Week6_Monitoring_&_Tuning

The Apache Spark User Interface

Start your application
Connect to the application UI using the following URL:

`http://<driver-node>:4040`

The SparkContext starts a web server for the application UI

- The driver program is the host
- The port defaults to 4040
- The UI is available only while the application is active

Why use the application UI

The Spark user interface shows information about the running application:

- Shows jobs, stages and tasks
- Storage of persisted RDDs and DataFrames
- Environment configuration and properties
- Executor summary
- SQL information (if SQL queries exist)
- The **Jobs** tab displays the aplication's jobs, including job status and the **Stages** tab reports the state of tasks within a stage
- The **Storage** tab shows the size of RDDs or DataFrames that persisted to memory or disk
- The **Environment** tab information includes any environment variables and system properties for Spark or the JVM
- The **Executors** tab displays a summary that shows memory and disk usage for any executors in use
- If the application runs SQL queries, select the **SQL** tab and the **Description** hyperlink to display a query's details

Monitoring Application Progress

Monitoring an application using the Spark Application UI provides these benefits:

- Quickly identify failed jobs and tasks
- Fast access to locate inefficient operations

- Application workflow optimization

How does the application flow

- Multiple related jobs
  - From different data sources
  - One or more DataFrames
  - Actions applied to the DataFrames
- Workflows can include:
  - Jobs created by the SparkContext in the driver program
  - Jobs in progress running as tasks in the executors
  - Completed jobs transferring resutls back to the driver or writing to disk

How Do Jobs Progress?

1. Spark jobs divide into stages, which connect as a directed acyclic graph (DAG)
2. Tasks for the current stage are scheduled on the cluster
3. As the stage completes all its tasks, the next dependent stage in the DAG begins
4. The job continues through the DAG until all stages complete
   If any tasks within a stage fail, after several attempts, Spark marks the task, stage, and job as failed and stops the application

- The application jobs complete
- The SparkContext stops
- The application UI is no longer available
- If event logging is enabled, view the application UI by starting the Spark History Server and connecting to it

Viewing the UI with History Server
Before the application is started, verify that event logging is enabled
`spark.eventLog.enabled true`
`spark.eventLog.dir <path-for-log-files>`
View the application UI by connecting to Spark History Server
`http://<host-url>:18080`
Start the History Server
`./sbin/start-history-server.sh`

The Spark application workflow includes jobs crated by the SparkContext in the driver program, jobs in progress running as taks in the executors, and completed jobs transferring results back to the driver or writing to disk

The Spark application UI centralizes critical information, including status information. You can quickly identify failures, then drill down to the lowest levels of the application to discover their root causes

Debugging Apache Spark Application Issues

Common Application Issues

- User Code
  - Driver program: code that runs in the driver process
  - Serialized closures contain the code's necessary functions, classes, and variables
  - The serialized closures are distributed into the cluster to run in parallel by the executors
  - User code can error in the driver: syntax, serialization, data validation, errors located in other code locations. Reports task errors to the driver program and the application will terminate
- Configuration
- Application Dependencies
  - Source files examples: Python script files, Java JAR files
  - Required data files
  - Application libraries
  - Dependencies myust be available for all nodes of the cluster
- Resource Allocation
  - CPU and memory resources must be available for tasks to run
  - Driver and executer processes require some amount of CPU and memory to run
  - Any worker with free resources can start processes
  - Resources are acquired until a process completes
  - If resources are not available, Spark retries until a worker is free
  - Lack of resources result in task time-outs, also called task starvation
- Network Communication

Exaining the log Files

- Application logs are found in `work/` directory named as `work/<application-id>/<stdout|error>`
- Spark standalone wirtes `master/worker` logs to the `log/` directory

Understanding Memory Resources

Configuring Spark Process Memory

Spark applications allow configuration of driver and executor memory

> Upper limit on useable memory enables apps to run without using all available cluster memory
> Exceeding memory requirementes causes disk spill or out-of-memory errors

Memory Setting Considerations

- Executor Memory:
  - Processing
  - Caching
  - Excessive caching leads to issues
- Driver Memory
  - Loads data, broadcasts variables
  - Handles results, such as collections

