Lessons from Large-Scale Cloud Software at Databricks

Matei Zaharia
@matei_zaharia
Outline

The cloud is eating software, but why?

About Databricks

Challenges, solutions and research questions
Outline

The cloud is eating software, but why?

About Databricks

Challenges, solutions and research questions
Traditional Software

Vendor

Dev Team

6-12 months

Release

6-12 months

Customers

Users
Ops

Users
Ops

Users
Ops

Cloud Software

Vendor

Dev + Ops Team

1-2 weeks

Customers

Users
Ops

Users
Ops

Users
Ops
Why Use Cloud Software?

1. Management built-in: much more value than the software bits alone (security, availability, etc)

2. Elasticity: pay-as-you-go, scale on demand

3. Better features released faster
Differences in Building Cloud Software

+ Release cycle: send to users faster, get feedback faster

+ Only need to maintain 2 software versions (current & next), in fewer configurations than you’d have on-prem

– Upgrading without regressions: very hard, but critical for users to trust your cloud (on-prem apps don’t require this)
  - Includes API, semantics, and performance regressions
Differences in Building Cloud Software

- **Building a multitenant service**: significant scaling, security and performance isolation work that you don’t need on-prem.

- **Operating the service**: security, availability, monitoring, etc (but customers would have to do it themselves on-prem).

+ **Monitoring**: see usage live for ops & product analytics.

Many of these challenges aren’t studied in research.
About Databricks

Founded in 2013 by the Apache Spark team at UC Berkeley

Data and ML platform on AWS and Azure for >5000 customers

- Millions of VMs launched/day, processing exabytes of data/day
- 100,000s of users

1000 employees, 200 engineers, >$200M ARR
VMs Managed / Day
Example Use Cases

**Regeneron**  
Correlate 500K patient records with DNA to design therapies

**FINRA**  
Identify fraud using machine learning on 30 PB of trade data

**Shell**  
Large-scale simulations for inventory management
Our Product

Built around open source:

- Apache Spark
- Scala
- Delta Lake
- mlflow

Interactive data science
Scheduled jobs
SQL frontend
ML platform
Data catalog
Security policies

Compute Clusters

Cloud Storage

Data scientists -> Interactive data science
Data engineers -> Scheduled jobs
Business users -> SQL frontend

Customer’s Cloud Account

Databricks Service

Databricks Runtime

aws

Azure
Our Specific Challenges

All the usual challenges of SaaS:
- Availability, security, multitenancy, updates, etc

Plus, **the workloads themselves are large-scale!**
- One user job could easily overload control services
- Millions of VMs $\Rightarrow$ many weird failures
Four Lessons

1. What goes wrong in cloud systems?
2. Testing for scalability & stability
3. Developing control planes
4. Evolving big data systems for the cloud
Four Lessons

1. What goes wrong in cloud systems?
2. Testing for scalability & stability
3. Developing control planes
4. Evolving big data systems for the cloud
What Goes Wrong in the Cloud?

Academic research studies many kinds of failures:

- Software bugs, network config, crash failures, etc

These matter, but other problems often have larger impact:

- Scaling and resource limits
- Workload isolation
- Updates & regressions
Causes of Significant Outages

- Scaling problem in our services: 30%
- Scaling problem in underlying cloud services: 20%
- Insufficient user isolation: 20%
- Deployment misconfiguration: 10%
- Other: 20%
Causes of Significant Outages

- Scaling problem in our services: 30%
- Scaling problem in underlying cloud services: 20%
- Insufficient user isolation: 20%
- Deployment misconfiguration: 10%
- Other: 20%

70% scale related
Some Issues We Experienced

Cloud networks: limits, partitions, slow DHCP, hung connections

Automated apps creating large load

Very large requests, results, etc

Slow VM launches/shutdowns, lack of VM capacity

Data corruption writing to cloud storage
Example Outage: Aborted Jobs

Jobs Service launches & tracks jobs on clusters

1 customer running many jobs/sec on same cluster

Cloud’s network reaches a limit of 1000 connections/VM between Jobs Service & clusters
  - After this limit, new connections hang in state SYN_SENT

Resource usage from hanging connections causes memory pressure and GC

Health checks to some jobs time out, so we abort them
Surprisingly Rare Issues

1 cloud-wide VM restart on AWS (Xen patch)
1 misreported security scan on customer VM
1 significant S3 outage
1 kernel bug (hung TCP connections due to SACK fix)
Lessons

Cloud services must handle load that varies on many dimensions, and rely on other services with varying limits & failure modes.

Problems likely to get worse in a “cloud service economy”
Four Lessons

1. What goes wrong in cloud systems?
2. Testing for scalability & stability
3. Developing control planes
4. Evolving big data systems for the cloud
Testing for Scalability & Stability

Software correctness is a Boolean property: does your software give the right output on a given input?

