Understanding MLOps with Azure

February 2020

Introductions

Raki Rahman

Solution Architect

slalom | Data & Analytics

https://rakirahman.com

Professional Experience

7+ years at Technology Consulting and Software Engineering firms:

- Big Data and Software Engineering on Azure + Open source stack
- Environment Operationalization: DevOps, MLOps, DataOps + ***anything else that ends with "Ops"***
- Delivering and operationalizing Azure environments for large Canadian Enterprise Clientele

Education

B.A.Sc. Aerospace Engineering (Engineering Science) – University of Toronto

databricks champion

Associate Spark Engineer Certified

Microsoft CERTIFIED AZURE SOLUTIONS ARCHITECT EXPERT

Microsoft CERTIFIED PROFESSIONAL Microsoft MCSA Cloud Platform Earned 2018

Google Cloud Certified PROFESSIONAL CLOUD ARCHITECT

Google Cloud Certified ASSOCIATE CLOUD ENGINEER

Agenda

00:00* **Deployment Kickoff:** Azure Environment Spinup for demo

00:02 **Big Data Demystified**: Hadoop, Spark and the Cloud

00:30 ## Project Showcase

Azure Synapse Analytics + Azure Databricks + **Bloomberg**

Bloomberg overview and Project Goal

Azure Solution Architecture

Bitcoin POC use case with Natural Language Processing

00:50 ## Useful resources

NLP applied - Harry Potter Q&A - BERT on Azure Kubernetes

Big Data Deep Dive - Spark Certification Study Guide

MLOps Deep Dive - MLOps on Azure with Malware Detection

01:00 5 Minute Q/A

01:05 **Mlflow Deep Dive** – Predicting Airbnb listing prices

01:50 Demo Q&A

*Timings are approximate ☺

Demo Prep

Azure Environment Deployment Kickoff

Big Data Demystified

Hadoop, Spark and the Cloud

Recurring questions

Prevalent across multiple Industry Verticals

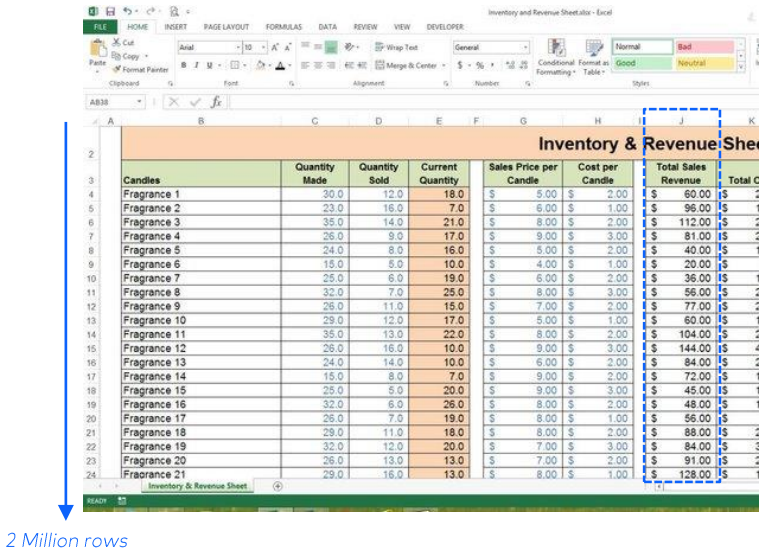
- **What's so BIG about Big Data** - that I can't solve by just throwing more hardware at my On-Premise SQL Databases?
- Does Big Data processing use some sort of **special computers**?
- If not, how does it **actually leverage** regular Computer Architecture (that you and I are using right now – our laptops) to **process large amounts of data**?
- What's **Distributed Computing** and what do I need to know? Why should I care and how is it different from my On-Premise SQL Databases?
- What's **Hadoop**? What's **Spark**? And why do both exist?
- I already have expensive servers I bought from Dell in my Data Center, who should I **care about the Cloud**?
- What's the **big deal about Azure Databricks**?

"Open this 1 Terabyte XSLX file in Excel"

Motivation Use Case

- You work for a popular online shopping website
- Your boss comes by with a 2TB external Samsung hard drive containing a **1 TB XSLX** file that contains *10 years* worth of transaction information
- You're asked to perform a straightforward routine task:
  1. Open the file in Excel
  2. Do a sum across the "Revenue (column J)" to generate the Revenue to date
  3. Send that single number over to your boss via e-mail

Inventory and Revenue sheet.xslx
1,073,741,824 KB

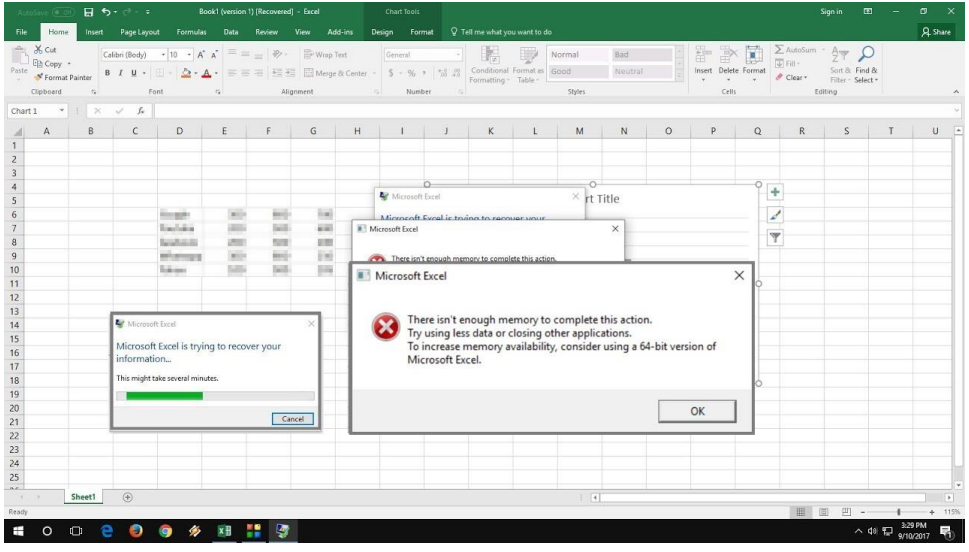


2 Million rows

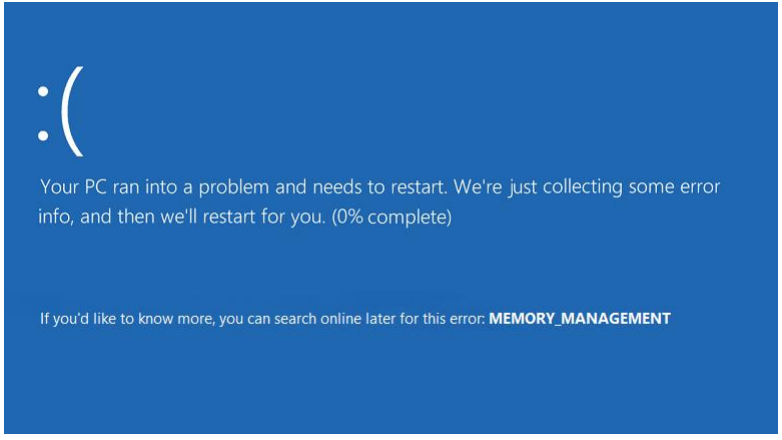
"Open this 1 Terabyte XSLX file in Excel"

