood of Spark. I am here in your time of need!

Similar to Kafka, a Spark cluster is comprised of a set of nodes. We'll call these nodes workers. Workers report to the cluster manager. The cluster manager is in charge of figuring out which workers are available for work so that when an application is submitted to the cluster by a **driver**, the workers are appropriately tasked.

Once the cluster manager has assigned workers to a job the workers then report directly back to the driver (also known as the Application Master). In this way, coordination of job execution itself can be done by the driver.



Spark Architecture

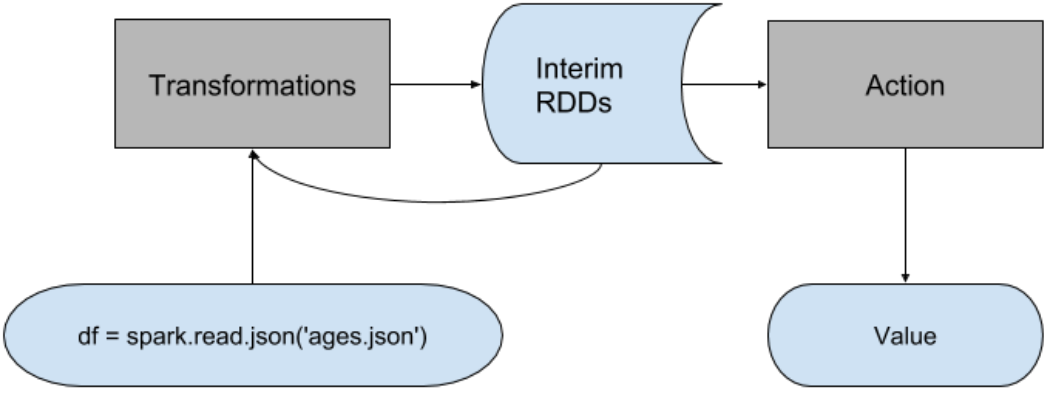
Think of it like a construction company. Workers at the construction company are scheduled for projects by managers at the company, but once they are on a project they report to the project supervisor for instructions. The project supervisor lets management know when workers are no longer needed for a project so they can be made available for other projects.

To do its part, the driver needs to figure out what each worker should work on. It examines the submitted job code and converts it into a directed acyclic graph (DAG) of various tasks. At a high level, these tasks typically include reading in data, manipulating it, and storing that data somewhere. The DAG enables the driver to ensure that work is done as efficiently as possible. This fits well because Spark is lazy! No, that doesn't mean it won't work for you, it just means it tries to do as little as possible at each step. Spark may actually not execute anything until it is time to store data as its final step. That is because Spark separates tasks into **transformations** and **actions**.

A **transformation** manipulates the data in place or converts data into a new RDD. An example of this may be converting a string to uppercase or filtering for lines which contain a specific keyword.

An **action** is a task which requires calculations to be performed on the transformation. These

include counting records in an RDD, storing these records as a file or to a data store, or taking a sample of the RDD.



Spark Workflow

In the above example, we read in the `ages.json` file as a DataFrame. The resulting DataFrame may be **transformed** with a filter such as:

```
filtered = df.filter("name = 'bilbo'")
```

This transformation creates a new DataFrame called `filtered`, but `filtered` is not calculated until we perform an **action** on it. We can do that by counting the number of records found in `filtered`:

```
count = filtered.count()
```

To provide a count of the records, an action must be performed, thereby calculating the entire filtered DataFrame. What if we are then asked to return all the records in `filtered`? Guess what, the entire DataFrame has to be recalculated since returning the records (calling `filtered.collect()`) is yet another action. This is a critical distinction and can cause major performance issues. In this case, we should call `filtered.cache()` on the DataFrame prior to calling any actions so that it is cached for each subsequent action and need not be recalculated.

Spark Deployment Options

There are a variety of ways you can deploy Spark:

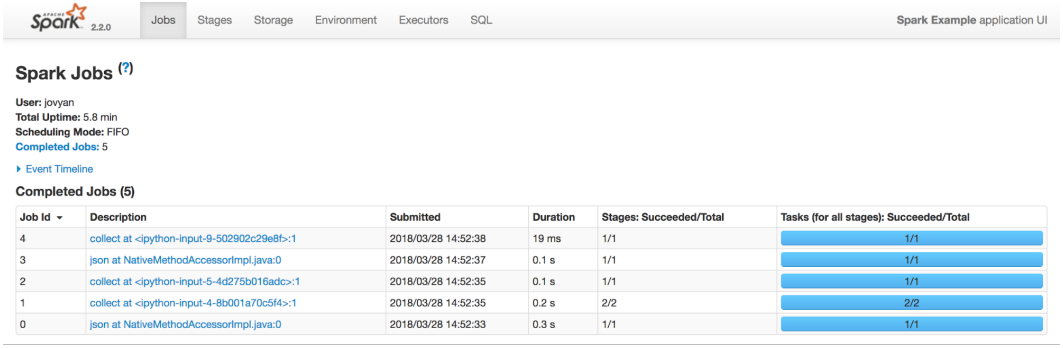
- **Standalone**: Spark provides the cluster manager using one of the nodes as the Spark Master/cluster manager. This is the easiest to set up and is what we will use for the remainder of this book.
- **Apache Mesos**: a cluster manager which uses the Mesos master as the cluster manager
- **Hadoop YARN**: the native Hadoop cluster manager which uses the YARN resource manager as the cluster manager. This is most useful in circumstances where you have other processing (Hive, Impala, etc.) running on the cluster and Spark is competing for the same resources. If you run Spark on Amazon Web Service through their Elastic MapReduce Services you will encounter YARN.
- **Kubernetes**: if you are using a serverless architecture and run your processing in containers (such as Docker) then you can use Kubernetes for scheduling your Spark jobs. This is probably the most complex and bleeding edge approach for deploying Spark and really only makes sense if you have a cluster already managed by Kubernetes.

Within these options, there are numerous ways that the cluster manager can be fine-tuned. Spark defaults to a first-in, first-out (FIFO) scheduling approach. This means that whoever submits their job first gets first dibs on the cluster's resources. Generally speaking, unless you are involved in infrastructure management or DevOps you won't need to worry about how Spark was deployed.

For our purposes, we will run Spark in standalone mode using Docker. Jupyter has published some great tools for doing this (see https://github.com/jupyter/docker-stacks/tree/master/all-spark-notebook), which we've incorporated into the docker-compose file in the book's repository.

The Spark Web Interface

The application driver exposes a web user interface (UI) at port 4040. If there is more than one Spark job running on the driver, each subsequent job can be found at the next port (4041, 4042, and so on). We've opened up port 4040 on the Spark Docker container so that you can explore the UI. Once you've stepped through the *Lord of the Rings* examples above try going to `http://localhost:4040` in your web browser. You should see:



Spark Web UI

This is the application UI. You can see that it lists out the jobs which we sent to Spark. Note that only actions are listed here: we called `collect` three times, we read in JSON data once and we wrote JSON to disk once. The application UI can be useful for identifying bottlenecks in your Spark job. If certain jobs or tasks take longer than expected this can be used to perform a deep dive on specifically what is happening.

Since our dataset is small and our cluster has just one node, you'll notice that all jobs (except the `collect` associated with the aggregation) have just 1 task associated with them. This is because our DataFrame has only 1 partition. This means that only one core is used for each of these jobs. If we call `df = df.repartition(10).cache()` then run the same jobs, we will see 10 tasks per job. By repartitioning the DataFrame, we enable a higher degree of parallelism: if we had 10 cores on this node then all tasks would run in parallel. Similarly, if we had 5 nodes with 2 cores each we would split the job evenly among the nodes and all processing would run in parallel. By calling `.cache()` on the repartitioned DataFrame we instruct Spark to read in the data just once, repartition it, then cache it. Otherwise, each time we perform an action we would need to repartition the data. At scale, this involves shuffling data around the cluster so we should avoid doing this repeatedly!

We'll use the application UI later to examine the performance of our Spark Streaming application.

Elasticsearch

Elasticsearch (ES) is a distributed document store and search engine. It is free and open source. It excels at handling unstructured documents such as HTML pages as it is fundamentally non-relational. It can also be used to index JSON objects like the ones which we will be generating. It's currently used by Netflix, Facebook, Walmart, Slack, Uber, Tinder and

many others. It's incredibly powerful software that has an entire ecosystem built around it. Logstash is typically used to pull logs from servers and store them in Elasticsearch. Kibana is dashboarding and business intelligence software which sits on top of Elasticsearch to allow you to explore the data. This is affectionately known as the ELK stack (Elasticsearch, Logstash, Kibana). We'll be using Elasticsearch and Kibana for our pipeline.

If you've ever used a SQL based relational database you are probably familiar with the concepts of **databases** and **tables**. The functional equivalents with Elasticsearch are **indexes** and **types**.

- **Index**: this is a collection of documents that have generally similar characteristics. For example, you might have an index for Tweets that your bot has collected. You might have another index for grocery price data. Think of an index as a SQL database.
- **Type**: each index has a document type. Prior to Elasticsearch 6 there could be multiple types per index. However, now, Elasticsearch only supports one type per index. So, we don't need to worry about this too much. Think of a type as a SQL table.

Let's spin up our Elasticsearch cluster in docker! Run the following:

```
cd ~/repos/realtimecrypto/chapter-5/elasticsearch/
docker-compose up
```

Now, you can test that Elasticsearch is live by opening up another terminal window and executing:

```
curl localhost:9200
```

You should be returned some basic information about the cluster that looks like:

```
{
  "name" : "LTcEeOp",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "_na_",
  "version" : {
    "number" : "6.2.3",
    "build_hash" : "667b497",
    "build_date" : "2017-09-14T19:22:05.189Z",
    "build_snapshot" : false,
    "lucene_version" : "6.6.1"
  },
  "tagline" : "You Know, for Search"
}
```

Now that Elasticsearch is up and running let's open up a Jupyter Notebook and kick the tires on our ES cluster. In the terminal window make sure you are in this chapter's ES directory and that you've activated the conda environment for this book and open up a Jupyter Notebook (just run `jupyter notebook`).

You should open up the `Introduction-to-Elasticsearch.ipynb` notebook. We can now go ahead and initialize our connection to Elasticsearch.

```
from elasticsearch import Elasticsearch

es = Elasticsearch()
```

Using the official Python client for Elasticsearch, we've just initialized a connection to ES using the default port (9200). Now we can use the es object we've initialized to create an index and generate some documents.

```
es.indices.create(index='test-index')
```

We've just created an index called `test-index`. Next, we can index some documents:

```
from datetime import datetime, timezone
import random
for i in range(0,100):
    doc = {
        "description": "random price data",
        "timestamp": datetime.now(tz=timezone.utc),
        'price': random.randint(1,100)
    }
    es.index(index="test-index", doc_type="test-type", id=i, body=doc)
```

We just generated 100 random price objects. We stored each with a randomized "price" between 1 and 100 and captured the time at which we indexed the object. Note that we also defined the `doc_type` on the fly as we indexed the objects. We called the type `test-type`. One of the helpful things about ES is that it can infer a schema from objects you provide it. In ES terminology this is called a **mapping**. Note that if your objects' schema change down the road, ES can accommodate that as well. Let's try it just to prove the point:

```
from datetime import datetime, timezone
import random
coin_types = ['dogecoin','bitcoin','ethereum','litecoin','dash']
for i in range(100,200):
    doc = {
        "description": "random price data",
        "timestamp": datetime.now(tz=timezone.utc),
        'price': random.randint(1,100),
        'coin_type': coin_types[random.randint(0,4)]
    }
    es.index(index="test-index", doc_type="test-type", id=i, body=doc)
```

We just created objects that have a new field (`coin_type`) and indexed them to the same type. Note that we randomly selected the coin type from a list so our coins are either dogecoin, bitcoin, ethereum, litecoin, or dash. Be careful! ES handles this gracefully, but it can also cause problems down the road if you expect all objects to have certain fields but they do not. We can compare two objects we indexed to see how they look. Let's grab `id` 42 and 142.

