The Big eBook of Data Science and Engineering Use Cases

8 leading companies that have accelerated innovation with Databricks

databricks

Table of Contents

Introduction

AI is a hot topic right now. From autonomous cars to predicting customer churn, companies across industries are continuing to harness the value of big data and trying to take advantage of machine learning and AI to deliver on innovative use cases previous assumed to require human perception and cognition.

But putting AI into practice is another story. With tech giants including Baidu and Google spending between $20B to $30B on AI in 2016, with 90% of this spent on R&D and deployment[1], it's no wonder there's a growing gap between the early adopters and the greater majority of companies that can't keep up. They simply lack the technical expertise and resources to implement and reap the benefits of machine learning and AI. There are three primary challenges that are making AI inaccessible to the majority of the companies in the world:

PEOPLE: Most companies don't have armies of data scientists with PhDs solely focused on finding ways to deploy AI to extract insights from their massive troves of data. Furthermore, AI is not achieved in isolation. It requires all stakeholders including data engineering, data science, and the business to work together towards that common goal.

PROCESS: To be successful with AI, you've got to know which technologies—both open source projects and proprietary solutions—are best for your needs and figure out how to configure and stitch them together to make them work.

INFRASTRUCTURE: Making AI a reality requires a heavy investment in both technology and human resources to support every step of the workflow from processing streams of data at scale to providing the facilities for data scientists to train and deploy machine learning models.

This eBook provides a deeper dive into eight companies that have figured out how to unify analytics across data science, engineering and the business — tapping into the benefits of machine learning and AI with Databricks to deliver transformative business outcomes. This collection of technical blogs provide experiences, insights, and practical tips that will help both data engineers and data scientists succeed.

[1]McKinsey Global Institute Study: Artificial Intelligence, The Next Digital Frontier)

Increase Business Agility with Smarter Data Engineering

Case studies for data engineering teams

How Apache Spark on Databricks is Taming the Wild West of Wi-Fi

by Tomasz Magdanski
Director of Big Data and Analytics, iPass

iPass is the world's largest Wi-Fi provider, yet we don't own a single hotspot. You can think of us as the Uber of Wi-Fi. And though it may sound simple, it is actually quite complicated. That is because integrating more than 160 commercial Wi-Fi providers from around the world into a single network is difficult. Not only do hotspots require constant monitoring to ensure a consistent user experience, but as a technology, Wi-Fi is also more fragile than most people think. You can expect a consistent user experience from your home Wi-Fi network, but commercial-grade hotspots do not share the same characteristics.

Signal strength at home is usually good. Bandwidth is in the tens of megabits, with only a handful of devices usually attaching to the home network. Single access point architecture prevails, and there is no provisioning requirement beyond the access key.

Public and commercial hotspots are usually farther away from the user, their signal strength is worse, bandwidth more limited and shared across many devices. Often, the hotspots themselves break and need to be replaced, or at least regularly maintained. Add community hotspots to the mix, and this Wi-Fi ecosystem becomes fairly unmanageable.

Although Wi-Fi can be unpredictable, here at iPass, we had a big idea. What if we could monitor every single, solitary hotspot in the world? Measuring speed, availability, performance and location so that we could recommend which hotspot a device should connect to and which one it should not. I'm saying a device, not a user. Think about cellular networks for a second; you as the user do not decide which cell tower to attach to, the device does. In terms of the user experience, it just happens.

Luckily for us, our years of connectivity experience and our own cutting-edge technology came together. iPass industry knowledge and experience met "big data" and our Wi-Fi service platform, iPass SmartConnect™, was born.

With iPass SmartConnect, we aimed to keep our users always best connected. Not only to Wi-Fi but to the best available network, Wi-Fi or cellular. That was the company's founding mission, after all. Our backend measures and analyses connectivity and quality data to ensure a device is connected to the best network every time. Our iPass SmartConnect SDK and iPass SmartConnect backend can make the same recommendation for IoT devices, ensuring they are also always best connected.

First off, we needed to group and categorize individual access points. For example, hotels can have many access points that share common infrastructure. So, instead of learning about every hotspot separately, we wanted to group them into logical “venues” and monitor them as an entity.

Easier said than done, though, especially if you don’t actually know where the venue is or how many hotspots there are at any given location.

Moreover, from experience, we knew that many connection failures are the result of a user simply walking away from an access point. Here in the Bay Area, for instance, you’ll find access points at virtually every traffic light. Once you connect to one of them, your device will remember it and try to connect you every time you are nearby.

So here you are, driving along, listening to Pandora or talking on Skype, while your phone decides to jump onto a hotspot. But in the meantime, you pull away. Wi-Fi connection will fail and cellular interface will need to renegotiate data connection.

Nowadays, your device knows when you’re on the move, as your device has an accelerometer. But your device can’t tell whether the hotspot is moving as well. If iPass SmartConnect can detect motion patterns and tag hotspots as moving, we can recommend hotspots that are moving with you, like hotspots on a plane or train.

That was our plan, technically challenging and innovative. And in early 2016, iPass asked me to relocate to headquarters in Redwood Shores, California to lead this exciting project.

We had a ton of technical debt and the company was not architected with big data in mind. We were a hardcore RDBMS shop.

The team started small, just an engineer and myself. So in the spirit of a true startup, we rolled up our sleeves and got to work.

We had some previous experience playing with Kafka, Storm, Hadoop and Hbase, but being a team of two, we did not have the time to build an in-house big data platform.

It was obvious to us that in order to build and grow our platform, we had to move out of our self-managed data center and into the cloud. So in order to allow for rapid development and on-demand scaling, I started vetting cloud big data vendors.

At that time, iPass was already using AWS for newly released products and services.

Being small and smart, we did not see a lot of value in managing two “clouds.”

Platform-wise, one solution stood out above all the others: Apache Spark. It can batch, it can stream, it is mostly actively developed, and their offices are close by. That comes in handy. And it leads the way in efficiency among all other big data technologies.

So Apache Spark it was.

A new wave of business meant we needed to turn iPass SmartConnect from concept to reality and quick. We needed to start developing, right away.

We could not afford to build and maintain Spark clusters, or manage EMR. Instead, we needed to focus on writing business logic. So we scheduled our first call with Databricks.

The Databricks platform was very promising – almost too good to be true. Full separation of storage and computation, easy to use, high-level API, each job creates its own cluster, no more library dependency conflicts, various cluster sizes and instance types, all Spark versions, web-based development and a fantastic team of very smart people to help us out.

We signed on the dotted line. And shortly thereafter, we got our Databricks instance up and running.

It took us a few weeks to write the first iteration of the tracking and ranking job. iPass SmartConnect was launched.

Since then, we've developed all of the features mentioned above. Some of them are in beta, but the speed of development has been like nothing we had experienced or managed before.

Apart from iPass SmartConnect, we also use Databricks for ETLs, streaming, analytics, machine learning, data lake processing and reporting. It has become a central point of our entire data flow.

The team has grown, and we are now delivering new data products in very agile, short sprint cycles. We have accomplished all of this with a small team. And we no longer worry about hardware, maintenance, scalability or (lack of) resource challenges, like maintaining and expanding old infrastructure in our own datacenter, not to mention hardware and licensing costs.

Our success story would not be complete if I did not mention the tremendous help we received from the engineering team at Databricks. We have been using Spark in a fully mature production environment, and the team's response was truly incredible. We are very happy to have the brainpower and dedication of Databricks' engineers on our side.

So what have we learned? Simply this. If you have a vision and need to get to market quickly, do not try to re-invent the wheel by building everything yourself. Instead, focus on business logic, and the delivery of sellable and scalable products. And let the guys at Spark do what they do best.

Delivering Exceptional Care Through Data-Driven Medicine

Connecting Healthcare Providers to Insights That Elevate Clinical Quality Using Apache Spark

by Jorge A. Caballero
CEO, Distal

Today, 96% of U.S. health care providers use electronic health records (EHRs) – up from 10% in 2008. When I was a clinical intern, paper charts were the norm. Today, we practice medicine in a data-driven world. Large payers, such as Medicare and health insurance companies, would like to see providers leverage clinical data in ways that improve health outcomes, lower costs, and improve the patient experience.

In order to effect such change, payers have begun to shift some of the costs associated with suboptimal care to providers; this is a fundamental shift in the healthcare industry. Just a few years go, healthcare claims were paid without regard to clinical outcomes. Today, over 30% of all Medicare payments ($117B annually) are linked to clinical quality through a number of "value-based payment" mechanisms. Beginning this year, healthcare providers that fail to demonstrate progress toward data-driven quality improvement will face substantial financial penalties.

Providers have responded by assembling teams of clinical quality managers, informaticians, and health IT experts to navigate the transition to value-based payment. These clinical and technical leaders must overcome a number of technical challenges, and it remains to be seen whether all providers will ultimately succeed. An emerging theme among providers that are reporting early wins is the central role of big data technologies, such as Apache Spark.

Healthcare providers have a big data problem

- **Massive volume and overwhelming variety:** Collectively, U.S. hospitals generate over 20 petabytes of data each year. This figure doesn't include data collected in outpatient medical offices, pharmacies, third-party clinical labs, consumer wearables, or portable monitors.

- **Disparate systems:** Clinical data doesn't reside exclusively in electronic health records (EHRs). It is common to have clinical data scattered across radiology, lab, and pharmacy information systems (to name a few).

- **Siloed data stores:** Sometimes multiple EHR instances are deployed at a single physical location. For example: Stanford

Hospital and Lucile Packard Children's Hospital at Stanford both use the same EHR software but they operate separate instances. As a result, there is no way to retrieve a complete patient record from the adult hospital through the children's hospital EHR (and vice versa). This is a big problem in the setting of medical emergencies where patient care teams are comprised of staff from both hospitals (e.g. STAT C-sections or pediatric trauma). Poor data interoperability in healthcare creates information gaps that expose patients to harm and contribute billions to wasteful spending.