Motivation Use Case

- You realize your laptop only has 500 GB of space – **so you can't even copy the 1000 GB file to local storage (C:\ Drive)**
- You try to open the file directly from the External drive, and Excel hits you with an "**Out of Memory**" error



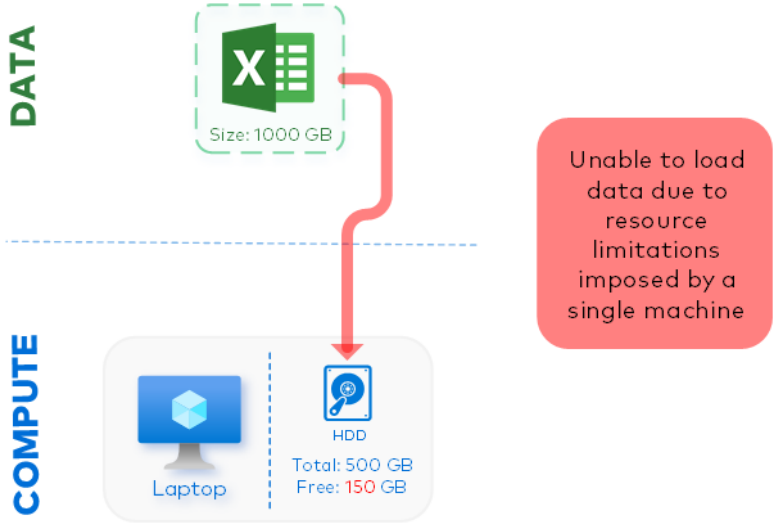
You keep on trying, and get hit with a **Blue Screen of Death**



Current Setup – 1 Machine

Motivation Use Case

Loading a large file onto a single machine



In conclusion, you realize there's no way given the **resource constraints of 1 Machine** to open this file and perform this simple revenue aggregation.

💡 Brilliant Idea – "Divide and Conquer"

Motivation Use Case

You realize you're able to pass files to your office colleagues' laptops since you're all connected to the office's LAN!

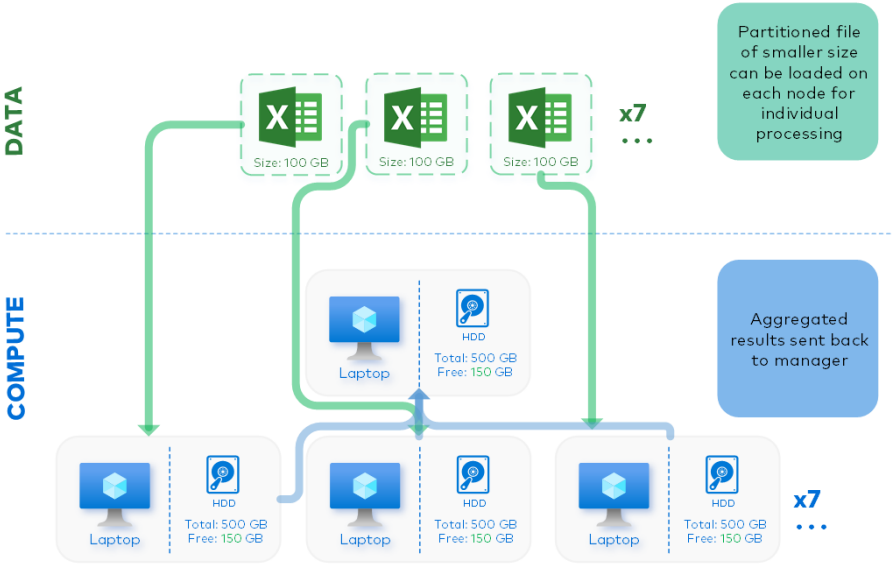
So what if you tried using **10 laptops** to open **10 chunks** of this file? That should work, since most of them have at least 100 GB of space free, so you can:

1. Chunk, or **partition** the 1000 GB file into 10 files of 100 GB each (or say, 1 years worth of transaction per chunk - i.e. **partition by date**)
2. Send each partition to each identical laptop via the office's **LAN** (local area network – **connecting the machines over wire**), and copy it to the laptop's local hard drive so Excel can open it within the machine
3. Open the partitioned file with Excel on each laptop, and do a sum across the price column
4. Take that price aggregation, and save it as a **tiny separate CSV file** (one line containing the price aggregation) on each laptop
5. You send your colleagues an email containing a **network accessible folder** hosted on your laptop, and they each send you all 10 tiny CSVs to that location
6. You open each CSV file, sum up the 10 numbers, and report the full sum back to your boss

Current Setup – 10 Machines

Motivation Use Case

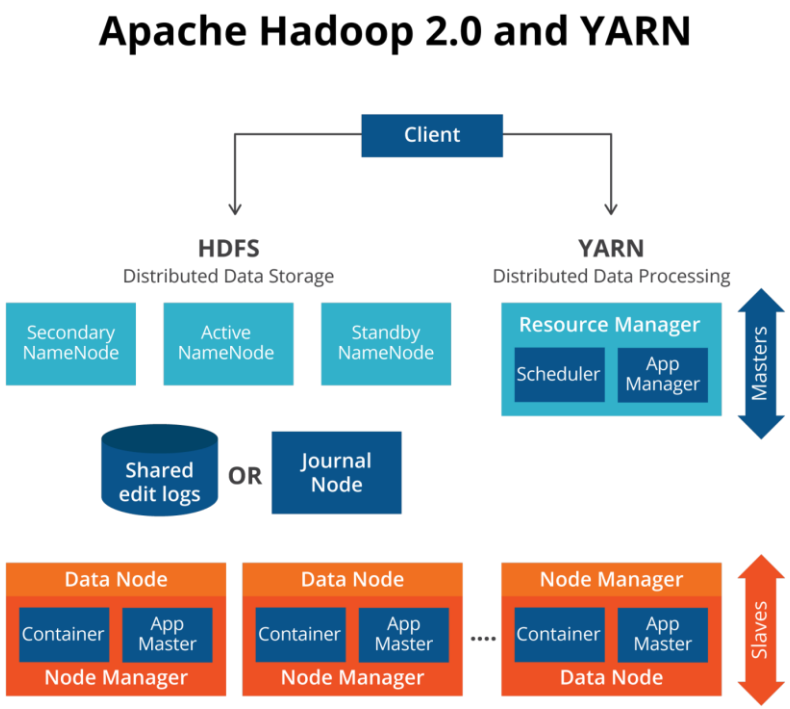
Partitioning data for divide-and-conquer



What you just implemented above is called a "**Master/Slave Architecture**", and what is essentially a primitive rendition of Hadoop MapReduce (and as we'll get to – Driver/Executor rendition of Spark).

Hadoop MapReduce

Master/Slave Architecture



HDFS (hadoop-distributed-file-system) Distributed file system engine designed to run on commodity hardware (your laptop, Samsung External Drive etc). Can store an infinite amount of information because you can throw an infinite of hard drives at the cluster (i.e. ignoring the space-time constraint)

HDFS: You + Your colleagues' laptops - Cheap, easily replaceable, interconnected, fault tolerant storage

Master/Slave

A **master server** manages the file system namespace and regulates access to files by clients. **Slave server(s)** take partitioned data from the Master, process them, and send the results back to the specified directory.

Master: You + Your laptop was the Master that assigned the workloads, gathered the partitioned results back, and aggregated the final value.

