RFC: Root Cause Analysis

Introduction

The Open Distro for Elasticsearch Performance Analyzer captures OpenSearch and JVM layer activity, and lower-level resource usage (e.g., disk, network, CPU and memory) of these activities. Based on this instrumentation, Performance Analyzer computes and exposes *diagnostic* metrics, with the goal of enabling OpenSearch users and administrators to measure and understand bottlenecks in their OpenSearch clusters. The Open Distro for Elasticsearch PerfTop client provides real time visualization of these diagnostic metrics to surface bottlenecks to OpenSearch users and operators.

This RFC proposes a design for a framework that builds on the Performance Analyzer to support root cause analysis (RCA) of performance and reliability problems for OpenSearch instances. This framework would conduct real time root cause analysis of such problems using Performance Analyzer metrics. Root cause analysis can significantly improve operations, administration and provisioning of OpenSearch clusters, and it can enable OpenSearch client teams to tune their workloads to reduce errors.

Root Cause Analysis

The first step is to define what we mean by a “root cause". We define a root cause as a function of one or more *symptoms*. A symptom is an operation over metrics or *other* symptoms. Root causes may also be a function of other root causes. Note that these operations may involve aggregations; for example, a symptom could consume a time average of a metric. In addition, for confidence, a root cause could be a computation over a sufficiently long window of time. This definition does not allow for cycles in the dependency graph between metrics and root causes. The following equations show an example of these relationships:

$$\begin{aligned}\mathrm{root\_cause}_i &= f_{\mathrm{RCA}}(\mathrm{symptom}_k, \mathrm{root\_cause}_j)\\ \mathrm{symptom}_k &= f_{\mathrm{Symptom}}(\mathrm{metric}_1, f_{\mathrm{Avg}}(\mathrm{metric}_2))\end{aligned}$$

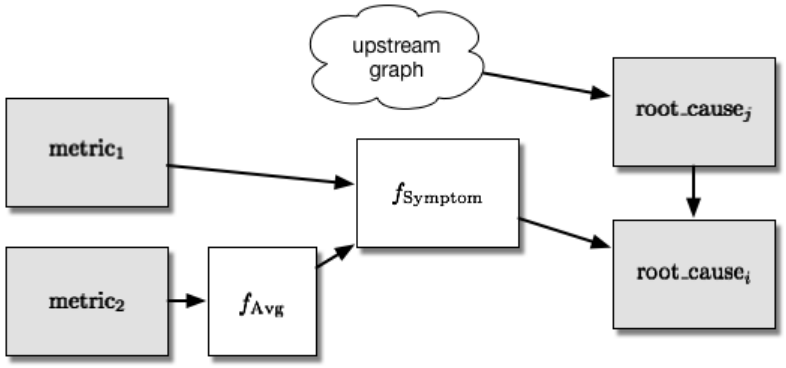
Note that any of the functions above can take metadata as inputs, such as thresholds.

Root causes can span the entire stack, from the infrastructure layers (e.g., the OS, host, virtualization layers and the network) to the Java Virtual Machine to the OpenSearch engine. Root causes also include problems related to the input workload to OpenSearch.

Root Cause Analysis Framework

A root cause analysis framework helps diagnose root causes in real time. Based on the recursive definition above, one design choice would be to build an acyclic data flow graph that would take metric streams generated by the Performance Analyzer plugin as input. Nodes of the data flow graph would include computations such as metrics output (source nodes), aggregations, symptoms and root causes (sink nodes). The data flow graph across all root causes would span all nodes of an OpenSearch cluster (including master nodes).

Edges of the graph transfer the output of a parent node to all child nodes.The framework would treat this output as an opaque stream since the data format between nodes is a contract between each pair of nodes. The framework explicitly requires nodes to send timestamps – this is necessary for a node to diagnose issues with a parent node and handle staleness in data (e.g., data delivered late). Message delivery is ordered and provides at most once semantics (i.e., messages could be dropped to keep up with stream rate); small message loss isn’t a significant issue for root cause analysis because such algorithms rely significantly on statistical data. The following figure shows the above equations as a data flow graph (sources and sinks are shaded):



The framework needs to be fault tolerant for OpenSearch, JVM and infrastructure performance and reliability problems.

