Auto-Scaling in Apache Storm

List Of Topics

1. Abstract
2. End Goals and Learning
3. Deliverables
4. System Used
5. Programming Languages
6. Overview Of Implementation
7. Methodologies of Evaluation
8. Scope and Work Plan
9. Conclusion
10. References
Abstract

Storm is an Apache incubator project which was originally created by Twitter. It was built to process 'tweets' at real-time and perform transformations on the incoming data. Since being made open-source the use-cases for Storm have increased exponentially.

Spouts in the Storm cluster can produce unbounded data. But, being a real-time system, Storm needs to complete its assigned work-load in a reasonable time frame. If a work-load takes too long to finish, the property of being real-time will be violated.

Storm’s users are required to create a topology which guides the system to perform a desired set of tasks. Users are also expected to create data sources (Spouts) and worker processes (bolts) which Storm uses to drive its work. Furthermore, the degree of parallelism, which is needed to complete workloads on time, has to be determined manually.

It is a well-known problem that scaling large systems to work at a acceptable threshold is a trial-and-error process. It is very time-consuming and error-prone. In Storm, it has to be done for each topology that gets created. The goal of this project is to automate scaling within Storm by dynamically modifying parallelism hints. A new Rule-Based Feedback System will be used to achieve this.

Summary of Manual Scaling Process in the Current System
1. Storm Bolts have parallelism hints to scale to a desired threshold
2. Users need to use the CLI tool to change the parallelism of Spouts and Bolts till a desired threshold is reached
3. The UI Console can be used to check the threshold metrics each time they are changed

End Goals And Learning
1. Eliminate the need for manual scaling of bolts
   • Create an alternative approach to provide parallelism hints for Storm bolts
2. Allow Rule-based scaling for topologies – THRESHOLD_PER_SEC will be a single rule that will determine scaling for a topology
   • Auto-scaling the system when the threshold value changes in configuration (or in code) is out of scope for this project
3. The desired scaling (threshold) should be achieved within minutes of receiving the first tuple.
   • Manual scaling can even take hours to achieve desired thresholds in large topologies
   • Every time the input load increases, the system should scale-up automatically
   • Scale-down during low loads is out of scope for this project
4. Learn and compare performance against existing methodologies
   • Automated topology against default manual topologies
   • Automated topology against manual tuning
5. Learn and compare performance against other methodologies
   • Automated topology with feedback system against linear regression
   • Automated topology with feedback system against FFT scaling algorithms (Will fit into scope if possible)
6. Learn and compare performance against similar systems
   • Best manual topology Vs Automated topology Vs Spark Streaming System (nice to have)

**Deliverables**

**Project Report**
- Progress from start to finish
- Documentation of new feature
  - Setup of Nimbus, Threshold Monitor, Zookeeper, Spouts and Bolts
  - Metrics which will be used by the Threshold Monitor
  - CLI script - Command line interface for dynamic scaling
  - Scaling Algorithm
  - Data points for logging
  - Interesting insights/problems encountered - solved, left for future work
- Independent data from all experiments! (with environment information, data loads etc)
- Comparison charts on the performance of auto-scaling Vs other systems for varying data loads and data types
- Metrics for auto-scaling feature (not limited to this list) - StartUp Time of threshold monitor, Time to reach threshold for a topology, Amount of data processed before reaching threshold, Time to adjust to dynamic changes in threshold (future), documentation overhead for each topology (logging), performance overhead of running new daemon on topologies, performance overhead of using the CLI client
- Suggest alternate techniques and algorithms to optimize speed

**Code base**
- A branch from the Storm Git Repo with auto-scaling changes

**Apache Foundation**
- JIRA task for the project showing progress

**Meetings**
- All communication with members of the Apache Storm community
- Detailed minutes of meeting with Professor. Nick during the course of the project

**System Used**
Apache Storm
Overview of Implementation

The Threshold Monitor will intelligently allocate the resources of a bolt (threads and tasks) to reach a pre-defined processing threshold. The threshold should be provided as a configuration value in the Storm yaml file or in the Topology Builder (in code).

A daemon process (ThresholdMonitor) will be started in parallel to Nimbus (or within Nimbus) to monitor threshold. It will employ a feedback system for tuning resources and reach the pre-defined threshold. The feedback system will contain three components: an auto-scaling algorithm, a data point to the Storm MetricsProvider and, a data point to the Logging system.