Slave: You + Your colleagues' laptops

YARN – Yet Another Resource Negotiator

Resource manager that negotiates resources and workloads across the Cluster to ensure each partition is getting processed for the job to finish timely. This is the engine responsible for intelligent workload assignment and performance tracking.

YARN: You (if a colleague is out sick, you find another, or reassign re-calculated workloads with 9). If one colleague is being specially slow, you yell at them.

Hadoop MapReduce

Birth

Apache Hadoop was released to the public in 2006 under an Open-Source license, made possible thanks to this scientific paper Google published in 2003 as the "**The Google File System**"

MapReduce is the actual engine (replacing your **pseudo Project/Resource Manager role**). It's responsible for 2 main activities:

1. **Data Partitioning and Mapping** – sending partitioned data to available slave servers
2. **Reduction** – bringing the tiny files back, and aggregating the result

Fun Fact: the founder - Doug Cutting who worked at Yahoo! at the time - named the project after his son's toy elephant "Hadoop".

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.

Categories and Subject Descriptors

D [4]: 3—*Distributed file systems*

General Terms

Design, reliability, performance, measurement

Keywords

Fault tolerance, scalability, data storage, clustered storage

*The authors can be reached at the following addresses: {*sanjay,hgobioff,shuntak*}*@google.com.*

SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Second, files are huge by traditional standards. Multi-GB files are common. Each file typically contains many application objects such as web documents. When we are regularly working with fast growing data sets of many TBs comprising billions of objects, it is unwieldy to manage billions of approximately KB-sized files even when the file system could support it. As a result, design assumptions and parameters such as I/O operation and block sizes have to be revisited.

Third, most files are mutated by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Once written, the files are only read, and often only sequentially. A variety of data share these characteristics. Some may constitute large repositories that data analysis programs scan through. Some may be data streams continuously generated by running applications. Some may be archival data. Some may be intermediate results produced on one machine and processed on another, whether simultaneously or later in time. Given this access pattern on huge files, appending becomes the focus of performance optimization and atomicity guarantees, while caching data blocks in the client loses its appeal.

Fourth, co-designing the applications and the file system API benefits the overall system by increasing our flexibility.

HDFS

Fun fact about Cloud Storage

HDFS – founded 2003 by Google – is what made Cloud Storage and Distributed computing of today possible. This is how we store data dirt cheap in the Cloud (iCloud, Dropbox, OneDrive etc.)

Companies (Microsoft, Amazon, Google) have distributed, interconnected Data Centers across the world, with millions of Hard Drives. **HDFS makes the storage capacity of these drives ~infinite.**

These companies have an army of folks that take care of these servers (air conditioned, contained environments etc.). **Their job is to make sure any data that's put in there is in good health** (99.999...% SLA).

When you and I put our photos in the OneDrive Cloud, we're basically **renting out a tiny portion of Microsoft's massive HDFS cluster – which is why Cloud Storage is dirt cheap**.

MapReduce – Pain Points

Inherent design-based limitations

MapReduce Summary: a reliable, good-for-its-time, Big Data processing engine, that did the best it could with the technology it originated from.

The Fundamental Problem:

- MapReduce persists the dataset at **each step along the way on the hard disk**
  - **Hard disks were the best we had in 2003 to resiliently store data** *(we're talking Computer Science of Data Structures)* – MapReduce reflected this limitation
  - **Hard disks are slow.** Specially if you're using the cheap kind (non Solid-State-Drives).

So basically, combine:

1. **Hard disks**, specially mechanical ones (i.e. non SSDs), which are fundamentally slow (thanks to the laws of Physics), but reliable/resilient at storing data
2. **Unoptimized I/O Operations**, to move data across the network (you can think of this as a sophisticated spider-web of ethernet cables) back and forth between Hard disks on different machines, which is also slow

Note that none of this really takes proper advantage of the fact that there's a much faster way to access data in a Computer or Cluster of Computers -

the RAM.

- **RAM** – **R**andom **A**ccess **M**emory – was always used for temporary caching of information, so MapReduce couldn't use RAM for anything related to transforming Data
- The person who invented RAM back in 1968 - Robert Dennard - never meant for it to be a resilient data store (*write cached data quickly, read quickly, forget-about-it*)

RAM is faster

Putting things into Perspective

~100x faster
than HDD based architecture

	Hard drive		Solid State Drive		RAM	
MB/s	100	100	500	250	6000	8500
	Read	Write	Read	Write	Read	Write
	Booting up your laptop from 2001 – 30 seconds		Booting up your laptop from today – 10 seconds		Playing a video game on your PS4 – ~instant response to action	

Apache Spark

Birth

Matei Zaharia was at UC Berkeley when the Netflix Challenge (solving Sparse Matrix for Recommendations Engine) was released in 2009. He tried to solve the problem with Hadoop's MapReduce before quickly realizing the algorithm was being **bottlenecked by the Platform's Read-Write throughput**, rather than execution efficiency.

Over that weekend, he invented a premature rendition of the **RDD** – **R**esilient **D**istributed **D**ataset – which essentially allowed for his Server Cluster to store data resiliently at the RAM layer - UC Berkley RDD Paper on the right.

Apache Spark was released to the public in 2014 under an Open-Source license, fueled primarily by the power of the RDD, and the capabilities it brought.

- Spark runs workloads **100x faster** than Hadoop's MapReduce.
- Spark has over 1450 contributors, the **most adopted Big Data open-source project**.
- From 2014, Spark has been widely adopted in the industry as the **fastest way to process data** on modern computer paradigm.

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that *reuse* intermediate results across multiple computations. Data reuse is common in many *iterative* machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is *interactive* data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (*e.g.*, between two MapReduce jobs) is to write it to an external stable storage system, *e.g.*, a distributed file system. This incurs substantial overheads due to data replication, disk I/O, and serialization, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.*, looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.*, to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance *efficiently*. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (*e.g.*, cells in a table). With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to these systems, RDDs provide an interface based on *coarse-grained* transformations (*e.g.*, map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.[1] If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

[1]Checkpointing the data in some RDDs may be useful when a lineage chain grows large, however, and we discuss how to do it in §5.4.

Why Spark is a big deal

It's not going anywhere soon

From 2014, Spark has been widely adopted in the industry as the fastest way to process data on modern computer paradigm.

Simple concept:

1. **RAM is the fastest** way to perform data operations on modern computer architecture the world runs on today (Motherboard, CPU, Hard Drive, RAM) – **this is an indisputable truth**
2. **Spark's Engine runs on the RDD** – which is the brilliant way Matei invented to store data resiliently on the RAM layer – **you can't really reinvent this**

= Spark is the fastest possible way to analyze Data

...Until Quantum Computers become publicly accessible:

- Manufacturing of Quantum computers becomes as streamlined as manufacturing Binary computers (**good luck with 50+ years in the making**)
- Running an ETL job in Quantum stops costing a few million dollars per run

Quantum Computing can dethrone Spark's Execution capabilities, once we figure out all **the logistical issues in operationalizing it for public use** (don't hold your breath).