First let's grab `id` 42:

```
es.get(index="test-index", doc_type="test-type", id=42)['_source']
```

returns

```
{"description": "random price data",
 "price": 30,
 "timestamp": "2017-10-05T07:41:17.088744"}
```

and now we can try `id` 142:

```
es.get(index="test-index", doc_type="test-type", id=142)['_source']
```

returns

```
{"coin_type": "doge",
 "description": "random price data",
 "price": 52,
 "timestamp": "2017-10-05T07:45:01.241136"}
```

As expected, 142 contains the field `coin_type` while 42 does not. We can run searches against this data using the ES query syntax. This syntax is JSON based and can look a little confusing at first but once you've dialed in the queries you need for your pipeline you should be all set. Here's our query:

```
query = {
    "query": {
            "range" : {
            "price" : {
                    "gt" : 90
                    }
                }
        }
}
```

It's a range query, meaning that we are searching for objects where price is within a specified range. In our case, that is simply greater than (`gt`) 90. We could have asked for greater than or equal to 90 and less than 95. That would look like

```
query = {
    "query": {
        "range" : {
            "price" : {
                "gte" : 90,
                "lt" : 95
            }
        }
    }
}
```

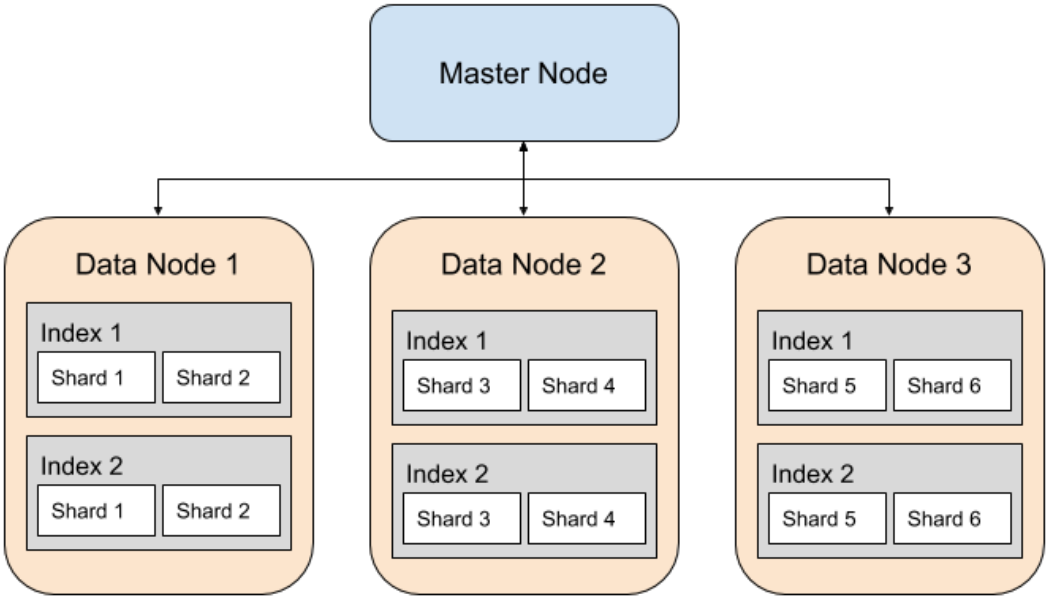
To actually run the search, try:

```
es.search(body=query, index='test-index', doc_type='test-type')
```

We have told ES that the query or `body` is our `query` JSON object. You should be returned all the objects of type `test-type` which matched our range query.

Elasticsearch Under the Hood

Elasticsearch is typically deployed as a cluster with a set of data nodes managed by a master node. The master node is responsible for creating and deleting indices and determining the status of the data nodes. Data nodes do pretty much what you'd expect: they store the data! When you create an index, its data will be shared across the cluster through a process known as sharding.



Elasticsearch Architecture

Data from any index can be stored across all available data nodes. Each data node will contain a configurable number of shards for each index. In a production environment it is likely (ahem, advisable!) that some of these shards will be replica shards. That means that if there is an issue with a data node the master node can route traffic to a data node which holds replicas of the down node's data. Elasticsearch's default replication factor is 2, meaning that in the above diagram each index would have 6 primary shards and 6 replica shards.

Sharding is critical to Elasticsearch because it provides redundancy in case of failure. It also provides parallelization of operations such as indexing new data and searching across an index. Finally, sharding enables horizontal scaling of an Elasticsearch cluster. Adding a new data node to the cluster isn't a big deal; shards can be reallocated to the new data node so that the workload across the cluster remains balanced.

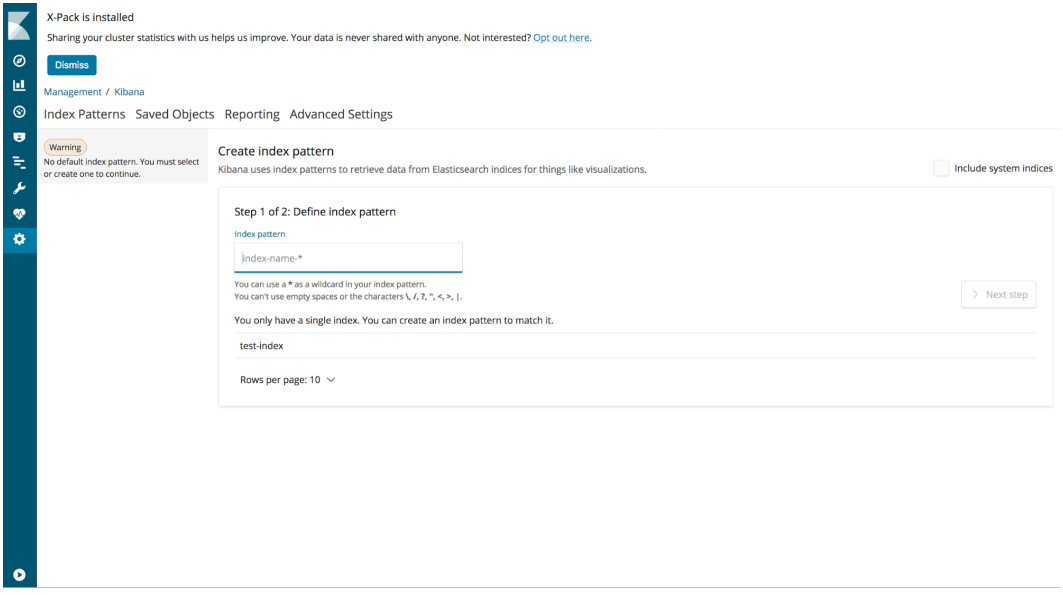
Kibana

Now we get to make pretty charts! Kibana is an open source data visualization platform which sits on top of Elasticsearch. The two work hand in glove; as you index data into Elasticsearch it becomes available for visualization in Kibana. Let's get Kibana set up and play around with the data we've just indexed.

If you've shut down the Elasticsearch docker container go ahead and restart it with the snippet below. Otherwise, you'll actually already have Kibana up and running (because we are sneaky like that and included it in the `docker-compose` file for ES!).

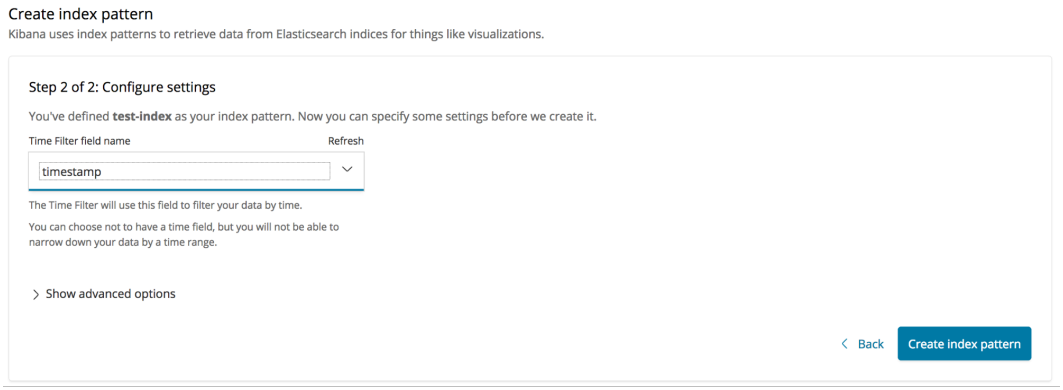
```
cd ~/repos/realtimecrypto/chapter-5/elasticsearch/
docker-compose up
```

Now, in your browser navigate to `localhost:5601`. By default, Kibana is configured to run on port 5601. When Kibana loads, navigate to the **Management** console. It should look like:



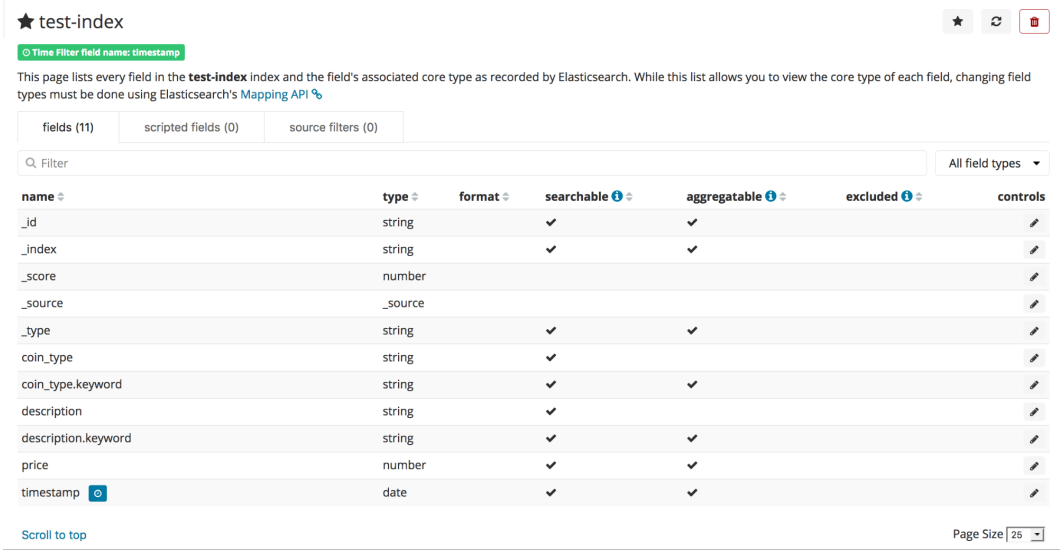
Management Console

Next, we need to explain to Kibana which index we would like to visualize. So, for the Index Pattern, you should insert `test-index`. You should leave `timestamp` as the value for Time Filter field name as this just lets Kibana know how we want to slice data by date. Now hit the Create button! You should then select `timestamp` as the Time Filter field name for `test-index`:



Create Index Pattern

You have successfully set up the index pattern and Kibana has interpreted the index's mapping:

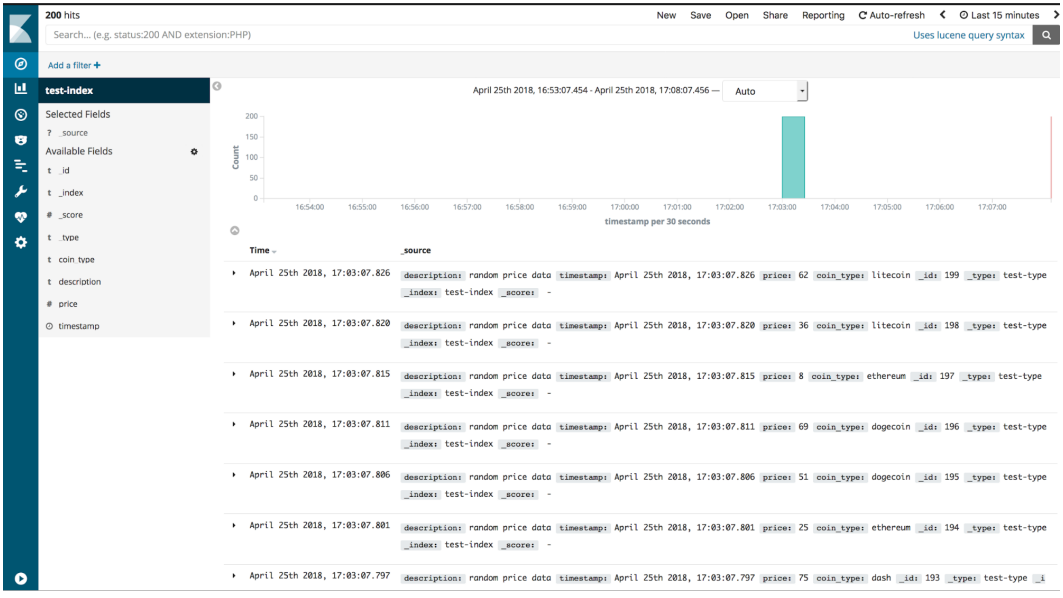


Index Mapping

Next, let's set the default time zone for Kibana to be UTC. Since we indexed our data using a UTC timestamp, which is a best practice, we want to make sure that Kibana is aware of this. So, navigate to Advanced Settings for `test-index` and change **dateFormat:tz** to **UTC**. That setting should now look like:

Time Zones

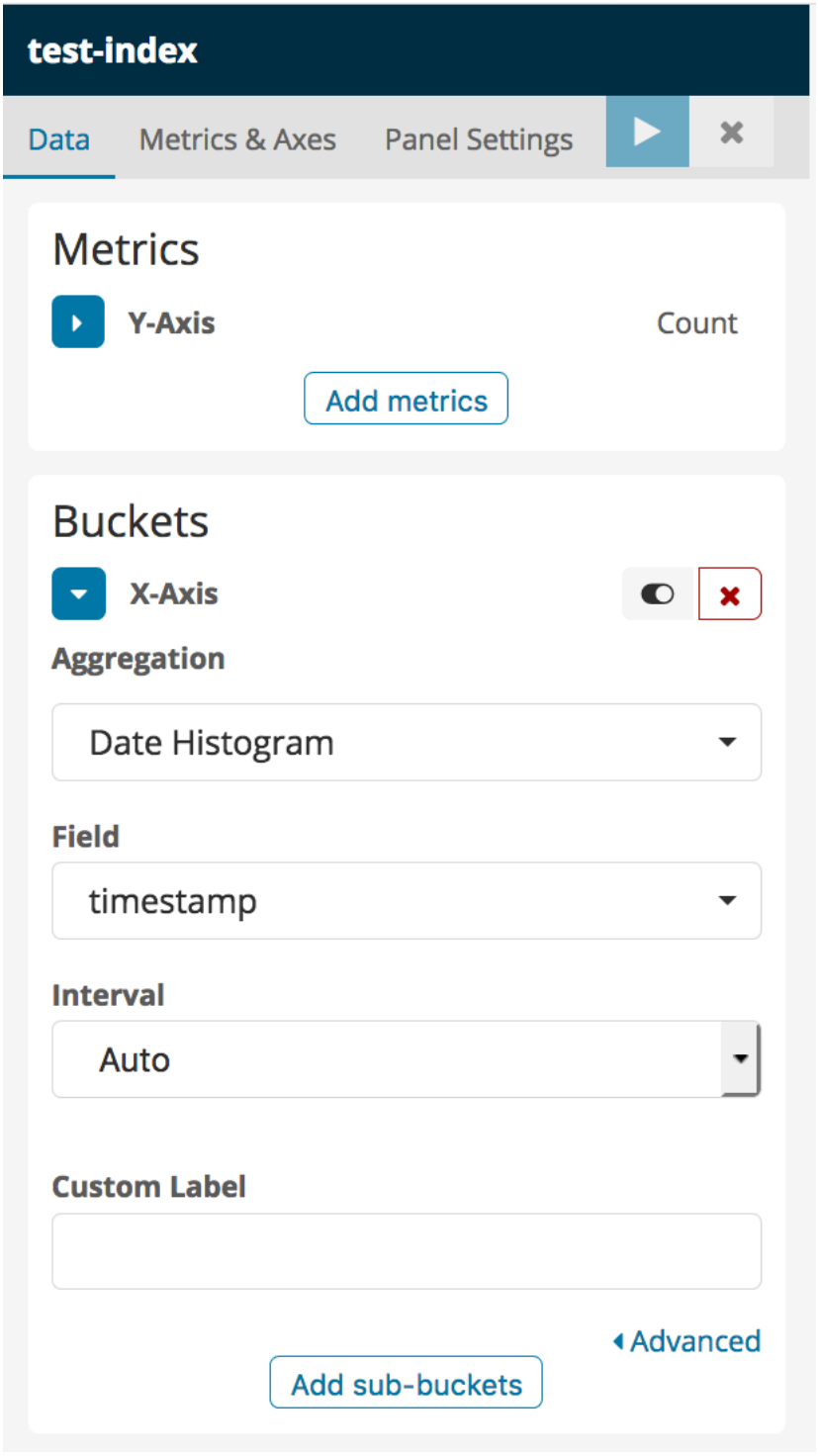
Navigate over to the **Discover** console. This is the most basic way in Kibana to visualize and interrogate data. Notice the time range selected in the top right-hand corner of your **Discover** console. This defaults to the last 15 minutes but if you indexed your data prior to 15 minutes ago you will need to change this. Kibana offers a robust capability to filter by time so you should have no problem changing the time range visualized to locate the data you indexed.



Discover Console

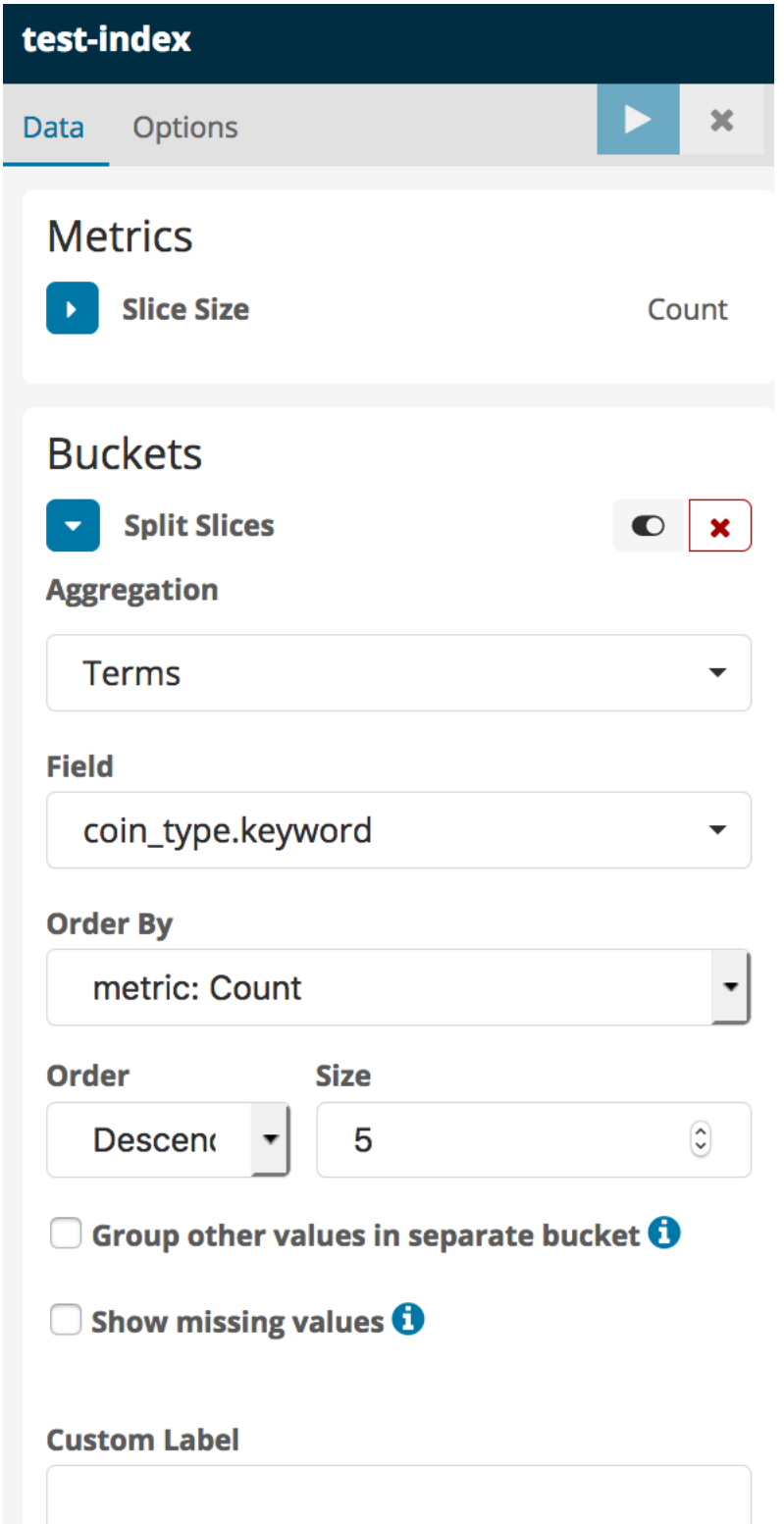
Click around! Try clicking on one of the horizontal bars to drill down by time until you can see a more granular representation of your data. This simple yet powerful visualization is critical for managing streaming data as it allows you to easily monitor stream volume.

Now let's try to make a dashboard. First, we need to build our visualizations. Let's create a date histogram like the one in the **Discover** console. Let's also create a pie chart of coin types. Navigate to the **Visualize** console and click "Create a visualization". Pick "Vertical Bar" from the options and select `test-index` as the index to build your visualization from. Now you should click "X-Axis" from the "Select buckets type" menu and "Date Histogram" from the "Aggregation" menu. It should look like this:

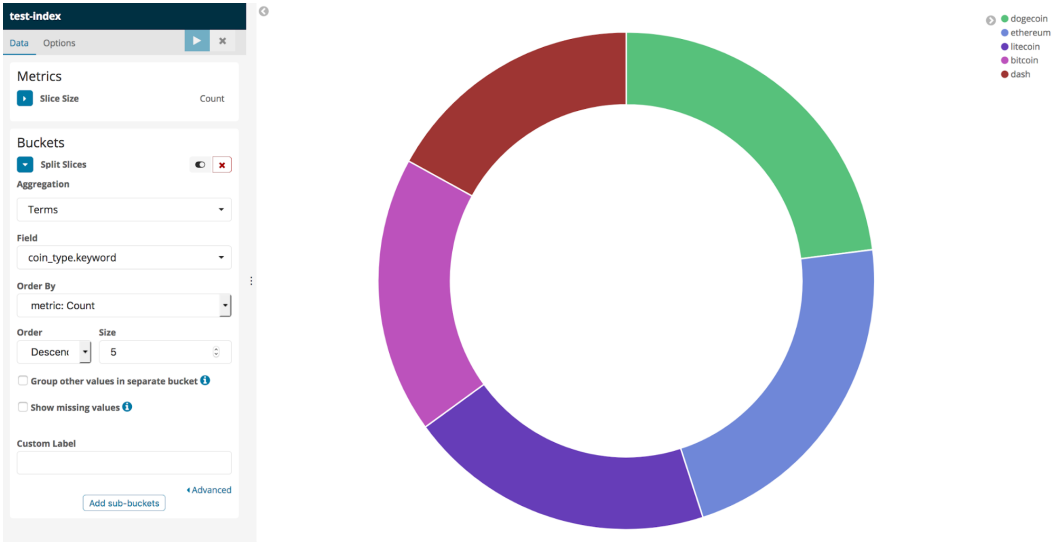


Date Histogram

Next, go ahead and hit the play button to check out your visualization. Again, you may need to change the time range to be able to visualize the entirety of your data. At the top of the page click “Save” and title this chart “Date Histogram”. Then, go back to the **Visualize** console and you should see “Date Histogram” listed. Let’s create another visualization with the + button. Let’s select a “Pie”. Again choose test-index as the index to operate on. From “Select buckets type” pick “Split Slices”. Now choose a “Terms” aggregation and pick “coin_type.keyword” from the “Field” menu. Your setup should look like:

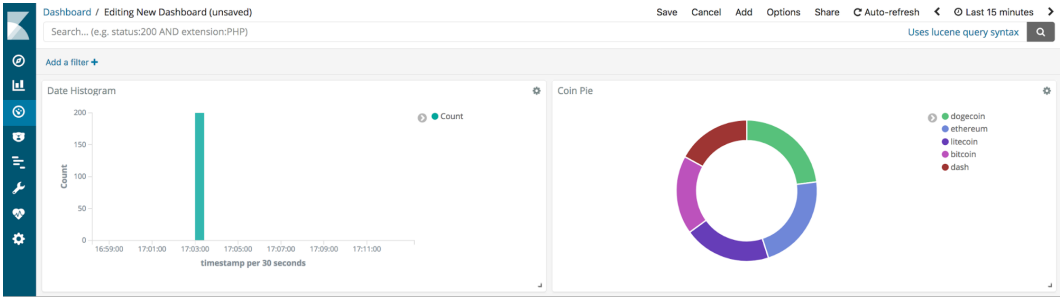

test-index
Data
Options
Metrics
Slice Size
Count
Buckets
Split Slices
Aggregation
Terms
Field
coin_type.keyword
Order By
metric: Count
Order
Descen
Size
5
Group other values in separate bucket
Show missing values
Custom Label

Now, when you hit the play button you'll see a pie chart visualizing the volume of coins indexed by coin type! Very nifty indeed.



Pie Chart Visualization

Let's save this visualization as "Coin Pie". Navigate to the **Dashboard** console and click "Create a dashboard". Using the top menu, click "Add" and select your "Date Histogram" visualization and your "Coin Pie" visualization. Your dashboard should look like:



Kibana Dashboard

Now, you can click anywhere on the dashboard and see how the data is filtered across the other visualization. Note that filters are added at the top, directly below the search bar. To remove a filter hover over it and click the trash can.

Filter

You can save this dashboard for later. You can create new visualizations and add them to it to create a rich reporting mechanism for tracking streaming data.

Bitcoin Price Forecasting

Now that we have a solid understanding of cryptocurrency from Chapter 4 and understand the pieces of our pipeline from Chapter 5, we are ready to train a machine learning model. In this chapter, we will use historic BTC price data to train a model which forecasts out future prices. This will be a critical component in our BTC trading pipeline. We're going to be training our model using Facebook's tool for time series forecasting at scale, Prophet: https://facebook.github.io/prophet/.

The Data

The data we'll use comes from Kaggle: https://www.kaggle.com/mczielinski/bitcoin-historical-data. You'll need to register with Kaggle in order to download the data. What are you waiting for! Note that the latest data available from Kaggle at the time of writing runs up to March 27, 2018. Make sure you use the most up to date version of the data. Let's use the Bitstamp historic data as that is a popular exchange.

Once you've downloaded the data, navigate to the directory for this chapter:

```
cd ~/repos/realtimecrypto/chapter-6
```

Next you should run `jupyter notebook` and open the `Bitcoin-Price-Forecasting.ipynb` notebook. Let's read the data into a Pandas DataFrame with:

```
import pandas as pd
df = pd.read_csv('bitstampUSD_1-min_data_2012-01-01_to_2018-03-27.csv')
```

We can inspect the data with:

```
df.tail()
```

This shows us the last records in the DataFrame. You should see a table like this (note, Kaggle continues to update the data so you may download more recent data):

Timestamp	Open	High	Low	Close	Volume_(BTC)	Volume_(Currency)	Weighted_Price
1508457360	5690.88	5690.88	5690.88	5690.88	0.168941	961.421706	5690.880000
1508457420	5698.13	5704.10	5695.63	5704.10	2.311662	13174.852877	5699.300163

BTC Trading Data

The timestamps here are in Unix Epoch Time which is the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970. Each row represents one minute of trading on the Bitstamp Exchange (https://www.bitstamp.net/). This includes the opening price, closing price, the volume, and weighted price. We'll use the weighted price–trade prices weighted by the volume of the trade–for training our model.

