Mastering apache storm pdf editor software reviews ratings

I'm not robot!

rate_reviewWrite a Reviewfile_downloadDownload PDFOverviewReviewsLikes and Dislikes VDOC.PUB Authors: JAIN , ANKIT PDF Download Embed This document was uploaded by our user. The uploader already confirmed that they had the permission to publish it. If you are author/publisher or own the copyright of this documents, please report to us by using this DMCA report form. Report DMCA Mastering Apache Storm Processing big data streams in real time Ankit Jain BIRMINGHAM - MUMBAI Mastering Apache Storm Copyright © 2017 Packt Publishing All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews. Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book. Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information. First published: August 2017 Production reference: 1140817 Published by Packt Publishing Ltd. Livery Place 35 Livery Street Birmingham B3 2PB, UK. ISBN 978-1-78712-563-6 www.packtpub.com Credits Author Ankit Jain Copy Editor Safis Editing Reviewers Doug Ortiz Oleg Okun Project Coordinator Nidhi Joshi Commissioning Editor Veena Pagare Proofreader Safis Editing Acquisition Editor Divya Poojari Indexer Tejal Daruwale Soni Content Development Editor Mayur Pawanikar Graphics Tania Dutta Technical Editor Dinesh Pawar Production Coordinator Arvindkumar Gupta About the Author Ankit Jain holds a bachelor's degree in computer science and engineering. He has 6 years, experience in designing and architecting solutions for the big data domain and has been involved with several complex engagements. His technical strengths include Hadoop, Storm, S4, HBase, Hive, Sqoop, Flume, Elasticsearch, machine learning, Kafka, Spring, Java, and J2EE. He also shares his thoughts on his personal blog. You can follow him on Twitter at @mynameisanky. He spends most of his time reading books and playing with different technologies. When not at work, he spends time with his family and friends watching movies and playing games. About the Reviewers Doug Ortiz is a senior big data artechitect at ByteCubed who has been architecting, developing and integrating enterprise solutions throughout his whole career. Organizations that leverage high skillset have been able to rediscover and reuse their underutilized data via existing and emerging technologies such as: Microsoft BI Stack, Hadoop, NoSQL Databases, SharePoint, Hadoop, related toolsets and technologies. He is the founder of Illustris, LLC and can be reached at Interesting aspects of his profession: He has experience integrating multiple platforms and products He has big data, data science certifications, R and Python certifications He helps organizations gain a deeper understanding of, and value, their current investments in data and existing resources, turning them into useful sources of information He has improved, salvaged, and architected projects by utilizing unique and innovative techniques His hobbies are yoga and scuba diving. Oleg Okun is a machine learning expert and the author/editor of four books, numerous journal articles, and conference papers. His career spans more than a quarter of a century. He was employed in both academia and industry in his mother country, Belarus, and abroad (Finland, Sweden, and Germany). His work experience includes document image analysis, fingerprint biometrics, bioinformatics, online/offline marketing analytics, credit scoring analytics, and text analytics. He is interested in all aspects of distributed machine learning and the Internet of Things. Oleg currently lives and works in Hamburg, Germany. I would like to express my deepest gratitude to my parents for everything that they have done for me. www.PacktPub.com For support files and downloads related to your book, please visit www.PacktPub.com. Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com, and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at for more details. At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks. Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career. Why subscribe? Fully searchable across every book published by Packt Copy and paste, print, and bookmark content On demand and accessible via a web browser Customer Feedback Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at . If you'd like to join our team of regular reviewers, you can e-mail us at We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products. Table of Contents Preface Chapter 1: Real-Time Processing and Storm Introduction Apache Storm Features of Storm Storm components Nimbus Supervisor nodes The ZooKeeper cluster The Storm data model Definition of a Storm topology Operation modes in Storm Programming languages Summary Chapter 2: Storm Deployment, Topology Development, and Topology Options Storm prerequisites Installing Java SDK 7 Deployment of the ZooKeeper cluster Setting up the Storm cluster Developing the hello world example The different options of the Storm topology Deactivate Activate Rebalance Kill Dynamic log level settings Walkthrough of the Storm UI Cluster Summary section Nimbus Summary section Supervisor Summary section Nimbus Configuration section Topology Summary section Dynamic log level settings 1 7 8 9 10 10 10 11 11 12 16 17 17 18 18 19 20 24 26 32 32 33 33 33 34 34 35 35 36 36 37 40 Updating the log level from the Storm UI Updating the log level from the Storm CLI Summary 40 42 43 Chapter 3: Storm Parallelism and Data Partitioning 44 Parallelism of a topology Worker process Executor Task Configure parallelism at the code level Worker process, executor, and task distribution Rebalance the parallelism of a topology Rebalance the parallelism of a SampleStormClusterTopology topology Different types of stream grouping in the Storm cluster Shuffle grouping Field grouping All grouping Global grouping Direct grouping Local or shuffle grouping None grouping Custom grouping Guaranteed message processing Tick tuple Summary Chapter 4: Trident Introduction 44 44 45 45 46 47 48 49 50 51 51 52 52 53 53 54 54 55 57 59 60 Trident introduction Understanding Trident's data model Writing Trident functions, filters, and projections Trident function Trident filter Trident projection Trident repartitioning operations Utilizing shuffle operation Utilizing partitionBy operation Utilizing global operation Utilizing broadcast operation Utilizing batchGlobal operation Utilizing partition operation [ii] 60 61 61 61 63 64 65 65 65 67 68 69 70 Trident aggregator partitionAggregate aggregate 71 71 72 73 73 74 75 76 77 78 78 ReducerAggregator Aggregator CombinerAggregator persistentAggregate Aggregator chaining Utilizing the groupBy operation When to use Trident Summary Chapter 5: Trident Topology and Uses Trident groupBy operation groupBy before partitionAggregate groupBy before aggregate Non-transactional topology Trident hello world topology Trident state Distributed RPC When to use Trident Summary Chapter 6: Storm Scheduler 79 80 80 80 81 83 88 89 93 93 94 Introduction to Storm scheduler Default scheduler Isolation scheduler Resource-aware scheduler Component-level configuration Memory usage example CPU usage example Worker-level configuration Node-level configuration Global component configuration Custom scheduler Configuration changes in the supervisor node Configuration setting at component level Writing a custom supervisor class Converting component IDs to executors Converting supervisors to slots Registering a CustomScheduler class [iii] 94 94 95 96 96 97 97 98 98 98 99 100 100 100 107 109 113 Summary 113 Chapter 7: Monitoring of Storm Cluster Cluster statistics using the Nimbus thrift client Fetching information with Nimbus thrift Monitoring the Storm cluster using JMX Monitoring the Storm cluster using Ganglia Summary 114 114 115 124 127 136 Chapter 8: Integration of Storm and Kafka 137 Introduction to Kafka Kafka architecture Producer Replication Consumer Broker Data retention Installation of Kafka brokers Setting up a single node Kafka cluster Setting up a three node Kafka cluster 137 138 138 139 139 140 140 140 141 144 145 146 147 151 158 159 Multiple Kafka brokers on a single node Share ZooKeeper between Storm and Kafka Kafka producers and publishing data into Kafka Kafka Storm integration Deploy the Kafka topology on Storm cluster Summary Chapter 9: Storm and Hadoop Integration Introduction to Hadoop Hadoop Common Hadoop Distributed File System Namenode Datanode HDFS client Secondary namenode YARN ResourceManager (RM) NodeManager (NM) ApplicationMaster (AM) Installation of Hadoop Setting passwordless SSH Getting the Hadoop bundle and setting up environment variables [iv] 160 160 161 161 161 162 162 162 164 164 164 164 165 166 167 Setting up HDFS Setting up YARN Write Storm topology to persist data into HDFS Integration of Storm with Hadoop Setting up Storm-YARN Storm-Starter topologies on Storm-YARN Summary Chapter 10: Storm Integration with Redis, Elasticsearch, and HBase Integrating Storm with HBase Integrating Storm with Redis Integrating Storm with Elasticsearch Integrating Storm with Esper Summary Chapter 11: Apache Log Processing with Storm Apache log processing elements Producing Apache log in Kafka using Logstash