Automated Performance Management for the Big Data Stack

Anastasios Arvanitis, Shivnath Babu, Eric Chu, Adrian Popescu, Alkis Simitsis, Kevin Wilkinson

unravel

Automated Performance Management for the Big Data Stack

Anastasios Arvanitis, Shivnath Babu, Eric Chu, Adrian Popescu, Alkis Simitsis, Kevin Wilkinson

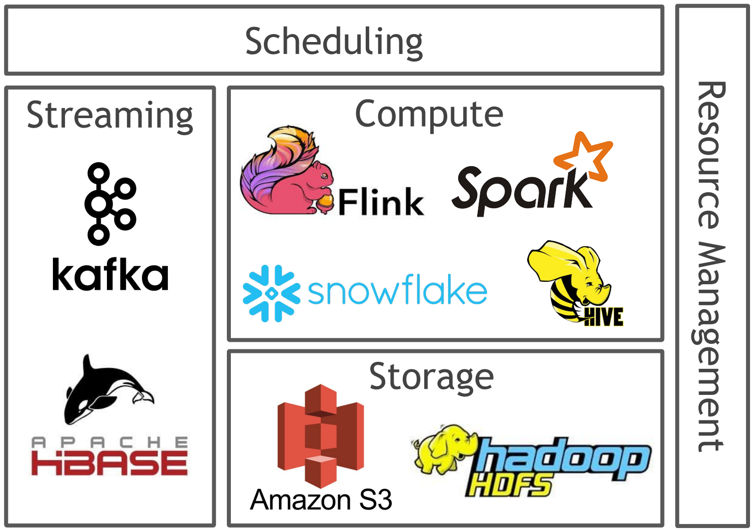
unravel

Tuning Database Configuration Parameters with iTuned. VLDB’09

Application Performance Management (APM) software: helps with the monitoring and management of the performance and availability of applications.

unravel

BIG DATA STACK



Roadmap

- Performance management requirements
- Architecture of a performance management solution
- Solutions deep dive
- Conclusion

