

unravel™

APACHE  
Spark™



# Optimizing Spark Job Performance

Parts 1 - 3: TROUBLESHOOTING SPARK



# Troubleshooting Spark Challenges

**Part 1:** TOP TEN SPARK DIFFICULTIES

“The most difficult thing is finding out why your job is failing, which parameters to change. Most of the time, it’s OOM errors...”

– Jagat Singh, Quora

Spark has become one of the most important tools for processing data – especially non-relational data – and deriving value from it. And Spark serves as a platform for the creation and delivery of analytics, AI, and machine learning applications, among others. But troubleshooting Spark applications is hard – and we’re here to help.

In this guide, we’ll describe ten challenges that arise frequently in troubleshooting Spark applications. We’ll start with issues at the job level, encountered by most people on the data team – operations people/administrators, data engineers, and data scientists, as well as analysts. Then, we’ll look at problems that apply across a cluster. These problems are usually handled by operations people/administrators and data engineers.

For more on Spark and its use, please see [this piece in Infoworld](#). And for more depth about the problems that arise in creating and running Spark jobs, at both the job level and the cluster level, please see the links below. There is also a good [introductory guide](#).

## Five Reasons Why Troubleshooting Spark Applications Is Hard

Some of the things that make Spark great also make it hard to troubleshoot. Here are some key Spark features, and some of the issues that arise in relation to them:

**1. Memory-resident.** Spark gets much of its speed and power by using memory, rather than disk, for interim storage of source data and results. However, this can cost a lot of resources and money, which is especially visible in the cloud. It can also make it easy for jobs to crash due to lack of sufficient available memory. And it makes problems hard to diagnose – only traces written to disk survive after crashes.

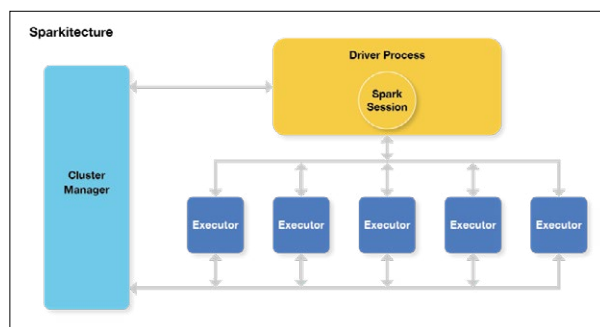
**2. Parallel processing.** Spark takes your job and applies it, in parallel, to all the data partitions assigned to your job. (You specify the data partitions, another tough and important decision.) But when a processing workstream runs into trouble, it can be hard to find and understand the problem among the multiple workstreams running at once.

**3. Variants.** Spark is open source, so it can be tweaked and revised in innumerable ways. There are major differences among the Spark 1 series, Spark 2.x, and the newer Spark 3.

And Spark works somewhat differently across platforms – on-premises; on cloud-specific platforms such as AWS EMR, Azure HDInsight, and Google Dataproc; and on Databricks, which is available across the major public clouds. Each variant offers some of its own challenges, and a somewhat different set of tools for solving them.

**4. Configuration options.** Spark has hundreds of configuration options. And Spark interacts with the hardware and software environment it’s running in, each component of which has its own configuration options. Getting one or two critical settings right is hard; when several related settings have to be correct, guesswork becomes the norm, and over-allocation of resources, especially memory and CPUs (see below) becomes the safe strategy.

**5. Trial and error approach.** With so many configuration options, how to optimize? Well, if a job currently takes six hours, you can change one, or a few, options, and run it again. That takes six hours, plus or minus. Repeat this three or four times, and it’s the end of the week. You may have improved the configuration, but you probably won’t have exhausted the possibilities as to what the best settings are.



**The Spark application is the Driver Process, and the job is split up across executors.** (Source: [Apache Spark for the Impatient on DZone](#).)

## Three Issues with Spark Jobs, on-Premises and in the Cloud

Spark jobs can require troubleshooting against three main kinds of issues:

- **Failure.** Spark jobs can simply fail. Sometimes a job will fail on one try, then work again after a restart. Just finding out that the job failed can be hard; finding out why can be harder. (Since the job is memory-resident, failure makes the evidence disappear.)
- **Poor performance.** A Spark job can run slower than you would like it to; slower than an external service level agreement (SLA); or slower than it would do if it were optimized. It’s very hard to know how long a job “should” take, or where to start in optimizing a job or a cluster.

- **Excessive cost or resource use.** The resource use or, especially in the cloud, the hard dollar cost of a job may raise concern. As with performance, it's hard to know how much the resource use and cost "should" be, until you put work into optimizing and see where you've gotten to.

All of the issues and challenges described here apply to Spark across all platforms, whether it's running on-premises, in Amazon EMR, or on Databricks (across AWS, Azure, or GCP). However, there are a few subtle differences:

- **Move to cloud.** There is a big movement of big data workloads from on-premises (largely running Spark on Hadoop) to the cloud (largely running Spark on Amazon EMR or Databricks). Moving to cloud provides greater flexibility and faster time to market, as well as access to built-in services found on each platform.
- **Move to on-premises.** There is a small movement of workloads from the cloud back to on-premises environments. When a cloud workload "settles down," such that flexibility is less important, then it may become significantly cheaper to run it on-premises instead.
- **On-premises concerns.** Resources (and costs) on-premises tend to be relatively fixed; there can be a leadtime of months to years to significantly expand on-premises resources. So the main concern on-premises is maximizing the existing estate: making more jobs run in existing resources, and getting jobs to complete reliably and on-time, to maximize the pay-off from the existing estate.
- **Cloud concerns.** Resources in the cloud are flexible and "pay as you go" – but as you go, you pay. So the main concern in the cloud is managing costs. (As [AWS puts it](#), "When running big data pipelines on the cloud, operational cost optimization is the name of the game.") This concern increases because reliability concerns in the cloud can often be addressed by "throwing hardware at the problem" – increasing reliability, but at greater cost.
- **On-premises Spark vs Amazon EMR.** When moving to Amazon EMR, it's easy to do a "lift and shift" from on-premises Spark to EMR. This saves time and money on the cloud migration effort, but any inefficiencies in the on-premises environment are reproduced in the cloud, increasing costs. It's also fully possible to refactor before moving to EMR, just as with Databricks.
- **On-premises Spark vs Databricks.** When moving to Databricks, most companies take advantage of Databricks' capabilities, such as ease of starting/shutting down clusters, and do at least some refactoring as part of the cloud migration effort. This costs time and money in the cloud migration effort, but results in lower costs and, potentially, greater reliability for the refactored job in the cloud.

All of these concerns are accompanied by a distinct lack of needed information. Companies often make crucial decisions – on-premises vs. cloud, EMR vs. Databricks, "lift and shift" vs. refactoring – with only guesses available as to what different options will cost in time, resources, and money.

## Ten Spark Challenges

Many Spark challenges relate to configuration, including the number of executors to assign, memory usage (at the driver level, and per executor), and what kind of hardware/machine instances to use. You make configuration choices per job, and also for the overall cluster in which jobs run, and these are interdependent – so things get complicated, fast.

Some challenges occur at the job level; these challenges are shared right across the data team. They include:

1. How many executors should each job use?
2. How much memory should I allocate for each job?
3. How do I find and eliminate data skew?
4. How do I make my pipelines work better?
5. How do I know if a specific job is optimized?

Other challenges come up at the cluster level, or even at the stack level, as you decide what jobs to run on what clusters. These problems tend to be the remit of operations people and data engineers. They include:

6. How do I size my nodes, and match them to the right servers/instance types?
7. How do I see what's going on across the Spark stack and apps?
8. Is my data partitioned correctly for my SQL queries?
9. When do I take advantage of auto-scaling?
10. How do I get insights into jobs that have problems?

For easy access, these challenges are listed below, linked to the appropriate page in this guide:

### Job-Level Challenges

1. [Executor and core allocation](#)
2. [Memory allocation](#)
3. [Data skew/small files](#)
4. [Pipeline optimization](#)
5. [Finding out whether a job is optimized](#)

### Cluster-Level Challenges

6. [Resource allocation](#)
7. [Observability](#)
8. [Data partitioning vs. SQL queries/inefficiency](#)
9. [Use of auto-scaling](#)
10. [Troubleshooting](#)

**Impacts:** Resources for a given job (at the cluster level) or across clusters tend to be significantly under-allocated (causes crashes, hurting business results) or over-allocated (wastes resources and can cause other jobs to crash, both of which hurt business results).



## Section 1: Five Job-Level Challenges

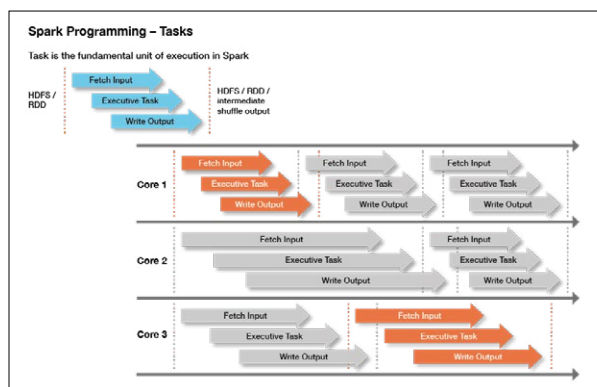
These challenges occur at the level of individual jobs. Fixing them can be the responsibility of the developer or data scientist who created the job, or of operations people or data engineers who work on both individual jobs and at the cluster level.