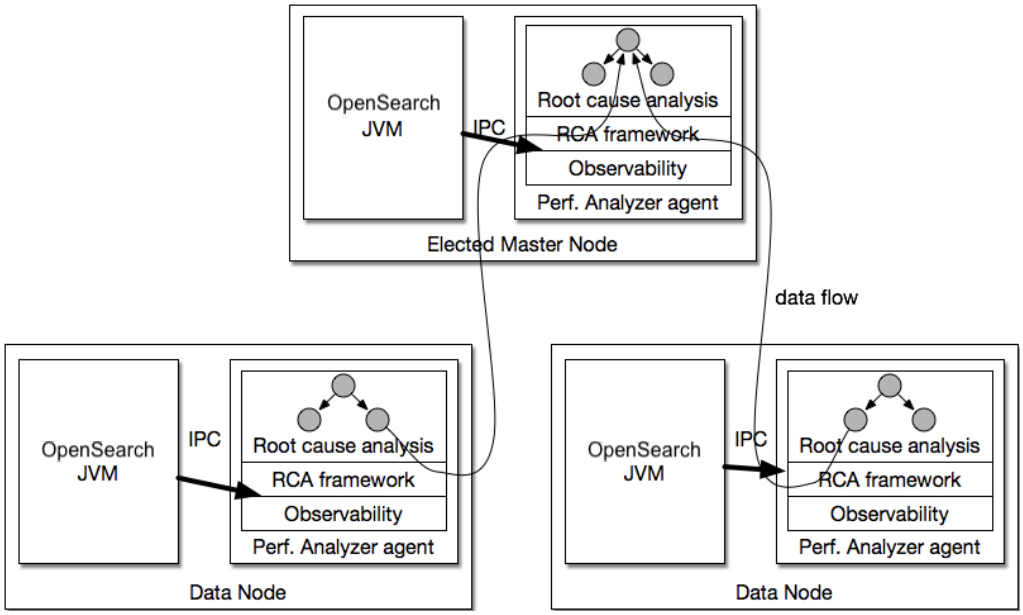
The framework would expose an API to query the current (or recent) set of diagnoses across some nodes or the entire cluster. This could be used by diagnostic tools (e.g., the Performance Analyzer PerfTop) or automated control plane actions.

Proposed implementation

Architecture

The root cause analysis framework builds on the Performance Analyzer architecture. Performance Analyzer consists of two components: an OpenSearch plugin that records OpenSearch events (and their context), and an agent process that provides a view of the system state at that node periodically. The agent also exposes metrics with a number of dimensions (also referred to as observability).

The root cause analysis framework builds on the observability component to construct a data flow graph that computes root causes. All RCAs must be registered with the framework. This allows the framework to de-dup computations and optimize the streaming runtime. It exposes root causes and their context for applications to consume. Note that the framework resides in the agent process, so it is isolated from failures and performance problems in the OpenSearch JVM. The architecture is shown below.



rca configuration

This is the primary configuration for the RCA framework on a node. When the framework starts up, it looks for this configuration in the Performance Analyzer configuration directory. The required fields are:

- ***analysis-graph-implementor***: main class instantiated by the run-time framework, and
- ***threshold-store-location***: location where threshold related information is present

The configuration also includes metadata (`tags`) for the node, notably, the `locus`, which is an identifier string for the node "type" (e.g., data and master nodes). The locus is used by the RCA framework to instantiate certain nodes in the RCA graph. The `datastore` block allows the user to persist the RCAs.

```
{
  // This package is instantiated by the runtime to run the RCA dataflow graph.
  // Only one RCA graph is run on one OpenSearch cluster.
  "analysis-graph-implementor": "com.amazon.opendistro.opensearch.performanceanalyzer.

  "rca-store-location": "/user/rcas/rcas.jar",

  "threshold-store-location": "/user/thresholds/",

  // How often to check if the files in the thresholds directory are updated.
  "new-threshold-check-minutes": 30,

  // metadata
  "tags": {
    "locus": "loc1",
    "disk": "ssd",
    "region": "use1",
    "instance-type": "i3.8xl",
    "domain": "rca-test-cluster"
  },

  // This is the persistent store (sqlite or in-memory)for the RCAs.
```

```
  "datastore": {
    "type": "sqlite",
    "location-dir": "/user",
    "filename": "rca.sqlite",

    // How often the sqlite file (containing RCAs) will be rotated, measured in second
    "rotation-period-seconds": 21600
  }
}
```

