

Personalization at Spotify using Cassandra

Posted on January 9, 2015 by Kinshuk Mishra (<https://labs.spotify.com/author/kinshukm/>) and Matt Brown (<https://labs.spotify.com/author/mattbrownspotify/>)

[Like](#) 52 people like this. [Sign Up](#) to see what your friends like.

By **Matt Brown** (<https://twitter.com/mattnworb>) and **Kinshuk Mishra** (https://twitter.com/_kinshukmishra)

At Spotify we have over 60 million active users (<https://press.spotify.com/us/information/>) who have access to a vast music catalog of over 30 million songs. Our users have a choice to follow thousands of artists and hundreds of their friends and create their own music graph. On our service they also discover new and existing content by experiencing a variety of music promotions (album releases, artist promos), which get served over our ad platform. These options have empowered our users and made them really engaged. Over time they have created over 1.5 billion playlists and just last year they streamed over 7 billion hours (<http://www.idigitaltimes.com/spotify-users-stream-7-billion-hours-music-2014-ed-sheeran-katy-perry-earning-most-398195>) worth of music.

But at times an abundance of options has also made our users feel a bit lost. How do you find that right playlist for your workout from over a billion playlists? How do you discover new albums which are relevant to your taste? We help our users discover and experience relevant content by personalizing their experience on our platform.

Personalizing user experience involves learning their tastes and distastes in different contexts. A metal genre listener might not enjoy an announcement for a metal genre album when they are trying to put their kid to sleep and playing kid's music at night. Serving them a recommendation for a kid's music album might be more relevant in that context. But this experience might not be relevant for another metal genre listener who doesn't mind receiving metal genre album recommendations during any context. These two users with similar listening habits might have different preferences. Personalizing their experiences on Spotify according to their respective taste in different contexts helps us make them more engaged.

Given these product insights we set out to build a personalization system which could analyze both real-time and historic data to understand user's context and behavior respectively. Over time we've evolved our personalization tech stack due to a flexible architecture and ensured we used the right tools to solve the problem at scale.

Overall Architecture

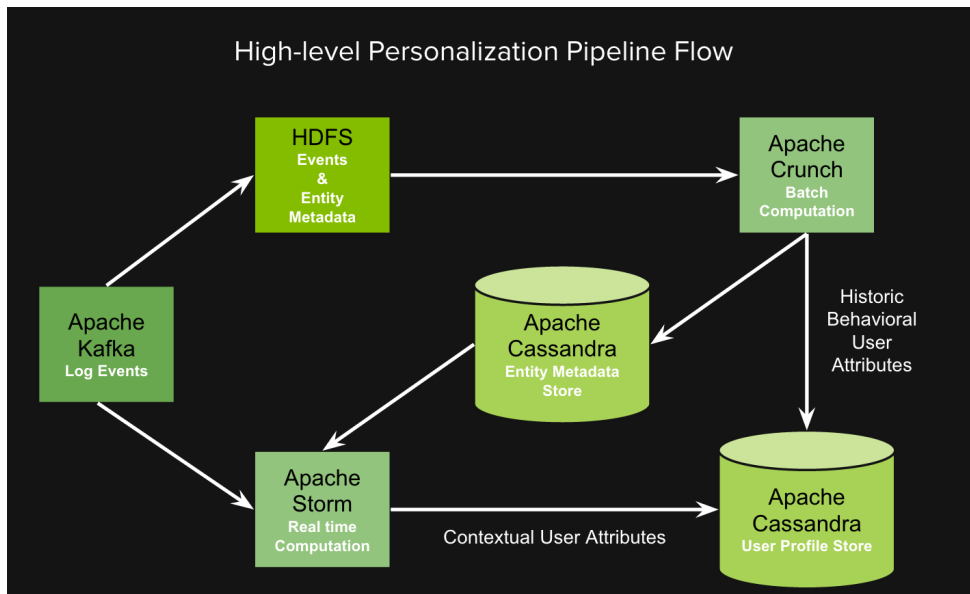
In our personalization stack we are using *Kafka* (<http://kafka.apache.org/>) for log collection, *Storm* (<https://storm.apache.org/>) for real-time event processing, *Crunch* (<https://labs.spotify.com/2014/11/27/crunch/>) for running batch map-reduce jobs on *Hadoop* (<http://hadoop.apache.org/>) and *Cassandra* (<http://cassandra.apache.org/>) to store user profile attributes and metadata about entities like playlists, artists, etc.