First, some data cleaning. Let's go ahead and convert the Epoch timestamp into Python datetime format and use this new field, `ds`, as the DataFrame index. We'll follow the naming conventions used in Prophet for now, hence ds.

```
from datetime import datetime
df['ds'] = df['Timestamp'].apply(lambda x: datetime.fromtimestamp(x))
df = df.set_index(df['ds'])
```

Next, let's take the log of the weighted price and set that as a variable called `y`. We'll use `log(weighted_price)` as opposed to just weighted price since we are trying to predict change in price, rather than raw price itself. Think of it this way: we can't say whether or not BTC price will trend upwards or downwards in the future. What we are trying to figure out is whether there are minute by minute, hourly, daily or even weekly patterns which we can exploit to make a profit (or, as our hedge fund friends might call it: Alpha).

```
import numpy as np
df['y'] = np.log(df['Weighted_Price'])
```

Then, let's filter down the data to just incorporate records from 2016 onward. This serves us two practical purposes:

1. We can assume that there has been more active trading in more recent years so patterns much older than 2 years ago may simply no longer hold
2. Less data means less computation required to train the model. We are lazy!

So, we'll create a new DataFrame called `df_subset`:

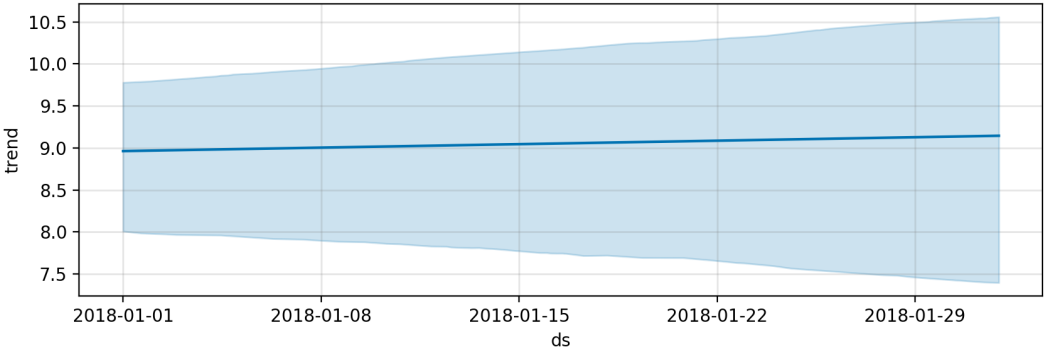
```
df_subset = df[df['ds'] >= pd.Timestamp('2016-01-01')]
```

Now, we are ready to do some forecasting!

Prophet for Profit

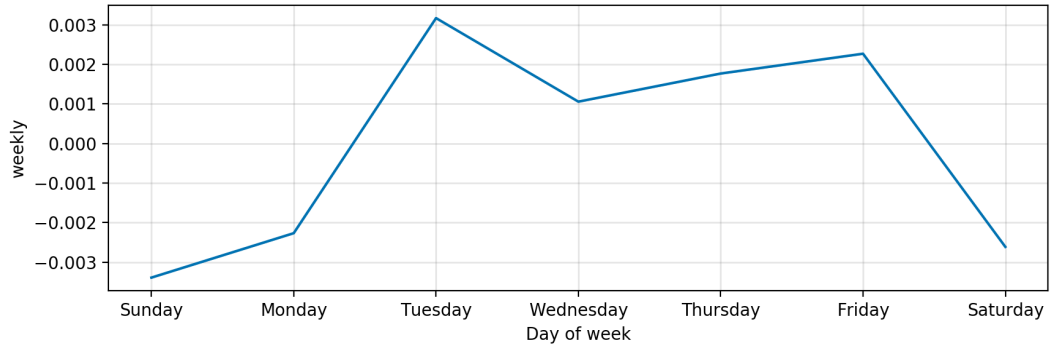
In 2017, Facebook published a paper called Forecasting at Scale (https://research.fb.com/wp-content/uploads/2017/11/forecastingatscale.pdf). As part of their research, they unveiled an open source project for time series forecasting called Prophet (https://github.com/facebook/prophet). Prophet is useful for taking historic time series data, such as BTC prices, and forecasting out where the data will be in the future. In our use case, we are interested in future prices of BTC. BTC prices are, of course, extremely volatile and no one can accurately guess whether prices will be up or down in a month, let alone 6 months from now. Which is exactly why we chose to use Prophet.

Prophet offers an additive regression model which extracts the general trend from monthly, weekly, and daily seasonality components. This means that we can separate out whether BTC prices are *generally* going up or down and make trades on whether BTC prices *specifically* rise and fall on particular days of the week and during particular weeks of the year.



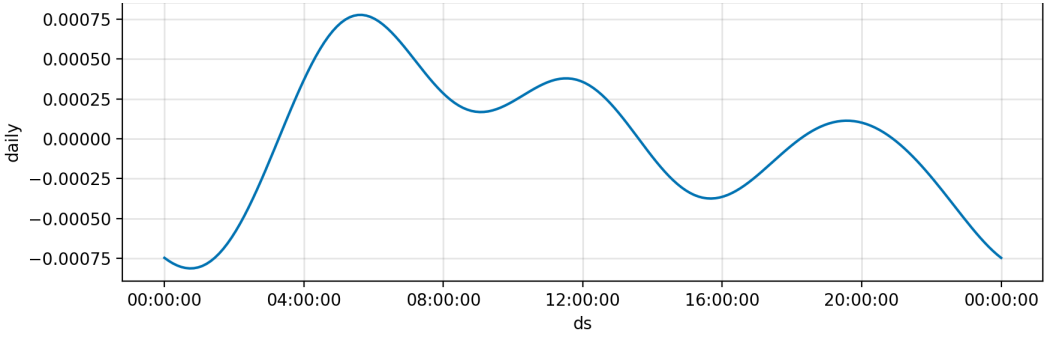
Forecast Trend

For example, we made a forecast for January 2018 using 2017 data. You can see that the general trend (chart above) is upwards.



Weekly Cyclicality

The weekly trend shows that prices typically rise Tuesday to Friday then drop over the weekend and continue falling through Mondays.



Daily Cyclicality

Within a day, price typically rises from about 4:00 AM GMT through a little after 12:00 PM GMT. Price then generally spends the rest of the day falling. Note that in the above charts the y-axis is the log of price, which represents *change* in price. So, any positive value is an increase in price and any negative value represents a decrease in price.

Under the hood, Prophet models time series data as a function of a trend (`g`), seasonality (`s`), and holidays (`h`) plus an error term (`e`).

```
y(t) = g(t) + s(t) + h(t) + t
```

The trend is extracted using a logistic growth model with automatic changepoint detection. Prophet uses Fourier series for seasonality detection. The user has the capacity to add holidays or other major events which are added as a regression component. In our use case, the hacking

of the Mt. Gox exchange is a worthy example of a "holiday" which we may wish to explicitly account for.

This may sound complicated, but in practice implementing Prophet is straight-forward. Let's go!

```
import pickle
from fbprophet import Prophet
m = Prophet()
```

First, we need to import prophet and initialize a Prophet object we'll call `m`. Next, we can actually train our model and save it as a pickle file for later use:

```
m.fit(df_subset[['ds','y']]);

with open('btc_model.pickle', 'wb') as pickle_file:
    pickle.dump(m, pickle_file)
    pickle_file.close()
```

That's it! Pretty simple, right? Don't be surprised if training the model takes several minutes. The good news is you can always unpickle the model whenever you need to, so you don't need to retrain it. To do that, just run:

```
with open('btc_model.pickle', 'rb') as pickle_file:
    m = pickle.Unpickler(pickle_file)
    m = m.load()
```

We have just naively trained a model using Prophet on our data and let it automatically detect trend and seasonality. Now we can use the model to forecast forward in time:

```
future = m.make_future_dataframe(periods=150000, freq='1min', include_history=False)
```

This generates an empty time series dataframe looking forward 150,000 minutes from the last datapoint in our BTC price data set. Note that if the data you use is older than a couple months you should forecast further out in time so that your bot has sufficient data to trade with. Now we can fit the model to this `future` dataframe:

```
fcst = m.predict(future)
```

Since we forecast out log(price) let's get it back into something that is human-readable by taking the exponent of the forecast value so that we can get back to price:

```
fcst['yhat_exp'] = fcst['yhat'].apply(lambda x: math.exp(x))
output = fcst[fcst['ds'] > '2018-01-01'][['ds','yhat_exp']]
output['delta'] = output.yhat_exp.diff()
```

Now we have a new DataFrame called `output` which contains the predicted price changes from minute to minute (`delta`). If `delta` is positive (price increases during that minute), we will consider that a "buy" minute; we should buy at the beginning of the minute as we expect price to rise over the course of the minute. If `delta` is negative, we will consider that a minute in which we should sell. We can add a feature called `side` which captures this information:

```
def choose_side(x):
    if x > 0:
        return 'buy'
    else:
        return 'sell'

output['side'] = output['delta'].apply(lambda x: choose_side(x))
```

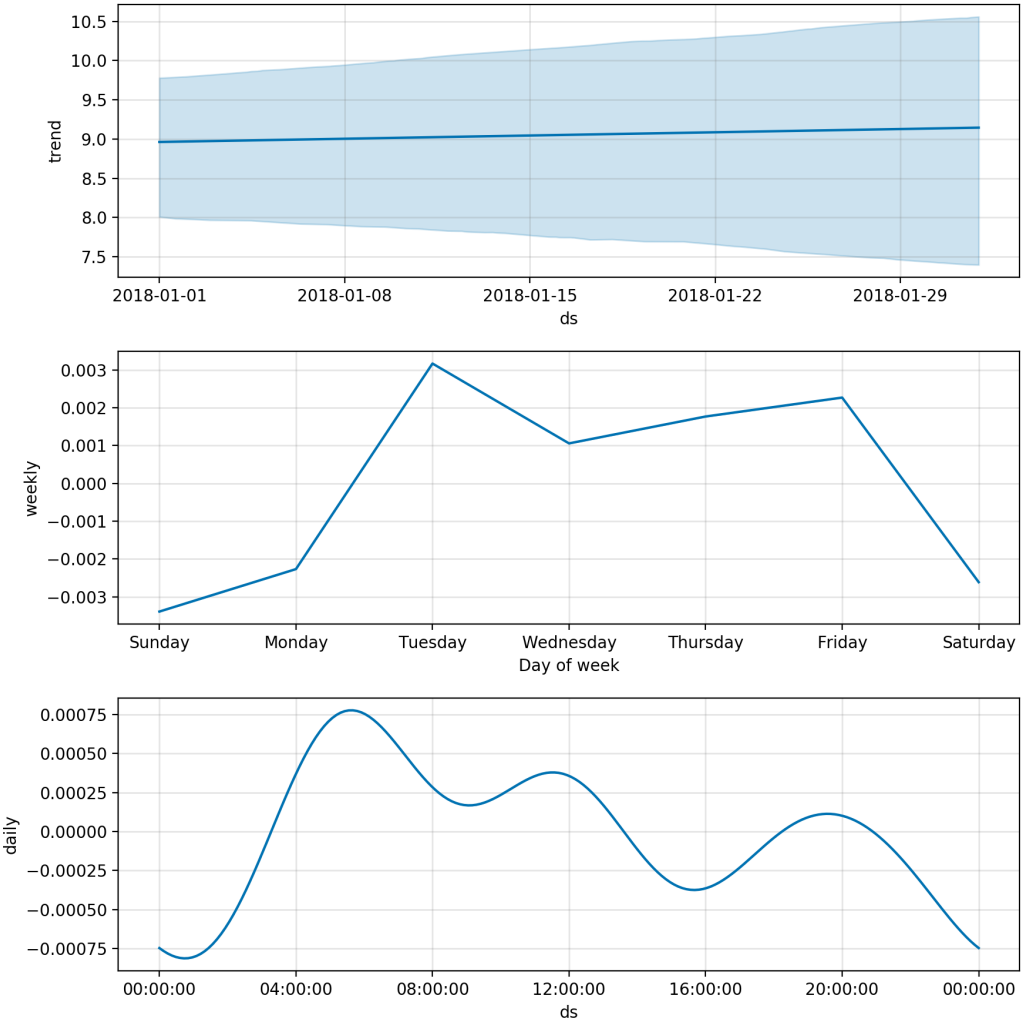
Finally, we can save our forecast as a CSV for later use.

```
output[['ds','side']].to_csv('2018_full_trade_forecast.csv', index=False)
```

If you want to plot the component chart just try:

```
fcst_sampled = fcst[fcst['ds'] > '2018-01-01 00:00:00']
a = m.plot_components(fcst_sampled)
```

Which will return the full component chart:



Components Chart

Congratulations, you've now trained a BTC price forecasting model using state of the art forecasting technology developed by Facebook!