Spark Data Persistence

Spark data persistence:

- Store intermediate calculations
- Persist to memory/disk
- Spark has configurable memory for executor and driver processes
- Executor memory and Storage memory share a region that can be tuned as needed
- Caching data can help improve application performance

Understanding Processor Resources

- Sparks assigns CPU cores to driver and executor processes
- Parallelist is limited by the number of cores available
- Executor process tasks in parallel up to the number of cores assigned to the application
- After processing, CPU cores become available for future tasks
- Workers in the cluster contain a limited number of cores
- If no cores are available to an application, the application must wait for currently running tasks to finish

Default CPU Core Usage

- Spark queues tasks and waits for avaialable executors and cores for maximized parallel processing
- Parallel processing tasks mainly depend on the number of data partitions and operations
- Application settings will override default behavior

Setting Executor Cores on Submit

- Specifies the number of executor cores for a Spark standalone cluster **per executor process**

```
$ ./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://<spark-master-URL>:7077 \
  --executor-cores 8 \
  /path/to/examples.jar \
  1000
```

- Specifies the executor cores for a Spark standalone cluster **for the application**

```
$ ./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://<spark-master-URL>:7077 \
  --total-executor-cores 50 \
  /path/to/examples.jar \
  1000
```

Setting worker node resources

- By default, Spark uses all avaialable memory minus 1GB and all available cores

```
# Start standalone worker with MAX 10Gb memory, 8
cores
$ ./sbin/start-worker.sh \
    spark://<spark-master-URL> \
    --memory 10G --cores 8
```

- Spark assigns CPU cores to driver and executor processes during application processing
- Executors process tasks in parallel according to the number of cores available or assigned by the application
- If using the Spark Standalone cluster manager, you can specify the total memory and CPU cores workers can use

Summary

To connect to the Apache Spark user interface web server, start your application and connect to the application UI using the
following URL: http://:4040

The Spark application UI centralizes critical information, including status information into the Jobs, Stages, Storage, Environment and Executors tabbed regions. You can quickly identify failures, then drill down to the lowest levels of the application to discover their root causes. If the application runs SQL queries, select the SQL tab and the Description hyperlink to display the query's details.

The Spark application workflow includes jobs created by the Spark Context in the driver program, jobs in progress running as tasks in the executors, and completed jobs transferring results back to the driver or writing to disk.

Common reasons for application failure on a cluster include user code, system and application configurations, missing dependencies, improper resource allocation, and network communications. Application log files, located in the Spark installation directory, will often show the complete details of a failure.

User code specific errors include syntax, serialization, data validation. Related errors can happen outside the code If a task fails due to an error, Spark can attempt to rerun tasks for a set number of retries. If all attempts to run a task fail, Spark reports an error to the driver and terminates the application. The cause of an application failure can usually be found in the driver event log.

Spark enables configurable memory for executor and driver processes. Executor memory and Storage memory share a region that can be tuned.

Setting data persistence by caching data is one technique used to improve application performance.

The following code example illustrates configuration of executor memory on submit for a Spark Standalone cluster:

```
$ ./bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master
spark://<spark-master-URL>:7077 \
--executor-memory 10G \
/path/to/examples.jar \1000
```

The following code example illustrates setting Spark Standalone worker memory and core parameters:

```
Start standalone worker with MAX 10Gb memory, 8 cores
$
./sbin/start-worker.sh \
```

```
spark://<spark-master-URL> \
––memory 10G ––cores 8
```

Spark assigns processor cores to driver and executor processes during application processing. Executors process tasks in parallel according to the number of cores available or as assigned by the application.

You can apply the argument '--executor-cores 8 \' to set executor cores on submit per executor. This example specifies eight cores.

You can specify the executor cores for a Spark standalone cluster for the application using the argument ''--total-executor-cores 50' followed by the number of cores for the application. This example specifies 50 cores.

When starting a worker manually in a Spark standalone cluster, you can specify the number of cores the application uses by using the argument '--cores' followed by the number of cores. Spark's default behavior is to use all available cores.