Performance Management Requirements

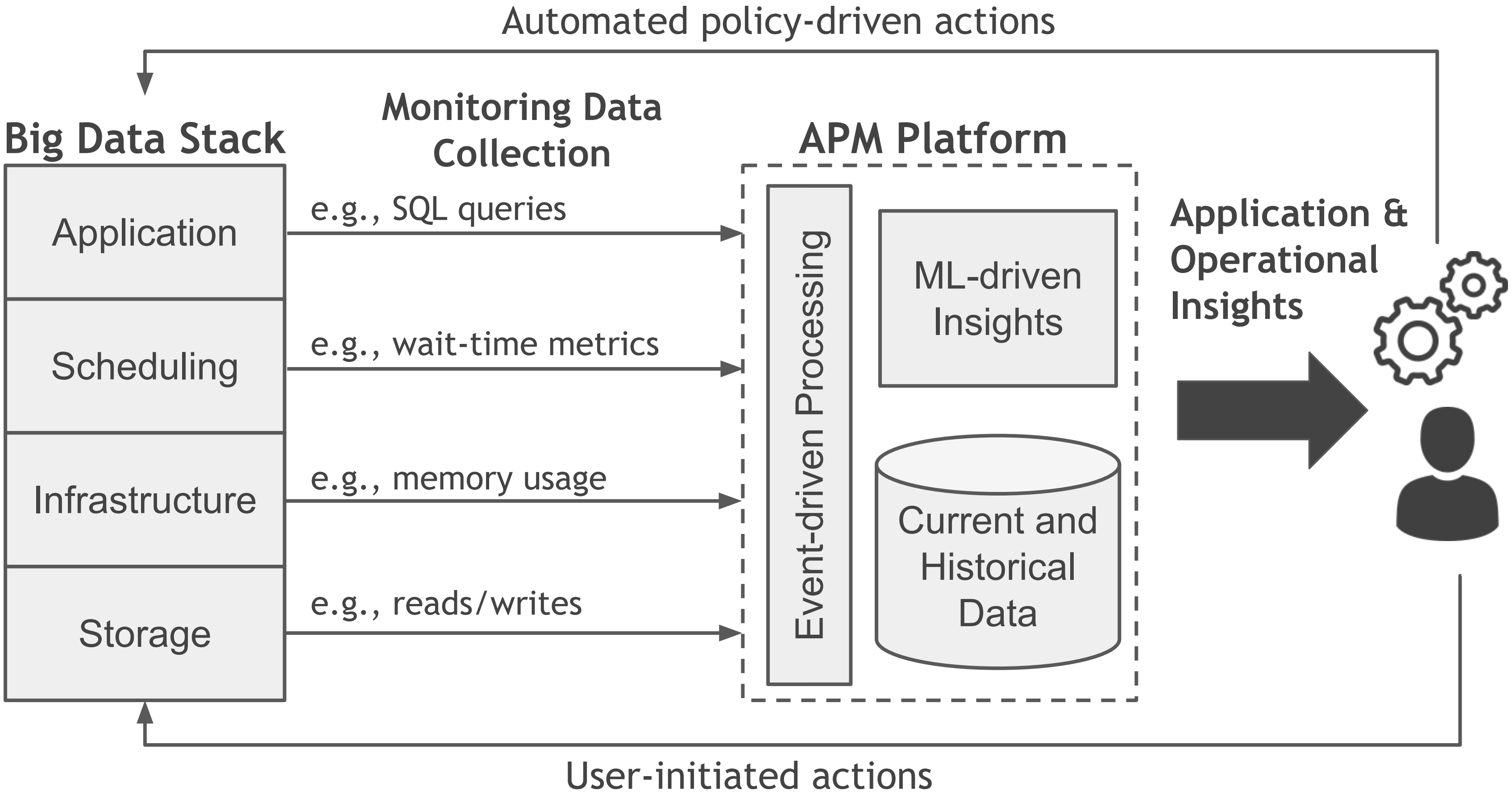
Application:

- Failures
- Stalls
- Runaways
- SLA compliance
- Changes over time
- Rogue/victim apps

Operational:

- Resource allocation policies
- Rogue application detection
- Tuning configuration knobs, data partitioning, and storage layout
- Optimizing cloud costs
- Capacity planning
- Efficient ‘chargeback’