Scalability and stability are a matter of degree

- **What load** will your system fail at? (any system with limited resources will)
- **What failure behavior** will you have? (crash all clients, drop some, etc)
Example Scalability Problems

Large result: can crash browser, notebook service, driver or Spark

Large record in file

Large # of tasks

Code that freezes a worker

+ All these affect other users!
Databricks Stress Test Infrastructure

1. Identify dimensions for a system to scale in (e.g. # of users, number of output rows, size of each output row, etc)

2. Grow load in each dimension until a failure occurs

3. Record failure type and impact on system
   - Error message, timeout, wrong result?
   - Are other clients affected?
   - Does the system auto-recover? How fast?

4. Compare over time and on changes
<table>
<thead>
<tr>
<th>Suite</th>
<th>Test</th>
<th>MaxValue</th>
<th>State</th>
<th>MaxStep</th>
<th>Flags</th>
<th>Message</th>
<th>MaxStep diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScalaClusterSuite</td>
<td>big broadcast</td>
<td>1000000000</td>
<td>FAILED</td>
<td>4</td>
<td>at sun.nio.ch.FileChannel</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>ScalaClusterSuite</td>
<td>big tasks</td>
<td>1000000000</td>
<td>FAILED</td>
<td>4</td>
<td>at java.io.ByteArrayInputStream</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>ScalaClusterSuite</td>
<td>caching large objects</td>
<td>1000000000</td>
<td>TIMED_OUT</td>
<td>3</td>
<td>java.lang.Exception</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>ScalaClusterSuite</td>
<td>caching small objects</td>
<td>1000000000</td>
<td>SUCCEEDED</td>
<td>4</td>
<td>failed</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>ScalaClusterSuite</td>
<td>crashing executors</td>
<td>1000000000</td>
<td>SUCCEEDED</td>
<td>1000</td>
<td>failed</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>ScalaClusterSuite</td>
<td>crashing tasks</td>
<td>1000000000</td>
<td>SUCCEEDED</td>
<td>4</td>
<td>failed</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>ScalaClusterSuite</td>
<td>display large rows</td>
<td>1000000000</td>
<td>TIMED_OUT</td>
<td>4</td>
<td>CBLS java.lang.Exception</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>ScalaClusterSuite</td>
<td>lots of shuffle tasks</td>
<td>1000000000</td>
<td>TIMED_OUT</td>
<td>3</td>
<td>failed</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>ScalaClusterSuite</td>
<td>lots of tasks</td>
<td>1000000000</td>
<td>TIMED_OUT</td>
<td>3</td>
<td>failed</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>ScalaClusterSuite</td>
<td>popular key in groupBy</td>
<td>1000000000</td>
<td>TIMED_OUT</td>
<td>4</td>
<td>failed</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>ScalaDriverSuite</td>
<td>allocate big arrays</td>
<td>1000000000</td>
<td>TIMED_OUT</td>
<td>3</td>
<td>failed</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>ScalaDriverSuite</td>
<td>allocate small arrays</td>
<td>1000000000</td>
<td>FAILED</td>
<td>3</td>
<td>at NotebookS$anonfun</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>ScalaDriverSuite</td>
<td>infinite loop</td>
<td>1000000000</td>
<td>TIMED_OUT</td>
<td>1</td>
<td>CBLS java.lang.Exception</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>ScalaDriverSuite</td>
<td>no such method error</td>
<td>1000000000</td>
<td>SUCCEEDED</td>
<td>4</td>
<td>failed</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>ScalaDriverSuite</td>
<td>print a lot</td>
<td>1000000000</td>
<td>TIMED_OUT</td>
<td>4</td>
<td>CBLS java.lang.Exception</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>ScalaDriverSuite</td>
<td>system exit</td>
<td>1000000000</td>
<td>FAILED</td>
<td>1</td>
<td>at com.databricks.scala</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>ScalaDriverSuite</td>
<td>thread sleep</td>
<td>1000000000</td>
<td>TIMED_OUT</td>
<td>2</td>
<td>failed</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>SQLClusterSuite</td>
<td>broadcast join</td>
<td>1000000000</td>
<td>FAILED</td>
<td>4</td>
<td>at com.databricks.scala</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>SQLClusterSuite</td>
<td>broadcast join on cached data</td>
<td>1000000000</td>
<td>SUCCEEDED</td>
<td>4</td>
<td>at com.databricks.scala</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>SQLClusterSuite</td>
<td>count distinct</td>
<td>1000000000</td>
<td>TIMED_OUT</td>
<td>4</td>
<td>failed</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>SQLClusterSuite</td>
<td>count distinct with common keys</td>
<td>1000000000</td>
<td>FAILED</td>
<td>2</td>
<td>at com.databricks.scala</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>SQLClusterSuite</td>
<td>self join</td>
<td>1000000000</td>
<td>FAILED</td>
<td>2</td>
<td>at com.databricks.scala</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>SQLClusterSuite</td>
<td>self join on cached data</td>
<td>1000000000</td>
<td>TIMED_OUT</td>
<td>4</td>
<td>failed</td>
<td>1000000000</td>
<td>0</td>
</tr>
<tr>
<td>SQLClusterSuite</td>
<td>self join with common keys</td>
<td>1000000000</td>
<td>TIMED_OUT</td>
<td>2</td>
<td>failed</td>
<td>1000000000</td>
<td>0</td>
</tr>
</tbody>
</table>
Four Lessons