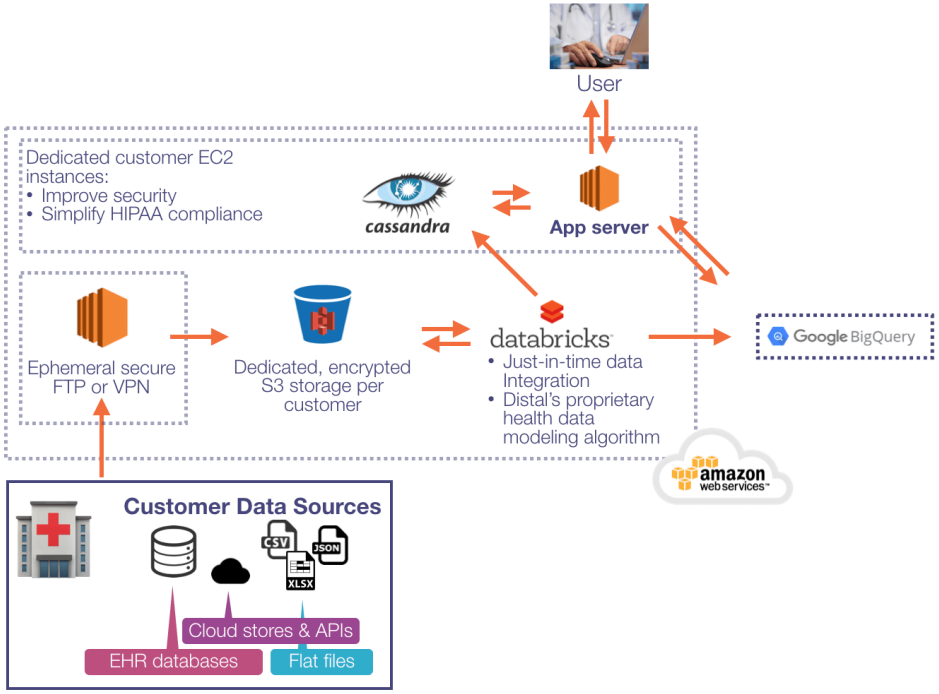
- **Inconsistent data schemas:** The RDBMS representation of raw clinical data consists of hundreds of tables that relate to one another in ways that are poorly documented (if at all). That's because the underlying schemas evolve as customizations are made to the front-end. This also means that Stanford's data schema is very different from that of Kaiser's, Sutter's, and UCSF – even though they all license their EHR software from the same vendor.

How Distal tames healthcare big data with Databricks

Distal offers a turnkey software solution that connects healthcare providers to the information they need to deliver exceptional care. Our clinical intelligence platform delivers clinically-meaningful insights through an easy-to-use web application. We use Databricks to build ETL pipelines that blend multiple data sources into a canonical source of truth. We also combine Distal's proprietary algorithm with MLlib's DataFrame-based pipelines to automate data modeling and clinical concept mapping.

Databricks has been key to building a cost-effective software solution that enables clinicians to engage in data-driven quality improvement. Databricks allows us to leverage the power of Apache Spark to:

- ingest massive volumes of structured, semi-structured, and unstructured data from a variety of sources,
- apply a common semantic layer to raw data,
- integrate and deduplicate data elements,
- analyze the data,
- map the results to a common set of API resources,
- surface the insights to the end-user, and
- protect patient privacy through encryption, granular access controls, and detailed audit trails.



How Databricks allows Distal to leverage the power of Apache Spark.

Databricks components that play particularly important roles include:

- **Notebooks and collaboration features:** Databricks notebooks allowed us to quickly iterate on new ETL components and to test how they fit into complementary pipelines. We began with a single notebook that included all of the code to get the job done. At this stage, there were snippets of Java, Scala, and Python in a single notebook. Once we had a working concept, we broke out functional code blocks into separate notebooks in order to run them as independent jobs. Throughout the process, we relied on Databricks' collaboration features to a) fix bugs, b) track changes, and c) share production-ready libraries internally.

- **Databricks library manager:** We felt the need to move quickly, so we incorporated open source libraries/packages into our ETL processes whenever possible. Very early on, we used Databricks to search through Spark Packages and Maven Central for the most stable, best-supported libraries. Once the pipelines began to take shape, we used the Databricks package manager to automatically attach required libraries to new clusters.

- **Integration with Amazon Web Services:** The fact that Databricks is built on Amazon Web Services (AWS) allowed us to seamlessly integrate with the full spectrum of AWS services while adhering to security best practices. Rather than relying exclusively on AWS access keys, which can be lost or stolen, Databricks enabled us to use IAM roles to restrict read/write access to S3 buckets. Furthermore, we restricted access to pipelines that handle sensitive data by applying user-level access controls through the Databricks admin panel.

A hint of things to come

With Apache Spark serving as the cornerstone of our ETL and analytics stack, we are confident in our ability to develop new features and products. In the near term, we are looking to augment our ETL pipelines and knowledge representation by developing novel ML algorithms and borrowing from existing methods. We are also looking

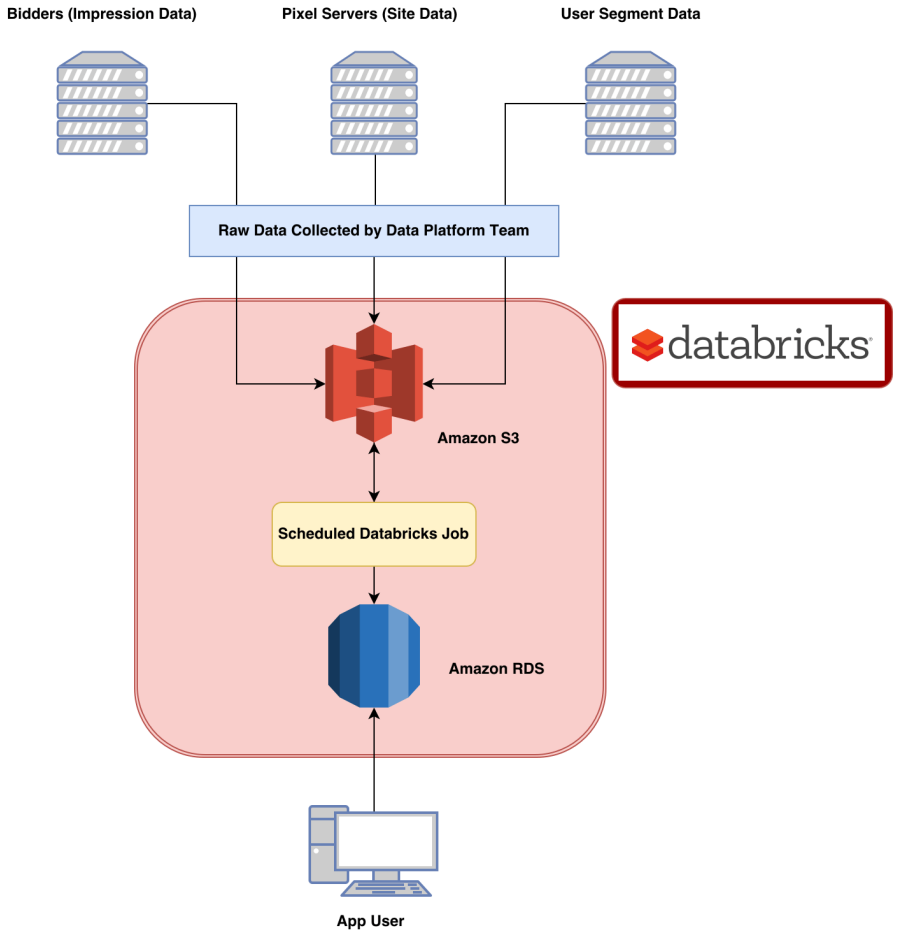
grow our team. If you love Databricks and would like to help us build products that have a measurable impact on millions of people, please introduce yourself by sending a quick note to team@distal.co.

Take Reports From Concept to Production with PySpark and Databricks

by Andrew Candela,
Senior Data Engineer at MediaMath

Introduction: What is MediaMath?

MediaMath is a demand-side media buying and data management platform. This means that brands and ad agencies can use our software to programmatically buy advertisements as well as manage and use the data that they have collected from their users. We serve over a billion ads each day, and track over 4 billion events that occur on the sites of our customers on a busy day. This wealth of data makes it easy to imagine novel reports in response to nearly any situation. Turning these ideas into scalable products consumable by our clients is challenging however.



The typical lifecycle of a new report. Popular reports are first built into a custom web-app for clients. Extremely popular ones are built into the UI of our core product.

Reporting at MediaMath

Typically the life cycle of a new report we dream up is:

Proof of concept is easy. All it takes is a novel client request or a bright idea combined with a scrappy analyst, and you've got a great idea for a new product. Building a proof of concept into a custom web app is harder, but should be achievable by a savvy team with a few days to dedicate to the project. Including the report in the core product is often prohibitively hard, as it requires coordination between potentially many teams with potentially competing priorities. This blog will focus on the second stage of this process: Turning a concept report into a scalable web app for client consumption, a process that Databricks has significantly streamlined for us.

The Audience Index Report: What is it?

The Audience Index Report (AIR) was developed to help current MediaMath clients understand the demographic makeup of users visiting their sites. Using segment data containing (anonymized of course) demographic and behavioral data of our users, the AIR provides a measure of the observed number of site-visitors from a segment compared to the expected number of site-visitors from that segment. The measure of this difference is referred to as the Index for segment s and site (also referred to as 'pixel') p. Please refer to the appendix for a more detailed description of the Index. For now, you should know that in order to compute the index for a site-segment you need to know four quantities:

1. $|s \cap p|$: the number of users in segment s who also fired pixel p (a pixel is used to track site visits)
2. $|Gs \cap p|$: the number of users in segment group G who also fired pixel p (G is the collection of segments to which s belongs)
3. $|s|$: the number of users in segment s
4. $|Gs|$: the number of users in the segment group G

Knowing the index of a site-segment is useful because it allows MediaMath clients to quantify the demographic and behavioral features of the users browsing their web content. It also is beneficial to our partners, because our clients can take insights gleaned from the report and then target an appropriate audience population (also known as a segment) by buying data from our partners.