Architecture of a Performance Management Solution



Solutions Deep Dive

1. Application failure

2. Cluster optimization

Solution Deep Dive #1: Application Failure

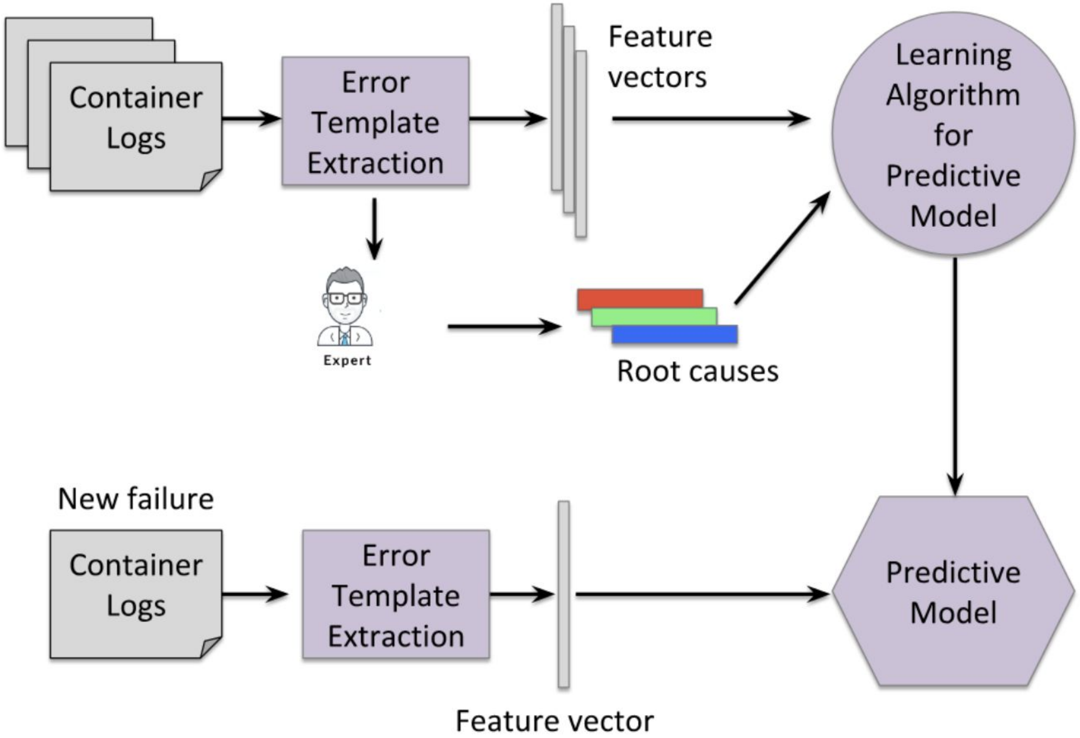
Example: Distributed Spark applications

- 1 driver container, 1+ executor containers
- Many verbose, messy logs are generated each time an application fails

Two parts:

1. Automatic identification of the root cause of the failure
2. Automatic fixes for failed applications

Part 1: Supervised Approach for Automatic Root Cause Analysis (RCA)



Training Data Collection and Preprocessing

Training data:

- Logs from real-life Spark application failures
- Logs generated by their lab framework that artificially injects failures

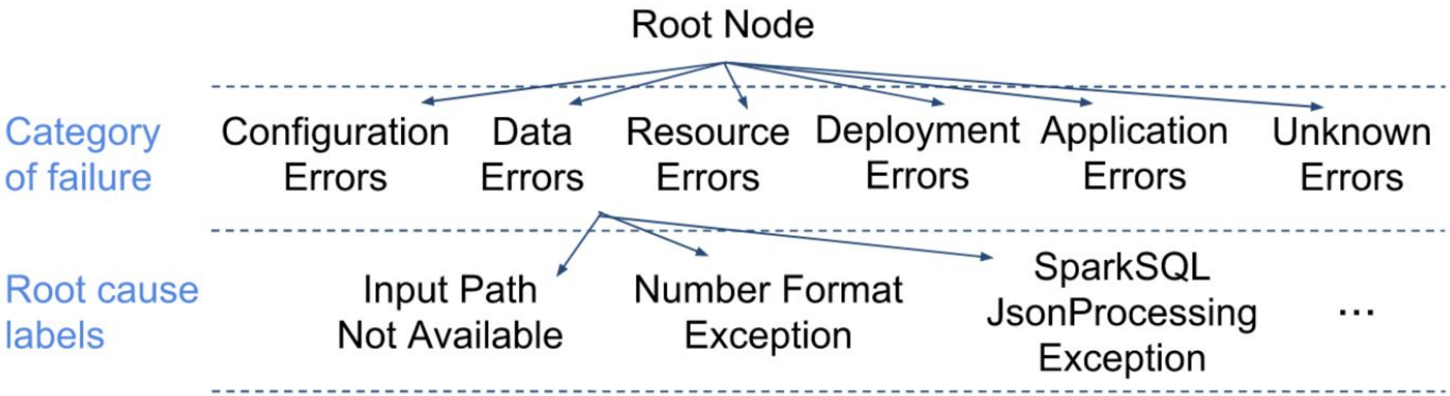
Preprocessing:

- Extract all possible error message templates from each log

Labeling the Root Cause

- Logs generated from their lab framework: *labels already known*
- Logs from real-life Spark failures: *labeled by a* ***human expert***