1. What goes wrong in cloud systems?
2. Testing for scalability & stability
3. Developing control planes
4. Evolving big data systems for the cloud
Developing Control Planes

Cloud software consists of interacting, independently updated services, many of which call other services

What is the right programming model for this software?
Examples

Cluster manager service:
- API: requests to launch, scale and shut down clusters
- Behavior: request VMs, set up clusters, reuse VMs in pools
- State: requests, running VMs, etc

Jobs service:
- API: scheduled or API-triggered jobs to execute
- Behavior: acquire a cluster, run job, monitor state, retry
- State: jobs to be run, what’s currently active, where is it, etc
Examples

Cluster manager service:
- API: requests to launch, scale and shut down clusters
- Behavior: request VMs, set up clusters, reuse VMs in pools
- State: requests, running VMs, etc

Jobs service:
- API: scheduled or API-triggered jobs to execute
- Behavior: acquire a cluster, run job, monitor state, retry
- State: jobs to be run, what’s currently active, where is it, etc
Control Plane Infrastructure

We designed a shared service framework that handles:

- Deployment: AWS, Azure, local, special environments
- Storage: databases, schema updates, etc
- Security tokens & roles
- Monitoring
- API routing & limiting
- Feature flagging

Our service stack:

- Kubernetes
- Scala
- JSONnet
- Prometheus
- envoy
- databricks
Best Practices

**Isolate state:** relational DB is usually enough with org sharding

**Isolate components that scale differently:** allows separate scaling

**Manage changes through feature flags:** fastest, safest way

**Watch key metrics:** most outages could be predicted from one of CPU load, memory load, DB load or thread pool exhaustion

**Test pyramid:** 70% unit tests, 20% integration, 10% end-to-end
Example: Cluster Manager

Cluster manager v1

Cluster Manager

Cloud VM API

Customer Clusters

Cluster Manager v2

CM Master

Delegate

Delegate

Usage, billing, etc

VM launch, setup, monitoring, etc

Cloud VM API

Customer Clusters
Four Lessons

1. What goes wrong in cloud systems?
2. Testing for scalability & stability
3. Developing control planes
4. Evolving big data systems for the cloud
Evolving Big Data Systems for the Cloud

MapReduce, Spark, etc were designed for on-premise datacenters

How can we evolve these to leverage the benefits of the cloud?

- Availability, elasticity, scale, multitenancy, etc

Two examples from Databricks:

- Delta Lake: ACID on cloud object stores
- Cloudifying Apache Spark
Delta Lake Motivation

Cloud object stores (S3, Azure blob, etc) are the largest storage systems on the planet

- Unmatched availability, parallel I/O bandwidth, and cost-efficiency

Open source big data stack was designed for on-prem world

- Filesystem API for storage
- RDBMS for table metadata (Hive metastore)
- Other distributed systems, e.g. ZooKeeper

How can big data systems fully leverage cloud object stores?
Example: Atomic Parallel Writes

Spark on HDFS

Input Files

Output Partitions

Atomic Rename

/tmp-job-1

Spark on S3 (Naïve)

Input Files

Output Partitions

Full object names (no cheap rename)

/my-output/part-1

/my-output/part-2

/my-output/part-3

/my-output/part-4

/_DONE

Real cases are harder (e.g. appending to a table)
Delta Lake Design

1. Track metadata that says *which* objects are part of a dataset

2. Store this metadata itself in a cloud object store
   - Write-ahead log in S3, compressed using Apache Parquet

*Before Delta Lake:* 50% of Spark support issues were about cloud storage

*After:* fewer issues, increased perf

Input Files

```
Input Files
```

Output Partitions

```
/my-output/part-X
/my-output/part-Y
/my-output/part-Z
/my-output/part-W
/my-output/_delta_log
```

https://delta.io
Major Benefits of Delta Lake

Once we had transactions over S3, we could build much more:

- UPSERT, DELETE, etc (GDPR)
- Caching
- Multidimensional indexing
- Audit logging
- Time travel
- Background optimization

Result: greatly simplified customers’ data architectures
Other Cloud Features

Scheduler-integrated autoscaling for Apache Spark

Autoscaling local storage volumes

User isolation for high-concurrency Spark clusters
- Serverless experience for users inside an org
- Separate library envs, IAM roles, performance & fault isolation
Conclusion

The cloud is eating software by enabling much better products
  - Self-managing, elastic, more reliable & scalable

But building cloud products is understudied and hard

Many opportunities, from service fabrics to cloud-native systems