Putting it all Together

Here we are! This is it. This is where we put together everything we've learned and build a cohesive data pipeline, enabled by machine learning, operating on Bitcoin exchange data. In this chapter we will:

1. Stream order data from Gemini
2. Pipe the order data to Kafka
3. Process data with Spark Streaming
4. Pipe processed data back to Kafka
5. Apply our Prophet forecasting model and execute trades on Gemini
6. Pipe our results into Elasticsearch
7. Visualize the pipeline in Kibana

Before we dive in, if you'd like to follow along make sure you navigate to the directory for this chapter and run docker-compose up:

```
cp ~/repos/realtimecrypto/chapter-6/2018_full_trade_forecast.csv  ~/repos/realtimecry\
pto/chapter-7/
cd ~/repos/realtimecrypto/chapter-7/
mkdir volumes
sudo chmod +777 -R volumes
docker-compose up
```

You should then look for logs from the spark-node which provide the link/token to the `spark-node`'s Jupyter Notebook server:

```
spark-node       |     Copy/paste this URL into your browser when you connect for the\
 first     time,
spark-node       |     to login with a token:
spark-node       |         http://localhost:8888/?token=4ce1b8dc72d53451de2511cba0f38\
fca140928f3dcead842
```

In this case, you'd want to go to `http://localhost:8888/?token=4ce1b8dc72d53451de2511cba0f38fca14` but note that your token will be different than this one. We'll actually be running the pipeline directly off the spark-node so it's important that you get it running.

Let's emphasize this point: ***all the notebooks for this chapter should be run from the Jupyter instance created by the `spark-node`*** This is critical as it enables Spark to communicate with Kafka. You should not need to run the `jupyter notebook` command at all, you should simply use the notebook instance created via `docker-compose up`.

> Note: if the Elasticsearch docker container fails to start it is likely unable to write data to its directory. In that case, try running `sudo chmod +777 -R volumes` once more to ensure that it has write access to the `volumes` directory.

Gemini Order Data

Streaming order data from the Gemini exchange is our first order of business. We touched on this in Chapter 4 but will revisit it here. To get started open up the `Gemini-to-Kafka.ipynb` notebook and run:

```
1  !pip install kafka websocket-client
```

Using a `!` before a command in a Jupyter Notebook is a really handy way to pass shell commands without having to directly access a shell on the server (or in our case, Docker container). We can use this to `pip install` a couple requirements we have: one is for Kafka and the other is for websockets. This is handy because we are able to install Python packages to our Docker container and we don't have to worry about environment and package management on our local machine.

Now we are going to create a class which will interact with Gemini:

```
import json
from kafka import KafkaProducer, KafkaConsumer
import websocket
import datetime
import json
import time
import base64
import hmac
from hashlib import sha384

class gemini_websocket(object):
    """
    An object for interacting with the Gemini Websocket. Full Gemini API documentatio\
n is available at https://docs.gemini.com
    """

    def __init__(self, kafka_bootstrap_servers):
        """
        Initializes gemini object.

        Args:
            gemini_api_key: your API key for the exchange
            gemini_api_secret: secret associated with your API key
        """
        self.kafka_bootstrap_servers = kafka_bootstrap_servers
        self.producer = self.create_producer(self.kafka_bootstrap_servers)

    def create_producer(self, bootstrap_servers):
        return KafkaProducer(bootstrap_servers=bootstrap_servers)

    def on_message(self, ws, message):
        message = json.loads(message)
        if message["type"] == "update":
            for i in message["events"]:
                if "side" in i:
                    payload = {"side": i["side"], "price": i["price"], "remaining": i\
["remaining"]}
                    sent = self.producer.send("gemini-feed", bytes(json.dumps(payload\
), "utf-8"))

    def on_error(self, ws, error):
        print("Error {0}, {1}".format(error, datetime.datetime.now()))

```

```
44      def on_close(self, ws):
45          print("Closed, {}".format(datetime.datetime.now()))
46
47      def on_open(self, ws):
48          print("Opened, {}".format(datetime.datetime.now()))
49
50      def run_websocket(self):
51          ws = websocket.WebSocketApp("wss://api.gemini.com/v1/marketdata/BTCUSD", on_m\
52  essage=self.on_message, on_open=self.on_open,on_close=self.on_close, on_error=self.on\
53  _error)
54
55          ws.run_forever(ping_interval=5)
```

There are a lot of pieces to this class so let's break it down function by function. First, we define an initialization function.

```
1   def __init__(self, kafka_bootstrap_servers):
2       """
3       Initializes gemini object.
4
5       Args:
6           gemini_api_key: your API key for the exchange
7           gemini_api_secret: secret associated with your API key
8       """
9       self.kafka_bootstrap_servers = kafka_bootstrap_servers
10      self.producer = self.create_producer(self.kafka_bootstrap_servers)
```

This function generates our websocket object and creates a connection with Kafka. The Kafka connection is then used to create a Kafka producer which will allow us to pipe a data feed from the websocket to Kafka. Note that this is actually accomplished by calling the class's function `create_producer`:

```
1   def create_producer(self, bootstrap_servers):
2       return KafkaProducer(bootstrap_servers=bootstrap_servers)
```

This simple function takes in the location of the Kafka servers, which is defined as an array of node:port objects: `[kafka-node-1:port, kafka-node-2:port, ... kafka-node-n:port]`. In our case we only have one Kafka node and it runs on port 9092. The function above uses Kafka's official Python library to use its built-in `KafkaProducer` class.

Next, we have a function which tells our websocket connection how to handle each object that comes through from Gemini:

```
def on_message(self, ws, message):
    message = json.loads(message)
    if message["type"] == "update":
        for i in message["events"]:
            if "side" in i:
                payload = {"side": i["side"], "price": i["price"], "remaining": i["re\
maining"]}
                sent = self.producer.send("gemini-feed", bytes(json.dumps(payload), "\
utf-8"))
```

In our case, we are going to check for updates to the order book which are either bid/ask orders. We extract the side (bid/ask), the price, and the amount of BTC remaining remaining in the order. We will then send this object to a Kafka topic called `gemini-feed`.

The `on_error`, `on_open`, and `on_close` functions are just basic loggers that print out information when we kick off the websocket. The final function actually creates the websocket:

```
def run_websocket(self):
    ws = websocket.WebSocketApp("wss://api.gemini.com/v1/marketdata/BTCUSD", on_messa\
ge=self.on_message, on_open=self.on_open, on_close=self.on_close, on_error=self.on_er\
ror)

    ws.run_forever(ping_interval=5)
```

This function, when called, will generate a websocket connection to Gemini's websocket for market data (Bitcoin in U.S. Dollars) at `wss://api.gemini.com/v1/marketdata/BTCUSD`. We provide the functions we described above for handling messages and we tell it to run forever (or until we cancel the process). We put it all together with:

```
gem = gemini_websocket(["kafka-node:9092"])
gem.run_websocket()
```

We are now streaming Gemini order data to Kafka! To test this out, open up a new notebook and run:

```
import json
from kafka import KafkaConsumer

consumer = KafkaConsumer("gemini-feed", bootstrap_servers=["kafka-node:9092"])

for message in consumer:
    print(json.loads(message.value))
```

You are consuming from the `gemini-feed` Kafka topic and should see the order data streaming into the notebook.

Calculating Market Liquidity with Spark Streaming

Now we are ready to begin processing data with Spark Streaming. We are going to use Spark Streaming to identify changes in market liquidity which could prohibit us from making smart trades. To get started, open up the `Spark-Streaming.ipynb` notebook. We should already have Kafka installed on this node, but just in case let's run:

```
!pip install kafka
```

Before we kick off Spark, we need to make sure that it initializes with Kafka configured. We do this by ensuring that the Spark Streaming Kafka integration library is called when we run Spark. We can do this within a Notebook by running:

```
import os
os.environ["PYSPARK_SUBMIT_ARGS"] = "--packages org.apache.spark:spark-streaming-kafk\
a-0-8_2.11:2.0.2 pyspark-shell"
```

Next, we can initialize a Spark Streaming context:

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils
from kafka import KafkaProducer
import json
import time

sc = SparkContext(appName="SparkStreamingPipeline")
sc.setLogLevel("ERROR")
ssc = StreamingContext(sc, 60)
```

We've just initialized a Spark Context called `SparkStreamingPipeline` with the logging set to `ERROR`. This means that Spark will only log errors. We could alternatively set this to `WARN`, `INFO`, or `DEBUG` to get progressively more logging information. We don't want to be drowning in logs so will stick with `ERROR` for now, but if you ever need to look under the hood of Spark this the best starting point.

In the final line above, we set the `StreamingContext` to have a batch interval of 60 seconds. This means that Spark Streaming will process 60 seconds of Gemini order data before doing anything. It will collect data off Kafka for 60 seconds, then treat that data as one batch to process. This parameter can be tuned depending on your use case. If you need real-time data processing you want to keep your batch interval as small as possible.

What is as small as possible? If your batch interval is too small and the data rate too fast then Spark Streaming gets behind on its data processing and you end up with a backlog. That can be completely counterproductive and means that your stream processing gets further and further from real-time the longer that it runs. We'll discuss this in a bit more detail once we kick off the Spark Streaming job.

For now, let's set up our DStream (discretized stream) from Kafka to Spark.

```
kafkaStream = KafkaUtils.createStream(ssc, "kafka-node:2181", "spark-streaming", {"ge\
mini-feed":1})
```

Here we establish an object called `kafkaStream` which we have connected to our kafka broker and port (`kafka-node:2181`). We assign `kafkaStream` the consumer group name `spark-streaming`. Having an assigned consumer group enables Spark to interact with Kafka in a distributed manner; since potentially many nodes in our Spark cluster could be interacting with Kafka we need to let Kafka know that these nodes are working together. This avoids issues like data duplication or data loss. Finally, we specify the topic we wish to consume

(`gemini-feed`) and the number of partitions to consume per Spark executor (1). Since we have one spark node and one kafka node (and our topic has 1 partition) this makes sense.

If we had a production cluster with multiple Spark nodes and multiple Kafka nodes we might want to consume more than one partition with our DStream receiver. We could also create multiple DStreams from the same Kafka topic and union them together; if they are all from the same consumer group (`spark-streaming`) they will consume different partitions of our topic, thereby increasing the parallelism of our data ingest. This more advanced usage would look like:

```
num_streams = 5
kafkaStreams = [KafkaUtils.createStream(ssc, "kafka-node:2181", "spark-streaming", {"\
gemini-feed":1}) for i in range(num_streams)]

unified_stream = ssc.union(*kafkaStreams)
```

Next, we can actually begin processing our stream (we'll use the single receiver `kafkaStream`). First, let's parse the message from `kafkaStream` into a JSON object:

```
parsed = kafkaStream.map(lambda v: json.loads(v[1]))
```

We can define a function to help us calculate the price to volume ratio. First, we will get the bid and ask volume weighted by price. For example, if the ask price is $1,000 and the volume is 500 then we can calculate the volume weighted by price as 500 * $1,000 or 500000.

```
def price_volume(x):
    price = float(x["price"])
    remaining = float(x["remaining"])
    x["price_volume"] = price * remaining
    return (x["side"], x["price_volume"])
```

This function, applied to each order from Gemini, will return a tuple of the side (bid or ask) and the volume weighted by price. Now we can apply this to each order using a `map` function:

```
parsed_pv = parsed.map(lambda x: price_volume(x))
```

Mapping lets us take a function (in our case the price_volume function) and apply it to each record passing through the DStream. Now we'd like to aggregate the price-weighted volume by each side for our 60-second batch:

```
grouped = parsed_pv.reduceByKey(lambda accum, n: accum + n)
```

Here we can use reduceByKey since each tuple in `parsed_pv` has a key that is either bid or ask. When we `reduceByKey` and use an accumulation function as we have above we instruct Spark to sum the price-weighted volume for each side for the minute in question. Our results will be a set of 2 tuples: one for bid and one for ask. Each tuple will contain the summed price-weighted volume for that side for the minute.

Since we don't know the order of these two tuples, the simplest thing we can do to organize our output is to sort by key. This will ensure that the first tuple is the ask tuple and the second is the bid tuple:

```
grouped_sorted = grouped.transform(lambda rdd: rdd.sortByKey())
```

Now, we'd like to ship these results off to Kafka for use in our trading bot. So, first let's make sure we have a Kafka producer available to be used by Spark to ship its processed data back to Kafka:

```
producer = KafkaProducer(bootstrap_servers="kafka-node:9092")
```

Then we can define a handler function which takes the DStream and sends its records to Kafka:

```
def handler(message):
    records = message.collect()
    for record in records:
        output = {}
        output["type"] = record[0]
        output["value"] = record[1]
        output["timestamp"] = int(time.time())
        producer.send("spark.out", bytes(json.dumps(output), "utf-8"))
        producer.flush()
```

This function collects the records in the message and formats them as JSON objects to be shipped to a Kafka topic called `spark.out`. Along the way, we generate an epoch timestamp so that each record passed to Kafka will look something like this:

```
{
  "type": "ask",
  "value": 10000000,
  "timestamp": 1523306670
}
```

We can use this handler function to send the records from `grouped_sorted` to Kafka using a `foreachRDD` function. The `foreachRDD` function maps a function to the RDD(s) created by the DStream. In our case, we'll send our price-weighted volumes per side to Kafka:

```
grouped_sorted.foreachRDD(handler)
```

Now we can calculate the price-weighted volume of asks to bids and ship that ratio to Kafka as well:

```
results = grouped_sorted.map(lambda x: ("price_volume", x[1]))
ratio = results.reduceByKey(lambda x, y: x/y)
ratio.foreachRDD(handler)
```

First, we create an interim RDD called results where we basically ensure that each tuple in the RDD has the key `price_volume` so that in the next `reduceByKey` step we are able to collapse the 2 tuples (bid/ask) into 1 tuple (`price_volume`). In this reduceByKey step, we are simply dividing the price-weighted volume for asks by the same for bids. Then we can ship the ratio to Kafka!

We have the option to print results as we process them. This isn't necessary but can be helpful for checking out our processing as it happens:

```
grouped_sorted.pprint()
ratio.pprint()
```

Guess what? Up to this point, we've been instructing Spark on what we want it to do, but it hasn't actually been doing anything! To get this job started we need to run:

```
ssc.start()
```

And with that, we are now processing Gemini data from Kafka, with Spark Streaming, and shipping our results back to a new Kafka topic.

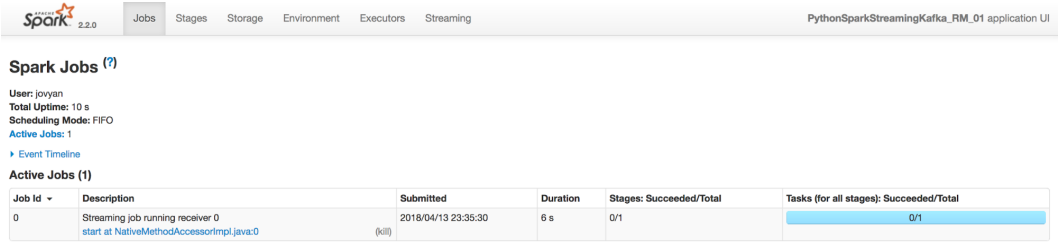
Monitoring Spark

With Spark Streaming the most important thing to keep track of is whether or not our Spark application is keeping up with the streaming data source. Tuning this requires adjusting the batch interval. If you recall, we set ours to a 60-second batch with:

```
1  ssc = StreamingContext(sc, 60)
```

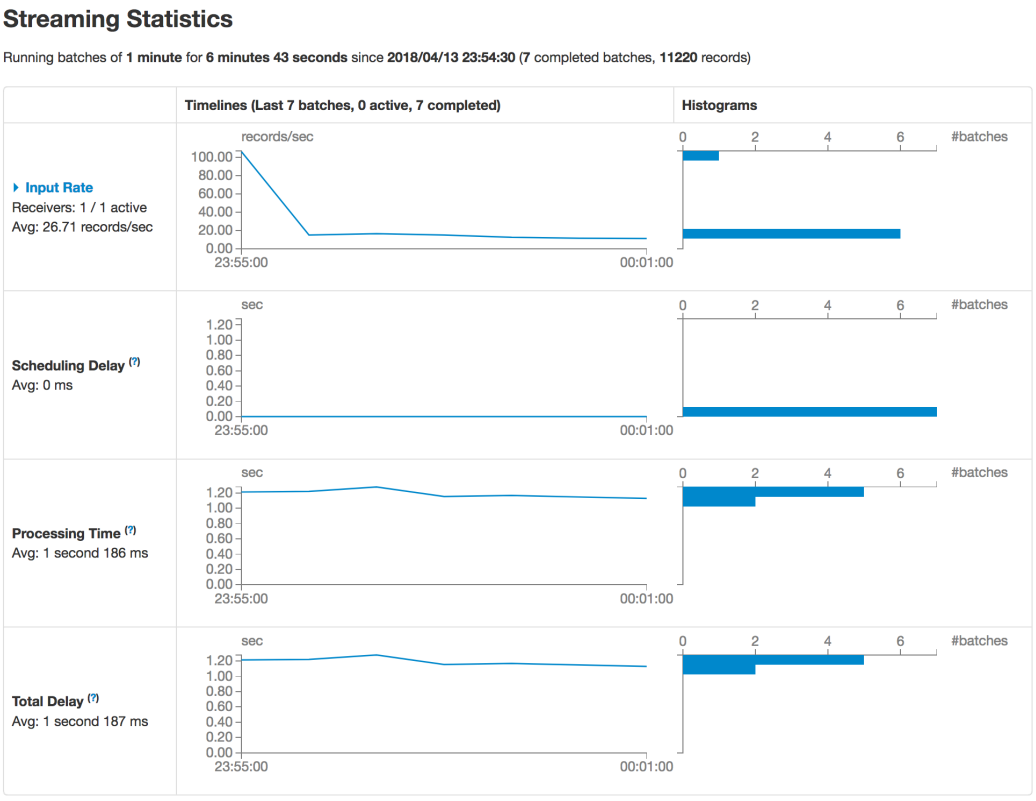
This is a very generous batch interval. It means that data streams in for 60 seconds and is then processed. The data must be processed in under 60 seconds so that it does not create a bottleneck. If data processing takes longer than the batch interval we start to see a delay in processing. This 60-second batch interval makes sense for us because our forecasting model is trained on minute-by-minute data. We need to aggregate data by the minute so we don't need real-time streaming data. If we had a different use case, we might start with a batch interval of 10 and then reduce it to a few seconds or even less than a second.

How can we tell whether we are experiencing processing delays? If you navigate to `http://localhost:4040` in your browser you should see the Spark Streaming job server:



Spark Job Server

At the top and right of the menu bar, click the link for "Streaming." This will show you the Spark Streaming job stats. What we are most interested in is whether "Total Delay" is exceeding our batch interval. In this case, it clearly isn't. We are well within reasonable operating parameters for Spark Streaming. Using this dashboard, you can also see how long it takes to process each batch, how many records have been processed in total, and how long each batch is taking to process. All systems nominal!



Spark Monitoring Dashboard

> One last thing to remember: we've only exposed port 4040 on our Spark docker container. In production, if you have multiple spark jobs running the first will run on 4040 then the next on 4041, then 4042 and so on.

Building a Trading Bot

We're finally here–the moment you've been waiting for! Now we get to the really fun part. We take our forecast which we created using Prophet and combine that with our analysis of the price-weighted volume of ask and bid information to execute trades. Let's recap what we've done and why we've done it:

1. We took the last 2 years of minute-by-minute Bitcoin price data and trained a model that will determine the price direction of Bitcoin each minute for 2018: in other words, whether we expect the price to rise or fall in a given minute.
2. We built a streaming connection to the Gemini Exchange which streams their order book (bids and asks with associated volume and price) and pushes that data to Kafka, a distributed message broker.
3. We created a Spark Streaming application to process the Gemini data off of a Kafka topic in order to calculate the ratio of bid and ask liquidity on the exchange for a given minute. This is a good measure of trading volatility and if it exceeds a certain threshold we may want to avoid making a trade. We pushed this information back to a different Kafka topic for use by our trading bot

Importing our forecasting model

That's a lot of moving pieces to keep track of, but that is the nature of a big data pipeline and I want this to be representative of what you will see in the real world! Now let's get started with our trading bot. The first thing it needs is our forecast information. Let's load it in and get it into an appropriate format with a little Pandas jiu jitsu:

```
import json
import pandas as pd
from datetime import datetime, timedelta
import time
from geminipy import Geminipy
from elasticsearch import Elasticsearch
from kafka import KafkaProducer, KafkaConsumer

# load in csv of forecasts
fcst = pd.read_csv("2018_full_trade_forecast.csv")

# generate a datetime field from the string timestamp `ds`
fcst["timestamp"] = fcst.ds.apply(lambda x: datetime.strptime(x, "%Y-%m-%d %H:%M:%S"))

# set the index as the timestamp so we can easily make lookups based on time
fcst = fcst.set_index("timestamp")[["side"]]
```

Connecting to Kafka

Since we'll be receiving data from Kafka we need to set up a Kafka Consumer:

```
consumer = KafkaConsumer("spark.out", bootstrap_servers=["kafka-node:9092"])
```

Connecting to Elasticsearch

Now we need to think about where we're going to store the output of our trades so we can track them. Here's where Elasticsearch comes into the picture. We're going to be keeping track of our trades with Elasticsearch and visualizing the outcomes with Kibana. So, let's initialize a connection to the Elasticsearch node. We need to connect to `elasticsearch:9200` since we are going to be running this bot from the `spark-node` docker container, not localhost; therefore, the Elasticsearch container is accessible via its name (`elasticsearch`).

```
es = Elasticsearch("elasticsearch:9200")
```

Note that from `localhost` (your computer) you can try curl `localhost:9200` to receive information about the Elasticsearch cluster or to run queries if you'd like.

Now, we need to create the `gemini` Elasticsearch index. A good practice is to check whether the index exists before trying to create it. If it already exists, then no need to try to create it since that will lead to an error:

```
if not es.indices.exists("gemini"):
    es.indices.create(index="gemini")
```

Connecting to Gemini

Next, we need to connect our bot to Gemini and set some thresholds for trading.

```
gemini_api_key = "some_key"
gemini_api_secret = "some_secret"
con = Geminipy(api_key=gemini_api_key, secret_key=gemini_api_secret, live=False)
```

Make sure to use API keys you received when you registered with Gemini. We've just established a connection to Gemini we'll call `con`. Note that we have set `live=False` which means that we are trading in the Gemini Sandbox. You can trade with real money if you'd like, (which I've done, by the way), but for the purpose of this book let's keep it in test mode.

Now let's think about two things:

1. How much should we trade: with each trade what percentage of our Gemini account balance should we put at risk?
2. What threshold should we set for minimum and maximum liquidity ratios?

For the first question, we will try trading 90% of our balance with each trade but you can configure this however you'd like. For the second question, let's require that the liquidity ratio threshold be between 0.1 and 2. This indicates that sell side volume is at least 1/10th to double that of buy side liquidity, suggesting (relatively) normal liquidity thresholds. If sell side liquidity is above 2x that of the buy side we would expect that the price will not rise in the next minute, despite whatever our model may tell us. That might indicate an external event causing a sharp sell-off which our model could not account for.

```
trade_pct = 0.9
threshold = (0.1, 2)
```

Defining basic trading functions

Now we need to define some basic functions that will let us smoothly interact with Gemini. These will help us:

1. Get our account balance
2. Get the current market clearing price (the ticker)
3. Create an order (so we can execute a trade!)
4. Format an order so that we can ship it to the exchange in the expected format

We also need some functions for pulling data from our model and for interacting with Elasticsearch:

1. Get the model's forecast for a given minute (either to buy or to sell)
2. Score our trade: determine whether or not we are making money or losing money!
3. Query Elasticsearch for the last time we made a buy trade (to help us score our trade)

This may sound a bit complicated but if we step through it one function at a time, by the end we'll have everything we need to create a trading bot.

Let's start by creating functions to find our account balance denominated in BTC and USD:

```
def get_btc_balance(con):
    for i in con.balances().json():
        if i["currency"] == "BTC":
            i["amount"] = float(i["amount"])
            i["available"] = float(i["available"])
            i["availableForWithdrawal"] = float(i["availableForWithdrawal"])
            i["doc_type"] = "balance"
            return i

def get_usd_balance(con):
    for i in con.balances().json():
        if i["currency"] == "USD":
            i["amount"] = float(i["amount"])
            i["available"] = float(i["available"])
            i["availableForWithdrawal"] = float(i["availableForWithdrawal"])
            i["doc_type"] = "balance"
            return i
```

Using a similar approach, we can use our connection to Gemini (con) to grab the latest ticker price for BTC:

```
def get_ticker(con):
    """
    NOTE: We need to ensure that numbers are numbers, not strings, for ES.
    Otherwise we would need to specify a mapping.
    """
    ticker = con.pubticker().json()
    ticker["ask"] = float(ticker["ask"])
    ticker["bid"] = float(ticker["bid"])
    ticker["last"] = float(ticker["last"])
    ticker["volume_BTC"] = float(ticker["volume"].pop("BTC"))
    ticker["volume_USD"] = float(ticker["volume"].pop("USD"))
    ticker["timestamp"] = datetime.fromtimestamp(ticker["volume"].pop("timestamp")/10\
00)
    ticker.pop("volume")
    ticker["doc_type"] = "ticker"
    return ticker
```

This ticker function returns the latest price and volume data for the exchange. Now we need to create an order for trading:

```
def make_order(con, amount, side, ticker):
    # if we are buying we should take the last ask price
    if side == "buy":
        bid_ask = "ask"

    # if we are selling we should take the last bid price
    elif side == "sell":
        bid_ask = "bid"

    order = con.new_order(amount = amount, # set order amount
                          price = ticker[bid_ask], # grab latest bid or ask price
                          side = side, # set side (either buy/sell)
                          options = ["immediate-or-cancel"] # take liquidity with an imme\
diate trade)

    # format order for Elasticsearch
    order_dict = format_order(order)

    return order_dict
```

This function takes in an amount (in USD), a side (either "buy" or "sell") and uses the last price from the ticker to place an order. Note that before returning the order from the function we first need to format it. This is so that we can store the order information in Elasticsearch:

```
def format_order(order):
    """
    NOTE: We need to ensure that numbers are numbers, not strings, for ES.
    Otherwise we would need to specify a mapping.

    Also, we convert epoch time to Python datetime which is natively recognized as a \
time field by ES.
    Epoch time, without a custom mapping, would appear as a number.
    """
    order_dict = order.json()
    try:
        order_dict["timestamp"] = datetime.fromtimestamp(int(order_dict["timestamp"]))
    except:
        # no timestamp field, try timestampms
        try:
            order_dict["timestamp"] = datetime.fromtimestamp(int(order_dict["timestam\
pms"]))
        except:
```

```
            # no timestampms, set to now
            order_dict["timestamp"] = datetime.now()
    order_dict["price"] = float(order_dict["price"])
    order_dict["original_amount"] = float(order_dict["original_amount"])
    order_dict["remaining_amount"] = float(order_dict["remaining_amount"])
    order_dict["avg_execution_price"] = float(order_dict["avg_execution_price"])
    order_dict["executed_amount"] = float(order_dict["executed_amount"])
    order_dict["doc_type"] = "order"
    return order_dict
```

Now let's define a function which analyzes our forecast model and determines the current and last minute's forecasted side (either buy or sell). We are interested in when these two differ because that indicates that we should make a trade. If the last 10 minutes were buying minutes and the current minute is a sell minute we would have done nothing in the last 9 minutes since we put our money in play during the first minute. Now though, our model says "sell" so we need to sell!