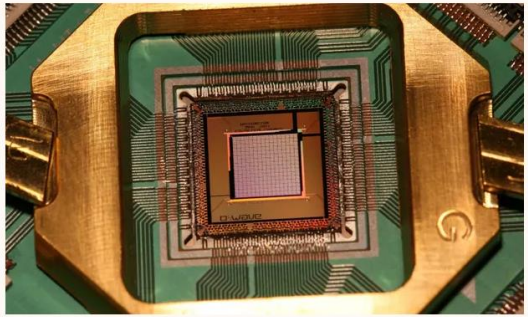
Computing

Quantum supremacy is coming. It won't change the world

Sabine Hossenfelder
Fri 2 Aug 2019 07.00 BST

193 323

If quantum computers are to help solve humanity's problems, they will have to improve drastically



▲ Quantum computers can solve certain problems much faster than conventional ones. However, such systems are exceptionally fragile, sensitive and expensive. Photograph: D-Wave

The unveiling of the marvel had the media gushing. It was Valentine's Day 1946, and the New York Times broke the story. The front page spoke of "an amazing machine" and "one of the war's top secrets". By crunching numbers at unprecedented speed, the Electronic Numerical Integrator and Computer, with its 18,000 vacuum tubes, was poised to "revolutionise modern engineering". Eniac would usher in a new epoch of industrial design, some said.

More than 70 years on, another overblown announcement is near. Several companies, notably Google, IBM and the California-based Rigetti, are racing to build a machine that achieves what is grandly termed "quantum supremacy". The feat will mark the moment when a quantum computer, for the first time, outperforms the best conventional computers. Google, the frontrunner, could claim the record this year.

Quantum computers can solve certain problems much faster than conventional ones. Drug development, materials design, weather forecasts, stock trades - all could potentially be helped by quantum computers that perform scores of calculations at once. Physicists have known this since the 1980s, but have only recently made working prototypes.

Operationalizing Spark

Spark + Scalability = Awesome

Now that we've established that Spark is amazing:

How does your organization actually implement it for real-life day-to-day benefit?

Where can you get:

1. **Enough servers in a Data Center** to run your Big Data workloads right In-Memory with Spark, with the ability to scale-up when you process data in bursts, and completely scale-down so you spend $0 during downtime
2. **A reliable resource negotiator** that takes away the overhead of managing YARN out of your hands and does it all behind the scenes (with room for tweaking, should you choose)
3. **Managed Clusters**, that removes the burden of updating Spark, Python, Scala and other Drivers as new versions are released, as well as dependency and library management
4. **An intuitive, easy, visual interface** to curate your workloads (like Jupyter notebooks) using a multitude of programming languages (like Python, SQL, R, Scala), depending on what your team is comfortable with (I don't know any R and Scala for example, but can get around in Python and SQL – this doesn't mean I should be forced to learn R to do my job)
5. **Secured integration** with your other Enterprise systems/Cloud Services (Like Azure Storage Account, SQL Data Warehouse, Azure Synapse, Tableau and Power BI)
6. **Keep up to date with the industry as it evolves** without you having to invest in architecture uplifts/data processing engine migrations every year

Azure Databricks

Spark + Scalability = Awesome

The Engine

Job execution engine able to efficiently distribute and execute tasks on RAM layer

The Platform

Scalable RAM + Processing power without management overhead

The Solution

Managed Apache Spark Platform able to seamlessly integrate within Azure Big Data/ML ecosystem

Matei Zaharia

Professor of Computer Science
Stanford University

The Engine

Apache Spark

Founded 2009

Unbeatable performance

+

Open Source transparency

=

The best Big Data processing engine

Founder of:
Apache Spark
Databricks
Delta Lake
MLflow

The Solution

Azure Databricks

Founded 2018

Cloud Scalability

+

Secret Sauce (Catalyst optimizer + Resource Negotiator)

=

The best platform to implement Spark

Project Showcase

Azure Real-time Streaming from Bloomberg

Bloomberg overview

Bloomberg overview

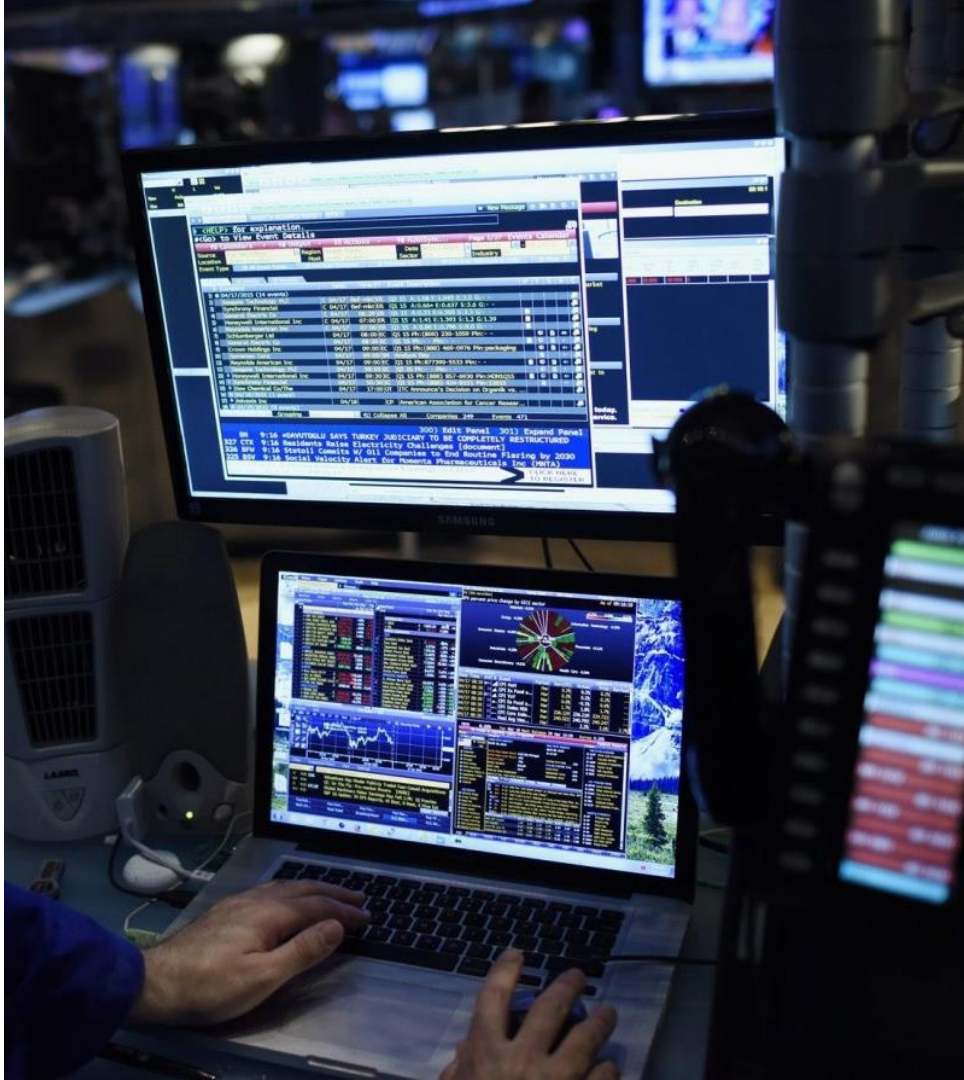
Massive global database of real-time and historical financial data – including trading, currency, supply chain, news, economic forecasts and more.

Bloomberg Terminal

The core revenue generator – implementing a client-side, physical terminal to connect to the Bloomberg Data Center.

Bloomberg Market Data Feed (B-PIPE)

B-PIPE enables real-time access to streaming data – allowing client applications access to the Bloomberg Data Center via the BLPAPI (C++, .NET, Java, Python).