Installation of Logstash What is Logstash? Why are we using Logstash? Installation of Logstash Configuration of Logstash Why are we using Kafka between Logstash and Storm? Splitting the Apache log line Identifying country, operating system type, and browser type from the log file Calculate the search keyword Persisting the process data Kafka spout and define topology Deploy topology MySQL queries Calculate the page hit from each country Calculate the count for each browser Calculate the count for each operating system Summary Chapter 12: Twitter Tweet Collection and Machine Learning Exploring machine learning Twitter sentiment analysis Using Kafka producer to store the tweets in a Kafka cluster [v] 168 171 173 177 178 182 184 185 185 196 201 206 212 213 213 214 214 215 215 215 215 216 217 220 224 227 232 236 238 238 239 240 240 241 241 243 243 Kafka spout, sentiments bolt, and HDFS bolt Summary Index 247 256 257 [vi] Preface Real-time data processing in no longer a luxury exercised by a few big companies but has become a necessity for businesses that want to compete, and Apache Storm is one of the de facto standards for developing real-time processing pipelines. The key features of Storm are that it is horizontally scalable, is fault tolerant, and provides guaranteed message processing. Storm can solve various types of analytic problem: machine learning, log processing, graph analysis, and so on. Mastering Storm will serve both as a getting started guide to inexperienced developers and as a reference for implementing advanced use cases with Storm for experienced developers. In the first two chapters, you will learn the basics of a Storm topology and various components of a Storm cluster. In the later chapters, you will learn how to build a Storm application that can interact with various other big data technologies and how to create transactional topologies. Finally, the last two chapters cover case studies for log processing and machine learning. We are also going to cover how we can use the Storm scheduler to assign delicate work to delicate machines. What this book covers Chapter 1, Real-Time Processing and Storm Introduction, gives an introduction to Storm and its components. Chapter 2, Storm Deployment, Topology Development, and Topology Options, covers deploying Storm into the cluster, deploying the sample topology on a Storm cluster, how we can monitor the storm pipeline using storm UI, and how we can dynamically change the log level settings. Chapter 3, Storm Parallelism and Data Partitioning, covers the parallelism of topology, how to configure parallelism at the code level, guaranteed message processing, and Storm internally generated tuples. Chapter 4, Trident Introduction, covers an introduction to Trident, an understanding of the Trident data model, and how we can write Trident filters and functions. This chapter also covers repartitioning and aggregation operations on Trident tuples. Preface Chapter 5, Trident Topology and Uses, introduces Trident tuple grouping, non-transactional topology, and a sample Trident topology. The chapter also introduces Trident state and distributed RPC. Chapter 6, Storm Scheduler, covers different types of scheduler available in Storm: the default scheduler, isolation scheduler, resource-aware scheduler, and custom scheduler. Chapter 7, Monitoring of the Storm Cluster, covers monitoring Storm by writing custom monitoring UIs using the stats published by Nimbus. We explain the integration of Ganglia with Storm using JMXTrans. This chapter also covers how we can configure Storm to publish JMX metrics. Chapter 8, Integration of Storm and Kafka, shows the integration of Storm with Kafka. This chapter starts with an introduction to Kafka, covers the installation of Storm, and ends with the integration of Storm with Kafka to solve any real-world problem. Chapter 9, Storm and Hadoop Integration, covers an overview of Hadoop, writing the Storm topology to publish data into HDFS, an overview of Storm-YARN, and deploying the Storm topology on YARN. Chapter 10, Storm Integration with Redis, Elasticsearch, and HBase, teaches you how to integrate Storm with various other big data technologies. Chapter 11, Apache Log Processing with Storm, covers a sample log processing application in which we parse Apache web server logs and generate some business information from log files. Chapter 12, Twitter Tweets Collection and Machine Learning, walks you through a case study implementing a machine learning topology in Storm. What you need for this book All of the code in this book has been tested on CentOS 6.5. It will run on other variants of Linux and Windows as well with appropriate changes in commands. [2] Preface We have tried to keep the chapters self-contained, and the setup and installation of all the software used in each chapter are included in the chapter itself. These are the software packages used throughout the book: CentOS 6.5 Oracle JDK 8 Apache ZooKeeper 3.4.6 Apache Storm 1.0.2 Eclipse or Spring Tool Suite Elasticsearch 2.4.4 Hadoop 2.2.2 Logstash 5.4.1 Kafka 0.9.0.1 Esper 5.3.0 Who this book is for If you are a Java developer and are keen to enter into the world of real-time stream processing applications using Apache Storm, then this book is for you. No previous experience in Storm is required as this book starts from the basics. After finishing this book, you will be able to develop not-so-complex Storm applications. Conventions In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning. Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Add the following line in the storm.yaml file of the Nimbus machine to enable JMX on the Nimbus node." A block of code is set as follows: org.apache.storm storm-core 1.0.2 provided [3] Preface Any command-line input or output is written as follows: cd $ZK_HOME/conf touch zoo.cfg New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Now, click on the Connect button to view the metrics of the supervisor node." Warnings or important notes appear like this. Tips and tricks appear like this. Reader feedback Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail , and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors. Customer support Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase. [4] Preface Downloading the example code You can download the example code files for this book from your account at acktpub.com. If you purchased this book elsewhere, you can visit om/support, and register to have the files e-mailed directly to you. You can download the code files by following these steps: 1. 2. 3. 4. 5. 6. 7. Log in or register to our website using your e-mail address and password. Hover the mouse pointer on the SUPPORT tab at the top. Click on Code Downloads & Errata. Enter the name of the book in the Search box. Select the book for which you're looking to download the code files. Choose from the drop-down menu where you purchased this book from. Click on Code Download. Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of: WinRAR / 7-Zip for Windows Zipeg / iZip / UnRarX for Mac 7-Zip / PeaZip for Linux The code bundle for the book is also hosted on GitHub at ishing/Mastering-Apache-Storm. We also have other code bundles from our rich catalog of books and videos available at . Check them out! Downloading the color images of this book We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from loads/MasteringApacheStorm_ColorImages.pdf. [5] Preface Errata Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the codewe would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title. To view the previously submitted errata, go to and enter the name of the book in the search field. The required information will appear under the Errata section. Piracy Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy. Please contact us at with a link to the suspected pirated material. We appreciate your help in protecting our authors and our ability to bring you valuable content. Questions If you have a problem with any aspect of this book, you can contact us at , and we will do our best to address the problem. [6] 1 Real-Time Processing and Storm Introduction With the exponential growth in the amount of data being generated and advanced datacapturing capabilities, enterprises are facing the challenge of making sense out of this mountain of raw data. On the batch processing front, Hadoop has emerged as the go-to framework to deal with big data. Until recently, there has been a void when one looks for frameworks to build real-time stream processing applications. Such applications have become an integral part of a lot of businesses as they enable them to respond swiftly to events and adapt to changing situations. Examples of this are monitoring social media to analyze public response to any new product that you launch and predicting the outcome of an election based on the sentiments of election-related posts. Organizations are collecting a large volume of data from external sources and want to evaluate/process the data in real time to get market trends, detect fraud, identify user behavior, and so on. The need for real-time processing is increasing day by day and we