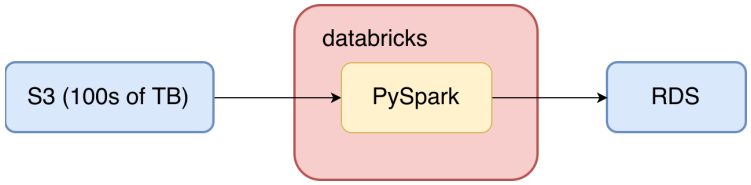
Producing the AIR – Why Databricks?

At a very high level, producing the AIR requires the following:

1. Process segment data to make it useable
2. Join processed segment data to first party data and aggregate
3. Write results to a relational database to serve our web app

I chose to implement this workflow with Apache Spark in the end, despite how primarily ETL heavy it was. I chose Spark for a couple of reasons, but it was primarily because much of the processing required was awkward to express with SQL. Spark's RDD APIs for Python provided the low-level customization I needed for the core ETL work. Additionally RDDs are readily transformed into DataFrames, so once I was done with the messy transformations I could slap a schema on my RDDs and use the very convenient DataFrame APIs to further aggregate them or write the results to S3 or a database.

The Databricks platform was also convenient because it brought all of the overhead required to run this workflow into one place. The Databricks UI is focused on notebooks, which suits this workflow well. I was able to create a couple of classes that handled all of the ETL, logic, logging and extra monitoring that I needed. I imported these classes into a handler notebook and used the Databricks job scheduler to configure the details of the cluster that this workflow runs on and to run the job itself. Now I've got my entire workflow running from just one Python notebook in Databricks! This convenience sped up the development process tremendously compared to previous projects and was just a lot more fun.



Databricks greatly reduces the overhead required for building an effective, back end workflow for our reports. Now many data engineering tasks are trivial and more energy can be focused on producing content, rather than maintaining overhead.

Processing segment data

Let's dig into how this report is generated. A user is defined to be 'in segment' if they have been added to the segment at least once in the last 30 days. Given this definition, the most useful format for the segment data is a key/value system. I will refer to this dataset as UDB (User DataBase). I chose to store the records as sequence files in S3 with the following structure:

KEY	VALUE
UserID	Nested Dictionary with segmentID and max and min timestamps corresponding to the time when the user was added to the segment.

Here is an example of one record from UDB:

```
(u'49ce54b1-c829-4d00-b490-a9443a2829b5',
  {11: {u'max': 1488499293, u'min': 1486658209},
  101: {u'max': 1488499293, u'min': 1486658209},
  123: {u'max': 1488499293, u'min': 1486230978}})
```

An added bonus here is that the first party data can be stored in exactly this same way, only in place of the segmentID we use pixelID (an identifier for the site). We produce this dataset by using the new day's data to update the current state of UDB each day. Here's what this step looks like:

```
# grab new data from a hive table
new_data = sqlContext.sql(self.query.format(self.current_date))
  .rdd.map(lambda x: (x[0],x))
  .combineByKey(createCombiner,mergeValue,mergeCombiners)
  .partitionBy(self.partitions)

# use the new data to update the current state
self.data = self.data.fullOuterJoin(new_data)
  .mapValues(combine_join_results)


# write out current state to S3 as sequence files
self.data
  .saveAsSequenceFile(
    self.path.format(self.current_date),
    compressionCodecClass="org.apache.hadoop.io.compress.DefaultCodec"
  )
```

We are well prepared now for Step 2: Joining and aggregating the data.

Joining data and aggregating

Since our intent is to understand the demographic information for each site, all we have to do is join the Site data to UDB. Site data and UDB are both stored as pairRDDs and are readily joined to produce a record like this:

```
(u'49ce54b1-c829-4d00-b490-a9443a2829b5',   #key
  ({11: {u'max': 1488499293, u'min': 1486658209},   #segments
    101: {u'max': 1488499293, u'min': 1486658209},
    123: {u'max': 1488499293, u'min': 1486230978}},
   {457455: {u'max': 1489356106, u'min': 1489355833}, #sites
    1016015: {u'max': 1489357958, u'min': 1489327086},
    1017238: {u'max': 1489355286, u'min': 1486658207}}))
```

After the join it's just a matter of counting up all of the siteID-segmentID and siteID-segmentGroup combinations we saw. This sounds easy, but it is the ugly part. Since one active user may have visited many sites and be in many segments, exploding the nested records actually causes quite a bit of extra data (up to $|p| \cdot |s|$ records for each user) so care must be taken to maintain an appropriate level of parallelism here. Using our example above, our result dataset would look like this:

```
(11, 457455),
(11, 1016015),
(101, 457455),
(101, 1016015),
(123,457455),
(123,1016015),
(123,1017238)
```

Notice how there are only seven lines here rather than nine. This is because we enforce the condition that a user must be in a segment before the first time they visit a site in order to be included in this report. Two records are scrubbed out here for that reason. Now I can convert this dataset into a DataFrame and aggregate it appropriately (`count()` grouping by site and segment). Since the result is itself a DataFrame, we are well set up for step 3 – writing to the relational database. This is the workflow for $|p \cap s|$. The workflow for $|p \cap G^s|$ is similar, and I'll omit it.

Writing to the relational database

We use an AWS hosted PostgreSQL RDS to serve data to our web-app. Spark's JDBC connector makes it trivial to write the contents of a DataFrame to a relational database such as this. Using PySpark and Postgres, you can run something like this:

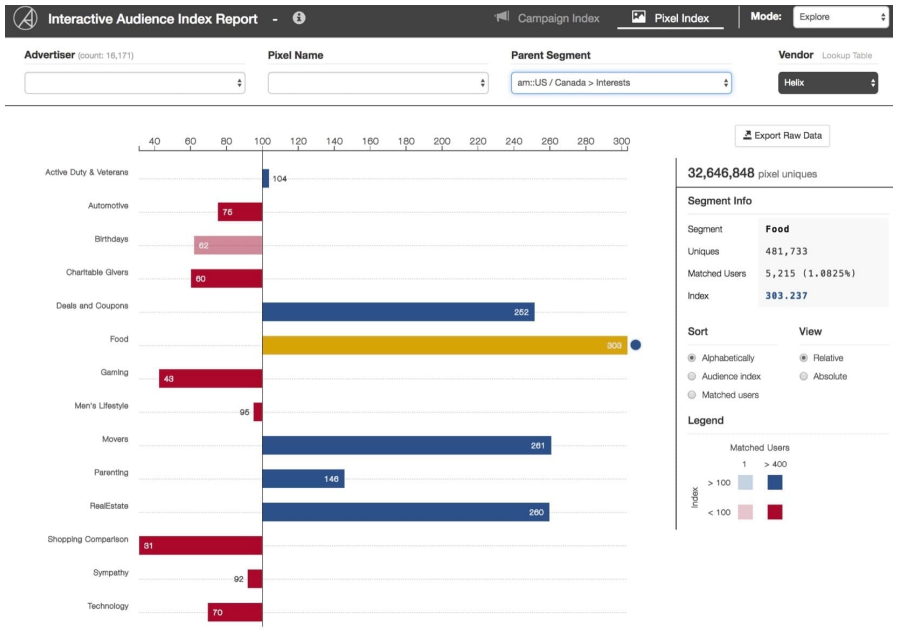
```
jdbcUrl='jdbc:postgresql://MY_HOST:PORT/MY_DATABASE?user=xx&password=xx'
yourDataframe.write.jdbc(jdbcUrl,YourTableName)
```

You can even bake table management into your class methods to streamline the process of updating a live table. For example if you can't use the mode=overwrite option of the write.jdbc() method (since the target table may be a production table you don't want to be down while you overwrite it), you can define a function like this:

```
def write_and_replace_table(TABLENAME):
  stage_table=TABLENAME + '_stage'
  temp_table=TABLENAME + '_tmp'
  #write the contents of the DF to RDS
  yourDataframe
    .write
    .jdbc(jdbcUrl,stage_table,mode='overwrite')
  #manage the table swap
  rename_table(TABLENAME,temp_table)
  rename_table(stage_table,TABLENAME)
  rename_table(temp_table,stage_table)
```

Conclusion

Now we've got everything in an Amazon RDS where careful database design allows our app to serve data quickly. We've managed to take hundreds of terabytes of data (this report aggregates the last 30 days) and condense it into a consumable report widely used by clients at MediaMath. Databricks provides a convenient platform to abstract away the painful parts of the orchestration and monitoring of this process. With Databricks we were able to focus on the features of the report rather than overhead, and have fun while we were doing it.



An example of the finished report. Segments that are over-indexing are shown in blue, while those that are under-indexing are shown in red. The advertiser and pixel names have been removed. Props to Sr Data Visualization Engineer Nicola Ricci for creating such an attractive visualization.

Sellpoints Develops Shopper Insights with Databricks

by Saman Keshmiri and Christopher Hoshino-Fish, Sellpoints

Saman is a Data Engineer at Sellpoints, where he responsible for developing, optimizing, and maintaining our backend ETL. Chris is a Data Scientist at Sellpoints, where he is responsible for creating and maintaining analytics pipelines, and developing user behavior models.