Project Goal

Financial Services

Platform Requirements

Architect a state-of-the-art, automated, operationalized Cloud Data Analytics Platform

Project Goal

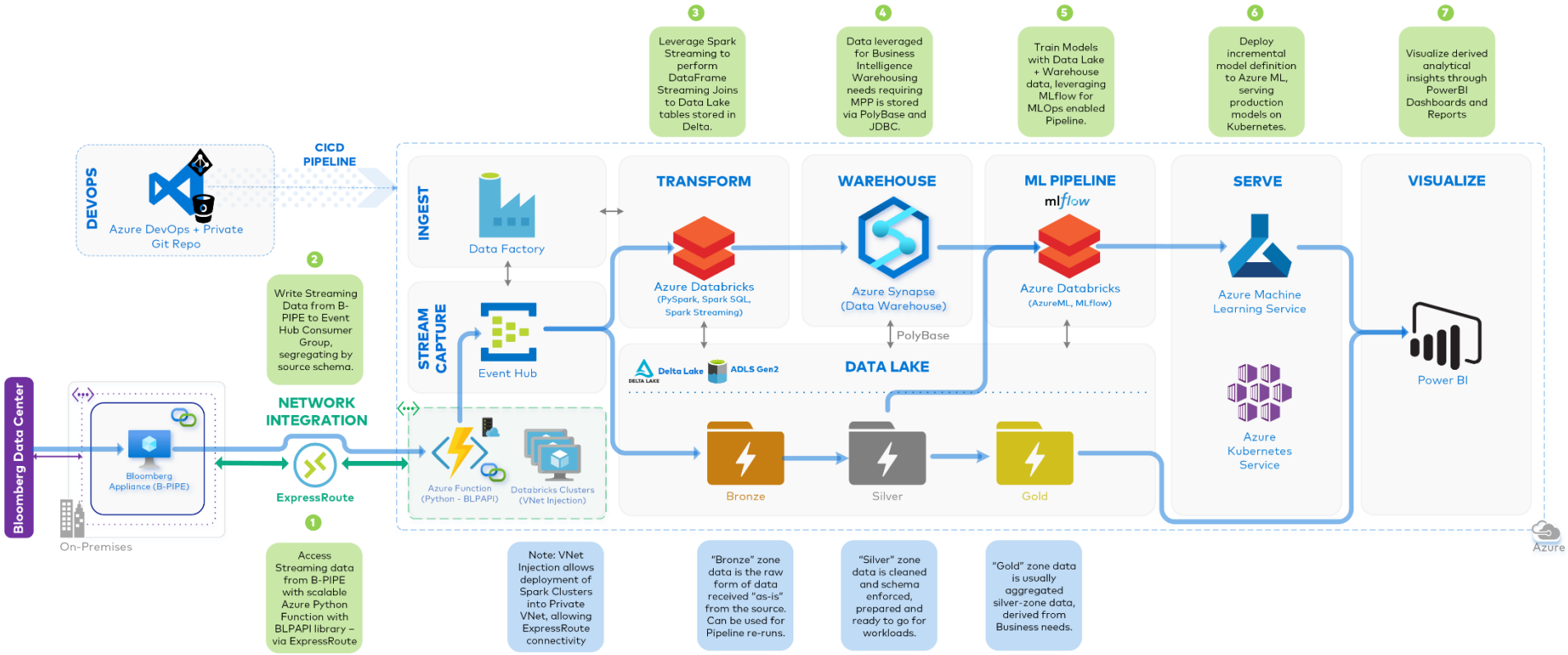
Financial Services

Platform Requirements

Azure Synapse Analytics + Azure Databricks + **Bloomberg**

Real-Time Ingestion	Secured API Integration	Rapid ETL	DevOps	Machine Learning Pipeline
				
Stream + Batch: • Ticker data • Share price • Company news • Supply chain updates • Historical data	ExpressRoute Integration to Bloomberg API	High-speed Data processing (Stream + Batch loads)	Continuous Integration Continuous Delivery (CICD) enabled Data Pipeline	MLOps enabled ML Pipeline capable of real time scoring and incremental model updates

Azure Solution Architecture



ML POC Use Case

Machine readable news with Bloomberg

Proof of Concept Objective

Use Machine readable news from Bloomberg to explore ML driven market trading strategy

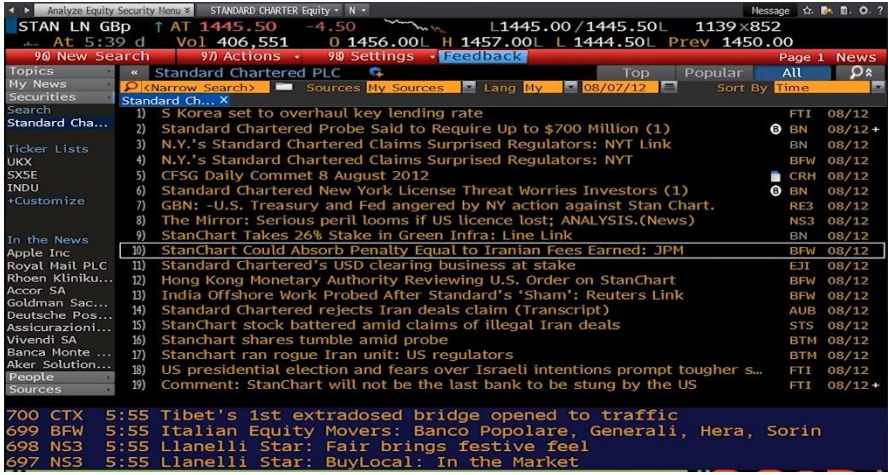
Machine readable news

The idea is – Bloomberg provides access to real-time event-driven news feeds in the form of a structured dataset (i.e. consistently formatted, readership stats, sentiment, ticker tagged e.g. STAN LN).

The goal is – train NLP* driven Machine Learning models to filter news and generate Buy/Sell signals.

*NLP: training ML models to process and analyze large amounts of language data.

On August 7th 2012, Standard Chartered stocks took a sharp decline.



Viewing news from the timestamp shows root cause

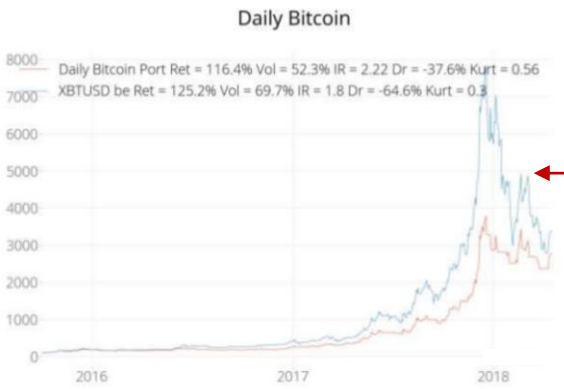
Summarized high level approach

#	To do	Approach	Technology
1	**Batch/Stream ingestion pipeline of ticker data (historical/real-time news + stock)**	Pull historical/real-time data from B-PIPE into Data Lake and store in "Bronze" Zone. Create Data Ingestion pipeline for Stream and Batch processing.	Azure Databricks DELTA LAKE Spark Streaming
2	**Create Language Data Cleansing Pipeline for news data, and curate into time series format**	Leverage Text Pre-Processing tools to perform cleaning operations (stemming, lemmatization, chunking/chinking) to filter noise and structure news data	Koalas BeautifulSoup python spaCy
3	**Perform tokenization, topic modelling and sentiment analysis** **Create indicators and apply trading rules**	Convert cleaned, structured text into mathematical (vectorized) representation, leveraging NLP specific mathematical techniques (TFIDF, TSNE, LSTM etc.). Convert into buy/sell signals and/or add other trading factors (e.g. carry)	Azure Machine Learning TensorFlow Keras learn Spark ML
4	**Score models on test data and deploy best performing model to serving layer**	Perform experiments and track parameters/results using Mlflow. Deploy containerized models to Azure Kubernetes using Azure ML version control.	mlflow Azure Machine Learning kubernetes
5	**Iterative Model Definition Updates**	Connect B-PIPE Streaming pipeline to ML Pipeline for iterative scoring and incremental model updates.	Azure Databricks mlflow DELTA LAKE Azure Machine Learning