require a real-time system/platform that should support the following features: Scalable: The platform should be horizontally scalable without any down time. Fault tolerance: The platform should be able to process the data even after some of the nodes in a cluster go down. No data lost: The platform should provide the guaranteed processing of messages. High throughput: The system should be able to support millions of records per second and also support any size of messages. Real-Time Processing and Storm Introduction Easy to operate: The system should have easy installation and operation. Also, the expansion of clusters should be an easy process. Multiple languages: The platform should support multiple languages. The end user should be able to write code in different languages. For example, a user can write code in Python, Scala, Java, and so on. Also, we can execute different language code inside the one cluster. Cluster isolation: The system should support isolation so that dedicated processes can be assigned to dedicated machines for processing. Apache Storm Apache Storm has emerged as the platform of choice for industry leaders to develop distributed, real-time, data processing platforms. It provides a set of primitives that can be used to develop applications that can process a very large amount of data in real time in a highly scalable manner. Storm is to real-time processing what Hadoop is to batch processing. It is open source software, and managed by Apache Software Foundation. It has been deployed to meet realtime processing needs by companies such as Twitter, Yahoo!, and Flipboard. Storm was first developed by Nathan Marz at BackType, a company that provided social search applications. Later, BackType was acquired by Twitter, and it is a critical part of their infrastructure. Storm can be used for the following use cases: Stream processing: Storm is used to process a stream of data and update a variety of databases in real time. This processing occurs in real time and the processing speed needs to match the input data speed. Continuous computation: Storm can do continuous computation on data streams and stream the results to clients in real time. This might require processing each message as it comes in or creating small batches over a short time. An example of continuous computation is streaming trending topics on Twitter into browsers. Distributed RPC: Storm can parallelize an intense query so that you can compute it in real time. Real-time analytics: Storm can analyze and respond to data that comes from different data sources as they happen in real time. [8] Real-Time Processing and Storm Introduction In this chapter, we will cover the following topics: What is a Storm? Features of Storm Architecture and components of a Storm cluster Terminologies of Storm Programming language Operation modes Features of Storm The following are some of the features of Storm that make it a perfect solution to process streams of data in real time: Fast: Storm has been reported to process up to 1 million tuples/records per second per node. Horizontally scalable: Being fast is a necessary feature to build a high volume/velocity data processing platform, but a single node will have an upper limit on the number of events that it can process per second. A node represents a single machine in your setup that executes Storm applications. Storm, being a distributed platform, allows you to add more nodes to your Storm cluster and increase the processing capacity of your application. Also, it is linearly scalable, which means that you can double the processing capacity by doubling the nodes. Fault tolerant: Units of work are executed by worker processes in a Storm cluster. When a worker dies, Storm will restart that worker, and if the node on which the worker is running dies, Storm will restart that worker on some other node in the cluster. This feature will be covered in more detail in Chapter 3, Storm Parallelism and Data Partitioning. Guaranteed data processing: Storm provides strong guarantees that each message entering a Storm process will be processed at least once. In the event of failures, Storm will replay the lost tuples/records. Also, it can be configured so that each message will be processed only once. Easy to operate: Storm is simple to deploy and manage. Once the cluster is deployed, it requires little maintenance. Programming language agnostic: Even though the Storm platform runs on Java virtual machine (JVM), the applications that run over it can be written in any programming language that can read and write to standard input and output streams. [9] Real-Time Processing and Storm Introduction Storm components A Storm cluster follows a master-slave model where the master and slave processes are coordinated through ZooKeeper. The following are the components of a Storm cluster. Nimbus The Nimbus node is the master in a Storm cluster. It is responsible for distributing the application code across various worker nodes, assigning tasks to different machines, monitoring tasks for any failures, and restarting them as and when required. Nimbus is stateless and stores all of its data in ZooKeeper. There is a single Nimbus node in a Storm cluster. If the active node goes down, then the passive node will become an Active node. It is designed to be fail-fast, so when the active Nimbus dies, the passive node will become an active node, or the down node can be restarted without having any effect on the tasks already running on the worker nodes. This is unlike Hadoop, where if the JobTracker dies, all the running jobs are left in an inconsistent state and need to be executed again. The Storm workers can work smoothly even if all the Nimbus nodes go down but the user can't submit any new jobs into the cluster or the cluster will not be able to reassign the failed workers to another node. Supervisor nodes Supervisor nodes are the worker nodes in a Storm cluster. Each supervisor node runs a supervisor daemon that is responsible for creating, starting, and stopping worker processes to execute the tasks assigned to that node. Like Nimbus, a supervisor daemon is also failfast and stores all of its states in ZooKeeper so that it can be restarted without any state loss. A single supervisor daemon normally handles multiple worker processes running on that machine. [10] Real-Time Processing and Storm Introduction The ZooKeeper cluster In any distributed application, various processes need to coordinate with each other and share some configuration information. ZooKeeper is an application that provides all these services in a reliable manner. As a distributed application, Storm also uses a ZooKeeper cluster to coordinate various processes. All of the states associated with the cluster and the various tasks submitted to Storm are stored in ZooKeeper. Nimbus and supervisor nodes do not communicate directly with each other, but through ZooKeeper. As all data is stored in ZooKeeper, both Nimbus and the supervisor daemons can be killed abruptly without adversely affecting the cluster. The following is an architecture diagram of a Storm cluster: The Storm data model The basic unit of data that can be processed by a Storm application is called a tuple. Each tuple consists of a predefined list of fields. The value of each field can be a byte, char, integer, long, float, double, Boolean, or byte array. Storm also provides an API to define your own datatypes, which can be serialized as fields in a tuple. [11] Real-Time Processing and Storm Introduction A tuple is dynamically typed, that is, you just need to define the names of the fields in a tuple and not their datatype. The choice of dynamic typing helps to simplify the API and makes it easy to use. Also, since a processing unit in Storm can process multiple types of tuples, it's not practical to declare field types. Each of the fields in a tuple can be accessed by its name, getValueByField(String), or its positional index, getValue(int), in the tuple. Tuples also provide convenient methods such as getIntegerByField(String) that save you from typecasting the objects. For example, if you have a Fraction (numerator, denominator) tuple, representing fractional numbers, then you can get the value of the numerator by either using getIntegerByField("numerator") or getInteger(0). You can see the full set of operations supported by org.apache.storm.tuple.Tuple in the Java doc that is located at tml. Definition of a Storm topology In Storm terminology, a topology is an abstraction that defines the graph of the computation. You create a Storm topology and deploy it on a Storm cluster to process data. A topology can be represented by a direct acyclic graph, where each node does some kind of processing and forwards it to the next node(s) in the flow. The following diagram is a sample Storm topology: [12] Real-Time Processing and Storm Introduction The following are the components of a Storm topology: Tuple: A single message/record that flows between the different instances of a topology is called a tuple. Stream: The key abstraction in Storm is that of a stream. A stream is an unbounded sequence of tuples that can be processed