The simplest way to describe Sellpoints is we help brands and retailers convert shoppers into buyers using data analytics. The two primary areas of business for Sellpoints include some of the largest and most complex data sets imaginable; we build and syndicate interactive content for the largest online retailers to help brands increase conversions, and we provide advanced display advertising solutions for our clients. We study behavior trends as users browse retail sites, interact with our widgets, or see targeted display ads. Tracking these events amounts to an average of 5,500 unique data points per second.

The Data Team's primary role is to turn our raw event tracking data into aggregated user behavior information, useful for decision-making by our account managers and clients. We track how people interact with retailer websites prior to purchasing. Our individual log lines represent a single interaction between a customer and some content related to a product, which we then aggregate to answer questions such as "how many visits does a person make prior to purchasing," or "how much of this demonstration video do people watch before closing the window, and are customers more likely to purchase after viewing?" The Data Team here at Sellpoints is responsible for creating user behavior models, producing analytical reports and visualizations for the content we track, and maintaining the visualization infrastructure.

In this blog, we will describe how Sellpoints is able to not only implement our entire backend ETL using the Databricks platform, but unify the entire ETL.

Use Case: Centralized ETL

Sellpoints needed a big data processing platform, one that would be able to replicate our existing ETL, based in Amazon Web Services (AWS), but improve speed and potentially integrate data sources beyond S3, including MySQL and FTP. We also wanted the ability to test MLlib, Apache Spark's machine learning library. We were extremely happy to be accepted as one of Databricks early customers, and outline our goals:

SHORT TERM GOALS:

- Replicate existing Hive-based ETL in SparkSQL – processing raw CSV log files into useable tables, optimizing the file size, and partitioning as necessary
- Aggregate data into tables for visualization
- Extract visualization tables with Tableau

LONG TERM GOALS:

- Rewrite ETL in Scala/Spark and optimize for speed
- Integrate MySQL via the JDBC
- Test and productize MLlib algorithms
- Integrate Databricks further into our stack

We have a small Data Team at Sellpoints, consisting of three Data Engineers, one Data Scientist, and a couple Analysts. Because our team is small and we do not have the DevOps resources to maintain a large cluster ourselves, Databricks is the perfect solution. Their platform allows the analysts to spin up clusters with the click of a button, without having to deal with installing packages or specific versions of software. Additionally, Databricks provides a notebook environment for Spark and makes data discovery incredibly intuitive. The ease of use is unparalleled and allows users across our entire Data Team to investigate our raw data.

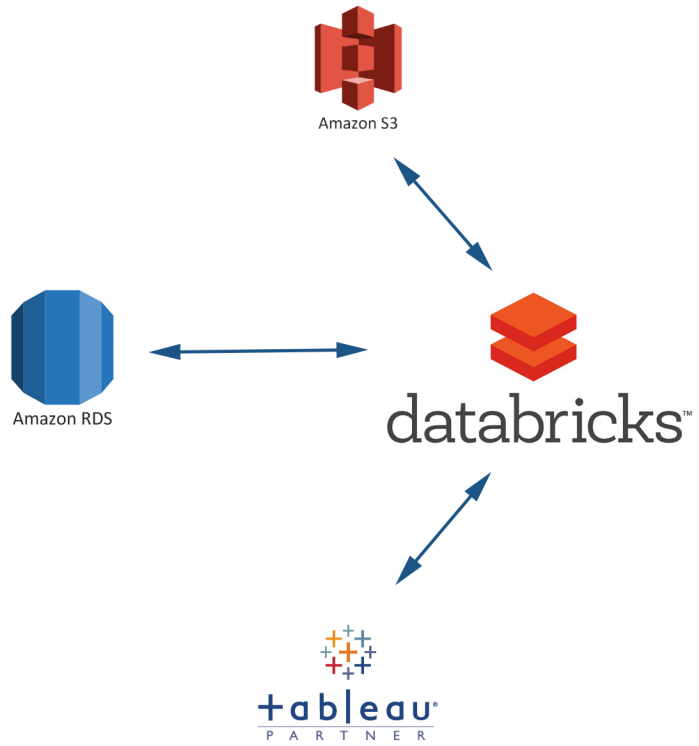


Illustration of Sellpoints' ETL architecture.

Architecture and Technical Details

Our immediate goal was to replicate our existing Hive-based ETL in SparkSQL, which turned out to be a breeze. SparkSQL can directly implement Hive libraries and utilizes a virtually identical syntax, so transferring our existing ETL was as simple as copy/paste and uploading the Hive CSV Serde. Soon after, Databricks released their native Spark CSV Serde, and we implemented it without issue. We also needed to extract these tables into Tableau, which Databricks again made simple. Databricks implements a Thrift server by default for JDBC calls, and Tableau has a SparkSQL connector that utilizes the JDBC. We needed to modify one setting here:

```
spark.sql.thriftServer.incrementalCollect = true
```

This tells the Thrift server that data should be sent in small increments rather than collecting all the data into the driver node and then pushing it out, which will cause the driver to run out of space quite quickly if you're extracting a non-trivial amount of data. With that, replicating our ETL in Databricks was complete. We were now leveraging Databricks to process 100GB/day of raw CSV data.

Our next steps involved learning Scala/Spark and the various optimizations that can be made inside Spark, while also starting to integrate other data sources. We had an SQL query that was taking over 2 hours to complete, but the output was being saved to S3 for consumption by Databricks. The query took so long because it involved joining 11 tables together and building a lookup with 10 fields. While optimizations could've been made to the structure of the MySQL db or the query itself, we thought, why not do it in Databricks? We were able to reduce the query time from 2 hours down to less than 2 minutes; and again, since the SparkSQL syntax encompasses all the basic SQL commands, implementing the query was as easy as copy/paste (N.B. we needed to modify our AWS settings in order to get this to work). A simplified version of the following code:

```
import com.mysql.jdbc.Driver
// create database connection
val host = "Internal IP Address of mysql db (10.x.x.x)"
val port = "3306"
val user = "username"
val password = "password"
val dbName = "databaseName"
val dbURL = s"jdbc:mysql://$host:$port/$dbName?user=$user&password=$password"
//load table as a val
val table =
  sqlContext
    .read
    .format("jdbc")
    .options(
      Map(
        "url" -> s"$dbURL",
        "dbtable" -> "dbName.tableToLoad",
        "driver"  -> "com.mysql.jdbc.Driver"))
    .load()
//register as a temporary SparkSQL table
table.registerTempTable("tableName")
//run the query
val tableData =
 sqlContext.sql("""
    SELECT *
    FROM tableName""")
```

We also receive and send out files on a daily basis via FTP. We need to download and store copies of these files, so we started downloading them to S3 using Databricks. This allowed us to further centralize our ETL in Databricks . A simplified version of the code below:

```
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPFile;
import org.apache.commons.net.ftp.FTPReply;
import java.io.File;
import java.io.FileOutputStream;
val ftpURL = "ftp.url.com"
val user = "username"
val pswd = "password"
val ftp = new FTPClient()
ftp.connect(ftpURL)
ftp.enterLocalPassiveMode()
ftp.login(user,pswd)
val folderPath = "pathToDownloadFrom"
val files = ftp.listFiles(s"/$folderPath").map(_.getName)
dbutils.fs.mkdirs(s"""/$savePath""")
val outputFile = new File(s"""/dbfs/$savePath/filename.txt""")
val output = new FileOutputStream(outputFile)
ftp.retrieveFile(s"/$folder/filename.txt", output)
output.close()
ftp.logout
```

We switched over from saving our processed data as CSV to Parquet. Parquet is columnar, which means that when you're only using some of the columns in your data, it is able to ignore the other columns. This has massive speed gains when you have trillions of rows, and allows us to decrease time waiting for initial results. We try to keep our individual parquet files around 120MB in size, which is the default block size for Spark and allows the cluster to load data quickly (which we found to be a bottleneck when using small CSV files).

Consolidating our ETL steps into a single location has added benefits when we get to actually analyzing the data. We always try to stay on the cutting edge of machine learning and offline analysis techniques for identifying new user segments, which requires that we maintain a recent set of testable user data, while also being able to compare the output of new techniques to what we're currently using. Being able to consolidate all of our data into a single platform has sped up our iteration process considerably, especially since this single platform includes the MLlib project.

Benefits and Lessons Learned

In the 1.5 years since implementing Databricks, we've been able to:

- Speed up our first-step ETL by ~4x
- Integrate mysql and ftp data sources directly into Databricks, and speed up those portions of the ETL by 100x
- Optimize our processed data for analysis

We have also learned a lot during this process. A few tips and lessons include:

- Store your data as Parquet files
- Leverage the processing power of Spark and implement any RDS ETL in Databricks via the JDBC
- Partition your data as granularly as possible while maintaining larger file sizes – loading data is slow, but processing is fast!
- To learn more about how Sellpoints is using Databricks, read the case study.

A special thank you to Ivan Dejanovic for his edits and patience.

Accelerate Innovation with Interactive Data Science

Case studies for data science teams

Parallelizing Large Simulations with Apache SparkR on Databricks

by Wayne W. Jones and Dennis Vallinga (Shell)
and Hossein Falaki (Databricks)

Introduction

Apache Spark 2.0 introduced a new family of APIs in SparkR, the R interface to Apache Spark to enable users to parallelize existing R functions. The new dapply, gapply and spark.lapply methods open exciting possibilities for R users. In this post, we present details on one use case jointly done by Shell Oil Company and Databricks.

Use Case: Stocking Recommendation

In Shell, current stocking practices are often driven by a combination of vendor recommendations, prior operational experience and "gut feeling." As such, a limited focus is directed towards incorporating historical data in these decisions, which can sometimes lead to excessive or insufficient stock being held at Shell's locations (e.g., an oil rig).