Taxonomy of Failures



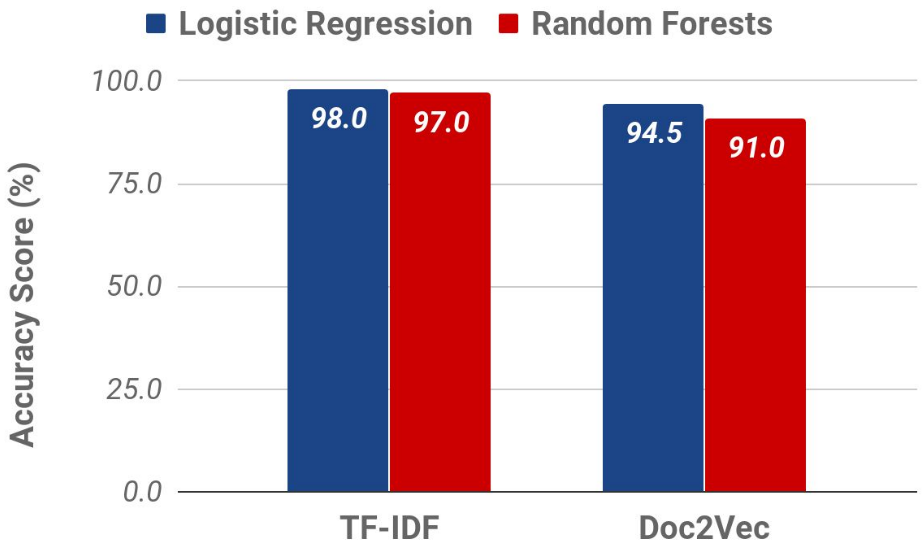
Transforming Logs Into Feature Vectors

1. Bit vector membership encoding (e.g., 100110)
    a. Each bit represents whether a specific error message template is present in the log
2. Bag of words + TF-IDF
    a. Ignores word order and semantics
3. Doc2Vec
    a. Incorporates word order and semantics information

Ready to train the predictive model.

Accuracy at Predicting the Root Cause

- Training data (failure logs) generated from injecting 14 root causes of failures
- Accuracy calculated using a 75%-25% split of training and test data



Solution Enhancements

- Make the degree of confidence in the predicted root cause easier for users to understand
- Speed up the ability to incorporate new types of application failures
    - Active learning techniques to prioritize failure log labeling tasks

Part 2: Automatic Fixes for Failed Applications

Key findings from analysis of the Spark application failure logs:

- “90-10” rule in the root cause of application failures
- Two most common causes:
  - Running out of memory (OOM) on some component
  - Timeouts while waiting for some resources

Configuring memory allocation and usage:

- Multiple configuration knobs at each component: Driver, Executor, container, JVM, and more...

Automatic Fixes for Failed Applications caused by OOM

Maintain 2 variables for each memory-specific configuration knob (*m*):

- *m_lo*: max known setting of *m* that causes OOM
- *m_hi*: min known setting of *m* that does not cause OOM
  - Most resource-efficient setting known to run the application successfully

Let *m_curr* be the current of setting of *m* while the application is running and *m_obs* be the observed usage of *m*

On application success:

- *m_hi = min(m_hi, m_obs)*

On application failure due to OOM:

- *m_lo = max(m_lo, m_curr)*

New run of the application: set *m = (m_hi + m_lo) / 2*

Example: Automatic Tuning of a Failed Spark App (OOM)

- Tuning the amount of memory allocated in an Executor container

	TYPE	STATUS	ID	DURATION
☐	SPARK	FAILED	..._1043	15m 34s
☐	SPARK	SUCCESS	..._1044	4m 29s
☐	SPARK	SUCCESS	..._1045	1m 3s
☐	SPARK	SUCCESS	..._1046	1m 5s
☐	SPARK	SUCCESS	..._1047	1m 4s

Solution Deep Dive #2: Cluster Optimization

Three parts (sort of):

1. Fine-tuning cluster-wide configuration parameters
2. Optimizing resource budget configurations
3. Capacity planning using predictive analysis