in parallel by Storm. Each stream can be processed by a single or multiple types of bolts (the processing units in Storm, which are defined later in this section). Thus, Storm can also be viewed as a platform to transform streams. In the preceding diagram, streams are represented by arrows. Each stream in a Storm application is given an ID and the bolts can produce and consume tuples from these streams on the basis of their ID. Each stream also has an associated schema for the tuples that will flow through it. Spout: A spout is the source of tuples in a Storm topology. It is responsible for reading or listening to data from an external source, for example, by reading from a log file or listening for new messages in a queue and publishing them--emitting in Storm terminology into streams. A spout can emit multiple streams, each of a different schema. For example, it can read records of 10 fields from a log file and emit them as different streams of seven-fields tuples and four-fields tuples each. The org.apache.storm.spout.ISpout interface is the interface used to define spouts. If you are writing your topology in Java, then you should use org.apache.storm.topology.IRichSpout as it declares methods to use with the TopologyBuilder API. Whenever a spout emits a tuple, Storm tracks all the tuples generated while processing this tuple, and when the execution of all the tuples in the graph of this source tuple is complete, it will send an acknowledgement back to the spout. This tracking happens only if a message ID was provided when emitting the tuple. If null was used as the message ID, this tracking will not happen. A tuple processing timeout can also be defined for a topology, and if a tuple is not processed within the specified timeout, a fail message will be sent back to the spout. Again, this will happen only if you define a message ID. A small performance gain can be extracted out of Storm at the risk of some data loss by disabling the message acknowledgements, which can be done by skipping the message ID while emitting tuples. [13] Real-Time Processing and Storm Introduction The important methods of spout are: nextTuple(): This method is called by Storm to get the next tuple from the input source. Inside this method, you will have the logic of reading data from external sources and emitting them to an instance of org.apache.storm.spout.ISpoutOutputCollector. The schema for streams can be declared by using the declareStream method of org.apache.storm.topology.OutputFieldsDeclarer. If a spout wants to emit data to more than one stream, it can declare multiple streams using the declareStream method and specify a stream ID while emitting the tuple. If there are no more tuples to emit at the moment, this method will not be blocked. Also, if this method does not emit a tuple, then Storm will wait for 1 millisecond before calling it again. This waiting time can be configured using the topology.sleep.spout.wait.strategy.time.ms setting. ack(Object msgId): This method is invoked by Storm when the tuple with the given message ID is completely processed by the topology. At this point, the user should mark the message as processed and do the required cleaning up, such as removing the message from the message queue so that it does not get processed again. fail(Object msgId): This method is invoked by Storm when it identifies that the tuple with the given message ID has not been processed successfully or has timed out of the configured interval. In such scenarios, the user should do the required processing so that the messages can be emitted again by the nextTuple method. A common way to do this is to put the message back in the incoming message queue. open(): This method is called only once--when the spout is initialized. If it is required to connect to an external source for the input data, define the logic to connect to the external source in the open method, and then keep fetching the data from this external source in the nextTuple method to emit it further. [14] Real-Time Processing and Storm Introduction Another point to note while writing your spout is that none of the methods should be blocking, as Storm calls all the methods in the same thread. Every spout has an internal buffer to keep track of the status of the tuples emitted so far. The spout will keep the tuples in this buffer until they are either acknowledged or failed, calling the ack or fail method, respectively. Storm will call the nextTuple method only when this buffer is not full. Bolt: A bolt is the processing powerhouse of a Storm topology and is responsible for transforming a stream. Ideally, each bolt in the topology should be doing a simple transformation of the tuples, and many such bolts can coordinate with each other to exhibit a complex transformation. The org.apache.storm.task.IBolt interface is preferably used to define bolts, and if a topology is written in Java, you should use the org.apache.storm.topology.IRichBolt interface. A bolt can subscribe to multiple streams of other components--either spouts or other bolts--in the topology and similarly can emit output to multiple streams. Output streams can be declared using the declareStream method of org.apache.storm.topology.OutputFieldsDeclarer. The important methods of a bolt are: execute(Tuple input): This method is executed for each tuple that comes through the subscribed input streams. In this method, you can do whatever processing is required for the tuple and then produce the output either in the form of emitting more tuples to the declared output streams, or other things such as persisting the results in a database. You are not required to process the tuple as soon as this method is called, and the tuples can be held until required. For example, while joining two streams, when a tuple arrives you can hold it until its counterpart also comes, and then you can emit the joined tuple. [15] Real-Time Processing and Storm Introduction The metadata associated with the tuple can be retrieved by the various methods defined in the Tuple interface. If a message ID is associated with a tuple, the execute method must publish an ack or fail event using OutputCollector for the bolt, or else Storm will not know whether the tuple was processed successfully. The org.apache.storm.topology.IBasicBolt interface is a convenient interface that sends an acknowledgement automatically after the completion of the execute method. If a fail event is to be sent, this method should throw org.apache.storm.topology.FailedException. prepare(Map stormConf, TopologyContext context, OutputCollector collector): A bolt can be executed by multiple workers in a Storm topology. The instance of a bolt is created on the client machine and then serialized and submitted to Nimbus. When Nimbus creates the worker instances for the topology, it sends this serialized bolt to the workers. The work will desterilize the bolt and call the prepare method. In this method, you should make sure the bolt is properly configured to execute tuples. Any state that you want to maintain can be stored as instance variables for the bolt that can be serialized/deserialized later. Operation modes in Storm Operation modes indicate how the topology is deployed in Storm. Storm supports two types of operation modes to execute the Storm topology: Local mode: In local mode, Storm topologies run on the local machine in a single JVM. This mode simulates a Storm cluster in a single JVM and is used for the testing and debugging of a topology. Remote mode: In remote mode, we will use the Storm client to submit the topology to the master along with all the necessary code required to execute the topology. Nimbus will then take care of distributing your code. In the next chapter, we are going to cover both local and remote mode in more detail, along with a sample example. [16] Real-Time Processing and Storm Introduction Programming languages Storm was designed from the ground up to be usable with any programming language. At the core of Storm is a thrift definition for defining and submitting topologies. Since thrift can be used in any language, topologies can be defined and submitted in any language. Similarly, spouts and bolts can be defined in any language. Non-JVM spouts and bolts communicate with Storm over a JSON-based protocol over stdin/stdout. Adapters that implement this protocol exist for Ruby, Python, JavaScript, and Perl. You can refer to to find out about the implementation of these adapters. Storm-starter has an example topology, sources, which implements one of the bolts in Python. Summary In this chapter, we introduced you to the basics of Storm and the various components that make up a Storm cluster. We saw a definition of different deployment/operation modes in which a Storm cluster can operate. In the next chapter, we will set up a single and three-node Storm cluster and see how we can deploy the topology on a Storm cluster. We will also see different types of stream groupings supported by Storm and the guaranteed message semantic provided by Storm. [17] 2 Storm Deployment, Topology Development, and Topology Options In this chapter, we are going to start with deployment of Storm on multiple node (three Storm and three ZooKeeper) clusters. This chapter is very important because it focuses on how we can set up the production Storm cluster and why we need the high availability of both the Storm Supervisor, Nimbus, and ZooKeeper (as Storm uses ZooKeeper for storing the metadata of the cluster, topology, and so on)? The following are the key points that we are going to cover in this chapter: Deployment of the Storm cluster Program and deploy the word count example Different options of the Storm UI--kill, active, inactive, and rebalance Walkthrough of the Storm UI Dynamic log level settings Validating the Nimbus high availability Storm prerequisites You should have the Java JDK and ZooKeeper ensemble installed before starting the deployment of the Storm cluster. Storm Deployment, Topology Development, and Topology Options Installing Java SDK 7 Perform the following steps to install the Java SDK 7 on your machine. You can also go with JDK 1.8: 1. Download the Java SDK 7 RPM from Oracle's site (. 