However, job-level challenges, taken together, have massive implications for clusters, and for the entire data estate. One of our Unravel Data customers has undertaken a right-sizing program for resource-intensive jobs that has clawed back nearly half the space in their clusters, even though data processing volume and jobs in production have been increasing.

For these challenges, we'll assume that the cluster your job is running in is relatively well-designed (see next section); that other jobs in the cluster are not resource hogs that will knock your job out of the running; and that you have the tools you need to troubleshoot individual jobs.

### 1. HOW MANY EXECUTORS AND CORES SHOULD A JOB USE?

One of the key advantages of Spark is parallelization – you run your job's code against different data partitions in parallel workstreams, as in the diagram below. The number of workstreams that run at once is the number of executors, times the number of cores per executor. So how many executors should your job use, and how many cores per executor – that is, how many workstreams do you want running at once?



**A Spark job using three cores to parallelize output. Up to three tasks run simultaneously, and seven tasks are completed in a fixed period of time. (Source: Lisa Hua, [Spark Overview](#), Slideshare.)**

You want high usage of cores, high usage of memory per core, and data partitioning appropriate to the job. (Usually, partitioning on the field or fields you're querying on.) This [beginner's guide](#) for Hadoop suggests two-three cores per executor, but not more than five; this [expert's guide](#) to Spark tuning on AWS suggests that you use three executors per node, with five cores per executor, as your starting point for all jobs.

You are likely to have your own sensible starting point for your on-premises or cloud platform, the servers or instances available, and experience your team has had with similar workloads. Once your job runs successfully a few times, you can either leave it alone, or optimize it. We recommend that you optimize it, because optimization:

- Helps you save resources and money (not over-allocating)
- Helps prevent crashes, because you right-size the resources (not under-allocating)
- Helps you fix crashes fast, because allocations are roughly correct, and because you understand the job better

### 2. HOW MUCH MEMORY SHOULD I ALLOCATE FOR EACH JOB?

Memory allocation is per executor, and the most you can allocate is the total available in the node. If you're in the cloud, this is governed by your instance type; on-premises, by your physical server or virtual machine. Some memory is needed for your cluster manager and system resources (16GB may be a typical amount), and the rest is available for jobs.

If you have three executors in a 128GB cluster, and 16GB is taken up by the cluster, that leaves 37GB per executor. However, a few GB will be required for executor overhead; the remainder is your per-executor memory. You will want to partition your data so it can be processed efficiently in the available memory.

This is just a starting point, however. You may need to be using a different instance type, or a different number of executors, to make the most efficient use of your node's resources against the job you're running. As with the number of executors (see previous section), optimizing your job will help you know whether you are over- or under-allocating memory, reduce the likelihood of crashes, and get you ready for troubleshooting when the need arises.

For more on memory management, see this widely read article, [Spark Memory Management](#), by our own Rishitesh Mishra.

### 3. HOW DO I HANDLE DATA SKEW AND SMALL FILES?

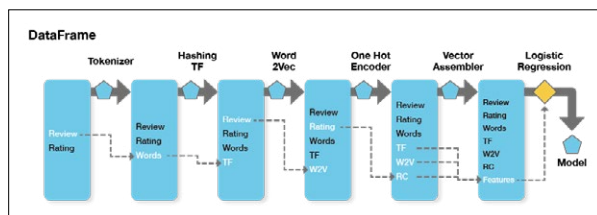
Data skew and small files are complementary problems. Data skew tends to describe large files – where one key value, or a few, have a large share of the total data associated with them. This can force Spark, as it's processing the data, to move data around in the cluster, which can slow down your task, cause low utilization of CPU capacity, and cause out-of-memory errors which abort your job. Several techniques for handling very large files which appear as a result of data skew are given in the popular article, [Data Skew and Garbage Collection](#), by Rishitesh Mishra of Unravel.

Small files are partly the other end of data skew – a share of partitions will tend to be small. And Spark, since it is a parallel processing system, may generate many small files from parallel processes. Also, some processes you use, such as file compression, may cause a large number of small files to appear, causing inefficiencies. You may need to reduce parallelism (undercutting one of the advantages of Spark), repartition (an expensive operation you should minimize), or start adjusting your parameters, your data, or both ([see details](#)).

Both data skew and small files incur a meta-problem that's common across Spark – when a job slows down or crashes, how do you know what the problem was? We will mention this again, but it can be particularly difficult to know this for data-related problems, as an otherwise well-constructed job can have seemingly random slowdowns or halts, caused by hard-to-predict and hard-to-detect inconsistencies across different data sets.

#### 4. HOW DO I OPTIMIZE AT THE PIPELINE LEVEL?

Spark pipelines are made up of dataframes, connected by transformers (which calculate new data from existing data), and Estimators. Pipelines are widely used for all sorts of processing, including extract, transform, and load (ETL) jobs and machine learning. Spark makes it easy to combine jobs into pipelines, but it does not make it easy to monitor and manage jobs at the pipeline level. So it's easy for monitoring, managing, and optimizing pipelines to appear as an exponentially more difficult version of optimizing individual Spark jobs.



**Existing Transformers create new Dataframes, with an Estimator producing the final model. (Source: [Spark Pipelines: Elegant Yet Powerful](#), InsightDataScience.)**

Many pipeline components are “tried and trusted” individually, and are thereby less likely to cause problems than new components you create yourself. However, interactions between pipeline steps can cause novel problems.

Just as job issues roll up to the cluster level, they also roll up to the pipeline level. Pipelines are increasingly the unit of work for DataOps, but it takes truly deep knowledge of your jobs and your cluster(s) for you to work effectively at the pipeline level. [This article](#), which tackles the issues involved in some depth, describes pipeline debugging as an “art.”

#### 5. HOW DO I KNOW IF A SPECIFIC JOB IS OPTIMIZED?

Neither Spark nor, for that matter, SQL are designed for ease of optimization. Spark comes with a monitoring and management interface, Spark UI, which can help. But Spark UI can be challenging to use, especially for the types of comparisons – over time, across jobs, and across a large, busy cluster – that you need to really optimize a job. And there is no “SQL UI” that specifically tells you how to optimize your SQL queries.

There are some general rules. For instance, a “bad” – inefficient – join can take hours. But it's very hard to find where your app is spending its time, let alone whether a specific SQL command is taking a long time, and whether it can indeed be optimized.

[Spark's Catalyst optimizer](#) does its best to optimize your queries for you. But when data sizes grow large enough, and processing gets complex enough, you have to help it along if you want your resource usage, costs, and runtimes to stay on the acceptable side.

### Section 2: Cluster-Level Challenges

Cluster-level challenges are those that arise for a cluster that runs many (perhaps hundreds or thousands) of jobs, in cluster design (how to get the most out of a specific cluster), cluster distribution (how to create a set of clusters that best meets your needs), and allocation across on-premises resources and one or more public, private, or hybrid cloud resources.

The first step toward meeting cluster-level challenges is to meet job-level challenges effectively, as described above. A cluster that's running unoptimized, poorly understood, slowdown-prone and crash-prone jobs is impossible to optimize. But if your jobs are right-sized, cluster-level challenges become much easier to meet. (Note that [Unravel Data](#), as mentioned in the previous section, helps you find your resource-heavy Spark jobs, and optimize those first. It also does much of the work of troubleshooting and optimization for you.)

Meeting cluster-level challenges for Spark may be a topic better suited for a graduate-level computer science seminar than for this guide, but here are some of the issues that come up, and a few comments on each:

#### 6. ARE NODES MATCHED UP TO SERVERS OR CLOUD INSTANCES?

A Spark node – a physical server or a cloud instance – will have an allocation of CPUs and physical memory. (The whole point of Spark is to run things in actual memory, so this is crucial.) You have to fit your executors and memory allocations into nodes that are carefully matched to existing resources, on-premises or in the cloud. (You can allocate more or fewer Spark cores than there are available CPUs, but matching them makes things more predictable, uses resources better, and may make troubleshooting easier.)

On-premises, poor matching between nodes, physical servers, executors and memory results in inefficiencies, but these may not be very visible; as long as the total physical resource is sufficient for the jobs running, there's no obvious problem. However, issues like this can cause datacenters to be very poorly utilized, meaning there's big overspending going on – it's just not noticed. (Ironically, the impending prospect of cloud migration may cause an organization to freeze on-premises spending, shining a spotlight on costs and efficiency.)

In the cloud, “pay as you go” pricing shines a different type of spotlight on efficient use of resources – inefficiency shows up in each month's bill. You need to match nodes, cloud instances, and job CPU and memory allocations very closely indeed, or incur what might amount to massive overspending. [This article](#) gives you some guidelines for running Apache Spark cost-effectively on AWS EC2 instances, and is worth a read even if you're running on-premises, or on a different cloud provider.