Approach for Fine-tuning Cluster-wide Configuration Parameters

- Collect performance data of prior completed applications
- Analyze the applications w.r.t. the cluster's current configuration
- Generate recommended cluster parameter changes
- Predict/quantify the impact these changes will have on the applications in the future

Example: Fine-tuning Cluster-wide Config Params

Tune the size of map containers

mapreduce.map.memory.mb

Recommendation: **2048**

Improvement Potential: **High**

Using the recommended default would save 50% of 1,263,087,616 MB from 7275 jobs using the current default, or 97% of total jobs in the workload

Tuning Instructions

Default: **4096**

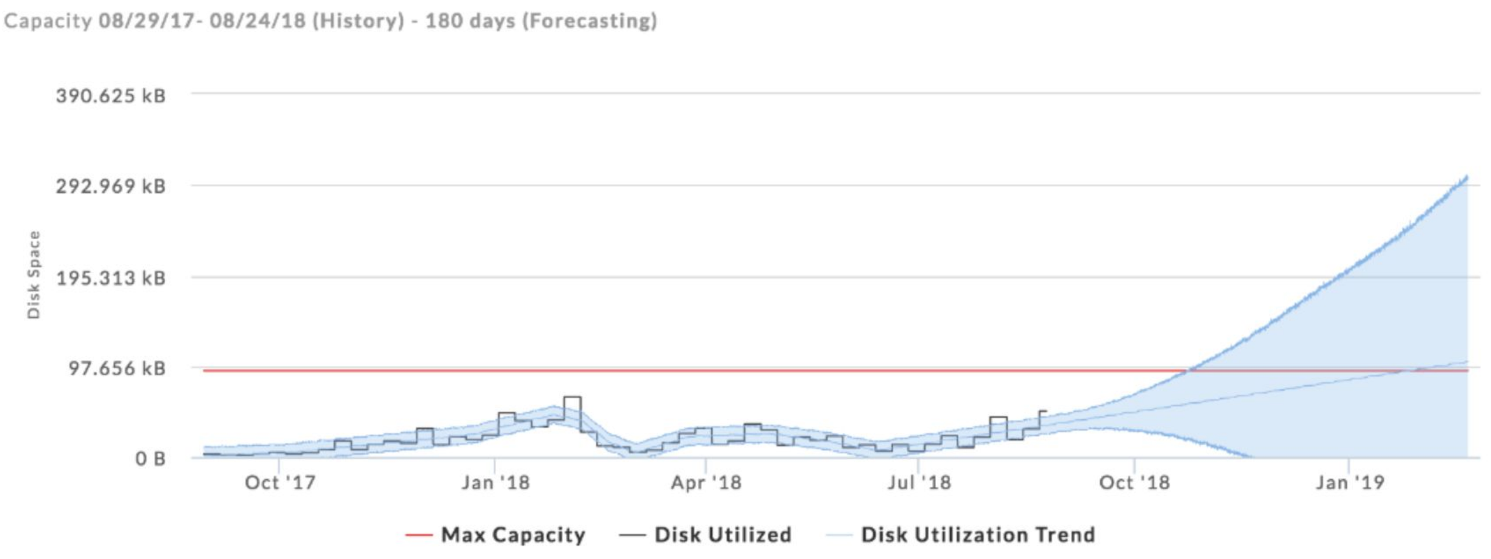
% Apps Default: **97%**

Candidate	% of memor...	% of jobs ...
1792	56%	80%
2048	50%	85%

Optimizing Resource Budget Configurations

- Track resource utilization
- Compare **pending resource requests** with the **resources currently allocated** to generate insights
- Recommend actions based on the insights

Capacity Planning Using Predictive Analysis



- Cites “Forecasting at Scale” by S. Taylor and B. Letham from Facebook

Conclusion

- Performance management requirements of big data stacks
- Architecture for providing automated solutions to these requirements
- Deep dive into some solutions

Thoughts:

- Wish deep dives went deeper and that there was a larger discussion of the challenges they have encountered along the way
- Glad to see they are still around and making the effort to publish