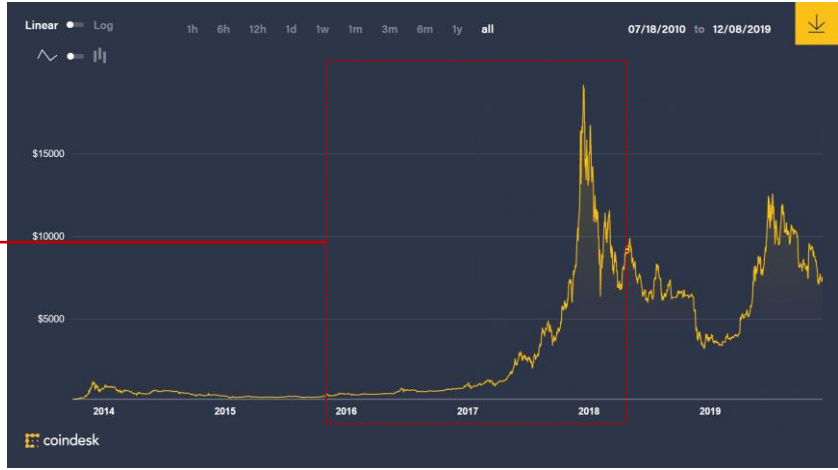
Trading Strategy Output from News Data

```
if __name__ == '__main__':
    model = TradingBitcoinDaily()

    model.construct_strategy()
    model.plot_strategy_group_benchmark_pnl()
```



Bitcoin (long) actual price **VS.**
Trading Strategy from NLP



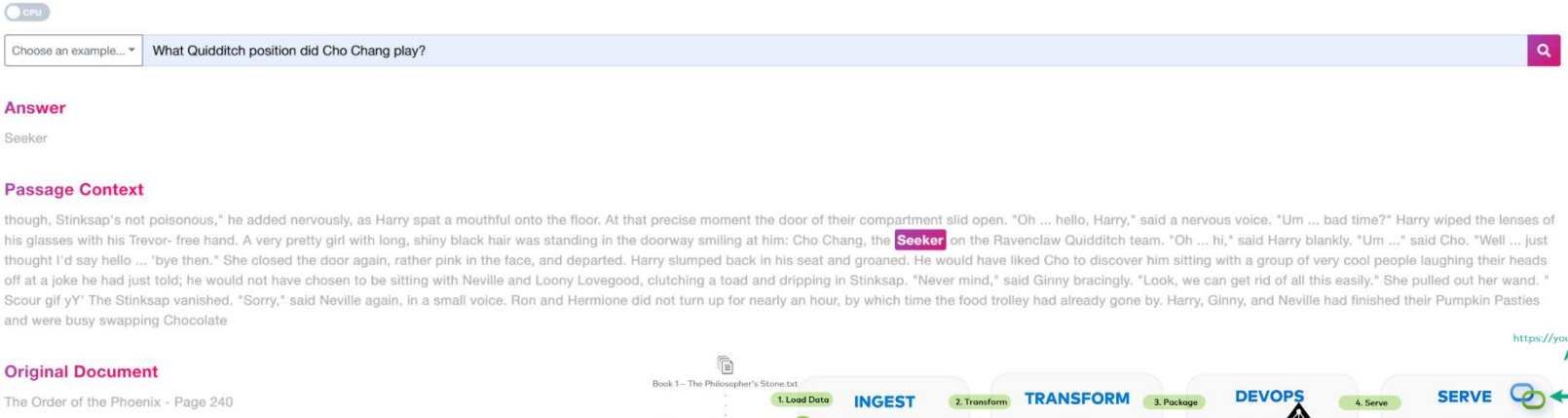
Bitcoin price: 2010 to 2019

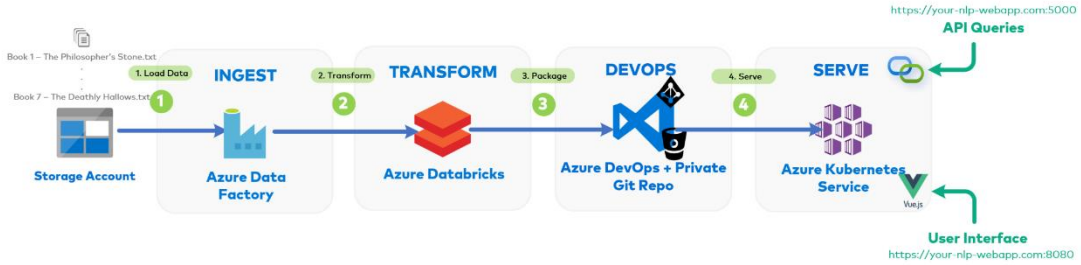
Useful Resources

Self-Study/Deep-Dive material

Intelligent Harry Potter Search Engine

Building a BERT-powered Natural Language Processing model, trained on Harry Potter books to answer context-specific questions, served in a containerized web-app.