The prototype tool, Inventory Optimization Analytics solution, has proven that Shell can use advanced data analysis techniques on SAP inventory data to:

- Optimize warehouse inventory levels
- Forecast safety stock levels
- Rationalize slow moving materials
- Review and re-assign non-stock and stock items on materials list
- Identify material criticality (e.g., via bill of materials linkage, past usage or lead time)

To calculate the recommended stocking inventory level requirement for a material, the Data Science team has implemented a Markov Chain Monte Carlo (MCMC) bootstrapping statistical model in R. The model is applied to each and every material (typically 3000+) issued across 50+ Shell locations. Each individual material model involves simulating 10,000 MCMC iterations to capture the historical distribution of issues. Cumulatively, the computational task is large but, fortunately, is one of an embarrassingly parallel nature because the model can be applied independently to each material.

Existing Setup

The full model is currently executed on a 48-core, 192GB RAM standalone physical offline PC. The MCMC bootstrap model is a custom

built set of functions which use a number of third-party R packages (`"fExtremes", "ismev", "dplyr", "tidyr", "stringr"`).

The script iterates through each of the Shell locations and distributes the historical material into roughly equally sized groups of materials across the 48 cores. Each core then iteratively applies the model to each individual material. We are grouping the materials because a simple loop for each material would create too much overhead (e.g. starting the R process etc.) as each calculation takes 2-5 seconds. The distribution of the material group jobs across the cores is implemented via the R parallel package. When the last of the individual 48 core jobs complete, the script moves on to the next location and repeats the process. The script takes a total time of approximately 48 hours to calculate the recommended inventory levels for all Shell locations.

Using Apache Spark on Databricks

Instead of relying on a single large machine with many cores, Shell decided to use cluster computing to scale out. The new R API in Apache Spark was a good fit for this use case. Two versions of the workload were developed as prototypes to verify scalability and performance of SparkR.

Prototype I: A proof of concept

For the first prototype, we tried to minimize the amount of code change as the goal was to quickly validate that the new SparkR API can handle the workload. We limited all the changes to the simulation step as following:

For each Shell location list element:

1. Parallelize input date as a Spark DataFrame
2. Use `SparkR::gapply()` to perform parallel simulation for each of the chunks.

With limited change to existing simulation code base, we could reduce the total simulation time to 3.97 hours on a 50 node Spark cluster on Databricks.

Prototype II: Improving performance

While the first prototype was quick to implement, it suffered from one obvious performance bottleneck: a Spark job is launched for every iteration of the simulation. The data is highly skewed and as a result, during each job most executors are waiting idle for the straglers to finish before they can take on more work from the next job. Further, at the beginning of each job, we spend time parallelizing the data as a Spark DataFrame while most of the CPU cores on the cluster are idle.

To solve these problems, we modified the pre-processing step to produce input and auxiliary date for all locations and material values up-front. Input data was parallelized as a large Spark DataFrame. Next, we used a single `SparkR::gapply()` call with two keys: location ID and material ID to perform the simulation

With these simple improvements, we could reduce the simulation time to 45 minutes on a 50 node Spark cluster on Databricks.

Improvements to SparkR

SparkR is one of the latest additions to Apache Spark, and the apply API family was the latest addition to SparkR at the time of this work. Through this experiment, we identified a number of limitations and bugs in SparkR and fixed them in Apache Spark.

- [SPARK-17790] Support for parallelizing R data.frame larger than 2GB
- [SPARK-17919] Make timeout to RBackend configurable in SparkR
- [SPARK-17811] SparkR cannot parallelize data.frame with NA or NULL in Date columns

What's Next?

If you are a SparkR developer and you want to explore SparkR, get an account on Databricks today and peruse through our SparkR documentation.

Analysing Metro Operations Using Apache Spark on Databricks

by Even Vinge, Senior Manager
EY Advisory, Data & Analytics

Introduction

Sporveien is the operator of the Oslo Metro, a municipally owned Metro network supplying the greater Oslo area with public transportation since 1898. Today, the Metro transports more than 200,000 passengers on a daily basis and is a critical part of the city's infrastructure.

Sporveien is an integrated operator, meaning they are responsible for all aspects of keeping the service running. Metro operations, train maintenance, infrastructure maintenance and even construction of new infrastructure falls under Sporveien's domain. As a result, the company employs a large workforce of experienced metro drivers, operators, mechanics and engineers to ensure smooth operation of the metro.

Over the next few years, with a growing population and political shifts towards green transportation, the Metro will need to be able transport even more people than today. Sporveien's stated goal for the coming years is to enable this expansion while keeping costs down and, just as importantly, punctuality high.

Punctuality — the percentage of Metro departures delivered on time — is a key metric being closely monitored by Sporveien, as any deviation in punctuality can cause big problems for Sporveien's end customers — the citizens of Oslo. The Metro is a closed loop system running on a tight schedule, and even small deviations from the schedule can become amplified throughout the network and end up causing large delays.

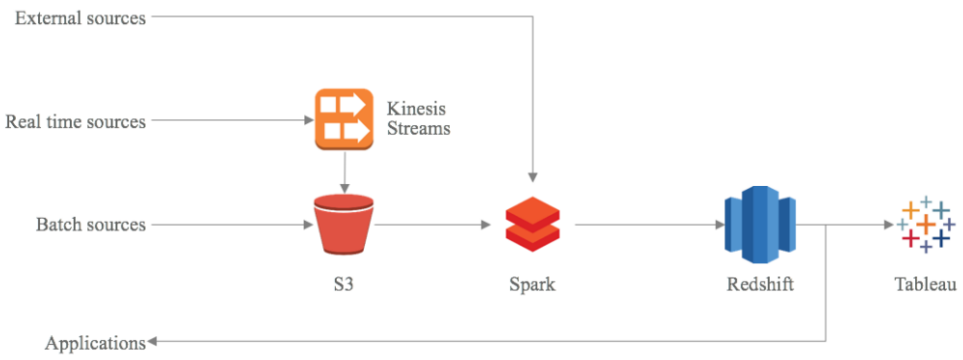
The Metro is being monitored continuously by a signalling system following every movement of every train, logging around 170,000 train movements and 3 million signal readings every day. Due to the volume and complexity of the signalling data, Sporveien was, until recently, not able to fully utilize this information source to get a deeper understanding of how the different aspects of Metro operations affect overall performance.

When we started looking at this data and the hypotheses we wanted to investigate, we realized this was a great case for Apache Spark. The combination of high data volumes with a need for near real time processing, as well as the need for extensive data cleaning and transformation to be able to extract value all pointed to the conclusion that a modern data analysis platform would be necessary. Wanting to start performing analyses and operationalizing them quickly without

getting bogged down in the complexities of setting up and administering a Spark cluster, we decided to use Databricks for both analysis and production deployment. Sporveien had already chosen AWS as their preferred cloud stack, so that Databricks was tightly integrated was an added advantage.

Our Data Pipeline: S3, Spark, Redshift and Tableau

In order to get insights from the signaling data, it is extracted from the source system and pushed to a Kinesis queue, before being loaded to S3 using a simple AWS Lambda function. Then the Spark processing sets off. First the data is cleaned and transformed to calculate precise timings of the actual metro departures. It is then combined with detailed scheduling data from the planning system using a multi-step matching routine which ensures that any single actual departure is only matched with a single planned departure. The departure data is then further enriched with passenger flow data fetched from the automated passenger counting system installed in each train. Finally, the enriched and cleaned data is dumped to our data warehouse in Redshift.



Sporveien Data Pipeline with Databricks

This whole process runs every five minutes throughout the day on real time data, using heuristics for some of the more complex calculations. To ensure precise results over time, the whole day's data is recalculated during the night using more detailed calculations. We like to think of this as our version of lambda architecture, where we are able to use the same stack and the same logic for both real time and batch processing.

Our data pipelines and our Redshift cluster have a myriad of uses within Sporveien, including operational analysis of operations performance and deviations, KPI reporting, real time dashboards, and recently also feeding analyzed sensor data back into production systems to enable condition based maintenance of some of Sporveien's railway infrastructure.

The signalling data is one of several data pipelines we have running in production. At the time of writing this, we have around 40 Databricks notebooks running nightly or continuously, coordinated using Databricks Notebook Workflows and scheduled in parallel using the Jobs feature. We have also set up an integration to Sporveien's Slack instance to make sure that everyone on the analytics team is aware of what's happening with our production jobs.

In all this, the Spark framework has proven to be an immensely powerful tool kit. An obvious advantage is being able to spread computing over several nodes — for instance we run our real-time jobs on a small cluster, and then launch a bigger one for recalculations at night. But equally important, we've found that PySpark SQL is a surprisingly succinct and communicative way of expressing data transformation logic. We get to write transformation logic that is understandable and auditable even to non-analysts, something that increases the trust in the results greatly.

The Challenge: How to do Continuous Integration on Databricks

As we're continuously extending our analyses and refining our algorithms, we've found ourselves needing to deploy updated transformation logic to our development and production repositories quite often.

We use Databricks for all aspects of writing and running Spark code, and were early adopters of the GitHub integration as we have the rest of our codebase in GitHub as well. However, we found the built in Github integration to be too simplistic to work for our needs. Since we are using Databricks Notebook Workflows, we have many dependencies between different notebooks, and as such we wanted to be able to perform commits across notebooks, being in control of all changes and what to deploy when. We also found it cumbersome trying to do larger deployments to production.

We have been working with Github Flow as our deployment workflow in previous projects, using branches for feature development and pull requests for both code reviews and automated deployments, and felt this could work well in this setting as well — if we only had the necessary CI setup in place. With this motivation, we spoke to Databricks to see what could be done, and agreed that an API for accessing the workspace would make a lot of what we wanted possible. A few months later we were very happy to be invited to try out the new Workspace API, and quickly confirmed that this was what we needed to be able to get a complete code versioning and CI setup going.

Bricker

The Workspace API makes it easy to list, download, upload and delete notebooks in Databricks. If we could get this working together, we would be able to sync a whole folder structure between our local machines and Databricks. And, of course, as soon as we had the

notebooks locally, we would have all our usual Git tools at our disposal, making it simple to stage, diff, commit, etc.