```
def lookup_side(fcst):
    # get current timestamp
    now = datetime.now()

    # we must add 1 minute to the time we lookup
    # this is because our forecasts are for whether we should have bought/sold in a g\
iven minute
    # so, we want trade with our prediction in mind (hence, add 1 minute)
    ts_prior = datetime(year=now.year, month=now.month, day=now.day, hour=now.hour, m\
inute=now.minute)
    ts = ts_prior + timedelta(minutes=1)

    last_fcst = fcst.ix[ts_prior].side
    curr_fcst = fcst.ix[ts].side
    return {"last_fcst": last_fcst, "side": curr_fcst} # return looked up side (eithe\
r buy or sell)
```

Now we need to define a function to score our trade: did we win on this one or lose money?

```
def score_trade(es, order, balance):
    last_price = get_last_buy(es)

    # get timestamp
    now = datetime.now()
    timestamp = datetime(year=now.year, month=now.month, day=now.day, hour=now.hour, \
minute=now.minute)

    # grab sell order price
    curr_price = order["price"]
    profit = (curr_price - last_price) * balance

    return {"profit": profit, "timestamp": timestamp, "doc_type": "score"}
```

Note that this function requires us to look up the last time we bought BTC and compare it to the price we are selling at in order to calculate profit or loss. The first line relies on a function called `get_last_buy` which queries Elasticsearch for this information:

```
def get_last_buy(es):
    query = {
      "sort" : [
            { "timestamp" : {"order" : "desc"}}
        ],
      "query": {
        "bool": {
          "must": [
            {
              "match_phrase": {
                "side": {
                  "query": "buy"
                }
              }
            }
          ]
        }
      }
    }

    results = es.search(index="gemini", doc_type="gem", body=query)
    last_buy = results["hits"]["hits"][0]
    last_price = last_buy["_source"]["price"]
    return last_price
```

This is a pretty standard Elasticsearch query. We have a `must` and `match_phrase` combination where we state that we are looking for only orders where the "side" was "buy". We are sorting by the "timestamp" field in descending order so that we can pull the most recent buy order we executed. Each time we sell BTC, we check the price at which we last bought BTC and calculate a profit on the trade. Later, we'll use Kibana to look at this over time to see our cumulative profit or loss.

With that, we are ready to create our core trading function:

```
def process_msg(message, fcst, con, es, threshold, trade_pct, last_price, traded):
    # load message as json
    msg = json.loads(message.value)

    # convert msg epoch time to datetime
    msg["timestamp"] = datetime.fromtimestamp(msg["timestamp"])

    # get ticker
    ticker = get_ticker(con)

    # check side from model
    lookup = lookup_side(fcst)
    side = lookup["side"]
    last_fcst = lookup["last_fcst"]

    msg["doc_type"] = msg.pop("type")

    # index msg to ES
    es.index(index="gemini", doc_type="gem", body=msg)

    # index account balance to ES
    es.index(index="gemini", doc_type="gem",body=get_btc_balance(con))

    # index ticker data to ES
    es.index(index="gemini", doc_type="gem", body=ticker)

    if side == "sell":
        balance = get_btc_balance(con)["amount"]
        if balance > 0:
            order = make_order(con, balance, "sell", ticker)
            score = score_trade(es, order, balance)
            print("Last trade yieled ${} in profit.".format(score["profit"]))

```

```
34              # index sell order data to ES
35              es.index(index="gemini", doc_type="gem", body=order)
36
37              # index score to ES
38              es.index(index="gemini", doc_type="gem", body=score)
39          else:
40              print("No balance to sell.")
41          return ticker["last"], False
42
43      else: # buy side
44
45          # only trade if we didn"t just trade
46          if not traded:
47
48              # Execute trade
49              if msg["doc_type"] == "price_volume":
50
51                  # only execute trade if price_volume within threshold
52                  if msg["value"] > threshold[0] and msg["value"] < threshold[1]:
53
54                      # trade with 90% of our balance
55                      trade_amount = round((get_usd_balance(con)["available"] * trade_p\
56  ct) / get_ticker(con)["last"],4)
57                      print(trade_amount)
58
59                      # execute order
60                      order = make_order(con, trade_amount, side, ticker)
61
62                      # index order data to ES
63                      es.index(index="gemini", doc_type="gem", body=order)
64
65                      # print order
66                      order.pop("timestamp")
67                      print(json.dumps(order, sort_keys=True,
68                          indent=4, separators=(",", ": ")))
69                      print("\n")
70
71                      return ticker["last"], True
72                  else:
73                      return last_price, False
74              else:
75                  return last_price, False
76          else:
```

```
77              return last_price, True
```

This is the most complex function with the most inputs so let's break it down. First, we consume messages from Kafka which are the output of our Spark Streaming application. Next, we grab ticker information and determine whether our model predicts this minute is either a buy or a sell minute. We also look up the last minute's forecast to check whether we need to execute a trade.

Before we do anything else, we push all the information we have from Kafka and Gemini to Elasticsearch. This includes the liquidity ratio we calculated, the ticker price, and our current balance.

Selling

Next, if our model forecasts that we need to make a sale because the price is dropping, we should check whether we've already sold off our BTC. If our BTC balance is greater than 0, let's place a sell order and index the results to Elasticsearch. We can also use this as an opportunity to score our last trade by comparing the price we last bought BTC at against our current selling price.

Buying

If our model forecasts that we should buy (that price is going up) we should first check whether we've already put our money into play. If we've just traded, no need to do anything. Otherwise, we should check that the liquidity ratio calculated by Spark falls within the threshold we set previously. If so, we can make a purchase and push the results to Elasticsearch.

Running the bot

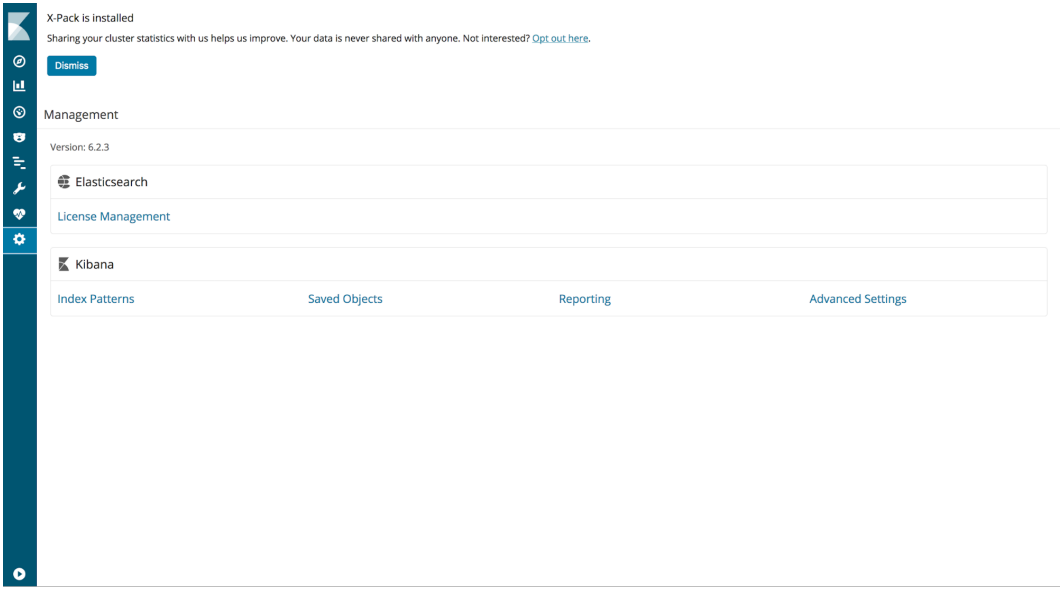
Finally, we can actually kick the tires on this thing and run the bot! We should initialize it by indicating that we haven't yet traded and we should provide the last ticker price as a starting point. Then, we can just start consuming from Kafka and processing messages with the functions we established above.

```
# get last price for keeping track of performance
last_price = get_ticker(con)["last"]
traded = False