Link

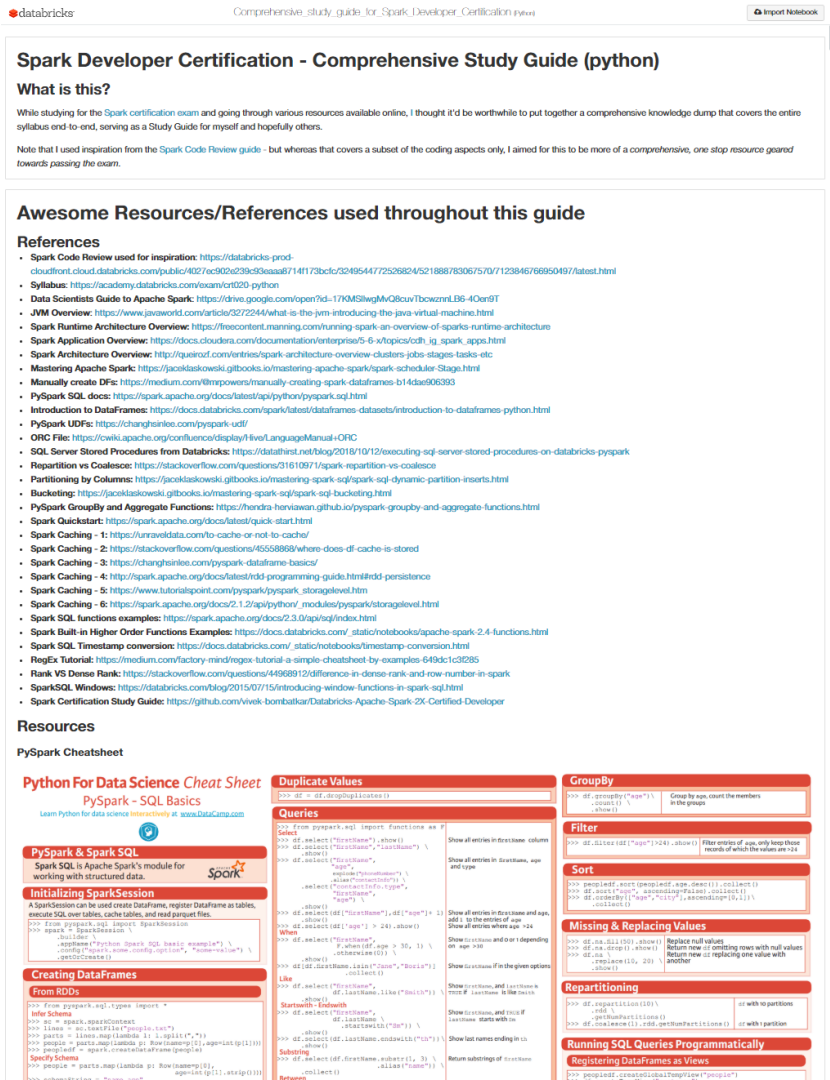
http://bit.ly/azure-harry-potter

Spark Certification Study Guide

While studying for the CRT020: Apache Spark 2.4 with Python 3 Developer Certification, I complied all my study notes into a thorough study-guide required to pass the exam:

Link

http://bit.ly/spark-study-guide


databricks

Comprehensive_study_guide_for_Spark_Developer_Certification (Python)

Import Notebook

Spark Developer Certification - Comprehensive Study Guide (python)

What is this?

While studying for the Spark certification exam and going through various resources available online, I thought it'd be worthwhile to put together a comprehensive knowledge dump that covers the entire syllabus end-to-end, serving as a Study Guide for myself and hopefully others.

Note that I used inspiration from the Spark Code Review guide - but whereas that covers a subset of the coding aspects only, I aimed for this to be more of a *comprehensive, one stop resource geared towards passing the exam.*

Awesome Resources/References used throughout this guide

References

- **Spark Code Review used for inspiration**: https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/3249544772526824/5218887830675 70/7123846766950497/latest.html
- **Syllabus**: https://academy.databricks.com/exam/crt020-python
- **Data Scientists Guide to Apache Spark**: https://drive.google.com/open?id=17KMSIlwgMvQ8cuvTbcwznnLB6-4Oen9T
- **JVM Overview**: https://www.javaworld.com/article/3272244/what-is-the-jvm-introducing-the-java-virtual-machine.html
- **Spark Runtime Architecture Overview**: https://freecontent.manning.com/running-spark-an-overview-of-sparks-runtime-architecture
- **Spark Application Overview**: https://docs.cloudera.com/documentation/enterprise/5-6-x/topics/cdh_ig_spark_apps.html
- **Spark Architecture Overview**: http://queirozf.com/entries/spark-architecture-overview-clusters-jobs-stages-tasks-etc
- **Mastering Apache Spark**: https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-scheduler-Stage.html
- **Manually create DFs**: https://medium.com/@mrpowers/manually-creating-spark-dataframes-b14dae906393
- **PySpark SQL docs:** https://spark.apache.org/docs/latest/api/python/pyspark.sql.html
- **Introduction to DataFrames:** https://docs.databricks.com/spark/latest/dataframes-datasets/introduction-to-dataframes-python.html
- **PySpark UDFs:** https://changhsinlee.com/pyspark-udf/
- **ORC File:** https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC
- **SQL Server Stored Procedures from Databricks:** https://datathirst.net/blog/2018/10/12/executing-sql-server-stored-procedures-on-databricks-pyspark
- **Repartition vs Coalesce:** https://stackoverflow.com/questions/31610971/spark-repartition-vs-coalesce
- **Partitioning by Columns:** https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-dynamic-partition-inserts.html
- **Bucketing:** https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-bucketing.html
- **PySpark GroupBy and Aggregate Functions:** https://hendra-herviawan.github.io/pyspark-groupby-and-aggregate-functions.html
- **Spark Quickstart:** https://spark.apache.org/docs/latest/quick-start.html
- **Spark Caching - 1:** https://unraveldata.com/to-cache-or-not-to-cache/
- **Spark Caching - 2:** https://stackoverflow.com/questions/45558868/where-does-df-cache-is-stored
- **Spark Caching - 3:** https://changhsinlee.com/pyspark-dataframe-basics/
- **Spark Caching - 4:** http://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence
- **Spark Caching - 5:** https://www.tutorialspoint.com/pyspark/pyspark_storagelevel.htm
- **Spark Caching - 6:** https://spark.apache.org/docs/2.1.2/api/python/_modules/pyspark/storagelevel.html
- **Spark SQL functions examples:** https://spark.apache.org/docs/2.3.0/api/sql/index.html
- **Spark Built-in Higher Order Functions Examples:** https://docs.databricks.com/_static/notebooks/apache-spark-2.4-functions.html
- **Spark SQL Timestamp conversion:** https://docs.databricks.com/_static/notebooks/timestamp-conversion.html
- **RegEx Tutorial:** https://medium.com/factory-mind/regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285
- **Rank VS Dense Rank:** https://stackoverflow.com/questions/44968912/difference-in-dense-rank-and-row-number-in-spark
- **SparkSQL Windows:** https://databricks.com/blog/2015/07/15/introducing-window-functions-in-spark-sql.html
- **Spark Certification Study Guide:** https://github.com/vivek-bombatkar/Databricks-Apache-Spark-2X-Certified-Developer