You still have big problems here. In the cloud, with costs both visible and variable, cost allocation is a big issue. It's hard to know who's spending what, let alone what the business results that go with each unit of spending are. But tuning workloads against server resources and/or instances is the first step in gaining control of your spending, across all your data estates.

## 7. HOW DO I SEE WHAT'S GOING ON IN MY CLUSTER?

“Spark is notoriously difficult to tune and maintain,” according to [an article in The New Stack](#). Clusters need to be “expertly managed” to perform well, or all the good characteristics of Spark can come crashing down in a heap of frustration and high costs. (In people's time and in business losses, as well as direct, hard dollar costs.)

Key Spark advantages include accessibility to a wide range of users and the ability to run in memory. But the most popular tool for Spark monitoring and management, Spark UI, doesn't really help much at the cluster level. You can't, for instance, easily tell which jobs consume the most resources over time. So it's hard to know where to focus your optimization efforts. And Spark UI doesn't support more advanced functionality – such as comparing the current job run to previous runs, issuing warnings, or making recommendations, for example.

Logs on cloud clusters are lost when a cluster is terminated, so problems that occur in short-running clusters can be that much harder to debug. More generally, managing log files is itself a big data management and data accessibility issue, making debugging and governance harder. This occurs in both on-premises and cloud environments. And, when workloads are moved to the cloud, you no longer have a fixed-cost data estate, nor the “tribal knowledge” accrued from years of running a gradually changing set of workloads on-premises. Instead, you have new technologies and pay-as-you-go billing. So cluster-level management, hard as it is, becomes critical.

## 8. IS MY DATA PARTITIONED CORRECTLY FOR MY SQL QUERIES? (AND OTHER INEFFICIENCIES)

Operators can get quite upset, and rightly so, over “bad” or “rogue” queries that can cost way more, in resources or cost, than they need to. One colleague describes a team he worked on that went through more than \$100,000 of cloud costs in a weekend of crash-testing a new application – a discovery made after the fact. (But before the job was put into production, where it would have really run up some bills.)

SQL is not designed to tell you how much a query is likely to cost, and more elegant-looking SQL queries (i.e., fewer statements) may well be more expensive. The same is true of all kinds of code you have running. So you have to do some or all of three things:

- Learn something about SQL, and about coding languages you use, especially how they work at runtime
- Understand how to optimize your code and partition your data for good price/performance
- Experiment with your app to understand where the resource use/cost “hot spots” are, and reduce them where possible

All this fits in the “optimize” recommendations from 1. and 2. above. We'll talk more about how to carry out optimization in Part 2 of this guide.

## 9. WHEN DO I TAKE ADVANTAGE OF AUTO-SCALING?

The ability to auto-scale – to assign resources to a job just while it's running, or to increase resources smoothly to meet processing peaks – is one of the most enticing features of the cloud. It's also one of the most dangerous; there is no practical limit to how much you can spend. You need some form of guardrails, and some form of alerting, to remove the risk of truly gigantic bills.

The need for auto-scaling might, for instance, determine whether you move a given workload to the cloud, or leave it running, unchanged, in your on-premises data center. But to help an application benefit from auto-scaling, you have to profile it, then cause resources to be allocated and de-allocated to match the peaks and valleys. And you have some calculations to make, because cloud providers charge you more for spot resources – those you grab and let go of, as needed – than for persistent resources that you keep running for a long time. Spot resources may cost two or three times as much as dedicated ones.

The first step, as you might have guessed, is to optimize your application, as in the previous sections. Auto-scaling is a price/performance optimization, and a potentially resource-intensive one. You should do other optimizations first.

Then profile your optimized application. You need to calculate ongoing and peak memory and processor usage, figure out

how long you need each, and the resource needs and cost for each state. And then decide whether it's worth auto-scaling the job, whenever it runs, and how to do that. You may also need to find quiet times on a cluster to run some jobs, so the job's peaks don't overwhelm the cluster's resources.

To help, Databricks has two types of clusters, and the second type works well with auto-scaling. Most jobs start out in an interactive cluster, which is like an on-premises cluster; multiple people use a set of shared resources. It is, by definition, very difficult to avoid seriously underusing the capacity of an interactive cluster.

So you are meant to move each of your repeated, resource-intensive, and well-understood jobs off to its own, dedicated, job-specific cluster. A job-specific cluster spins up, runs its job, and spins down. This is a form of auto-scaling already, and you can also scale the cluster's resources to match job peaks, if appropriate. But note that you want your application profiled and optimized before moving it to a job-specific cluster.

## 10. HOW DO I FIND AND FIX PROBLEMS?

Just as it's hard to fix an individual Spark job, there's no easy way to know where to look for problems across a Spark cluster. And once you do find a problem, there's very little guidance on how to fix it. Is the problem with the job itself, or the environment it's running in? For instance, over-allocating memory or CPUs for some Spark jobs can starve others. In the cloud, the noisy neighbors problem can slow down a Spark job run to the extent that it causes business problems on one outing – but leaves the same job to finish in good time on the next run.

The better you handle the other challenges listed in this guide, the fewer problems you'll have, but it's still very hard to know how to most productively spend Spark operations time. For instance, a slow Spark job on one run may be worth fixing in its own right, and may be warning you of crashes on future runs. But it's very hard just to see what the trend is for a Spark job in performance, let alone to get some idea of what the job is accomplishing vs. its resource use and average time to complete. So Spark troubleshooting ends up being reactive, with all too many furry, blind little heads popping up for operators to play Whack-a-Mole with.

## Impacts of These Challenges

If you meet the above challenges effectively, you'll use your resources efficiently and cost-effectively. However, our observation here at Unravel Data is that most Spark clusters are not run efficiently.

What we tend to see most are the following problems – at a job level, within a cluster, or across all clusters:

- **Under-allocation.** It can be tricky to allocate your resources efficiently on your cluster, partition your datasets effectively, and determine the right level of resources for each job. If you under-allocate (either for a job's driver or the executors), a job is likely to run too slowly, or to crash. As a result, many developers and operators resort to...
- **Over-allocation.** If you assign too many resources to your job, you're wasting resources (on-premises) or money (cloud). We hear about jobs that need, for example, 2GB of memory, but are allocated much more – in one case, 85GB.

Applications can run slowly, because they're under-allocated – or because some apps are over-allocated, causing others to run slowly. Data teams then spend much of their time fire-fighting issues that may come and go, depending on the particular combination of jobs running that day. With every level of resource in shortage, new, business-critical apps are held up, so the cash needed to invest against these problems doesn't show up. IT becomes an organizational headache, rather than a source of business capability.

## Conclusion

To jump ahead to the end of this series a bit, our customers here at Unravel are easily able to spot and fix over-allocation and inefficiencies. They can then monitor their jobs in production, finding and fixing issues as they arise. Developers even get on board, checking their jobs before moving them to production, then teaming up with Operations to keep them tuned and humming.

One Unravel customer, [Mastercard](#), has been able to reduce usage of their clusters by roughly half, even as data sizes and application density has moved steadily upward during the global pandemic. And everyone gets along better, and has more fun at work, while achieving these previously unimagined results.

So, whether you choose to use Unravel or not, develop a culture of right-sizing and efficiency in your work with Spark. It will seem to be a hassle at first, but your team will become much stronger, and you'll enjoy your work life more, as a result.

You need a sort of X-ray of your Spark jobs, better cluster-level monitoring, environment information, and to correlate all of these sources into recommendations. In [Troubleshooting Spark Applications, Part 2: Solutions](#), we will describe the most widely used tools for Spark troubleshooting – including the Spark Web UI and our own offering, Unravel Data – and how to assemble and correlate the information you need. If you would like to know more about Unravel Data now, you can [download a free trial](#) or [contact Unravel](#).



# Troubleshooting Spark Applications

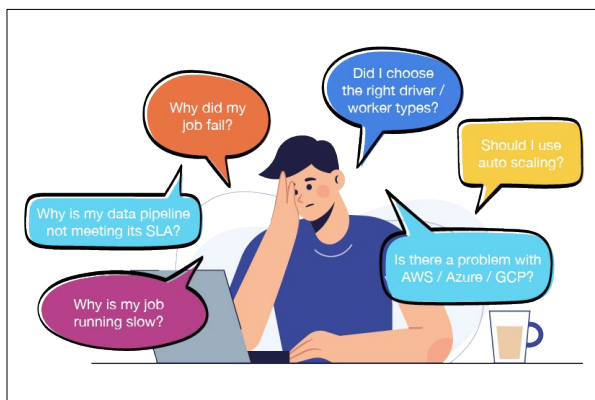
**Note:** This guide applies to running Spark jobs on any platform, including Cloudera platforms; cloud vendor-specific platforms – Amazon EMR, Microsoft HDInsight, Microsoft Synapse, Google DataProc; Databricks, which is on all three major public cloud providers; and Apache Spark on Kubernetes, which runs on nearly all platforms, including on-premises and cloud.

## Introduction

Spark is known for being extremely difficult to debug. But this is not all Spark's fault. Problems in running a Spark job can be the result of problems with the infrastructure Spark is running on, inappropriate configuration of Spark, Spark issues, the currently running Spark job, other Spark jobs running at the same time - or interactions among these layers. But Spark jobs are very important to the success of the business; when a job crashes, or runs slowly, or contributes to a big increase in the bill from your cloud provider, you have no choice but to fix the problem.