Threshold Definition

Threshold files are used in evaluation of symptoms and RCAs. Separating this metadata from the root cause analysis code allows thresholds to be updated without code changes, and enables support for dynamic thresholds. The threshold file allows for defining thresholds as a function of the cluster, specific instance types and instance properties. This is useful when administering multiple heterogeneous OpenSearch clusters.

```
{
  "name": "threshold1",
  "overrides": {
    "domain": {
      "domain1": "value1",
      "domain2": "value2",
    },
    "instance-type": {
      "i3.8xl": "value4",
      "m4.l": "value5"
    },
    "disk": {
      "ssd": "val-ssd",
      "spinning": "val-spin"
    }
  },
  "default": "value6",
  "override-precedence-order" : ["domain", "disk", "instance-type"]
}
```

The data flow graph

An RCA data flow graph threads all metrics nodes (source nodes) with multiple symptoms and RCAs. The framework expects a graph input , which can have multiple connected components.

We propose creating the RCA data flow graph as follows (the framework is in Java). The framework interface breaks down the root cause analysis data flow into two parts: defining the computation done in each node of the data flow, and defining the edges between these nodes (and the rest of the data flow). The framework's `RCA` class enables the node definition, by specifying two parts: the `evaluate` code and the frequency at which that code is called by the framework. The framework's `AnalysisGraph` class enables the edge definitions via the `construct` method. The framework streams data between nodes of the graph as objects of the `FlowUnit` type. Details:

1. Create the symptoms and RCAs as separate class files inside the package `com.amazon.opendistro.opensearch.performanceanalyzer.rca.store.rca`
2. Override the `evaluate` function to calculate the RCA or symptom using the data from upstream nodes.
3. Wrap the result of the evaluation in a `FlowUnit` and return.

4. Connect all nodes in by sub-classing the `AnalysisGraph` and implementing the `construct` method.

The class specification is as follows. The appendix also contains an example graph to evaluate a symptom that finds whether the CPU utilization for a shard is high, based on Performance Analyzer metrics.

```
// An RCA should extend the Rca class and it is required to Override the
// evaluate method.
public class HighShardCpuRca extends RCA {

    // The argument specifies how often 'evaluate' is called.
    public HighShardCpuRca(long evaluationIntervalSeconds);

    /**
     *
     * @param dependencies These are the upstream nodes, whose data it needs to
     *           evaluate itself. Dependencies are captured as a map where the
     *           'key' is the java.lang.Class of the Node and
     *           'value' is the output of the node (FlowUnit).
     *
     * @return The evaluated data. Data is encapsulated in what is called as FlowUnit
     *       FlowUnit is made of :
     *       timestamp: The time of genesis of the data
     *       data: A list of list structure representing a database table. This
     *       is similar to the structure that exists in the Performance Agents's
     *       sqlite tables. The author of 'evaluate' method chooses the dimensions
     *       and the values for each dimension, which will be persisted in the
     *       persistence store. Header row (row #0) is expected to contain the
     *       column name for each dimension and values.
     *       context: An arbitray number of (key, value) pairs serialized as string.
     *       It represents some context about how we arrived at an
     *       an RCA such as the thresholds and the actual values.
     */
    @Override
    public FlowUnit evaluate(Map<Class, FlowUnit> dependencies);
}
```

```
public class RCAGraph extends AnalysisGraph {
    @Override
    public void construct();
}
```

RESt API definitions

RCAs can be fetched from the internal RCA store using a rest API that Performance Analyzer exposes. Assume at a particular point in time, the root cause analysis system diagnoses two RCAs - rca-x and rca-y, the API would output the following:

```
GET <endpoint>:9600/_opendistro/_performanceanalyzer/rcas
```

```
{"name": "rca-x",
 "data": [
    {"Resource": "X",
     "State": "Unhealthy",
```

```
      "Context": {"actual": 34, "threshold": 20},
      "timestamp": 12345678},
     {"Resource": "X",
      "State": "Unhealthy",
      "Context": {"actual": 42, "threshold": 15},
      "timestamp": 12345678}
   ],
  "name": "rca-y",
 "data": [
     {"Resource": "Y",
      "State": "Contended",
      "Context": {"actual": 23.456, "threshold": 22},
      "timestamp": 12345678},
      {"Resource": "Y",
      "State": "Contended",
      "Context": {"actual": 21.456, "threshold": 17},
      "timestamp": 12345678}
   ],
 }
```

The endpoint would also allow querying for a specific root cause type.