So to make this work, we built a small Python CLI tool we've called bricker. Bricker allows us to sync a local folder structure to Databricks just by entering the command **bricker up**, and similarly to sync from Databricks to the local filesystem using **bricker down**. Bricker uses your current checked out Git branch name to determine the matching folder name in Databricks.

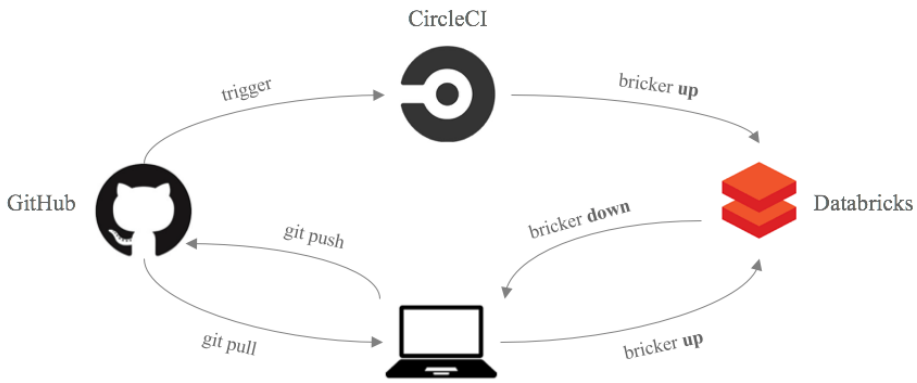
We have also added a feature to create a preconfigured cluster with the right IAM roles, Spark configs etc. using **bricker create_cluster** to save some time as we found ourselves often creating the same types of clusters.

We thought more people might be needing something similar, so we've open sourced the tool and published it on PyPI. You can find install instructions here.

Our Analysis Development Lifecycle: GitHub, Bricker, CircleCL and Databricks

Using bricker, our analysis lifecycle now works like this: We start a new feature by branching from the dev branch on our laptops, and **bricker up** to get a new cloned folder in Databricks where we develop and test new and updated analyses. When we're ready to merge, we use **bricker down** and commit to our feature branch, before pushing to Github and creating a pull request.

We've set up the cloud CI solution Circle CI to trigger on pull request merges, and so as soon as the pull request is merged into dev or production, CI fetches the merged code and runs **bricker up** to automatically deploy the merge to our Databricks production folder. With this, we've been able to reduce our entire production deployment into a one click approval in Github, allowing the entire team to iterate on analyses quickly and safely without worrying about broken dependencies, and to deploy into production as often as we need.



Sporveien Deployment Pipeline with Databricks

Impact

Using this setup, we have been able to very rapidly establish a robust and full featured analytics solution in Sporveien. We have spent less time thinking about infrastructure and more about creating analyses and data products to help Sporveien better understand how different aspects of the Metro operations affect punctuality.

In the last few years, Sporveien has really embraced data-driven decision making. We are using detailed analyses and visualizations of different aspects of performance in every setting from strategic planning to operational standups in the garages and with operations teams.

Looking forward, there is still a lot of exciting work to be done at Sporveien. We have only seen the beginning of what insights can be had by understanding and analysing the ever growing amounts of data generated by all kinds of operational systems. As we get more sensors and higher resolution measurements, we are now looking at applying machine learning to predict when railway infrastructure components will break down.

By continuously optimizing operations and improving performance, Sporveien is doing everything to be able to offer the citizens of Oslo a reliable and punctual service both today and in the many years to come.

Delivering a Personalized Shopping Experience with Apache Spark on Databricks

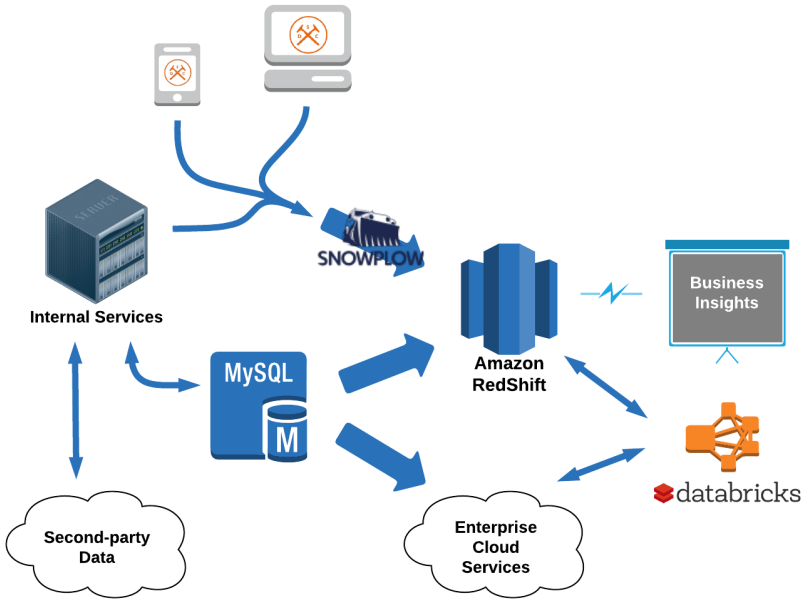
by Brett Bevers, Engineering Manager, Data Engineering
Dollar Shave Club

Dollar Shave Club (DSC) is a men's lifestyle brand and e-commerce company on a mission to change the way men address their shaving and grooming needs. Data is perhaps the most critical asset in achieving a cutting-edge user experience. Databricks has been an important partner in our efforts to build a personalized customer experience through data. This post describes how the Databricks platform supported all stages of development and deployment of a powerful, custom machine learning pipeline that we use to deliver targeted content to our members.

DSC's primary offering is a monthly subscription for razor cartridges, which are shipped directly to members. Our members join and manage their account on our single-page web app or native mobile apps. During their visit, they can shop our catalogue of grooming and bathroom products—we now have dozens of products organized under distinctive brands. Courtesy of the club, members and guests can enjoy *Original Content*, articles and videos created for people who enjoy our characteristic style. They can satisfy their curiosity on health and grooming topics with articles that don't remind them of their junior high health class. They can get quick tips on style, work and relationships, or they can read DSC's fun take on big questions, like "How long can civilization last on Earth?" DSC also seeks to engage people on social media channels, and our members can be enthusiastic about joining in. By identifying content and offers of the most interest to each individual member, we can provide a more personalized and better membership experience.

Data at Dollar Shave Club

DSC's interactions with members and guests generate a mountain of data. Knowing that the data would be an asset in improving member experience, our engineering team invested early in a modern data infrastructure. Our web applications, internal services and data infrastructure are 100% hosted on AWS. A Redshift cluster serves as the central data warehouse, receiving data from various systems. Records are continuously replicated from production databases into the warehouse. Data also moves between applications and into Redshift mediated by Apache Kafka, an open-source streaming platform. We use Snowplow, a highly-customizable open-source event pipeline, to collect event data from our web and mobile clients as well as server-side applications. Clients emit detailed records of page views, link clicks, browsing activity and any number of custom events and contexts. Once data reaches Redshift, it is accessed through various analytics platforms for monitoring, visualization and insights.

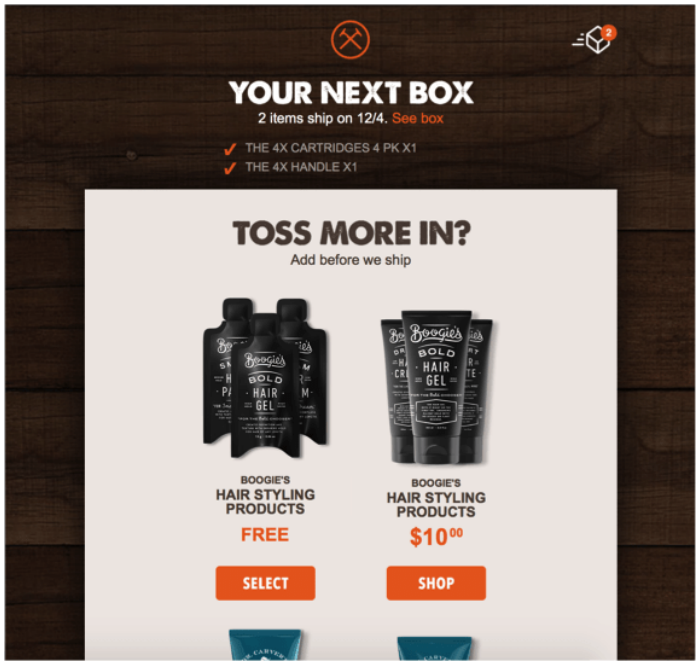


An illustration of Dollar Shave Club's data pipeline.

With this level of visibility, there are abundant opportunities to learn from our data and act on it. But identifying those opportunities and executing at scale requires the right tools. Apache Spark—a state-of-the-art cluster computing framework with engines for ETL, stream processing and machine learning—is an obvious choice. Moreover, Databricks' latest developments for data engineering make it exceedingly easy to get started with Spark, providing a platform that is apt as both an IDE and deployment pipeline. On our first day using Databricks, we were equipped to grapple with a new class of data challenges.

Use Case: Recommendation Engine

One of the first projects we developed on Databricks aimed to use predictive modeling to optimize the product recommendations that we make to our members over a particular email channel. Members receive a sequence of emails in the week before their subscription box is shipped. These emails inform them about the upcoming shipment and also suggest additional products that they can include in their box. Members can elect to add a recommended product from the email with just a few clicks. Our goal was to produce, for a given member, a ranking of products that prescribes which products to promote in their monthly email and with what priority.



An example of the email Dollar Shave Club sends to users to recommend new products.