In the diagram below logs are forwarded by Kafka producers, running on different services and emitting different kinds of events like completion of a song and delivery of an ad impression, to the Kafka broker. There are 2 sets of Kafka consumers, subscribed to different topics, for consuming events:

1. Kafka Consumers for Hadoop which write these events to HDFS. All the raw logs on HDFS are then processed in Crunch to remove duplicate events, filter out unwanted fields and convert records into Avro Format.
2. Kafka Consumer Spouts (<https://storm.apache.org/apidocs/backtype/storm/spout/ISpout.html>) running within Storm topologies (<https://storm.apache.org/documentation/Tutorial.html>) which stream the events for real-time computation.

There are also other Crunch pipelines which ingest and generate metadata (genre, tempo, etc.) for different entities. These data records are stored in HDFS and also exported using Crunch to Cassandra for real-time lookup in Storm pipelines. We will call the Cassandra cluster which stores entity metadata the Entity Metadata Store (EMS).

The Storm pipelines process raw log events from Kafka, filter out unwanted events, decorate the entities with metadata fetched from EMS, group it per user and determine user level attributes by some algorithmic combination of aggregation and derivation. These user attributes when combined represent a user's profile and they are stored in a Cassandra Cluster which we will call the User Profile Store (UPS).



(<https://spotifylabscom.files.wordpress.com/2015/01/screen-shot-2015-01-09-at-4-09-51-pm.png>)

Why is Cassandra a good fit?

Since UPS is central to our personalization system, in this post we will elaborate why we chose Cassandra for storage. When we started shopping around for different storage solutions for UPS we desired a solution which could:

- Scale horizontally
- Support replication – preferably cross-site
- Have low latency even at the cost of consistency since we aren't performing transactions
- Have the ability to load bulk and streaming data from Crunch and Storm respectively
- Have a decent ability to model different data schemas for different use-cases of entity metadata since we didn't want to invest in yet another solution for EMS as that would have increased our operational cost.

We considered variety of solutions that we commonly use at Spotify like Memcached, Sparkey (<https://github.com/spotify/sparkey>) and Cassandra. Only Cassandra fit the bill for all these requirements.

Horizontal scaling

Cassandra's ability to scale with the number of nodes added to a cluster is highly advertised and has been well documented already (<http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html>), so we were confident that it would be a good choice for our use case. Our project started with a relatively small size of data but has grown from a few GB to more than 100 GB today. Along the way, we've easily been able to expand our storage capacity by increasing the number of nodes in the cluster; for example we recently doubled the size of the cluster and observed that the latency (both at the median and 99th percentile) was nearly cut in half.

In addition, Cassandra's replication and availability features have been a huge help. While we have unfortunately had a few cases of nodes crashing because of GC or hardware issues, our services accessing the Cassandra cluster were barely affected, since all of the data was available on other nodes and the client drivers are smart enough to transparently failover.

Cross-site Replication

Spotify is available in nearly 60 countries across the globe. Our backend services run in two data centers in North America and two in Europe. To make sure that users could still be served by our personalization system in the event of failures in any one data center, it was very important for us to be able to store data in at least two of our data centers.

We are currently using the NetworkReplicationStrategy for our personalization clusters to replicate the data between the EU data center and NA data center. This allows users to access data in the Spotify data center nearest to them, and provides the redundancy capability as described above.