Resources

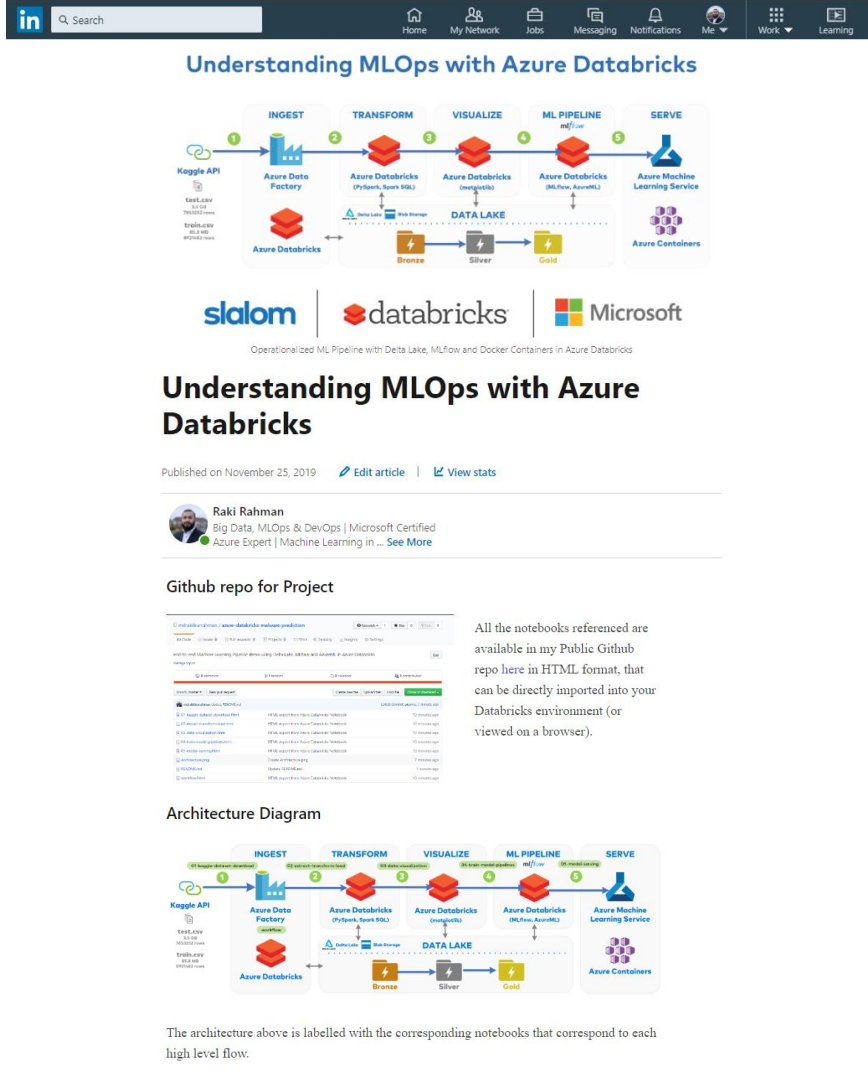
PySpark Cheatsheet

Understanding MLOps with Azure Databricks

"Because the focus has always been on the model's prediction capabilities and not how the model actually gets there, we've poured thousands of hours and developed hundreds of models, and now managing this beast of an environment is a never-ending battle every single day when you actually start using this pipeline to predict mission-critical features."

Link

http://bit.ly/mlops-databricks



MLFlow end-to-end demo

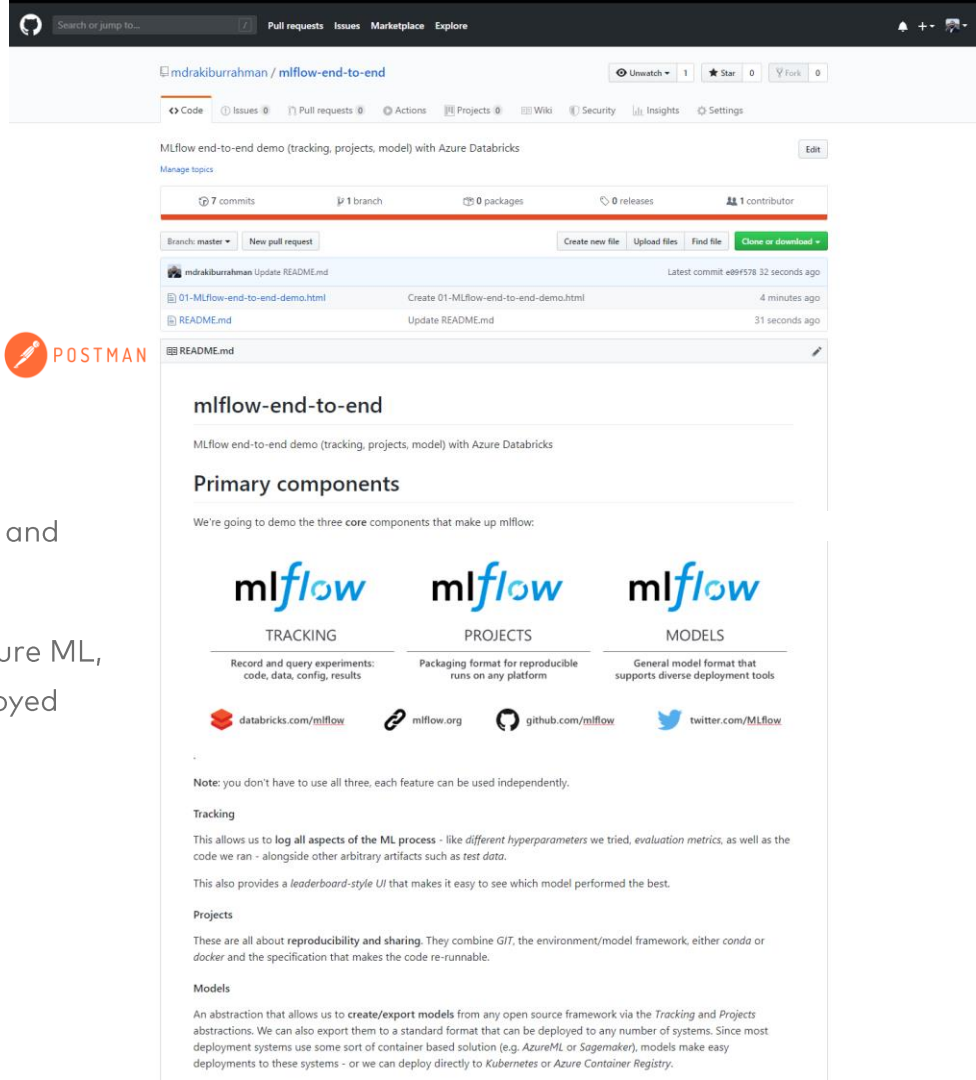
Our Demo for today

An end-to-end demo showcasing Mlflow **Tracking**, **Projects** and **Models**, using an Airbnb dataset.

We will finish by deploying a Random Forest model into Azure ML, and querying served model with Postman against the deployed Docker Container – mimicking a Production App.

Link

http://bit.ly/mlops-mmai



mdrakiburrahman / mlflow-end-to-end

MLflow end-to-end demo (tracking, projects, model) with Azure Databricks

7 commits · 1 branch · 0 packages · 0 releases · 1 contributor

mdrakiburrahman Update README.md — Latest commit e89f578 32 seconds ago

01-MLflow-end-to-end-demo.html — Create 01-MLflow-end-to-end-demo.html — 4 minutes ago

README.md — Update README.md — 31 seconds ago

README.md

mlflow-end-to-end

MLflow end-to-end demo (tracking, projects, model) with Azure Databricks

Primary components

We're going to demo the three **core** components that make up mlflow:

mlflow TRACKING — Record and query experiments: code, data, config, results

mlflow PROJECTS — Packaging format for reproducible runs on any platform

mlflow MODELS — General model format that supports diverse deployment tools

databricks.com/mlflow · mlflow.org · github.com/mlflow · twitter.com/MLflow

Note: you don't have to use all three, each feature can be used independently.

Tracking

This allows us to **log all aspects of the ML process** - like *different hyperparameters* we tried, *evaluation metrics*, as well as the code we ran - alongside other arbitrary artifacts such as *test data*.

This also provides a *leaderboard-style UI* that makes it easy to see which model performed the best.

Projects

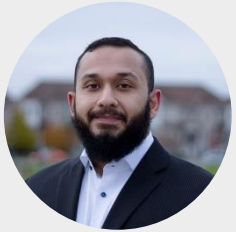
These are all about **reproducibility and sharing**. They combine *GIT*, the environment/model framework, either *conda* or *docker* and the specification that makes the code re-runnable.

Models

An abstraction that allows us to **create/export models** from any open source framework via the *Tracking* and *Projects* abstractions. We can also export them to a standard format that can be deployed to any number of systems. Since most deployment systems use some sort of container based solution (e.g. *AzureML* or *Sagemaker*), models make easy deployments to these systems - or we can deploy directly to *Kubernetes* or *Azure Container Registry*.

Before the Demo

Questions?



Raki Rahman

Solution Architect

slalom | Data & Analytics

https://rakirahman.com