The MetricsProvider will provide information to calculate the current threshold of the topology. As the daemon runs periodically, the current threshold will be fed into the scaling algorithm. This value will be used to calculate the percentage change in threshold compared to the previous run. The change can be an increase, decrease or remain same. It determines the next step in auto-tuning. Every run of the daemon will produce a tuning change, unless the threshold has already been reached or further scaling is not possible. If a change is made to tune performance, it will be logged per-topology.

Dynamic changes to the parallelism of bolts can be done using the Storm CLI (command-line interface) client. The feedback system will attempt to invoke the same code that CLI runs within Storm. If not possible, a script will be run to invoke CLI in a head-less mode. If the second method is chosen, the time-overhead of invoking the CLI client will be documented independent of the actual running time.

Example for Auto-Tuning:

Assume that the user has configured a threshold of 1000 tuples/sec through the configuration parameter, THRESHOLD_PER_SEC. Also, a topology required for processing the input data is created with no parallelism hints for bolts.

Once the topology is submitted two things happen. Nimbus allocates a default parallelism of one thread/one task for all bolts. The Threshold Monitor will receive the topology and its metadata.

Once the spouts and bolts are setup, the Threshold Monitor calculates the current threshold for the topology. Assume the current value is 500 tuples/s. It will then dynamically increases the number of tasks/threads/nodes in a “ideal bolt” to process
more tuples. After a pre-defined interval, the Threshold Monitor runs again and recalculates the current threshold. Assume that the value increases to 1100 tuples/sec. Since the desired threshold has been reached, no scaling is done for the topology. On the other hand, if the threshold is 700, the former process is repeated. By continuously updating the node configurations this way, the threshold can be gradually increased to reach the desired value. There could always be a decrease in performance due to too many threads/tasks or no change. In this case, the algorithm reverts its previous action and selects the next “ideal bolt” to work with.

Choosing the “ideal bolt” will be a key feature of the scaling algorithm. There is no clear answer to the problem. The selection can even vary depending on the kind of work each bolt is doing. In the proposed algorithm, the ideal bolt will be selected based on a breadth-first scan of the topology from each spout to its last bolt(s).

Architecture Diagram

Methodologies for Evaluation
The new feature will be tested in stages. Some of the later tests are nice to have and may not fit into the scope of the project.

**Test 1: Run a split-sentence / aggregate-word-count test over bounded data**

One Spout and two bolts
Spout: The complete collection of Harry potter books
- Determine a threshold before starting the test (Example: 5000 tuples/sec)
- First run the test in default mode. Each bolt has one executor and one task
- Record performance in terms of time taken, cpu used, nodes used etc
- Run the same test with a random parallelism hint in the topology. Try to manually increase/decrease the parallelism using CLI.
- Record time taken to achieve desired parallelism and other metrics as above
- Finally, run the auto-scaling module and record the performance
- Record whether auto-scaling did not use more resources than manual tuning to achieve the threshold
- Generate a report which compares the three cases

**Test 2: Use Twitter FireHose API to get real-time tweets. Find top 3 trending topics in an unbounded time frame**

Multiple spouts and multiple bolts
Spouts: Create dummy twitter account and automate tweets for 4 hours
- Run the same cases as above
- Generate a report which compares all the cases

**Test 3: Compare Spark Streaming Vs Manual Storm Tuning Vs Auto-Scaled Storm**
Setup one of the above tests to run against a Spark Streaming system. It will take some time to understand Spark’s methodologies for scaling and parallelism. This test will be subject availability of sufficient time.

**Nice to have:**
**Test 4: Compare Linear Regression Vs Feedback system**
**Test 5: Compare FFT Vs Feedback system**
Scope and Work Plan

November 1 2014 to January 4 2015 (9 weeks)

Week 1 & Week 2: Research Storm, Test Research Findings (Nov 1 to Nov 14)

- Research into storm architecture/infrastructure
- Determine the process by which executors and tasks are dynamically changed per spout and/or bolt
  - Already determined that the Storm CLI client allows this
  - Find out overhead/time-to-complete parameters of calling CLI through a script
  - Alternatively, find out if the code that CLI internally invokes can be called directly by the Feedback System