Widely used tools generally focus on part of the environment – the Spark job, infrastructure, the network layer, etc. These tools don't present a holistic view. But that's just what you need to truly solve problems. (You also need the holistic view when you're creating the Spark job, and as a check before you start running it, to help you avoid having problems in the first place. But that's another story.)

In this guide, Part 2 in a series, we'll show ten major tools that people use for Spark troubleshooting. We'll show what they do well, and where they fall short. In Part 3, the final piece, we'll introduce Unravel Data, which makes solving many of these problems easier.



Life as a Spark developer

## What's the Problem(s)?

The problems we mentioned in Part 1 of this series have many potential solutions. The methods people usually use to try to solve them often derive from that person's role on the data team. The person who gets called when a Spark job crashes, such as the job's developer, is likely to look at the Spark job. The person who is responsible for making sure the cluster is healthy will look at that level. And so on.

The following chart, from Part 1, shows the most common job-level and cluster-level challenges that data teams face in successfully running Spark jobs.

Job-Level Challenges	Cluster-Level Challenges
1. Executor and core allocation	6. Resource allocation
2. Memory allocation	7. Observability
3. Data skew/small files	8. Data partitioning vs. SQL queries/inefficiency
4. Pipeline optimization	9. Use of auto-scaling
5. Finding out whether a job is optimized	10. Troubleshooting
<b>Impacts:</b> Resources for a given job (at the cluster level) or across clusters tend to be significantly under-allocated (causes crashes, hurting business results) or over-allocated (wastes resources and can cause other jobs to crash, both of which hurt business results).	

In this guide, we highlight **five types of solutions** that people use – often in various combination – to solve problems with Spark jobs

### 1. Spark UI

### 2. Spark logs

### 3. Platform-level tools such as Cloudera Manager, the Amazon EMR UI, Cloudwatch, the Databricks UI, and Ganglia

### 4. APM tools such as Cisco AppDynamics, Datadog, and Dynatrace

### 5. DataOps platforms such as Unravel Data

As an example of solving problems of this type, let's look at the problem of an application that's running too slowly – a very common Spark problem, that may be caused by one or more of the issues listed in the chart. Here, we'll look at how existing tools might be used to try to solve it.

**Note:** Many of the observations and images in this guide originated in the July 2021 presentation, [Beyond Observability: Accelerate Performance on Databricks](#), by Patrick Mawyer, Systems Engineer at Unravel Data. We recommend this webinar to anyone interested in Spark troubleshooting and Spark performance management, whether on Databricks or on other platforms.

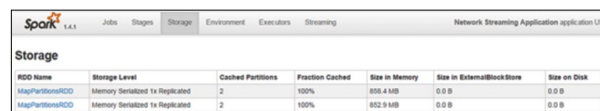


## Solving Problems Using Spark UI

Spark UI is the first tool most data team members use when there's a problem with a Spark job. It shows a snapshot of currently running jobs, the stages jobs are in, storage usage, and more. It does a good job, but is seen as having some faults. It can be hard to use, with a low signal-to-noise ratio and a long learning curve. It doesn't tell you things like which jobs are taking up more or less of a cluster's resources, nor deliver critical observations such as CPU, memory, and I/O usage.

In the case of a slow Spark application, Spark UI will show you what the current status of that job is. You can also use Spark UI for past jobs, if the logs for those jobs still exist, and if they were configured to log events properly. Also, the Spark history server tends to crash. When this is all working, it can help you find out how long an application took to run in the past – you need to do this kind of investigative work just to determine what “slow” is.

The following screenshot is for a Spark 1.4.1 job with a two-node cluster. It shows a Spark Streaming job that steadily uses more memory over time, which might cause the job to slow down. And the job eventually – over a matter of days – runs out of memory.



RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in ExternalBlockStore	Size on Disk
MapPartitionsRDD	Memory Serialized 1x Replicated	2	100%	855.4 MB	0.0 B	0.0 B
MapPartitionsRDD	Memory Serialized 1x Replicated	2	100%	852.9 MB	0.0 B	0.0 B

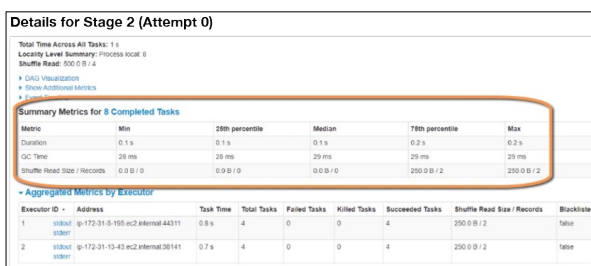
**A Spark streaming job that uses progressively more memory over time.** (Source: [Stack Overflow](#))

To solve this problem, you might do several things. Here's a brief list of possible solutions, and the problems they might cause elsewhere:

- **Increase available memory for each worker.** You can increase the value of the `spark.executor.memory` variable to increase the memory for each worker. This will not necessarily speed the job up, but will defer the eventual crash. However, you are either taking memory away from other jobs in your cluster or, if you're in the cloud, potentially running up the cost of the job.
- **Increase the storage fraction.** You can change the value of `spark.storage.memoryFraction`, which varies from 0 to 1, to a higher fraction. Since the Java virtual machine (JVM) uses memory for caching RDDs and for shuffle memory, you are increasing caching memory at the expense of shuffle memory. This will cause a different failure if, at some point, the job needs shuffle memory that you allocated away at this step.

- **Increase the parallelism of the job.** For a Spark Cassandra Connector job, for example, you can change `spark.cassandra.input.split.size` to a smaller value. (It's a different variable for other RDD types.) Increasing parallelism decreases the data set size for each worker, requiring less memory per worker. But more workers means more resources used; in a fixed resources environment, this takes resources away from other jobs; in a dynamic environment, such as Databricks job clusters, it directly runs up your bill.

The point here is that everything you might do has a certain amount of guesswork to it, because you don't have complete information. And, whichever approach you choose, you are putting the job in line for other, different problems, including later failure, failure for other reasons, or increased cost. Alternatively, your job may be fine, but at the expense of other jobs that then fail, and those failures will also be hard to troubleshoot.



Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.1 s	0.1 s	0.1 s	0.2 s	0.2 s
C/C Time	29 ms	29 ms	29 ms	29 ms	29 ms
Shuffle Read Size / Records	0.0 B / 0	0.0 B / 0	0.0 B / 0	200.0 B / 2	200.0 B / 2

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Shuffle Read Size / Records	Blocked
1	ip-172-31-1-155-ec2.internal/44311	0.9 s	4	0	0	4	200.0 B / 2	false
2	ip-172-31-13-42-ec2.internal/36141	0.7 s	4	0	0	4	200.0 B / 2	false

**Spark UI, showing metrics for completed tasks.**  
(Source: Unravel Data)

Here's a look at the Stages section of Spark UI. It gives you a list of metrics across executors. However, there's no overview or big picture view to help guide you in finding problems. And the tool doesn't make recommendations to help you solve problems, or avoid them in the first place.

Spark UI is limited to Spark – and Spark job for example may have data coming in from Kafka, and run alongside other tools. Each of those has its own monitoring and management tools, or does without; Spark UI doesn't work with them. It also lacks pro-active alerting, automatic actions, and AI-driven insights, all found in Unravel.

Spark UI is very useful for what it does, but its limitations – and the limitations of the other tool types described here – lead many organizations to build homegrown tools or toolsets, often built on Grafana. These solutions are resource-intensive, hard to extend, hard to support, and hard to keep up-to-date.

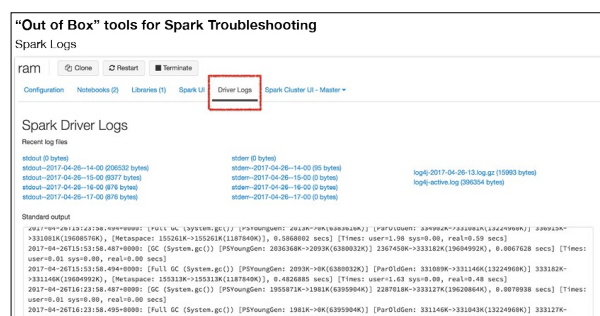
A few individuals and organizations even offer their homegrown tools as open source software for other organizations to use, but of course support, documentation, and updates are limited. Several such tools, such as Sparklint and DrElephant, do not support recent versions of Spark. At this writing, they have not had many, if any, fresh commits in recent months or even years.



## Spark Logs

Spark logs are the underlying resource for troubleshooting Spark jobs. Spark UI can even use Spark logs, if available, to rebuild a view of the Spark environment on an historical basis. You can use the logs related to the job's driver and executors to retrospectively find out what happened to a given job, and even some information about what was going on with other jobs at the time.

If you have a slow app, for instance, you can painstakingly assemble a picture to tell you if the slowness was in one task versus the other by scanning through multiple log files. But answering why and finding the root cause is hard. These logs don't have complete information about resource usage, data partitioning, correct configuration setting and many other factors than can affect the performance. There are also many potential issues that don't show up in Spark logs, such as "noisy neighbor" or networking issues that reduce resource availability within your Spark environment.