While we have not yet had any incidents that caused downtime of an entire cluster in an entire data center, we have performed several tests of migrating user traffic from one data center to another and our Cassandra setup has flawlessly handled the increase in traffic from handling requests from both sites in one site.

Low-latency operations and tunable consistency

Given Spotify's user base, computing personalized data about user's listening in real-time results in a large volume of data to store to a database. Beyond wanting queries to read this data to be fast, it was also important to us that the write path of storing data could also have very low latency.

Since writes in Cassandra (http://www.datastax.com/documentation/cassandra/2.1/cassandra/dml/dml_write_path_c.html) result in storage in an append-only structure (<https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveldb/>), writes are generally very fast. In fact in our usage of Cassandra for personalization, writes are typically an order of magnitude faster than reads.

Since the personalization data we compute in real-time is not transactional in nature, and lost data is easily replaced with new data in a few minutes from a user's listening stream, we can tune the consistency level (<http://www.datastax.com/documentation/cassandra/2.1/cassandra/dml/dmlAboutDataConsistency.html>) of our write and read operations to sacrifice consistency for even lower latency (by not waiting for all replicas to respond before having the operation succeed).

Bulk Data Transfer

At Spotify we have a large investment in Hadoop and HDFS, and nearly all of our insights about users come from running jobs on historical data.

Cassandra offers options for bulk importing data (http://www.datastax.com/documentation/cassandra/2.0/cassandra/tools/toolsBulkloader_t.html) from other data sources (such as HDFS) into the Cassandra cluster by building entire SSTables and then streaming the tables into the cluster. Streaming the tables into the cluster is much simpler, faster and more efficient than sending millions or more of individual INSERT statements for all of the data you want to load into Cassandra.

Spotify has open-sourced a tool on-top of the bulk SSTable loader named `hdfs2cass` (<https://github.com/spotify/hdfs2cass>) for the specific use case of ingesting data from HDFS.

While the availability of this feature didn't influence our decision to use Cassandra for personalization, it has made integrating our data from HDFS into Cassandra very easy and simple to operate.

Our Cassandra data model

Our data model for personalization data in Cassandra has undergone a few evolutions since we have started on this project.

Initially, we thought we should simply have two column families – one for attributes (key-value pairs) for a user, and a similar set of attributes for “entities” (such as artists, tracks, playlists etc). The former would only contain short-lived data with TTLs, while the latter would be populated infrequently with relatively static data.

The motivation behind storing pairs of keys and values as individual CQL rows rather than attempting to create one CQL column for each “feature” (and having one CQL row per user) is to allow the services and batch jobs producing this data to be independent of the service consuming the data. With this approach, passing a new type of “feature” in this data from the producer all the way to the end-user does not require any changes in the consuming service, since that service is simply querying for all pairs for a given user.

These column families looked something like:

```
CREATE TABLE entitymetadata (
  entityid text,
  featurename text,
  featurevalue text,
  PRIMARY KEY (entityid, featurekey)
)

CREATE TABLE userprofilelatest (
  userid text,
  featurename text,
  featurevalue text,
  PRIMARY KEY (userid, featurename)
)
```

This structure worked fine for initial prototypes, but we soon ran into a few issues which necessitated a rethinking of the structure for the metadata about “entities”:

1. The structure of the *entitymetadata* column meant that we could easily add new types of entitymetadata, but we could not remove data for certain *featurenames* if we experimented with a new type of data and found it was not useful and no longer needed.
2. For some types of metadata about entities, the data did not have a natural *string* representation, but could be stored more easily using one of the CQL collection types. For instance, there were some cases where it felt more natural to represent the value as a *list<text>*, because the value was a list of things with an ordering that we wanted to maintain; or a *map<text, int>* for storing a ranking of values for an entity.