- Determine how to access available metrics for throughput.
  - Storm exposes metrics through the internal implementation that should be sufficient for our formula to calculate current threshold
  - Additional metrics can be added as well
  - Need to find out if the metrics are topology-wide? Do we have metrics at the bolt-level
  - http://storm.incubator.apache.org/documentation/Metrics.html

- Determine how to run a daemon task parallel to Nimbus and send topologies to it

- Determine how to connect to the logging framework in Storm. Is it possible to build separate logger for scaling information?
- Determine how to access the logged information using keywords

- Set up a Storm cluster combining AWS and personal systems.
  - Install Nimbus, workers and the Feedback system daemon
  - Find contacts in the storm community that we can ask for feedback. Get added to the storm mailing list.
  - Create JIRA issue in Storm for this feature
  - Learning Clojure and storm-specific Java APIs and begin report

Week 3: Algorithm for parallelism, Test existing features (Nov 15 to Nov 21)
• Design the algorithm for automating parallelism (core completed)
  • Parallelism can be increased by the following order of granularity
    • Tasks
    • Executors/Threads
    • Bolts
  • Run sample topologies to gain insights about performance in a single-spout single-bolt environment. (1 worker with 1 executor and 1 task)
    • Storm runs in the above default configuration when no parallelism is provided in the topology.
    • Storm has a UI console in Nimbus which gives the health of a topology
  • Test Metrics and Logging
  • Document findings in Report
  • Meet with Professor Nick to discuss progress

**Week 4: Implementation and First Test (Nov 22 to Nov 30)**
• Begin implementing solution
  • Define interfaces for interaction with the system
  • Build the Threshold Monitor module
  • Conduct Test 1 from above. Test the new feature with the harry Potter book collection as spouts. We plan to reuse the “split sentence/aggregate word count” example
  • Document experiment results in report

**Week 5: Implementation and Second Test (Dec 1 to Dec 7)**
• Bulk of implementation
• Run second test with the Twitter Firehose API to work with live tweets and find trending topics
• Request feedback from storm community
• Meet with Professor Nick to discuss progress
• Update progress in report and details of any changes that the Professor and/or the Apache community suggests.

**Week 6: (Dec 8 to Dec 15)**
• Continue working on implementation, ideally get it to a near finished state.
• Track of further work and issues we encounter in the report
• Prepare for final test of the system.
• Determine the metrics and overhead of running our system. Is it computationally expensive or prohibitively slow to use it when creating new topologies?
• Meet with Professor Nick to discuss progress
Week 7: (Dec 16 - Dec 24)

- Finish implementation
- Perform tests comparing a static storm topology to one that can increase dynamically.
- Things to measure include but are not limited to:
  - Time to completion for a known set of tuples. How much faster is it?
  - Physical machine resource usage. How much more does this method cost us?

Week 8 and Week 9: (Dec 25 - Jan 4)

- Try to compare with Spark Streaming and Storm Auto-Scaling with Linear regression (Nice to have)
- Submit report to Professor. Nick
- Submit the feature to Apache

Conclusion

Process automation is a powerful tool to build on Big Data systems. One of the biggest challenges in these systems is to scale load and complete within a certain time and for varying types of loads. My research shows that this be a valuable addition to the Storm community. It will also serve as a template on future auto-scaling systems.

References

https://storm.apache.org/
http://storm.incubator.apache.org/documentation/Understanding_the_parallelism_of_a-Storm_topology.html
http://storm.incubator.apache.org/documentation/Configuration.html
http://storm.incubator.apache.org/documentation/Metrics.html
https://news.ycombinator.com/item?id=3014597
https://groups.google.com/forum/#!topic/storm_user/IDv0yaTJLLU
https://groups.google.com/forum/#!searchin/storm_user/auto$20scaling/storm_user/Rr9K7f-AMLc/LLZyoe9TITkI
https://groups.google.com/forum/#!searchin/storm_user/auto$20scaling/storm_user/IDv0yaTJLLU/uI3GAM_lywJ
https://groups.google.com/forum/#!searchin/storm_user/auto$20scaling/storm_user/Ix_2gFZ5u0A/tajDtPH_nNoI
http://techblog.netflix.com/2013/12/scryer-netflixs-predictive-auto-scaling.html
http://www.sersc.org/journals/IJHIT/vol5_no2_2012/23.pdf
https://scalr-wiki.atlassian.net/wiki/display/docs/Autoscaling+Algorithms