Source: Unravel Data

Spark logs are a tremendous resource, and are always a go-to for solving problems with Spark jobs. However, if you depend on logs as a major component of your troubleshooting toolkit, several problems come up, including:

- **Access and governance difficulties.** In highly secure environments, it can take time to get permission to access logs, or you may need to ask someone with the proper permissions to access the file for you. In some highly regulated companies, such as financial institutions, it can take hours per log to get access.
- **Multiple files.** You may need to look at the logs for a driver and several executors, for instance, to solve job-level problems. And your brain is the comparison and integration engine that pulls the information together, makes sense of it, and develops insights into possible causes and solutions.
- **Voluminous files.** The file for one component of a job can be very large, and all the files for all the components of a job can be huge - especially for long-running jobs. Again, you are the one who has to find and retain each part of the information needed, develop a complete picture of the problem, and try different solutions until one works.

- **Missing files.** Governance rules and data storage considerations take files away, as files are moved to archival media or simply lost to deletion. More than one large organization deletes files every 90 days, which makes quarterly summaries very difficult, and comparisons to, say, the previous year's holiday season or tax season impossible.
- **Only Spark-specific information.** Spark logs are just that - logs from Spark. They don't include much information about the infrastructure available, resource allocation, configuration specifics, etc. Yet this information is vital to solving a great many of the problems that hamper Spark jobs.

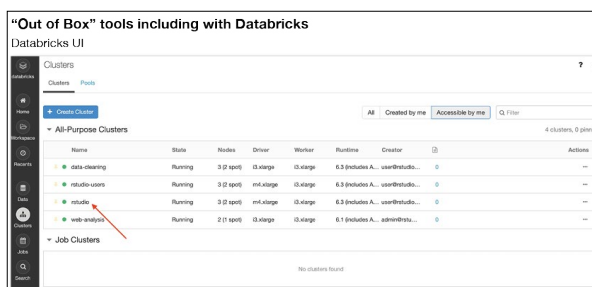
Because Spark logs don't cover infrastructure and related information, it's up to the operations person to find as much information they can on those other important areas, then try to integrate it all and determine the source of the problem. (Which may be the result of a complex interaction among different factors, with multiple changes needed to fix it.)

## Platform-Level Solutions

There are platform-level solutions that work on a given Spark platform, such as Cloudera Manager, the Amazon EMR UI, and Databricks UI. In general, these interfaces allow you to work at the cluster level. They tell you information about resource usage and the status of various services.

If you have a slow app, for example, these tools will give you part of the picture you need to put together to determine the actual problem, or problems. But these tools do not have the detail-level information in the tools above, nor do they even have all the environmental information you need. So again, it's up to you to decide how much time to spend researching, to pull all the information together, and to try to determine a solution. A quick fix might take a few hours; a comprehensive, long-term solution may take days of research and experimentation.

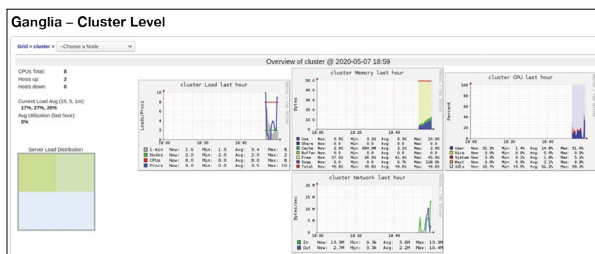
This screenshot shows Databricks UI. It gives you a solid overview of jobs and shows you status, cluster type, and so on. Like other platform-level solutions, it doesn't help you much with historical runs, nor in working at the pipeline level, across the multiple jobs that make up the pipeline.



Source: Unravel Data

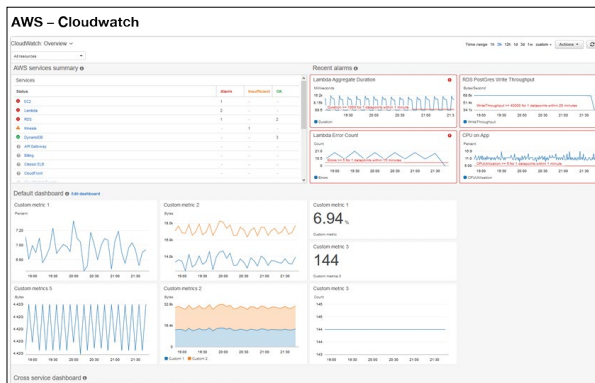
Another monitoring tool for Spark, which is included as open source within Databricks, is called Ganglia. It's largely complementary to Databricks UI, though it also works at the cluster and, in particular, at the node level. You can see host-level metrics such as CPU consumption, memory consumption, disk usage, network-level IO – all host-level factors that can affect the stability and performance of your job.

This can allow you to see if your nodes are configured appropriately, to institute manual scaling or auto-scaling, or to change instance types. (Though someone trying to fix a specific job is not inclined to take on issues that affect other jobs, other users, resource availability, and cost.) Ganglia does not have job-specific insights, nor work with pipelines. And there are no good output options; you might be reduced to taking a screen snapshot to get a JPEG or PNG image of the current status.



Source: Unravel Data

Support from the open-source community is starting to shift toward more modern observability platforms like Prometheus, which works well with Kubernetes. And cloud providers offer their own solutions – AWS Cloudwatch, for example, and Azure Log Monitoring and Analytics. These tools are all oriented toward web applications, not the backend apps used for analytics, AI, machine learning, and other use cases that are usually considered part of the big data world. They lack big data application and pipeline information which is essential to understand what's happening to your job and how your job is affecting things on the cluster or workspace.



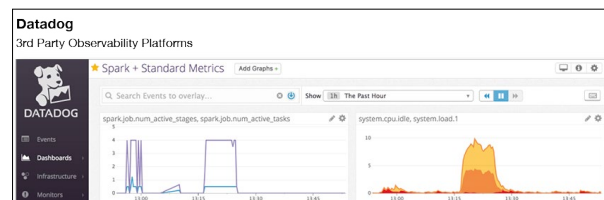
Source: Unravel Data

Platform-level solutions can be useful for solving the root causes of problems such as out-of-memory errors. However, they don't go down to the job level, leaving that to resources such as Spark logs and tools such as Spark UI. Therefore, to solve a problem, you are often using platform-level solutions in combination with job-level tools – and again, it's your brain that has to do the comparisons and data integration needed to get a complete picture and solve the problem.

Like job-level tools, these solutions are not comprehensive, nor integrated. They offer snapshots, but not history, and they don't make proactive recommendations. And, to solve a problem on Databricks, for example, you may be using Spark logs, Spark UI, Databricks UI, and Ganglia, along with Cloudwatch on AWS, or Azure Log Monitoring and Analytics. None of these tools integrate with the others.

## APM Tools

There is a wide range of monitoring tools, generally known as Application Performance Management (APM) tools. Many organizations have adopted one or more tools from this category, though they can be expensive, and provide very limited metrics on Spark and other modern data technologies. Leading tools in this category include Datadog, Dynatrace, and Cisco AppDynamics.



Source: Unravel Data

For a slow app, for instance, an APM tool might tell you if the system as a whole is busy, slowing your app, or if there were networking issues, slowing down all the jobs. While helpful, they're oriented toward monitoring and observability for Web applications and middleware, not data-intensive operations such as Spark jobs. They tend to lack information about pipelines, specific jobs, data usage, configuration setting, and much more, as they are not designed to deal with the complexity of modern data applications.

## DataOps Platforms

To sum up, there are several types of existing tools:

- **DIY with Spark logs.** Spark keeps a variety of logs, and you can parse them, in a do it yourself (DIY) fashion, to help solve problems. But this lacks critical infrastructure, container, and other metrics.
- **Open source tools.** Spark UI comes with Spark itself, and there are other Spark tools from the open source community. But these lack infrastructure, configuration and other information. They also do not help connect together a full pipeline view, as you need if you are using technologies such as Kafka to bring data in.
- **Platform-specific tools.** The platforms that Spark runs on - notably Cloudera platforms, Amazon EMR, and Databricks - each have platform-specific tools that help with Spark troubleshooting. But these lack application-level information and are best used for troubleshooting platform services.
- **Application performance monitoring (APM) tools.** APM tools monitor the interactions of applications with their environment, and can help with troubleshooting and even with preventing problems. But the applications these APM tools are built for are technologies such as .NET, Java, and Ruby, not technologies that work with data-intensive applications such as Spark.
- **DataOps platforms.** DataOps – applying Agile principles to both writing and running Spark, and other big data jobs – is catching on, and new platforms are emerging to embody these principles.

Each tool in this plethora of tools takes in and processes different, but overlapping, data sets. No one tool provides full visibility, and even if you use one or more tools of each type, full visibility is still lacking.

You need expertise in each tool to get the most out of that tool. But the most important work takes place in the expert user's head: spotting a clue in one tool, which sends you looking at specific log entries and firing up other tools, to come up with a hypothesis as to the problem. You then have to try out the potential solution, often through several iterations of trial and error, before arriving at a "good enough" answer to the problem.