Instead of a single column family with values that were keyed by (*entityid, featurename*), we adopted an approach of having one column family for each “featurename”, with the values using the appropriate CQL type. For instance:

```
CREATE TABLE playlisttag (
  entityid text,
  featurevalue list<text>,
  PRIMARY KEY (entityid)
)
```

Using the appropriate CQL type instead of a string-fits-all approach meant that we no longer had to make any awkward decisions about how to represent possibly-non-text-data as text (addressing point #2 above) and that we could easily drop column families for features that we added as an experiment but later decided were not useful. From an operational perspective this also allows us to inspect the number of read and write operations for each “feature”.

As of late 2014 we have close to a dozen column families for this type of data and have found it much easier to work with compared to trying to cram all pieces of data into a single representation.

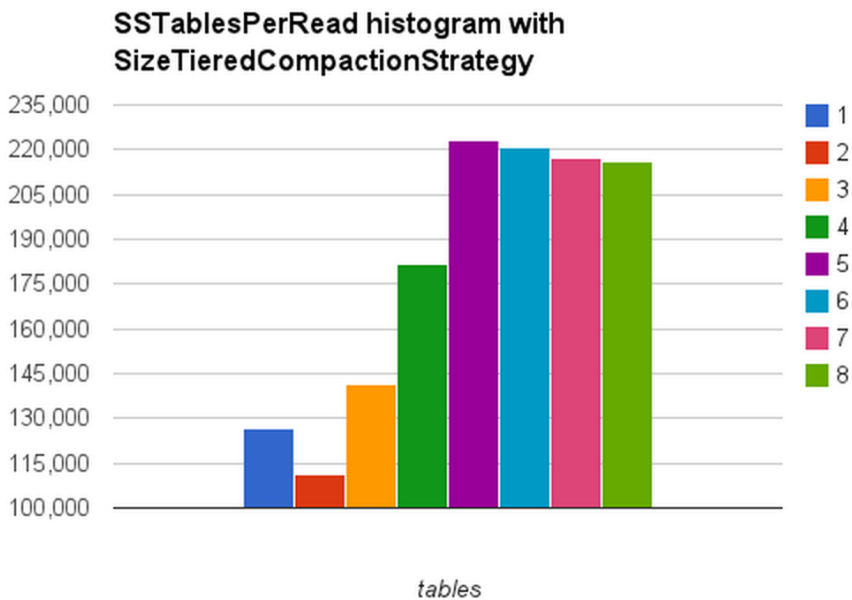
The column family for user data underwent a similar evolution when the `DateTieredCompactionStrategy` (<https://labs.spotify.com/2014/12/18/date-tiered-compaction/>) became available in Cassandra (which we are proud to say was a contribution to the Cassandra project from a fellow Spotify employee).

We were unhappy with the read and write latency to the `userprofilelatency` column family and thought that DTCS might be a great fit for our use case since all of this data was timestamp-oriented and had short TTLs, so we experimented with changing from using STCS to DTCS for our “`userprofilelatest`” table to improve the latencies.

Before making any changes, we made note of the `SSTablesPerRead` histograms from `nodetool` to have a “before” state to compare our changes against. A sample histogram from around this time looked like:

```
SSTables per Read
1 sstables: 126733
2 sstables: 111414
3 sstables: 141385
4 sstables: 181974
5 sstables: 222921
6 sstables: 220581
7 sstables: 217314
8 sstables: 216296
10 sstables: 380294
```

Note that not only is the histogram relatively flat, meaning that a large number of read requests need to touch many SSTables, but that the numbers actually increased as you go down the histogram.



(<https://spotifylabscom.files.wordpress.com/2015/01/screen-shot-2015-01-09-at-4-12-18-pm.png>)

We knew after examining the histograms that the latency was more-than-likely caused by the sheer number of SSTables that each read was going through, and that the key to reducing the latency would be in reducing the number of SSTables that had to be examined for each read.

Initially the results after enabling DTCS were not so promising, but this was caused not by any issues with the compaction strategy itself but by the fact that we had started to mix the short-lived TTL data in this column family with longer-lived “static” data for a user that had no TTL.