for message in consumer:
    try:
        last_price, traded = process_msg(message, fcst, con, es, threshold, trade_pct\
, last_price, traded)
    except Exception as e:
        print("Error occurred: {}".format(e))
```

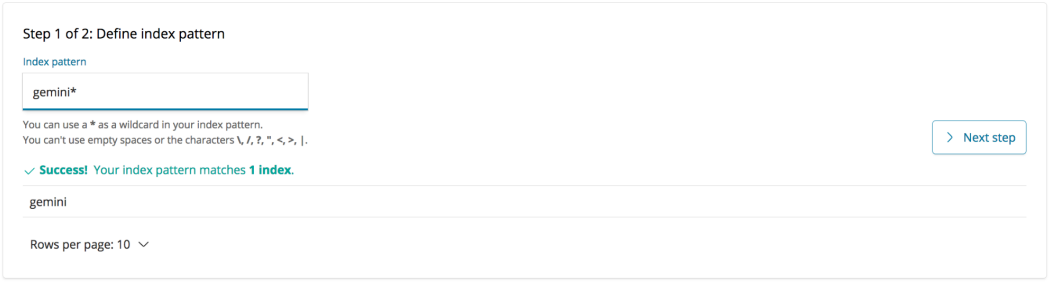
Configuring Kibana

Now that our bot is running, it should be shipping data to an Elasticsearch index called `gemini`. This means that we can now configure Kibana and start monitoring our progress! First, navigate to `http://localhost:5601` and click on the gear wheel at the bottom of the menu bar at the left to get to the **Management** console:



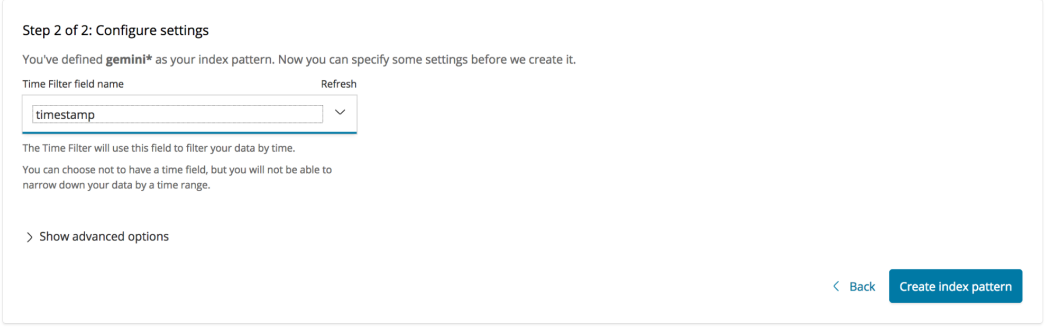
Management Console

Next, click the link under "Kibana" for "Index Patterns". Here you should see something like the above. In the Index Pattern box you can type "gemini":



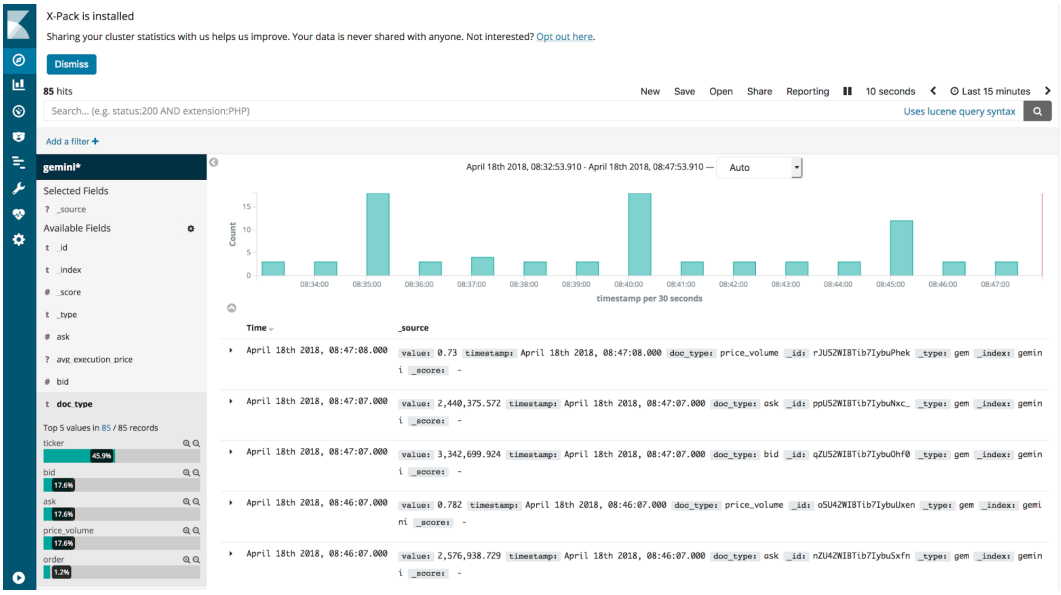
Create Index Pattern

You will want to configure the Time Filter field to be our field called “timestamp”:



Time Filter Field

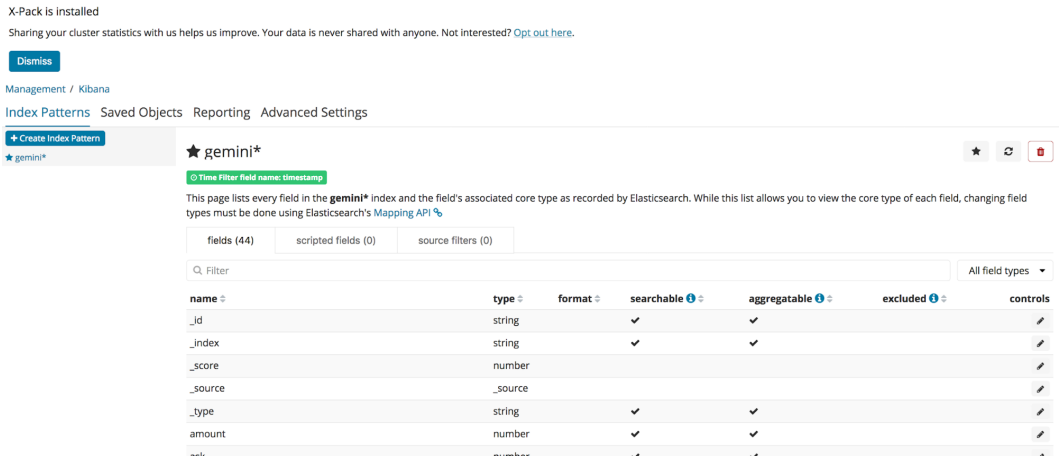
In the discover tab, by now you should see data streaming in!



Gemini Data

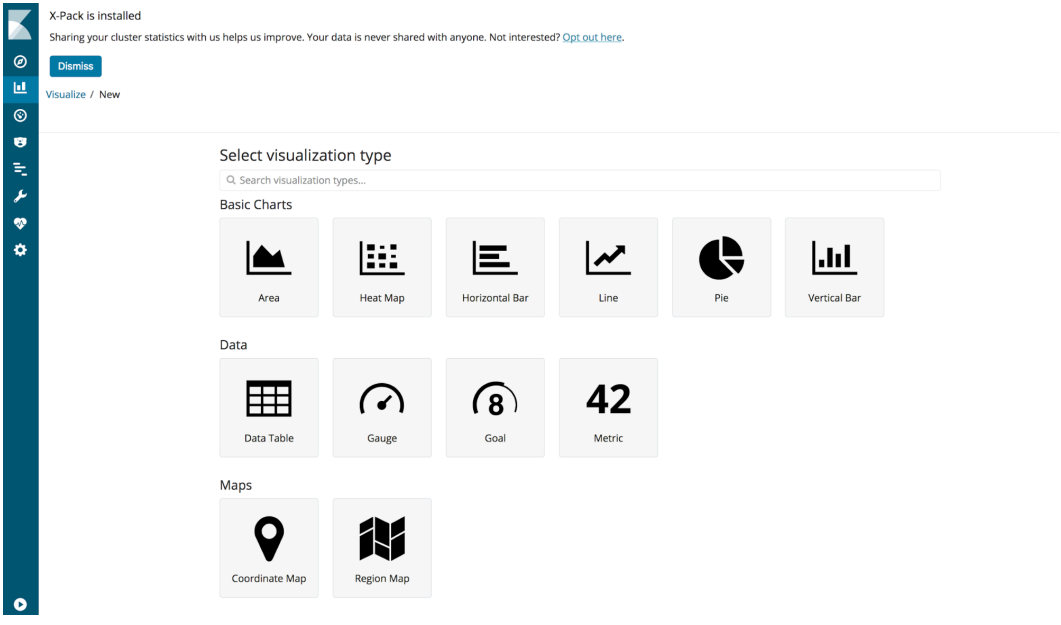
We can check our trading progress by creating a custom visualization. First though, let's make sure our mapping is up to date by refreshing the field index. Under the Manage tab, select "Index Patterns" once more and click the refresh button for the `gemini` index in the upper right of the screen. This ensures that Kibana is aware of the various fields which we have been indexing–including the `profit` field for when we score our trades.

> Depending on when you begin trading, you may get "stuck" in either a buy or sell cycle where the algorithm does not predict a price change for several minutes or hours. In this event, it may take some time to see scored trades appearing in Kibana.



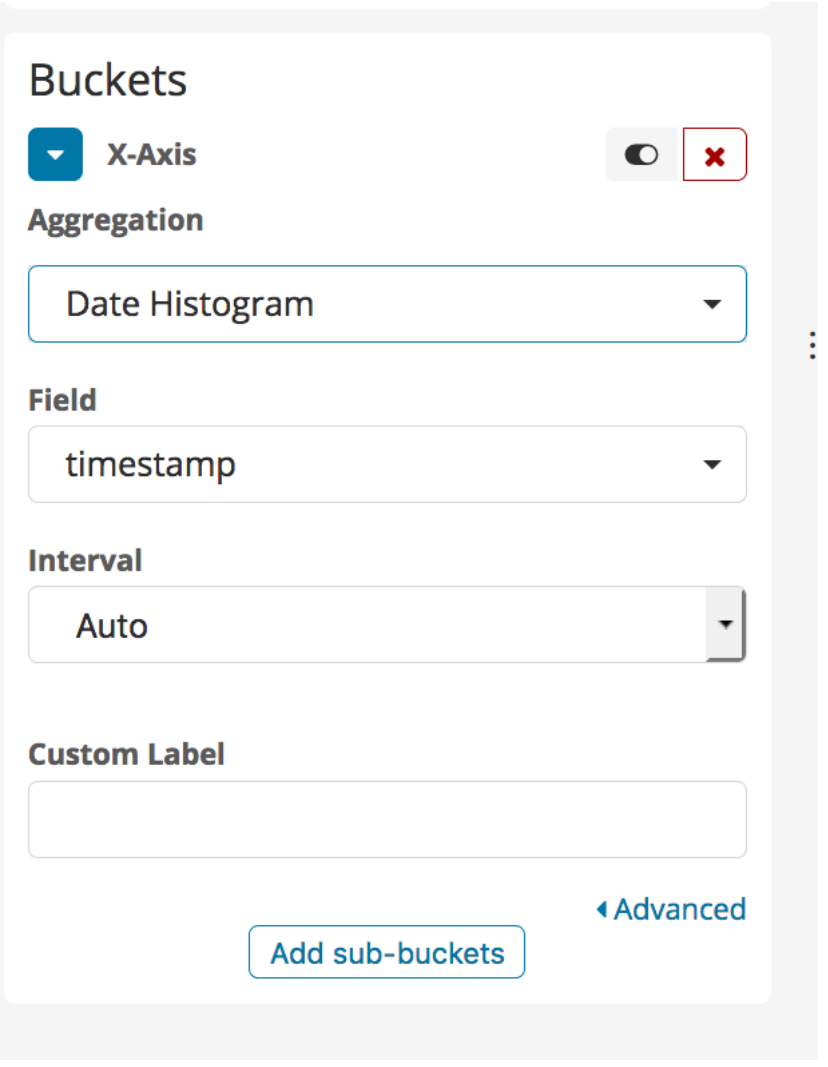
Gemini Mapping

Now we can set up a visualization where we examine the cumulative sum of our profits. Navigate to the Visualize tab and create a new "Line" visualization:



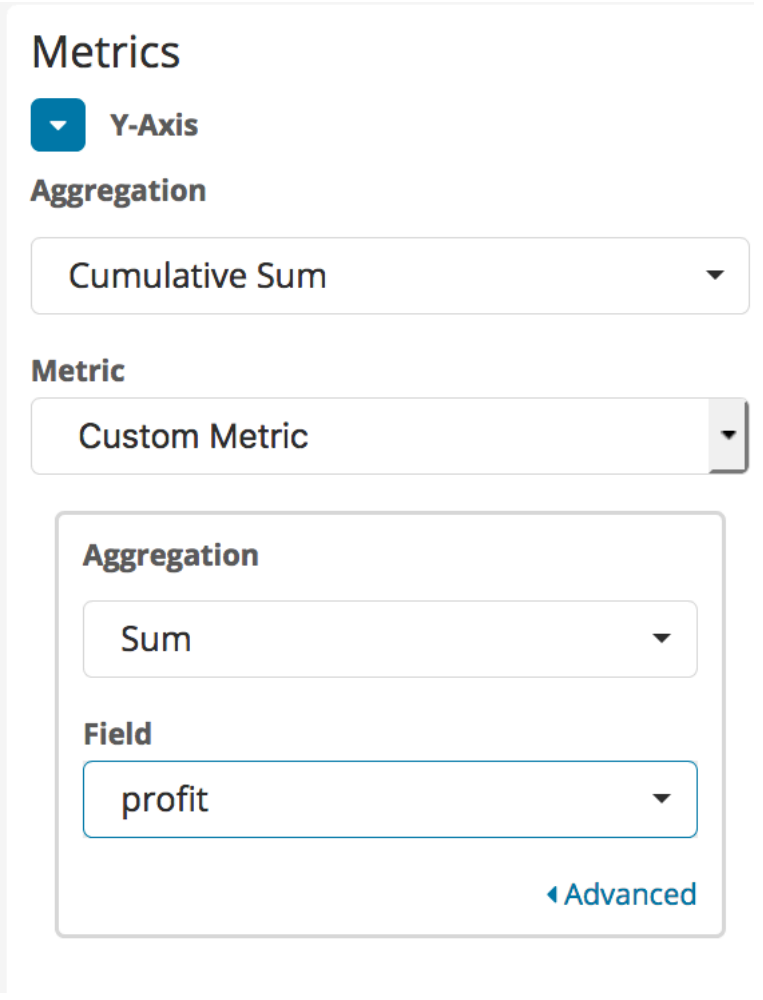
Visualization Menu

For the X-Axis select a Date Histogram using `timestamp`:



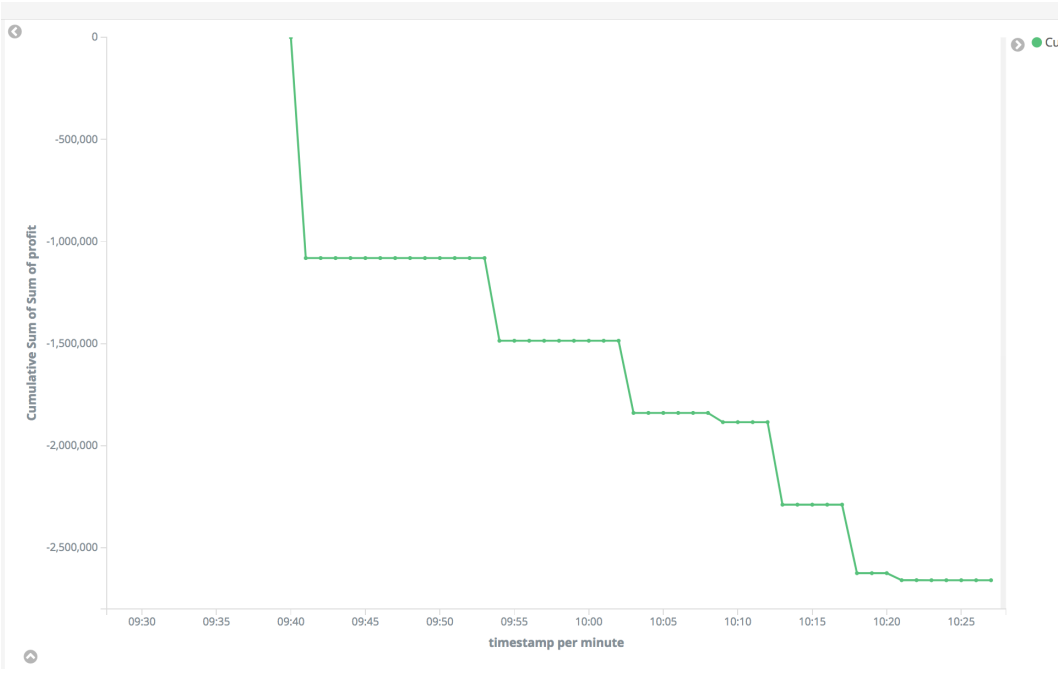
Date Histogram

For the Y-Axis select a Cumulative Sum on `profit`:



Gemini Profit

You can now see a trend line indicating how our trades are performing. Since we are trading in the Gemini Sand Box with fake data for illustrative purposes we shouldn't read too much into this performance as the numbers are all made up!



Gemini Performance

We can, if we choose to, add this chart to a dashboard and create numerous other visualizations to monitor our trading. Since we are indexing price-weighted volume of bids and asks, the ticker information, and each order we make we have a rich data source ingested into Elasticsearch from which to analyze the market and our trading bot's performance.

Conclusion

I hope you've enjoyed learning about building a real-time crypto pipeline. We've accomplished a lot together. We've:

1. learned about big data pipeline architectures
2. learned how to use Docker and Git
3. learned about cryptocurrency and blockchain
4. learned how to obtain streaming data from a cryptocurrency exchange
5. learned how to use Kafka as a message queue
6. learned how to process big data with Spark
7. learned how to perform a common machine learning task–time series forecasting
8. built a real-time cryptocurrency trading bot

I would love to get your feedback! You can find me on Twitter `@brandonmrose`. If you have questions about the code, feel free to open up an issue in the Github repo. Good luck trading!