Or, you might pursue two tried and trusted, but ineffective, "solutions": ratcheting up resources and retrying the job until it works, either due to the increased resources or by luck; or simply giving up, which our customers tell us they often had to do before they started using Unravel Data.

The situation is much worse in the kind of hybrid data clouds that organizations use today. To troubleshoot on each platform, you need expertise in the toolset for that platform, and all the others. (Since jobs often have cross-platform interdependencies, and the same team has to support multiple platforms.) In addition, when you find a solution for a problem on one platform, you should apply what you've learned on all platforms, taking into account their differences. In addition, you have issues that are inherently multi-platform, such as moving jobs from one platform to a platform that is better, faster, or cheaper for a given job. Taking on all this with the current, fragmented, and incomplete toolsets available is a mind-numbing prospect.

The biggest need is for a platform that integrates the capabilities from several existing tools, performing a five-step process:

1. Ingest all of the data used by the tools above, plus additional, application-specific and pipeline data.
2. Integrate all of this data into an internal model of the current environment, including pipelines.
3. Provide live access to the model.
4. Continually store model data in an internally maintained history.
5. Correlate information across the ingested data sets, the current, "live" model, and the stored historical background, to derive insights and make recommendations to the user.

This tool must also provide the user with the ability to put "triggers" onto current processes that can trigger either alerts or automatic actions. In essence, the tool's inbuilt intelligence and the user are then working together to make the right things happen at the right time.

A simple example of how such a platform can help is by keeping information per pipeline, not just per job - then spotting, and automatically letting you know, when the pipeline suddenly starts running slower than it had previously. The platform will also make recommendations as to how you can solve the problem. All this lets you take any needed action before the job is delayed



## Conclusion

In this Part 2 guide, we've taken a whirlwind tour of tools used to troubleshoot Spark applications. Most tools provide one or more pieces of the puzzle, but none of them – nor any combination of them – is a holistic solution.

As mentioned in Part 1 of this guide, Unravel customers can easily spot and fix over-allocation and inefficiencies, using job-level insights. Developers can check a job for stability and efficiency even before putting it into production.

Unravel Data, the solution our company offers, is the leading platform for DataOps. In **Troubleshooting Spark Applications, Part 3: Solutions**, we will describe in detail how Unravel Data compares to the tools described here. We will show how it helps you prevent and repair issues at the job, pipeline, and cluster levels, while also helping with additional challenges such as cost optimization, SLA adherence, and cloud migration.

You may be fine with the tools you already have, along with any custom work you have done in-house. But if you think there's room for improvement in how you troubleshoot Spark jobs, and manage Spark and related technologies, you may want to check out Part 3.

We hope you have enjoyed, and learned from, reading this guide. If you would like to know more about Unravel Data now, you can [download a free trial](#) or [contact Unravel](#).



unravel™



# Troubleshooting Spark Solutions

PART 3: THE ANSWER IS UNRAVEL

Current practice for Spark troubleshooting is messy. Part of this is due to Spark's very popularity; it's widely used on platforms as varied as open source Apache Spark, on all platforms; Cloudera's Hadoop offerings (on-premises and in the cloud); Amazon EMR, Azure Synapse, and Google Dataproc; and Databricks, which runs on all three public clouds. (Which means you have to be able to address Spark's interaction with all of these very different environments.)

Because Spark does so much, on so many platforms, "Spark troubleshooting" covers a wide range of problems - jobs that halt; pipelines that fail to deliver, so you have to find the issue; performance that's too slow; or using too many resources, either in the data center (where your clusters can suck up all available resources) or in the cloud (where resources are always available, but your costs rise, or even skyrocket.)

## Where Are the Issues – and the Solutions?

Problems in running Spark jobs occur at the job and pipeline levels, as well as at the cluster level, as described in Part 1 of this three-part series: the top ten problems you encounter in working with Spark. And there are several solutions that can help, as we described in Part 2: five types of solutions used for Spark troubleshooting. (You can also see our recent webinar, [Troubleshooting Apache Spark](#), for an overview and demo.)

Tools	Problem Location			Added Capabilities
	Job Level	Pipeline Level	Cluster Level	
				Add'l sensors; history; correlation; AI-powered recommendations; AutoActions; cloud migration
Spark UI, logs	✓	✗	✗	✗
Orchestration tools	✗	✓	✗	✗
Cluster mgmt. tools	✗	✗	✓	✗
Unravel Data	✓	✓	✓	✓

**Table: What each level of tool shows you – and what's missing**

Existing tools provide incomplete, siloed information. We created Unravel Data as a go-to DataOps platform that includes much of the best of existing tools. In this Part 3 of the series we'll give examples of problems at the job, pipeline, and cluster levels, and show how to solve them with Unravel Data. We'll also briefly describe how Unravel Data helps you prevent problems, providing AI-powered, proactive recommendations.

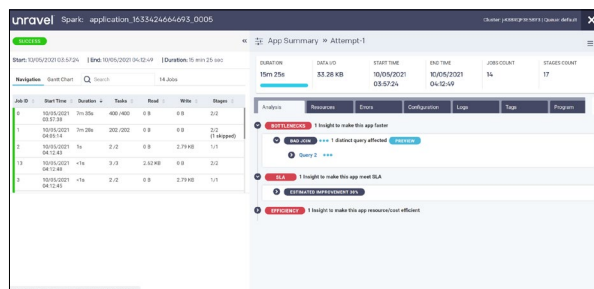
The Unravel Data platform gathers more information than existing tools by adding its own sensors to your stack, and by using all previously existing metrics, traces, logs, and available API calls. It gathers this robust information set together and correlates pipeline information, for example, across jobs.

The types of issues that Unravel covers are, broadly speaking: fixing bottlenecks; meeting and beating SLAs; cost optimization; fixing failures; and addressing slowdowns, helping you improve performance. Within each of these broad areas, Unravel has the ability to spot hundreds of different types of factors contributing to an issue. These contributing factors include data skew, bad joins, load imbalance, incorrectly sized containers, poor configuration settings, and poorly written code, as well as a variety of other issues.

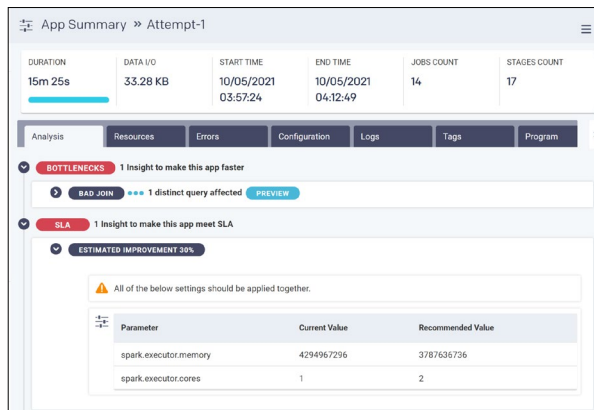
## Fixing Job-Level Problems with Unravel

Here's an example of a Spark job or application run that's monitored by Unravel.

In Unravel, you first see automatic recommendations, analysis, and insights for any given job. This allows users to quickly understand what the problem is, why it happened, and how to resolve it. In the example below, resolving the problem will take about a minute.



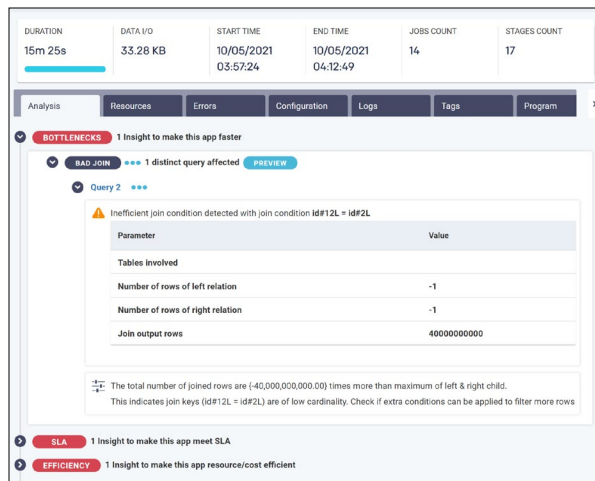
Let's dive into the insights for an application run, as shown below.



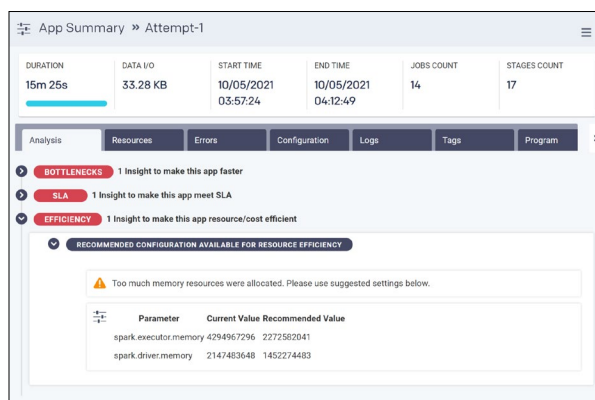


You can see here that Unravel has spotted bottlenecks, and also room for improving the performance of this app. It has narrowed down what the particular problem is with this application and how to resolve it. In this case, it has recommended to double the number of executors and reduce the memory for each executor, which will improve performance by about 30%, meeting the SLA.