2. Install the Java jdk-7u-linux-x64.rpm file on your CentOS machine using the following command: sudo rpm -ivh jdk-7u-linux-x64.rpm 3. Add the following environment variable in the ~/.bashrc file: export JAVA_HOME=/usr/java/jdk 4. Add the path of the bin directory of the JDK to the PATH system environment variable to the ~/.bashrc file: export PATH=$JAVA_HOME/bin:$PATH 5. Run the following command to reload the bashrc file on the current login terminal: source ~/.bashrc 6. Check the Java installation as follows: java -version The output of the preceding command is as follows: java version "1.7.0_71" Java(TM) SE Runtime Environment (build 1.7.0_71-b14) Java HotSpot(TM) 64-Bit Server VM (build 24.71-b01, mixed mode) [19] Storm Deployment, Topology Development, and Topology Options Deployment of the ZooKeeper cluster In any distributed application, various processes need to coordinate with each other and share configuration information. ZooKeeper is an application that provides all these services in a reliable manner. Being a distributed application, Storm also uses a ZooKeeper cluster to coordinate various processes. All of the states associated with the cluster and the various tasks submitted to Storm are stored in ZooKeeper. This section describes how you can set up a ZooKeeper cluster. We will be deploying a ZooKeeper ensemble of three nodes that will handle one node failure. Following is the deployment diagram of the three node ZooKeeper ensemble: In the ZooKeeper ensemble, one node in the cluster acts as the leader, while the rest are followers. If the leader node of the ZooKeeper cluster dies, then an election for the new leader takes places among the remaining live nodes, and a new leader is elected. All write requests coming from clients are forwarded to the leader node, while the follower nodes only handle the read requests. Also, we can't increase the write performance of the ZooKeeper ensemble by increasing the number of nodes because all write operations go through the leader node. It is advised to run an odd number of ZooKeeper nodes, as the ZooKeeper cluster keeps working as long as the majority (the number of live nodes is greater than n/2, where n being the number of deployed nodes) of the nodes are running. So if we have a cluster of four ZooKeeper nodes (3 > 4/2; only one node can die), then we can handle only one node failure, while if we had five nodes (3 > 5/2; two nodes can die) in the cluster, then we can handle two node failures. [20] Storm Deployment, Topology Development, and Topology Options Steps 1 to 4 need to be performed on each node to deploy the ZooKeeper ensemble: 1. Download the latest stable ZooKeeper release from the ZooKeeper site (. At the time of writing, the latest version is ZooKeeper 3.4.6. 2. Once you have downloaded the latest version, unzip it. Now, we set up the ZK_HOME environment variable to make the setup easier. 3. Point the ZK_HOME environment variable to the unzipped directory. Create the configuration file, zoo.cfg, at the $ZK_HOME/conf directory using the following commands: cd $ZK_HOME/conf touch zoo.cfg 4. Add the following properties to the zoo.cfg file: tickTime=2000 dataDir=/var/zookeeper clientPort=2181 initLimit=5 syncLimit=2 server.1=zoo1:2888:3888 server.2=zoo2:2888:3888 server.3=zoo3.2888.3888 Here, zoo1, zoo2, and zoo3 are the IP addresses of the ZooKeeper nodes. The following are the definitions for each of the properties: tickTime: This is the basic unit of time in milliseconds used by ZooKeeper. It is used to send heartbeats, and the minimum session timeout will be twice the tickTime value. dataDir: This is the directory to store the in-memory database snapshots and transactional log. clientPort: This is the port used to listen to client connections. initLimit: This is the number of tickTime values needed to allow followers to connect and sync to a leader node. syncLimit: This is the number of tickTime values that a follower can take to sync with the leader node. If the sync does not happen within this time, the follower will be dropped from the ensemble. [21] Storm Deployment, Topology Development, and Topology Options The last three lines of the server.id=host:port:port format specify that there are three nodes in the ensemble. In an ensemble, each ZooKeeper node must have a unique ID number between 1 and 255. This ID is defined by creating a file named myid in the dataDir directory of each node. For example, the node with the ID 1 (server.1=zoo1:2888:3888) will have a myid file at directory /var/zookeeper with text 1 inside it. For this cluster, create the myid file at three locations, shown as follows: At zoo1 /var/zookeeper/myid contains 1 At zoo2 /var/zookeeper/myid contains 2 At zoo3 /var/zookeeper/myid contains 3 5. Run the following command on each machine to start the ZooKeeper cluster: bin/zkServer.sh start Check the status of the ZooKeeper nodes by performing the following steps: 6. Run the following command on the zoo1 node to check the first node's status: bin/zkServer.sh status The following information is displayed: JMX enabled by default Using config: /home/root/zookeeper-3.4.6/bin/../conf/zoo.cfg Mode: follower The first node is running in follower mode. 7. Check the status of the second node by performing the following command: bin/zkServer.sh status The following information is displayed: JMX enabled by default Using config: /home/root/zookeeper-3.4.6/bin/../conf/zoo.cfg Mode: leader The second node is running in leader mode. [22] Storm Deployment, Topology Development, and Topology Options 8. Check the status of the third node by performing the following command: bin/zkServer.sh status The following information is displayed: JMX enabled by default Using config: /home/root/zookeeper-3.4.6/bin/../conf/zoo.cfg Mode: follower The third node is running in follower mode. 9. Run the following command on the leader machine to stop the leader node: bin/zkServer.sh stop Now, check the status of the remaining two nodes by performing the following steps: 10. Check the status of the first node using the following command: bin/zkServer.sh status The following information is displayed: JMX enabled by default Using config: /home/root/zookeeper-3.4.6/bin/../conf/zoo.cfg Mode: follower The first node is again running in follower mode. 11. Check the status of the second node using the following command: bin/zkServer.sh status The following information is displayed: JMX enabled by default Using config: /home/root/zookeeper-3.4.6/bin/../conf/zoo.cfg Mode: leader The third node is elected as the new leader. 12. Now, restart the third node with the following command: bin/zkServer.sh status [23] Storm Deployment, Topology Development, and Topology Options This was a quick introduction to setting up ZooKeeper that can be used for development; however, it is not suitable for production. For a complete reference on ZooKeeper administration and maintenance, please refer to the online documentation at the ZooKeeper site at . Setting up the Storm cluster In this chapter, we will learn how to set up a three nodes Storm cluster, of which one node will be the active master node (Nimbus) and the other two will be worker nodes (supervisors). The following is the deployment diagram of our three node Storm cluster: The following are the steps that need to be performed to set up a three node Storm cluster: 1. Install and run the ZooKeeper cluster. The steps for installing ZooKeeper are mentioned in the previous section. 2. Download the latest stable Storm release from at the time of writing, the latest version is Storm 1.0.2. [24] Storm Deployment, Topology Development, and Topology Options 3. Once you have downloaded the latest version, copy and unzip it in all three machines. Now, we will set the $STORM_HOME environment variable on each machine to make the setup easier. The $STORM_HOME environment contains the path of the Storm home folder (for example, export STORM_HOME=/home/user/storm-1.0.2). 4. Go to the $STORM_HOME/conf directory in the master nodes and add the following lines to the storm.yaml file: storm.zookeeper.servers: - "zoo1" - "zoo2" - "zoo3" storm.zookeeper.port: 2181 nimbus.seeds: "nimbus1,nimbus2" storm.local.dir: "/tmp/storm-data" We are installing two master nodes. 