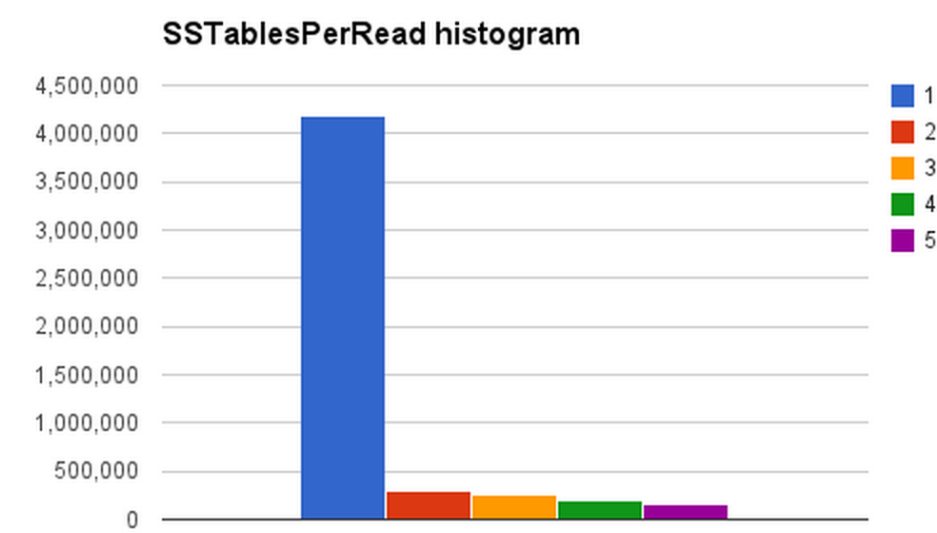
To test if DTCS could better cope with rows with a TTL if all of the rows in the table had a TTL, we split this column family into two column families, one column family for the “static” rows that had no TTL and one columnfamily for rows with a TTL.

After carefully migrating the backend service that is a consumer of this data (by first changing the service to read from both the old and new columnfamily, and then reading only from the new columnfamily once it was fully populated), our experiment was a success: DTCS on the columnfamily with only TTLed rows yielded `SSTablesPerRead` histograms where the ratio of reads that needed to touch only 1 SSTable compared to 2 SSTables was roughly between 6:1 and 12:1 (depending on the host).

An example output from `nodetool cfhistograms` with this change:

```
SSTables per Read
1 sstables: 4178514
2 sstables: 302549
3 sstables: 254760
4 sstables: 197695
5 sstables: 154961
...
```

or illustrated graphically:



(<https://spotifylabscom.files.wordpress.com/2015/01/screen-shot-2015-01-09-at-4-12-46-pm.png>)

In attacking the latency issues we had with the `userprofilelatest` columnfamily, we learned a few valuable lessons about Cassandra:

- DTCS is a great fit for time series, especially if the rows all have TTLs (and `SizeTieredCompactionStrategy` is a bad fit for this type of data)
 1. But DTCS is not so great if you mix rows with TTLs and rows without TTLs, so don't mix data in that way
 2. For tables with DTCS / TTL data we set `gc_grace_period` to 0 and effectively disable read repairs since we do not need them – the TTL is shorter than the grace period would be.
- `nodetool cfhistograms` and the number of SSTables-touched-per-read is probably the best resource for understanding why the latency numbers for a column family look the way they do, so make sure to frequently measure it and pipe it into your graphing system for observing changes over time.

With several tweaks to our data model and Cassandra configurations we've managed to build a robust storage layer for serving personalization data to multiple backend services. After fine-tuning the configurations we've incurred very little additional operational maintenance cost to keep our Cassandra clusters running. We've exposed a bunch of cluster and dataset specific metrics in a dashboard and set up alerts to fire if the metrics start trending in the wrong direction. This has helped us passively keep track of the cluster health. Other than doubling the size of our cluster to keep up with the additional load we haven't had to deal a lot with cluster maintenance. Even the cluster doubling part was fairly easy and seamless and deserves another post to explain all the details.