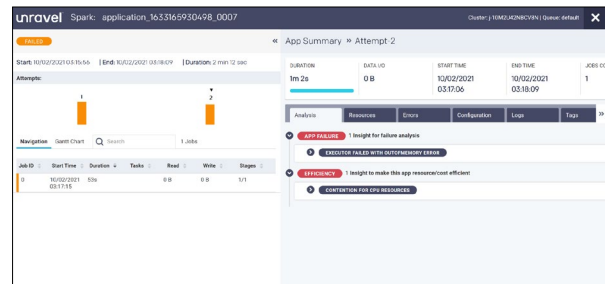
Additionally, Unravel has also spotted some bad joins which are slowing this application down, as shown below.



In addition to helping speed this application up, Unravel is also recommending resource settings which will lower the cost of running this application, as shown below - reductions of roughly 50% in executor memory and driver memory, cutting out half the total memory cost. Again, Unravel is delivering pinpoint recommendations. Users avoid a lengthy trial-and-error exercise; instead, they can solve the problem in about a minute.



Unravel can also help with jobs or applications that just didn't work and failed. It uses a similar approach as above to help data engineers and operators get to the root cause of the problem and resolve it quickly.

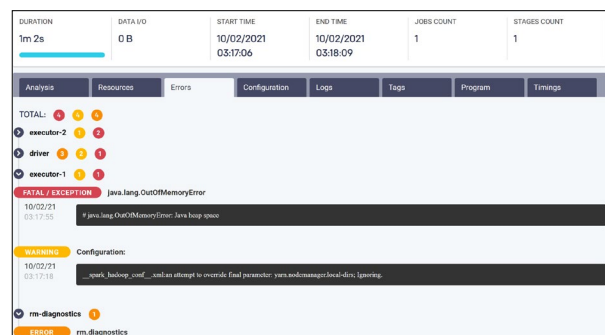


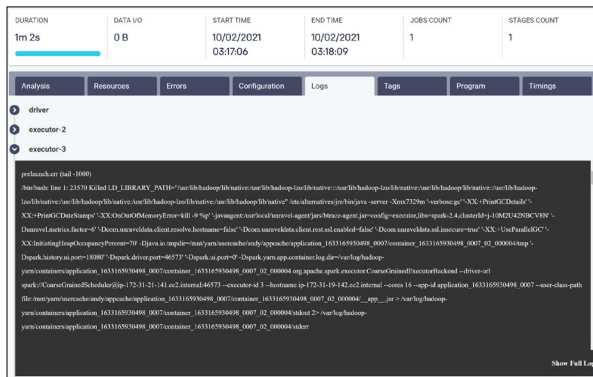
In this example, the job or application failed because of an out of memory exception error. Unravel surfaces this problem instantly and pinpoints exactly where the problem is.

For further information, and to support investigation, Unravel provides distilled and easy-to-access logs and error messages, so users and data engineers have all the relevant information they need at hand.

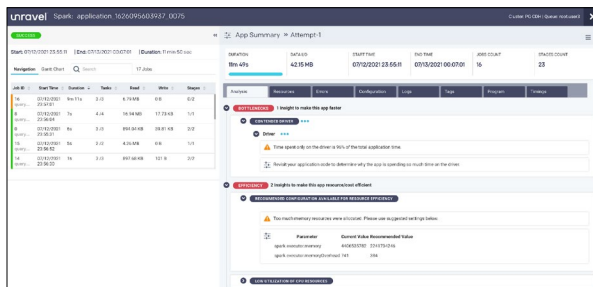
And once data teams start using Unravel, they can do everything with more confidence. For instance, if they try to save money by keeping resource allocations low, but overdo that a little bit, they'll get an out-of-memory error. Previously, it might have taken many hours to resolve the error, so the team might not risk tight allocations. But fixing the error only takes a couple of minutes with Unravel, so the data team can cut costs effectively.

Examples of logs that Unravel provides for easy access and error message screens follow.





Unravel strives to help users solve their problems with a click of a button. At the same time, Unravel provides a great deal of detail about each job and application, including displaying code execution, displaying DAGs, showing resource usage, tracking task execution, and more. This allows users to drill down to whatever depth needed to understand and solve the problem.



### Task stage metrics in Unravel Data

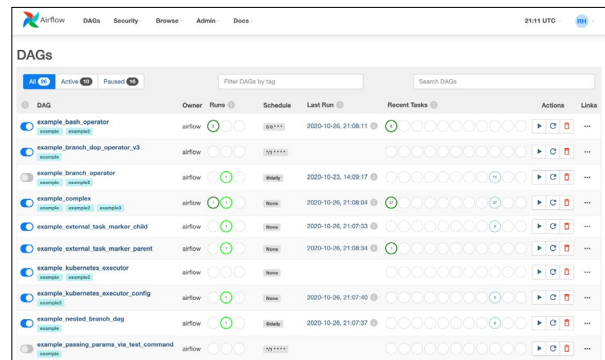
As another example, this screen shows details for task stage information:

- Left-hand side: task metrics. This includes the job stage task metrics of Spark, much like what you would see from Spark UI. However, Unravel keeps history on this information; stores critical log information for easy access; presents multiple logs coherently; and ties problems to specific log locations.
- Right-hand side: holistic KPIs. Information such as job start and end time, run-time durations, I/O in KB – and whether each job succeeded or failed.

## Data Pipeline Problems

The tools people use for troubleshooting Spark jobs tend to focus on one level of the stack or another – the networking level, the cluster level, or the job level, for instance. None of these approaches helps much with Spark pipelines. A pipeline is likely to have many stages, involving many separate Spark jobs.

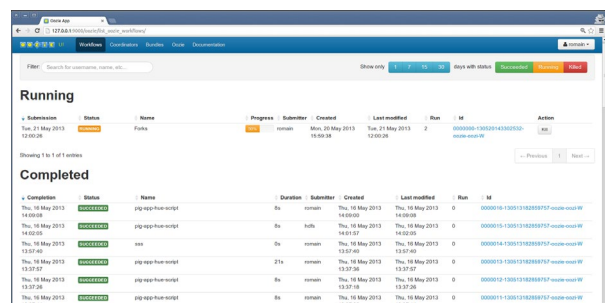
Here's an example. One Spark job can handle data ingest; a second job, transformation; a third job may send the data to Kafka; and a final job can be reading the data from Kafka and then putting it into a distributed store, like Amazon S3 or HDFS.



### Airflow being used to create and organize a Spark pipeline.

The two most important orchestration tools are Oozie, which tends to be used with on-premises Hadoop, and Airflow, which is used more often in the cloud. They will help you create and manage a pipeline; and, when the pipeline breaks down, they'll show you which job the problem occurred in.

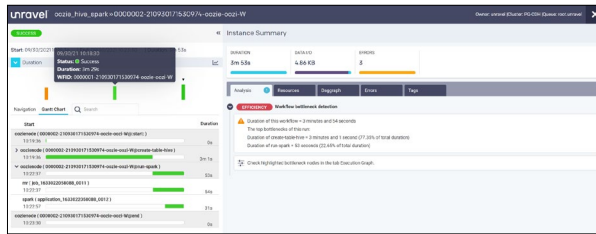
But orchestration tools don't help you drill down into that job; that's up to you. You have to find the specific Spark run where the failure occurred. You have to use other tools, such as Spark UI or logs, and look at timestamps, using your detailed knowledge of each job to cross-correlate and, hopefully, find the issue. As you see, just finding the problem is messy, intense, time-consuming, expert work; fixing it is even more effort.



### Oozie also gives you a big-picture view of pipelines.

Unravel, by contrast, provides pipeline-specific views that first connect all the components - Spark, and everything else in your modern data stack - and runs of the data pipeline together in one place. Unravel then allows you to drill down into the slow, failed, or inefficient job, identify the actual problem, and fix it quickly. And it gets even better; Unravel's AI-powered recommendations will help you prevent a pipeline problem from even happening in the first place.

You didn't have to look at Spark UI, plus dig through Spark logs, then check Oozie or Airflow. All the information is correlated into one view - a single pane of glass.



This view shows details for several jobs. In the graphic, each line has an instance run. The longest duration shown here is three minutes and 1 second. If the SLA is “under two minutes,” then the job failed to meet its SLA. (Because some jobs run scores or hundreds of times a day, missing an SLA by more than a minute - especially when that means a roughly 50% overshoot against the SLA - can become a very big deal.)

Unravel then provides history and correlated information, using all of this to deliver AI-powered recommendations. You can also set AutoActions against a wide variety of conditions and get cloud migration support.

## Cluster Issues

Resources are allocated at the cluster level. The screenshot shows ResourceManager (RM), which tracks resources, schedules jobs such as Spark jobs, and so on. You can see the virtual machines assigned to your Spark jobs, what resources they're using, and status - started or not started, completed or not completed.

ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus
application_1500262180460_0004	hadoop	HIVE-d724-280-0302-4180-836c-621e16e24c3	TEZ	root.hadoop	0	Mon Jul 17 11:45:49 +0800 2017	Mon Jul 17 11:46:09 +0800 2017	FINISHED	SUCCEEDED
application_1500262180460_0003	hadoop	HIVE-4a1316e02-8a01-4a20-8a6c-b298946315ef	TEZ	root.hadoop	0	Mon Jul 17 11:45:21 +0800 2017	Mon Jul 17 11:45:54 +0800 2017	FINISHED	SUCCEEDED
application_1500262180460_0002	hadoop	HIVE-3ae37043-47f5-4044-a89b-c28904e681ea	TEZ	root.hadoop	0	Mon Jul 17 11:45:01 +0800 2017	Mon Jul 17 11:45:25 +0800 2017	FINISHED	SUCCEEDED
application_1500262180460_0001	hadoop	Spark PI	SPARK	root.hadoop	0	Mon Jul 17 11:30:42 +0800 2017	Mon Jul 17 11:31:10 +0800 2017	FINISHED	SUCCEEDED

### Apache Hadoop ResourceManager

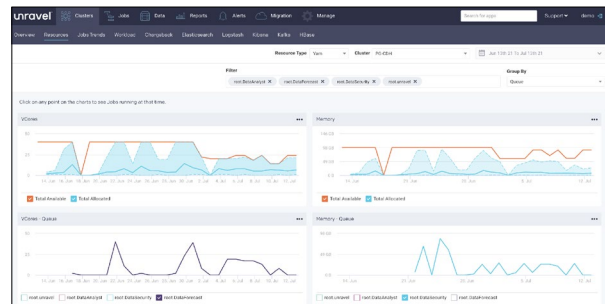
The first problem is that there's no way to see what actual resources your job is consuming. Nor can you see whether those resources are being used efficiently or not. So you can be over-allocated, wasting resources - or running very close to your resources limit, with the job likely to crash in the future.

Nor can you compare past to present; ResourceManager does not have history in it. Now you can pull logs at this level – the YARN level – to look at what was happening, but that's aggregated data, not the detail you're looking for. You also can't dig into potential conflicts with neighbors sharing resources in the cluster.

You can use site tools like Cloudwatch, Cloudera Manager or Ambari. They provide a useful holistic view, at the cluster level – total CPU consumption, disk I/O consumption, and network I/O consumption. But, as with some of the pipeline views we discussed above, you can't take this down to the job level.

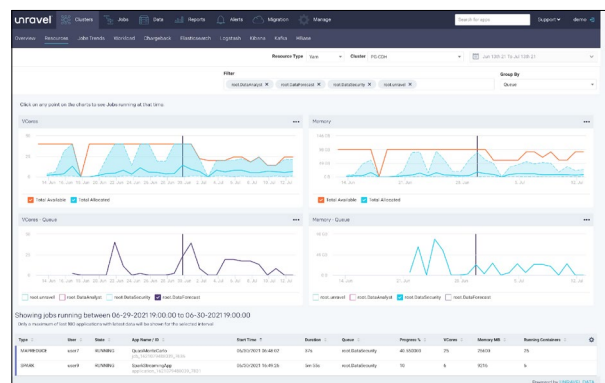
You may have a spike in cluster disk I/O. Was it your job that started that, or someone else's? Again, you're looking at Spark UI, you're looking at Spark logs, hoping maybe to get a bit lucky and figure out what the problem is. Troubleshooting becomes a huge intellectual and practical challenge. And this is all taking away from time making your environment better or doing new projects that move the business forward.

It's common for a job to be submitted, then held because the cluster's resources are already tied up. The bigger the job, the more likely it will have to wait. But existing tools make it hard to see how busy the cluster is. So later, when the job that had to wait finishes late, no one knows why that happened.



### A cluster-level view showing vCores, specific users, and a specific queue

By contrast, in this screenshot from Unravel, you see cluster-level details. This job was in the data security queue, and it was submitted on July 5th, around 7:30pm. These two rows show vCores – overall consumption on this Hadoop cluster's memory. The orange line shows maximum usage, and the blue line shows what's available.



### At this point in time, usage (blue line) did not exceed available resources (orange line)



You can also get more granular and look at a specific user. You can go to the date and time that the job was launched and see what was running at that point in time. And voilà – there were actually enough resources available.

So, it's not a cluster-level problem; you need to examine the job itself. And Unravel, as we've described, gives you the tools to do that. You can see that we've eliminated a whole class of potential problems for this slowdown – not in hours or days, and with no trial-and-error experimentation needed. We just clicked around in Unravel for a few minutes.

## Unravel Data: An Ounce of Prevention

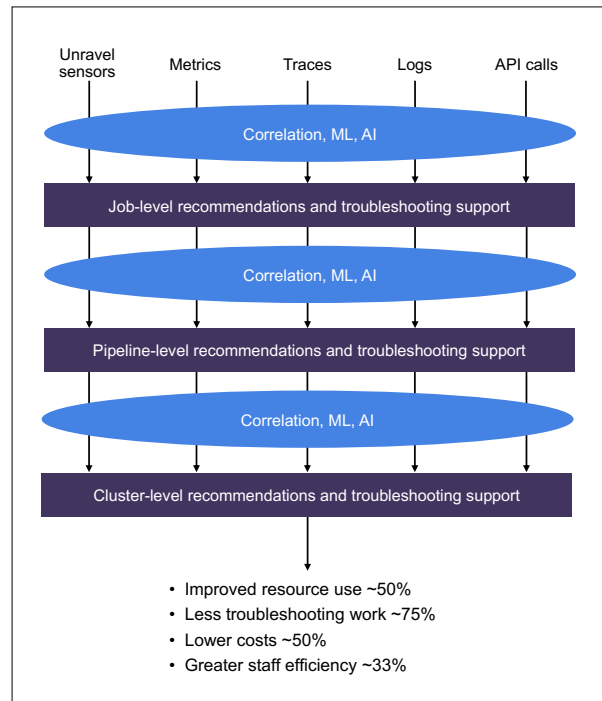
For the issues above, such as slowdowns, failures, missed SLAs or just expensive runs, a developer would have to be looking at YARN logs, ResourceManager logs, and Spark logs, possibly spending hours figuring it all out. Within Unravel, though, they would not need to jump between all those screens; they would get all the information in one place. They can then use Unravel's built-in intelligence to automatically root-cause the problem and resolve it.

Unravel Data solves the problem of Spark troubleshooting at all three levels – at the job, pipeline, and cluster levels. It handles the correlation problem – tying together cluster, pipeline, and job information – for you. Then it uses that information to give unique views at every level of your environment. Unravel makes AI-powered recommendations to help you head off problems; allows you to create AutoActions that execute on triggers you define; and makes troubleshooting much easier.

Unravel solves systemic problems with Spark. For instance, Spark tends to cause overallocation: assigning very large amounts of resources to every run of a Spark job, to try to avoid crashes on any run of that job over time. The biggest datasets or most congested conditions set the tone for all runs of the job or pipeline. But with Unravel, you can flexibly right-size the allocation of resources.

Unravel frees up your experts to do more productive work. And Unravel often enables newer and more junior-level people to be as effective as an expert would have been, using the ability to drill down, and the proactive insights and recommendations that Unravel provides.

Unravel even feeds back into software development. Once you find problems, you can work with the development team to implement new best practices, heading off problems before they appear. Unravel will then quickly tell you which new or revised jobs are making the grade.



### *The Unravel advantage – on-premises and all public clouds*

Another hidden virtue of Unravel is: it serves as a single source of truth for different roles in the organization. If the developer, or an operations person, finds a problem, then they can use Unravel to highlight just what the issue is, and how to fix it. And not only how to fix it this time, for this job, but to reduce the incidence of that class of problem across the whole organization. The same goes for business intelligence (BI) tool users such as analysts, data scientists, everyone. Unravel gives you a kind of X-ray of problems, so you can cooperate in solving them.

With Unravel, you have the job history, the cluster history, and the interaction with the environment as a whole – whether it be on-premises, or using Databricks or native services on AWS, Azure, or Google Cloud Platform. In most cases you don't have to try to remember, or discover, what tools you might have available in a given environment. You just click around in Unravel, largely the same way in any environment, and solve your problem.

Between the problems you avoid, and your new-found ability to quickly solve the problems that do arise, you can start meeting your SLAs in a resource-efficient manner. You can create your jobs, run them, and be a rockstar Spark developer or operations person within your organization.

## Conclusion

In this Part 3 of the guide, we've given you a wide-ranging tour of how you can use Unravel Data to troubleshoot Spark jobs – on-premises and in the cloud, at the job, pipeline, and cluster levels, working across all levels, efficiently, from a single pane of glass.

**In Troubleshooting Spark Applications, Part 1:**

**Top Ten Spark Difficulties**, we described the ten biggest challenges for troubleshooting Spark jobs across levels.

And in **Spark Troubleshooting, Part 2: Five Types of Solutions**, we describe the major categories of tools, several of which we touched on here.

This Part 3 of the guide builds on the other two to show you how to address the problems we described, and more, with a single tool that does the best of what single-purpose tools do, and more – our DataOps platform, Unravel Data.

We hope you have enjoyed, and learned from, reading this series of guides. If you would like to know more about Unravel Data now, you can [download a free trial](#) or [contact Unravel](#).