We planned to perform an exhaustive exploration in search of behavior that tends to indicate a member's level of interest in each of our products. We would extract a variety of metrics in about a dozen segments of member data, pivot that data by hundreds of categories, actions and tags, and index event-related metrics by discretized time. In all, we included nearly 10,000 features, for a large cohort of members, in the scope of our exploration. To contend with a large, high-dimensional and sparse dataset, we decided to automate the required ETL and Data Mining techniques using Spark Core, Spark SQL and MLlib. The final product would be a collection of linear models, trained and tuned on production data, that could be combined to produce product rankings.

We set out to develop a fully automated pipeline on Spark with the following stages:

1. Extract data from warehouse (Redshift)
2. Aggregate and pivot data per member
3. Select features to include in final models

Step 1: Extract Data

We start by looking at various *segments* of data in our relational databases; groupings of records that need to be stitched together to describe a domain of events and relationships. We need to understand each data segment—how it can be interpreted and how it might need to be cleaned up—so that we can extract accurate, portable, self-describing representations of the data. This is crucial work that collects together domain expertise and institutional knowledge about the data and its life-cycle, so it is important to document and communicate what is learned in the process. Databricks provides a "notebook" interface to the Spark shell that makes it easy to work with data interactively, while having complete use of Spark's programming model. Spark notebooks proved to be ideal for trying out ideas and quickly sharing the results or keeping a chronicle of your work for reference later.

For each data segment, we encapsulate the specifics of cleaning and denormalizing records in an *extractor* module. In many cases, we can simply export tables from Redshift, dynamically generate SQL queries, and then let Spark SQL do all the heavy lifting. If necessary, we can cleanly introduce functional programming using Spark's *DataFrames* API. And the application of domain-specific metadata has a natural home in an extractor. Importantly, the first steps for processing a particular data segment are neatly isolated from that for other segments and from other stages of the pipeline. Extractors can be developed and tested independently. And they can be reused for other explorations or for production pipelines.

```
def performExtraction(
extractorClass, exportName, joinTable=None, joinKeyCol=None,
startCol=None, includeStartCol=True, eventStartDate=None
):
customerIdCol = extractorClass.customerIdCol
timestampCol = extractorClass.timestampCol
extrArgs = extractorArgs(
customerIdCol, timestampCol, joinTable, joinKeyCol,
startCol, includeStartCol, eventStartDate
)
Extractor = extractorClass(**extrArgs)
exportPath = redshiftExportPath(exportName)
return extractor.exportFromRedshift(exportPath)
```

Example code for a data extraction pipeline. The pipeline uses the interface implemented by several extractor classes, passing arguments to customize behavior. The pipeline is agnostic to the details of each extraction.

```
def exportFromRedshift(self, path):
export = self.exportDataFrame()
writeParquetWithRetry(export, path)
return sqlContext.read.parquet(path)
.persist(StorageLevel.MEMORY_AND_DISK)

def exportDataFrame(self):
self.registerTempTables()
query = self.generateQuery()
return sqlContext.sql(query)
```

Example code for an extractor interface. In many cases, an extractor simply generates a SQL query and passes it to SparkSQL.

Step 2: Aggregate and Pivot

The data extracted from the warehouse is mostly detailed information about individual events and relationships; But what we really need is an aggregated description of activity over time, so that we can effectively search for behavior that indicates interest in one product or another. Comparing specific event types second by second is not fruitful—at that level of granularity, the data is much too sparse to be good fodder for machine learning. The first thing to do is to aggregate event-related data over discrete periods of time. Reducing events into counts, totals, averages, frequencies, etc., makes comparisons among members more meaningful and it makes data mining more tractable. Of course, the same set of events can be aggregated over several different dimensions, in addition to time. Any or all of these numbers could tell an interesting story.

Aggregating over each of several attributes in a dataset is often called *pivoting* (or *rolling up*) the data. When we group data by member and pivot by time and other interesting attributes, the data is transformed from data about individual events and relationships into a (very long) list of features that describe our members. For each data segment, we encapsulate the specific method for pivoting the data in a meaningful way in its own module. We call these modules *transformers*. Because the pivoted dataset can be extremely wide, it is often more performant to work with RDDs rather than DataFrames. We generally represent the set of pivoted features using a sparse vector format, and we use key-value RDD transformations to reduce the data. Representing member behavior as a sparse vector shrinks the size of the dataset in memory

and also makes it easy to generate training sets for use with MLlib in the next stage of the pipeline.

Step 3: Data Mining

At this point in the pipeline, we have an extremely large set of features for each member and we want to determine, for each of our products, which subset of those features best indicate when a member is interested in purchasing that product. This is a data mining problem. If there were fewer features to consider—say, dozens instead of several thousand—then there would be several reasonable ways to proceed. However, the large number of features being considered presents a particularly hard problem. Thanks to the Databricks platform, we were easily able to apply a tremendous amount of computing time to the problem. We used a method that involved training and evaluating models on relatively small, randomly sampled sets of features. Over several hundred iterations, we gradually accumulate a subset of features that each make a significant contribution to a high-performing model. Training a model and calculating the evaluation statistic for each feature in that model is computationally expensive. But we had no trouble provisioning large Spark clusters to do the work for each of our products and then terminating them when the job was done.

It is essential that we be able to monitor the progress of the data mining process. If there is a bug or data quality issue that is preventing the process from converging on the best performing model, then we need to detect it as early as possible to avoid wasting many hours of processing time. For that purpose, we developed a simple dashboard on Databricks that visualizes the evaluation statistics collected on each iteration.

Final Models

The evaluation module in MLlib makes it exceedingly simple to tune the parameters of its models. Once the hard work of the ETL and data mining process is complete, producing the final models is nearly effortless. After determining the final model coefficients and parameters, we were prepared to start generating product rankings in production. We used Databricks' scheduling feature to run a daily job to produce product rankings for each of the members who will be receiving an email notification that day. To generate feature vectors for each member, we simply apply to the most recent data the same extractor and transformer modules that generated the original training data. This not only saved development time upfront, it avoids the problems of dual maintenance of exploration and production pipelines. It also ensures that the models are being applied under the most favorable conditions—to features that have precisely the same meaning and context as in the training data.

Future Plans with Databricks and Apache Spark

The product recommendation project turned out to be a great success that has encouraged us to take on similarly ambitious data projects at DSC. Databricks continues to play a vital role in supporting our development workflow, particularly for data products. Large-scale data mining has become an essential tool that we use to gather information to address strategically important questions; and the resulting predictive models can then be deployed to power smart features in production. In addition to machine learning, we have projects that employ stream-processing applications built on Spark Streaming; for example, we consume various event streams to unobtrusively collect and report metrics, or to replicate data across systems in near real-time. And, of course, a growing number of our ETL processes are being developed on Spark.

Using Databricks to Democratize Big Data and Machine Learning at McGraw-Hill Education

Matt Hogan
Sr. Director of Engineering, Analytics and Reporting
at McGraw-Hill Education

McGraw-Hill Education is a 129-year-old company that was reborn with the mission of accelerating learning through intuitive and engaging experiences – grounded in research. When we began our journey, our data was locked in silos managed within teams and was secure, but not shared and not fully leveraged. The research needed to create machine learning models that could make predictions and recommendations to students and instructors was hard because the paths to the data needed to train those models were often unclear, slow, and sometimes completely closed off. To be successful, we first needed to change our organizational approach to burst through the silos that were preventing us from unlocking the full potential of shared, consumable information. We needed a culture of data. More precisely, a culture of secure data that protected the privacy of students and institutions that worked with us but allowed enough safe access to stoke intellectual curiosity and innovation across the organization. To guide our research, we founded the McGraw-Hill Education Learning Science Research Council (LSRC). The LSRC is chartered with guiding our own research as well as working with the education and technology communities to drive the field of learning science research forward.

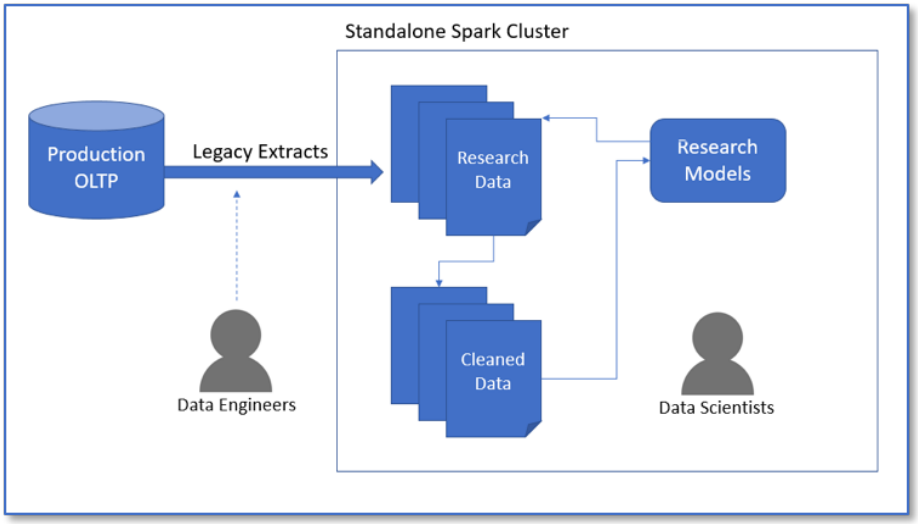
Our transformation is underscored by our deep understanding of learning, including knowledge of how students and instructors leverage both content and learning management solutions. Through exploring how content and learning management systems are used coupled with understanding a student's progress through material, we're able to leverage fully anonymized data to drive adaptive models to engage with students and deliver differentiated instruction to drive better outcomes. Aggregating, processing, developing models, and operationalizing those models as features in learning management systems requires a cohesive data engineering and data science platform, and Databricks has partnered with us to fill that engineering need. We have made some incredible strides forward by coupling the

technological edge gained from Databricks with a concerted organizational shift to align security, DevOps, platform, and product teams around a new data-centric paradigm of operating that allows interdisciplinary teams to cross-cut through data silos and unlock insights that would have otherwise not been possible.