We've overall been very satisfied with Cassandra as a solution for all our personalization needs and are confident to scale it up to serve personalized experience to our ever growing size of engaged user base.

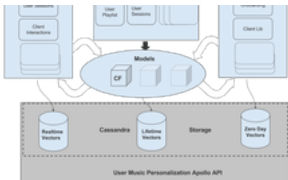
Thanks to PlanetCassandra (<http://planetcassandra.org/blog/personalization-at-spotify-using-apache-cassandra/>) for encouraging us to blog about our Cassandra experience.

[Like](#) 52 people like this. [Sign Up](#) to see what your friends like.

Share this:

- Twitter (<https://labs.spotify.com/2015/01/09/personalization-at-spotify-using-cassandra/?share=twitter&nb=1>)
- Facebook (<https://labs.spotify.com/2015/01/09/personalization-at-spotify-using-cassandra/?share=facebook&nb=1>)
- LinkedIn (<https://labs.spotify.com/2015/01/09/personalization-at-spotify-using-cassandra/?share=linkedin&nb=1>)
- Email (<https://labs.spotify.com/2015/01/09/personalization-at-spotify-using-cassandra/?share=email&nb=1>)

Related



(<https://labs.spotify.com/2016/08/07/commoditizing-music-ml-services/>)

Commoditizing Music Machine Learning : Services (<https://labs.spotify.com/2016/08/07/commoditizing-music-ml-services/>)

In "Labs"



(<https://labs.spotify.com/2013/03/15/backend-infrastructure-at-spotify/>)

Backend infrastructure at Spotify (<https://labs.spotify.com/2013/03/15/backend-infrastructure-at-spotify/>)

In "Labs"



(<https://labs.spotify.com/2017/10/16/big-data-processing-at-spotify-the-road-to-scio-part-1/>)

Big Data Processing at Spotify: The Road to Scio (Part 1) (<https://labs.spotify.com/2017/10/16/big-data-processing-at-spotify-the-road-to-scio-part-1/>)

In "Labs"

This entry was posted in **LABS ([HTTPS://LABS.SPOTIFY.COM/CATEGORY/LABS/](https://labs.spotify.com/category/labs/))** and tagged Apache Cassandra (<https://labs.spotify.com/tag/apache-cassandra/>), Apache Crunch (<https://labs.spotify.com/tag/apache-crunch/>), Apache Storm (<https://labs.spotify.com/tag/apache-storm/>), big data (<https://labs.spotify.com/tag/big-data/>), cassandra (<https://labs.spotify.com/tag/cassandra/>), data modeling (<https://labs.spotify.com/tag/data-modeling/>), personalization (<https://labs.spotify.com/tag/personalization/>). Bookmark the permalink (<https://labs.spotify.com/2015/01/09/personalization-at-spotify-using-cassandra/>).

← [HOW SPOTIFY SCALES APACHE STORM \(HTTPS://LABS.SPOTIFY.COM/2015/01/05/HOW-SPOTIFY-SCALES-APACHE-STORM/\)](https://labs.spotify.com/2015/01/05/how-spotify-scales-apache-storm/)

[DIVERSIFY – CREATING A HACKATHON WITH 50/50 FEMALE AND MALE PARTICIPANTS →](https://labs.spotify.com/2015/01/13/diversify-how-we-created-a-hackathon-with-50-50-female-male-participants/)
([HTTPS://LABS.SPOTIFY.COM/2015/01/13/DIVERSIFY-HOW-WE-CREATED-A-HACKATHON-WITH-50-50-FEMALE-MALE-PARTICIPANTS/](https://labs.spotify.com/2015/01/13/diversify-how-we-created-a-hackathon-with-50-50-female-male-participants/))

Comments



1. **Personalization at Spotify using Cassandra – thoughts... (<http://irrlab.com/2016/07/01/personalization-at-spotify-using-cassandra/>)** says:

July 2, 2016 at 12:12 am (<https://labs.spotify.com/2015/01/09/personalization-at-spotify-using-cassandra/#comment-615>)

[...] Personalization at Spotify using Cassandra [...]