Request for Comments

We seek comments and feedback on the proposed design for root cause analysis framework. Some specific questions we're seeking feedback on include:

- Types of production problems you face with OpenSearch, their symptoms and the root causes? Please add as many details as possible (e.g., cluster sizing, workload and configuration).
- What are some extensions you would like to see added to the proposed root cause analysis framework?
- How do you as an user of a RCA system consume root causes? For example, would you use alerts via email or push notifications to a chat.

Appendix: Example RCA code

The following is an example graph to evaluate a symptom that tracks CPU utilization for a shard, based on Performance Analyzer metrics. All code is written in Java (the same as that of the Performance Analyzer agent).

```
package com.amazon.opendistro.opensearch.performanceanalyzer.rca.store.rca;

public class HighShardCpuRca extends Rca {
    private static final Logger LOG = LogManager.getLogger(HighShardCpuRca.class);
    private Map<String, MovingAverage> averageMap;

    // The constructor takes one argument, which determines how often the evaluate
    // is called.
    public HighShardCpuRca(long evaluationIntervalSeconds) {
        super(evaluationIntervalSeconds);
        averageMap = new HashMap<>();
    }

    /**
     * If X% of the samples is above Y%, then we deem this symptom to have high-cpu us
```

```
     *
     * @param dependencies These are the upstream nodes whose data it needs to
     *           evaluate. The way the dependencies are captured is via a map where th
     *           key is the java.lang.Class of the Node and
     *           value is the output of the node.
     * @return The evaluated data. The data is encapsulated in FlowUnits.
     *
     */
    @Override
    public FlowUnit evaluate(Map<Class, FlowUnit> dependencies) {
        // A symptom for a hot shard can be defined in terms of moving average of metr
        // and threshold comparisons.
        // 1. We create a moving average (say 3 samples) object for each distinct shar
        // 2. Compare the result from #1 against a threshold (say 90%) and if such a s
        //    then report it as a FlowUnit
        FlowUnit cpuMetric = dependencies.get(CPU_Utilization.class);

        List<List<String>> allData = cpuMetric.getData();
        List<String> cols = allData.get(0);

        int shardIDIdx = IntStream.range(0, cols.size())
             .filter(i -> SHARD_ID.toString().equals(cols.get(i))).findFirst().getAsIr

        int maxColIdx = IntStream.range(0, cols.size())
             .filter(i -> "max".equals(cols.get(i))).findFirst().getAsInt();

        // This is something that should be inside a threshold file.
        final double HIGH_CPU_THRESHOLD = 90.0;

        // This is the evaluation result.
        List<List<String>> ret = new ArrayList<>();
        Map<String, String> context = new HashMap<>();

        allData.stream().skip(1).forEach(row -> {
            String shardId = row.get(shardIDIdx);
            MovingAverage entry = averageMap.get(shardId);
            if (null == entry) {
                entry = new MovingAverage(3);
                averageMap.put(shardId, entry);
            }
            double val = entry.next(Double.parseDouble(row.get(maxColIdx)));
            if (val > HIGH_CPU_THRESHOLD) {
                List<String> dataRow = Collections.singletonList(shardId);
                context.put("threshold", String.valueOf(HIGH_CPU_THRESHOLD));
                context.put("actual", String.valueOf(val));
                ret.add(dataRow);
            }
        });

        return new FlowUnit(System.currentTimeMillis(), ret, context);
    }

}
```

```
package com.amazon.opendistro.opensearch.performanceanalyzer.rca.store;

public class RCAGraph extends AnalysisGraph {
    @Override
    public void construct() {
        // Start with all the metrics we would need.
        Metric cpu = new CPU_Utilization(30);

        // class we defined above.
        HighShardCpuRca highShardCpuRca = new HighShardCpuRca(60);

        // assume that the metrics are generated on the same host as the RCA; we
        // add tags to make sure they both are evaluated. Not adding a tag makes
        // a graph node be always evaluated. Tags are key-value pairs and are
        // matched against what exists in the rca.conf.
        cpu.addTag("locus", "loc1");
        highShardCpuRca.addTag("locus", "loc1");

        // add the leaves to the graph
        addLeaf(cpu);

        // for each node, add all the upstream nodes.
        highShardCpuRca.addAllUpstreams(Collections.singletonList(cpu));
    }
}
```