5. Go to the $STORM_HOME/conf directory at each worker node and add the following lines to the storm.yaml file: storm.zookeeper.servers: - "zoo1" - "zoo2" - "zoo3" storm.zookeeper.port: 2181 nimbus.seeds: "nimbus1,nimbus2" storm.local.dir: "/tmp/storm-data" supervisor.slots.ports: - 6700 - 6701 - 6702 - 6703 If you are planning to execute the Nimbus and supervisor on the same machine, then add the supervisor.slots.ports property to the Nimbus machine too. [25] Storm Deployment, Topology Development, and Topology Options 6. Go to the $STORM_HOME directory at the master nodes and execute the following command to start the master daemon: $> bin/storm nimbus & 7. Go to the $STORM_HOME directory at each worker node (or supervisor node) and execute the following command to start the worker daemons: $> bin/storm supervisor & Developing the hello world example Before starting the development, you should have Eclipse and Maven installed in your project. The sample topology explained here will cover how to create a basic Storm project, including a spout and bolt, and how to build, and execute them. Create a Maven project by using com.stormadvance as groupId and storm-example as artifactId. Add the following Maven dependencies to the pom.xml file: org.apache.storm storm-core 1.0.2 provided Make sure the scope of the Storm dependency is provided, otherwise you will not be able to deploy the topology on the Storm cluster. Add the following Maven build plugins in the pom.xml file: maven-assembly-plugin 2.2.1 jar-with-dependencies [26] Storm Deployment, Topology Development, and Topology Options make-assembly package single Write your first sample spout by creating a SampleSpout class in the com.stormadvance.storm_example package. The SampleSpout class extends the serialized BaseRichSpout class. This spout does not connect to an external source to fetch data, but randomly generates the data and emits a continuous stream of records. The following is the source code of the SampleSpout class with an explanation: public class SampleSpout extends BaseRichSpout { private static final long serialVersionUID = 1L; private static final Map map = new HashMap(); static { map.put(0, "google"); map.put(1, "facebook"); map.put(2, "twitter"); map.put(3, "youtube"); map.put(4, "linkedin"); } private SpoutOutputCollector spoutOutputCollector; public void open(Map conf, TopologyContext context, SpoutOutputCollector spoutOutputCollector) { // Open the spout this.spoutOutputCollector = spoutOutputCollector; } public void nextTuple() { [27] Storm Deployment, Topology Development, and Topology Options // Storm cluster repeatedly calls this method to emita continuous // stream of tuples. final Random rand = new Random(); // generate the random number from 0 to 4. int randomNumber = rand.nextInt(5); spoutOutputCollector.emit(new Values(map.get(randomNumber))); try{ Thread.sleep(5000); }catch(Exception e) { System.out.println("Failed to sleep the thread"); } } public void declareOutputFields(OutputFieldsDeclarer declarer) { // emit the tuple with field "site" declarer.declare(new Fields("site")); } } Write your first sample bolt by creating a SampleBolt class within the same package. The SampleBolt class extends the serialized BaseRichBolt class. This bolt will consume the tuples emitted by the SampleSpout spout and will print the value of the field site on the console. The following is the source code of the SampleStormBolt class with an explanation: public class SampleBolt extends BaseBasicBolt { private static final long serialVersionUID = 1L; public void execute(Tuple input, BasicOutputCollector collector) { // fetched the field "site" from input tuple. String test = input.getStringByField("site"); // print the value of field "site" on console. System.out.println("######### Name of input site is : " + test); } public void declareOutputFields(OutputFieldsDeclarer declarer) { } } [28] Storm Deployment, Topology Development, and Topology Options Create a main SampleStormTopology class within the same package. This class creates an instance of the spout and bolt along with the classes, and chaines them together using a TopologyBuilder class. This class uses org.apache.storm.LocalCluster to simulate the Storm cluster. The LocalCluster mode is used for debugging/testing the topology on a developer machine before deploying it on the Storm cluster. The following is the implementation of the main class: public class SampleStormTopology { public static void main(String[] args) throws AlreadyAliveException, InvalidTopologyException { // create an instance of TopologyBuilder class TopologyBuilder builder = new TopologyBuilder(); // set the spout class builder.setSpout("SampleSpout", new SampleSpout(), 2); // set the bolt class builder.setBolt("SampleBolt", new SampleBolt(), 4).shuffleGrouping("SampleSpout"); Config conf = new Config(); conf.setDebug(true); // create an instance of LocalCluster class for // executing topology in local mode. LocalCluster cluster = new LocalCluster(); // SampleStormTopology is the name of submitted topology cluster.submitTopology("SampleStormTopology", conf, builder.createTopology()); try { Thread.sleep(100000); } catch (Exception exception) { System.out.println("Thread interrupted exception : " + exception); } // kill the SampleStormTopology cluster.killTopology("SampleStormTopology"); // shutdown the storm test cluster cluster.shutdown(); } } Go to your project's home directory and run the following commands to execute the topology in local mode: $> cd $STORM_EXAMPLE_HOME $> mvn compile exec:java -Dexec.classpathScope=compile Dexec.mainClass=com.stormadvance.storm_example.SampleStormTopology [29] Storm Deployment, Topology Development, and Topology Options Now create a new topology class for deploying the topology on an actual Storm cluster. Create a main SampleStormClusterTopology class within the same package. This class also creates an instance of the spout and bolt along with the classes, and chains them together using a TopologyBuilder class: public class SampleStormClusterTopology { public static void main(String[] args) throws AlreadyAliveException, InvalidTopologyException { // create an instance of TopologyBuilder class TopologyBuilder builder = new TopologyBuilder(); // set the spout class builder.setSpout("SampleSpout", new SampleSpout(), 2); // set the bolt class builder.setBolt("SampleBolt", new SampleBolt(), 4).shuffleGrouping("SampleSpout"); Config conf = new Config(); conf.setNumWorkers(3); // This statement submit the topology on remote // args[0] = name of topology try { StormSubmitter.submitTopology(args[0], conf, builder.createTopology()); } catch (AlreadyAliveException alreadyAliveException) { System.out.println(alreadyAliveException); } catch (InvalidTopologyException invalidTopologyException) { System.out.println(invalidTopologyException); } catch (AuthorizationException e) { // TODO Auto-generated catch block e.printStackTrace(); } } } Build your Maven project by running the following command on the projects home directory: mvn clean install The output of the preceding command is as follows: -- ---[INFO] -- ---[INFO] BUILD SUCCESS [INFO] ---------------------------- --------------------------- ---[INFO] Total time: 58.326s [30] Storm Deployment, Topology Development, and Topology Options [INFO] Finished at: [INFO] Final Memory: 14M/116M [INFO] -- ---- We can deploy the topology to the cluster using the following Storm client command: bin/storm jar jarName.jar [TopologyMainClass] [Args] The preceding command runs TopologyMainClass with the arguments arg1 and arg2. The main function of TopologyMainClass is to define the topology and submit it to the Nimbus machine. The storm jar part takes care of connecting to the Nimbus machine and uploading the JAR part. Log in on a Storm Nimbus machine and execute the following commands: $> cd $STORM_HOME $> bin/storm jar ~/storm_example-0.0.1-SNAPSHOT-jar-with-dependencies.jar com.stormadvance.storm_example.SampleStormClusterTopology storm_example In the preceding code ~/storm_example-0.0.1-SNAPSHOT-jar-withdependencies.jar is the path of the SampleStormClusterTopology JAR that we are deploying on the Storm cluster. The following information is displayed: 702 [main] INFO o.a.s.StormSubmitter - Generated ZooKeeper secret payload for MD5-digest: -8367952358273199959:-5050558042400210383 793 [main] INFO o.a.s.s.a.AuthUtils - Got AutoCreds [] 856 [main] INFO o.a.s.StormSubmitter - Uploading topology jar /home/USER/storm_example-0.0.1-SNAPSHOT-jar-with-dependencies.jar to assigned location: /tmp/storm-data/nimbus/inbox/stormjar-d3007821f87d-48af-8364-cff7abf8652d.jar 867 [main] INFO o.a.s.StormSubmitter - Successfully uploaded topology jar to assigned location: /tmp/storm-data/nimbus/inbox/stormjar-d3007821f87d-48af-8364-cff7abf8652d.jar 868 [main] INFO o.a.s.StormSubmitter - Submitting topology storm_example in distributed mode with conf {"storm.zookeeper.topology.auth.scheme":"digest","storm.zookeeper.topology. auth.payload":"-8367952358273199959:-5050558042400210383","topology.workers ":3} 1007 [main] INFO o.a.s.StormSubmitter - Finished submitting topology: storm_example Run the jps command to see the number of running JVM processes as follows: jps [31] Storm Deployment, Topology Development, and Topology Options The preceding command's output is: 26827 26530 26824 26468 26822 worker supervisor worker nimbus worker In the preceding code, a worker is the JVM launched for the SampleStormClusterTopology topology. The different options of the Storm topology This section covers the following operations that a user can perform on the Storm cluster: Deactivate Activate Rebalance Kill Dynamic log level settings Deactivate Storm supports the deactivating a topology. In the deactivated state, spouts will not emit any new tuples into the pipeline, but the processing of the already emitted tuples will continue. The following is the command to deactivate the running topology: $> bin/storm deactivate topologyName Deactivate SampleStormClusterTopology using the following command: bin/storm deactivate SampleStormClusterTopology The following information is displayed: 0 [main] INFO backtype.storm.thrift - Connecting to Nimbus at localhost:6627 76 [main] INFO backtype.storm.command.deactivate - Deactivated topology: SampleStormClusterTopology [32] Storm Deployment, Topology Development, and Topology Options Activate Storm also the supports activating a topology. When a topology is activated, spouts will again start emitting tuples. The following is the command to activate the topology: $> bin/storm activate topologyName Activate SampleStormClusterTopology using the following command: bin/storm activate SampleStormClusterTopology The following information is displayed: 0 [main] INFO backtype.storm.thrift - Connecting to Nimbus at localhost:6627 65 [main] INFO backtype.storm.command.activate - Activated topology: SampleStormClusterTopology Rebalance The process of updating a the topology parallelism at the runtime is called a rebalance. A more detailed information of this operation acn be in Chapter 3, Storm Parallelism and Data Partitioning. Kill Storm topologies are never-ending processes. To stop a topology, we need to kill it. When killed, the topology first enters into the deactivation state, processes all the tuples already emitted into it, and then stops. Run the following command to kill SampleStormClusterTopology: $> bin/storm kill SampleStormClusterTopology [33] Storm Deployment, Topology Development, and Topology Options The following information is displayed: 0 [main] INFO backtype.storm.thrift - Connecting to Nimbus at localhost:6627 80 [main] INFO backtype.storm.command.kill-topology - Killed topology: SampleStormClusterTopology Now, run the jps command again to see the remaining JVM processes as follows: jps The preceding command's output is: 26530 supervisor 27193 Jps 26468 nimbus Dynamic log level settings This allows the user to change the log level of topology on runtime without stopping the topology. The detailed information of this operation can be found at the end of this chapter. Walkthrough of the Storm UI This section will show you how we can start the Storm UI daemon. However, before starting the Storm UI daemon, we assume that you have a running Storm cluster. The Storm cluster deployment steps are mentioned in the previous sections of this chapter. Now, go to the Storm home directory (cd $STORM_HOME) at the leader Nimbus machine and run the following command to start the Storm UI daemon: $> cd $STORM_HOME $> bin/storm ui & By default, the Storm UI starts on the 8080 port of the machine where it is started. Now, we will browse to the page to view the Storm UI, where Nimbus node is the IP address or hostname of the the Nimbus machine. [34] Storm Deployment, Topology Development, and Topology Options The following is a screenshot of the Storm home page: Cluster Summary section This portion of the Storm UI shows the version of Storm deployed in the cluster, the uptime of the Nimbus nodes, number of free worker slots, number of used worker slots, and so on. While submitting a topology to the cluster, the user first needs to make sure that the value of the Free slots column should not be zero; otherwise, the topology doesn't get any worker for processing and will wait in the queue until a workers becomes free. Nimbus Summary section This portion of the Storm UI shows the number of Nimbus processes that are running in a Storm cluster. The section also shows the status of the Nimbus nodes. A node with the status Leader is an active master while the node with the status Not a Leader is a passive master. [35] Storm Deployment, Topology Development, and Topology Options Supervisor Summary section This portion of the Storm UI shows the list of supervisor nodes running in the cluster, along with their Id, Host, Uptime, Slots, and Used slots columns. Nimbus Configuration section This portion of the Storm UI shows the configuration of the Nimbus node. Some of the important properties are: supervisor.slots.ports storm.zookeeper.port storm.zookeeper.servers storm.zookeeper.retry.interval worker.childopts supervisor.childopts The following is a screenshot of Nimbus Configuration: [36] Storm Deployment, Topology Development, and Topology Options Topology Summary section This portion of the Storm UI shows the list of topologies running in the Storm cluster, along with their ID, the number of workers assigned to the topology, the number of executors, number of tasks, uptime, and so on. Let's deploy the sample topology (if it is not running already) in a remote Storm cluster by running the following command: $> cd $STORM_HOME $> bin/storm jar ~/storm_example-0.0.1-SNAPSHOT-jar-with-dependencies.jar com.stormadvance.storm_example.SampleStormClusterTopology storm_example We have created the SampleStormClusterTopology topology by defining three worker processes, two executors for SampleSpout, and four executors for SampleBolt. After submitting SampleStormClusterTopology on the Storm cluster, the user has to refresh the Storm home page. The following screenshot shows that the row is added for SampleStormClusterTopology in the Topology Summary section. The topology section contains the name of the topology, unique ID of the topology, status of the topology, uptime, number of workers assigned to the topology, and so on. The possible values of the Status fields are ACTIVE, KILLED, and INACTIVE. [37] Storm Deployment, Topology Development, and Topology Options Let's click on SampleStormClusterTopology to view its detailed statistics. There are two screenshots for this. The first one contains the information about the number of workers, executors, and tasks assigned to the SampleStormClusterTopology topology: The next screenshot contains information about the spouts and bolts, including the number of executors and tasks assigned to each spout and bolt: [38] Storm Deployment, Topology Development, and Topology Options The information shown in the previous screenshots is as follows: Topology stats: This section will give information about the number of tuples emitted, transferred, and acknowledged, the capacity latency, and so on, within the windows of 10 minutes, 3 hours, 1 day, and since the start of the topology Spouts (All time): This section shows the statistics of all the spouts running inside the topology Bolts (All time): This section shows the statistics of all the bolts running inside the topology Topology actions: This section allows us to perform activate, deactivate, rebalance, kill, and other operations on the topologies directly through the Storm UI: Deactivate: Click on Deactivate to deactivate the topology. Once the topology is deactivated, the spout stops emitting tuples and the status of the topology changes to INACTIVE on the Storm UI. Deactivating the topology does not free the Storm resource. Activate: Click on the Activate button to activate the topology. Once the topology is activated, the spout again starts emitting tuples. Kill: Click on the Kill button to destroy/kill the topology. Once the topology is killed, it will free all the Storm resources allotted to this topology. While killing the topology, the Storm will first deactivate the spouts and wait for the kill time mentioned on the alerts box so that the bolts have a chance to finish the processing of the tuples emitted by the spouts before the kill command. The following screenshot shows how we can kill the topology through the Storm UI: [39] Storm Deployment, Topology Development, and Topology Options Let's go to the Storm UI's home page to check the status of SampleStormClusterToplogy, as shown in the following screenshot: Dynamic log level settings The dynamic log level allows us to change the log level setting of the topology on the runtime from the Storm CLI and the Storm UI. Updating the log level from the Storm UI Go through the following steps to update the log level from the Storm UI: 1. Deploy SampleStormClusterTopology again on the Storm cluster if it is not running. 2. Browse the Storm UI at . 