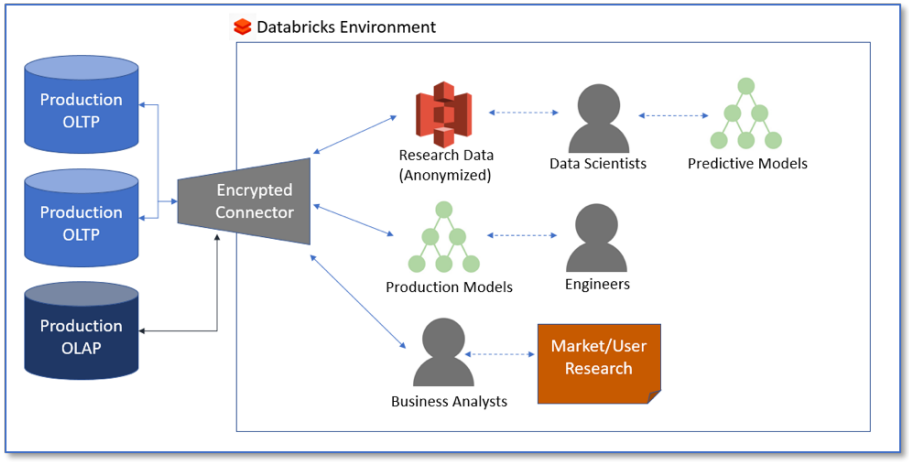
Democratizing Data and Machine Learning with Databricks

To test the efficacy of insights generated from learning science research, we first needed to democratize access to data and machine learning thereby reducing the cost of iterating on models and testing their effect in the market and on learners. Databricks provides us with a ubiquitous data environment that can be leveraged by business analysts, data engineers, and data scientists to work collaboratively toward a set of common goals.

Before bringing Databricks on, we had many Apache Spark clusters running on leased hardware. To get data to researchers, we created one-off extracts that would pull data from one of our product databases, scrub Personally Identifiable Information (PII), and store the data as flat files on one of the leased Spark clusters. This led to many problems including data quickly becoming stale, researchers working on models that were out of sync with the data in the extracted flat files, and no clear way to integrate the output of models developed by the data scientists.



After adopting Databricks, we could create secure connections between read-replicas of our learning systems and our data platform, ensuring that data was encrypted both in motion and at rest while allowing us to protect the confidentiality of learners. The connections allow researchers to have access to fresh data and current data models in a secure environment. Because the Spark systems are used by both engineering teams and researchers, output from the researchers could be leveraged to create new models and populate our data mart with new insights. As an added benefit, we could provide access to the business analytics teams who are now able to conduct their own research along with the data scientists on a single, unified analytics platform.



Iterating on the Connect Attrition Model

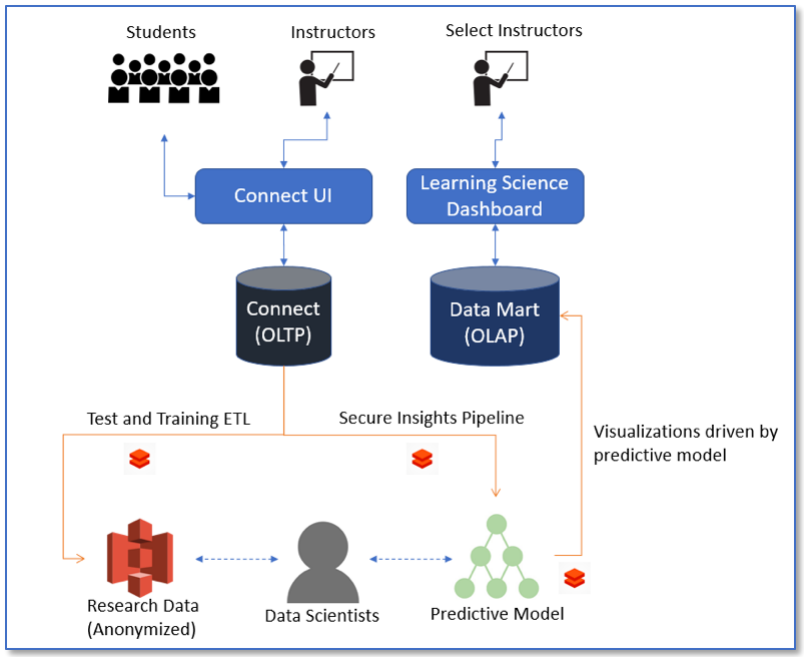
One of our first efforts catalyzed by Databricks was our Connect student attrition model – a classifier that can predict which students are at risk for dropping a course based on progress working through the material and outcomes on quizzes and assignments – within 2-4 weeks of a class starting. The McGraw-Hill Education Connect Platform is a highly reliable, secure learning management solution that covers homework, assessments, and course content delivery and does so leveraging many adaptive tools to improve student outcomes. While Connect appears to the world as a single, seamless platform and experience, it is made up of several pluggable technologies and products all generating their own data and metrics in real time. These components are hosted through a mix of in-house data centers and Amazon Web Services, making data aggregation, cleaning, and research a challenge.

To facilitate the data science work needed to train the model, we undertook several initiatives:

- Created secure access to read-replicas of the Connect OLTP system
- Created a new platform designed to facilitate rapid iteration and evaluation of models coupled with user experience and impact
- Worked to change McGraw-Hill Education's culture from one where data silos were the norm to one in which current, secure, anonymized data is made accessible to those who have a valid use for it
- Designed a system of leveraging encrypted S3 buckets (with Parquet in some cases) that researchers and developers could use to securely and easily move, evaluate, and read (using SparkSQL) data

To establish a cycle of rapid feedback with researchers in the academic community as well as a select number of instructors, we created a platform called the Learning Science Dashboard (LSD). The LSD leverages our single sign-on solution to provide access to visualizations that researchers create using Databricks through reading from and writing to S3 buckets. Once the first level of evaluation was completed, we engaged with designers and engineers to build an operational notebook that could use the connector to query data from live classes in Connect and write predictions to a staging area in S3. Using the Databricks job scheduler made the operationalization of the model a trivial task. A new UI was deployed (this time using AngularJS) into the LSD and select instructors with

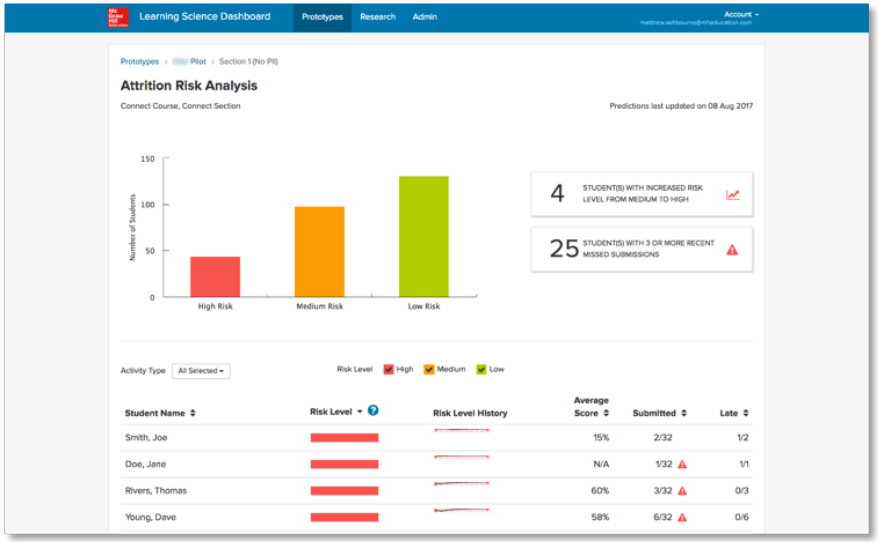
access to Connect could log in and see an evaluation version of the report to gain insights on how their classes are performing.



Deploying and Market Testing the Model

The initial visualizations created by the data scientists conveyed important meaning, but were not designed to be consumed by instructors focused on quickly gaining insights on class performance. The first round of visualizations that we tested were created by data scientists in Databricks and exported to S3 buckets with controlled access through the LSD.

The second round of iterations included engineering teams and designers and were designed to quickly convey insights – allowing instructors to focus interventions where they were needed.



Thoughts in Closing

Using Databricks, we could securely transform ourselves from a collection of data silos with limited access to data and minimal collaboration to an organization with democratized access to data and machine learning with data engineers, data scientists, business analysts, and front-end engineers all working in the same space and sharing ideas. By building out some complimentary technologies and processes, we reduced the time to market and created a path by which we could quickly and easily validate both models and insights with external researchers and instructors in the classroom.

Furthermore, our legacy data pipelines (ETL) were decommissioned, resulting in a significant cost savings and simplification of how we run our infrastructure. In the past, we had provisioned EC2 instances running all the time – even when data was not moving. We also had leased hardware running large but statically sized Spark clusters at a cost of thousands per month. Through leveraging Databricks' dynamic provisioning of instances/clusters when needed coupled with the notion of just-in-time data warehousing, we were able to reduce operational cost of our analytics infrastructure by 30%.

Simplify big data and AI with a Unified Analytics Platform from the team that created Apache Spark.

Our mission at Databricks is to accelerate innovation by unifying data science, data engineering and the business. We hope this collection of customer blog posts we've curated in this eBook will provide you with the insights and tools to help you solve your biggest data problems.

If you enjoyed the content in this eBook, visit the Databricks Blog for more technical tips, best practices, and customer case studies from the Spark experts at Databricks.

If you'd like to learn more about the Databricks Unified Platform, visit databricks.com/product
Or sign up for a free trial at databricks.com/try