2. **The Main Types of NoSQL Databases | Studio 3T (<https://studio3t.com/whats-new/nosql-database-types/>)** says:

February 22, 2018 at 8:53 am (<https://labs.spotify.com/2015/01/09/personalization-at-spotify-using-cassandra/#comment-1076>)

[...] Spotify uses Cassandra to store user profile attributes and metadata about artists, songs, etc. for better personalization [...]

3 Comments

Sort by **Newest**



Add a comment...



Samir Bessalah

Awesome

Like · Reply · 3y



Cassandra María Odqvist

Go Spotify

Like · Reply · 1 · 3y



Saeid SvH

why don't you use apache spark for batch processing?

Like · Reply · 3 · 3y

Facebook Comments Plugin

Recent Posts

Introducing Coördinator: A new open source project made at Spotify to inject some whimsy into datavisualizations


(<https://labs.spotify.com/2018/03/02/introducing-coordinator-a-new-open-source-project-made-at-spotify-to-inject-some-whimsy-into-data-visualizations/>)

2018/4/26	Personalization at Spotify using Cassandra Labs
Spotify Spotlight: Interview with Continuous Delivery Engineer DonnieThompson (https://labs.spotify.com/2018/02/05/spotify-spotlight-interview-with-continuous-delivery-engineer-donnie-thompson/)	
Spotify Spotlight: Interview with Data Engineer AnkitaPawar (https://labs.spotify.com/2018/01/22/spotify-spotlight-interview-with-data-engineer-ankita-pawar/)	
Testing of Microservices (https://labs.spotify.com/2018/01/11/testing-of-microservices/)	
Spotify Spotlight: Interview with Engineer DaenneySluijters (https://labs.spotify.com/2018/01/08/spotify-spotlight-interview-with-engineer-daenney-sluijters/)	
Spotify Retro Kit (https://labs.spotify.com/2017/12/15/spotify-retro-kit/)	
Spotify Spotlight: Interview with Data Engineer DaraElass (https://labs.spotify.com/2017/11/29/spotify-spotlight-interview-with-data-engineer-dara-elass/)	
Spotify Spotlight: Interview with Product Manager SabrinaAronson (https://labs.spotify.com/2017/11/27/spotify-spotlight-interview-with-product-manager-sabrina-aronson/)	
Autoscaling Pub/Sub Consumers (https://labs.spotify.com/2017/11/20/autoscaling-pub-sub-consumers/)	
Big Data Processing at Spotify: The Road to Scio (Part2) (https://labs.spotify.com/2017/10/23/big-data-processing-at-spotify-the-road-to-scio-part-2/)	

@SpotifyEng on Twitter

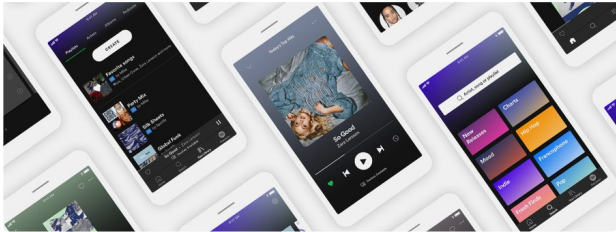
Tweets by @SpotifyEng

Spotify Engineering Retweeted




Spotify Press
@SpotifyPress

Personalized Discovery: Listen to Music You Love with All-New Free on Spotify: newsroom.spotify.com/2018-04-24/per...



Apr 24, 2018

Spotify Engineering Retweeted



Scala Days
@scaladays

Embed

View on Twitter

(<http://instagram.com/spotify>)

(<https://www.facebook.com/Spotify>)

(<https://twitter.com/spotify>)