3. Click on the storm_example topology. [40] Storm Deployment, Topology Development, and Topology Options 4. Now click on the Change Log Level button to change the ROOT logger of the topology, as shown in the following are the screenshots: 5. Configure the entries mentioned in the following screenshots change the ROOT logger to ERROR: 6. If you are planning to change the logging level to DEBUG, then you must specify the timeout (expiry time) for that log level, as shown in the following screenshots: [41] Storm Deployment, Topology Development, and Topology Options 7. Once the time mentioned in the expiry time is reached, the log level will go back to the default value: 8. Clear button mentioned in the Action column will clear the log setting, and the application will set the default log setting again. Updating the log level from the Storm CLI We can modify the log level from the Storm CLI. The following is the command that the user has to execute from the Storm directory to update the log settings on the runtime: bin/storm set_log_level [topology name] -l [logger name]=[LEVEL]:[TIMEOUT] In the preceding code, topology name is the name of the topology, and logger name is the logger we want to change. If you want to change the ROOT logger, then use ROOT as a value of logger name. The LEVEL is the log level you want to apply. The possible values are DEBUG, INFO, ERROR, TRACE, ALL, WARN, FATAL, and OFF. The TIMEOUT is the time in seconds. The log level will go back to normal after the timeout time. The value of TIMEOUT is mandatory if you are setting the log level to DEBUG/ALL. The following is the command to change the log level setting for the storm_example topology: $> bin/storm set_log_level storm_example -l ROOT=DEBUG:30 The following is the command to clear the log level setting: $> ./bin/storm set_log_level storm_example -r ROOT [42] Storm Deployment, Topology Development, and Topology Options Summary In this chapter, we have covered the installation of Storm and ZooKeeper clusters, the deployment of topologies on Storm clusters, the high availability of Nimbus nodes, and topology monitoring through the Storm UI. We have also covered the different operations a user can perform on running topology. Finally, we focused on how we can change the log level of running topology. In the next chapter, we will focus on the distribution of topologies on multiple Storm machines/nodes. [43] 3 Storm Parallelism and Data Partitioning In the first two chapters, we have covered the introduction to Storm, the installation of Storm, and developing a sample topology. In this chapter, we are focusing on distribution of the topology on multiple Storm machines/nodes. This chapter covers the following points: Parallelism of topology How to configure parallelism at the code level Different types of stream groupings in a Storm cluster Guaranteed message processing Tick tuple Parallelism of a topology Parallelism means the distribution of jobs on multiple nodes/instances where each instance can work independently and can contribute to the processing of data. Let's first look at the processes/components that are responsible for the parallelism of a Storm cluster. Worker process A

Storm topology is executed across multiple supervisor nodes in the Storm cluster. Each of the nodes in the cluster can run one or more JVMs called worker processes, which are responsible for processing a part of the topology. Storm Parallelism and Data Partitioning A worker process is specific to one of the specific topologies and can execute multiple components of that topology. If multiple topologies are being run at the same time, none of them will share any of the workers, thus providing some degree of isolation between topologies. Executor Within each worker process, there can be multiple threads executing parts of the topology. Each of these threads is called an executor. An executor can execute only one of the components, that is, any spout or bolt in the topology. Each executor, being a single thread, can execute only tasks assigned to it serially. The number of executors defined for a spout or bolt can be changed dynamically while the topology is running, which means that you can easily control the degree of parallelism of various components in your topology. Task This is the most granular unit of task execution in Storm. Each task is an instance of a spout or bolt. When defining a Storm topology, you can specify the number of tasks for each spout and bolt. Once defined, the number of tasks cannot be changed for a component at runtime. Each task can be executed alone or with another task of the same type, or another instance of the same spout or bolt. The following diagram depicts the relationship between a worker process, executors, and tasks. In the following diagram, there are two executors for each component, with each hosting a different number of tasks. Also, as you can see, there are two executors and eight tasks defined for one component (each executor is hosting four tasks). If you are not getting enough performance out of this configuration, you can easily change the number of executors for the component to four or eight to increase performance and the tasks will be uniformly distributed between all executors of that component. The following diagrams show the relationship between executor, task, and worker: [45] Storm Parallelism and Data Partitioning Configure parallelism at the code level Storm provides an API to set the number of worker processes, number of executors, and number of tasks at the code level. The following section shows how we can configure parallelism at the code level. We can set the number of worker processes at the code level by using the setNumWorkers method of the org.apache.storm.Config class. Here is the code snippet to show these settings in practice: Config conf = new Config(); conf.setNumWorkers(3); In the previous chapter, we configured the number of workers as three. Storm will assign the three workers for the SampleStormTopology and SampleStormClusterTopology topology. [46] Storm Parallelism and Data Partitioning We can set the number of executors at the code level by passing the parallelism_hint argument in the setSpout(args,args,parallelism_hint) or setBolt(args,args,parallelism_hint) methods of the org.apache.storm.topology.TopologyBuilder class. Here is the code snippet to show these settings in practice: builder.setSpout("SampleSpout", new SampleSpout(), 2); // set the bolt class builder.setBolt("SampleBolt", new SampleBolt(), 4).shuffleGrouping("SampleSpout"); In the previous chapter, we set parallelism_hint=2 for SampleSpout and parallelism_hint=4 for SampleBolt. At the time of execution, Storm will assign two executors for SampleSpout and four executors for SampleBolt. We can configure the number of tasks that can execute inside the executors. Here is the code snippet to show these settings in practice: builder.setSpout("SampleSpout", new SampleSpout(), 2).setNumTasks(4); In the preceding code, we have configured the two executors and four tasks of SampleSpout. For SampleSpout, Storm will assign two tasks per executor. By default, Storm will run one task per executor if the user does not set the number of tasks at the code level. Worker process, executor, and task distribution Let's assume the numbers of worker processes set for the topology is three, the number of executors for SampleSpout is three, and the number of executors for SampleBolt is three. Also, the number of tasks for SampleBolt is to be six, meaning that each SampleBolt executor will have two tasks. The following diagram shows what the topology would look like in operation: [47] Storm Parallelism and Data Partitioning Rebalance the parallelism of a topology As explained in the previous chapter, one of the key features of Storm is that it allows us to modify the parallelism of a topology at runtime. The process of updating a topology parallelism at runtime is called rebalance. There are two ways to rebalance the topology: Using Storm Web UI Using Storm CLI [48] Storm Parallelism and Data Partitioning The Storm Web UI was covered in the previous chapter. This section covers how we can rebalance the topology using the Storm CLI tool. Here are the commands that we need to execute on Storm CLI to rebalance the topology: > bin/storm rebalance [TopologyName] -n [NumberOfWorkers] -e [Spout]=[NumberOfExecutos] -e [Bolt1]=[NumberOfExecutos] [Bolt2]=[NumberOfExecutos] The rebalance command will first deactivate the topology for the duration of the message timeout and then redistribute the workers evenly around the Storm cluster. After a few seconds or minutes, the topology will revert to the previous state of activation and restart the processing of input streams. Rebalance the parallelism of a SampleStormClusterTopology topology Let's first check the numbers of worker processes that are running in the Storm cluster by running the jps command on the supervisor machine: Run the jps command on supervisor-1: > jps 24347 23940 24593 24349 worker supervisor Jps worker Two worker processes are assigned to the supervisor-1 machine. Now, run the jps command on supervisor-2: > jps 24344 worker 23941 supervisor 24543 Jps One worker process is assigned to the supervisor-2 machine. A total of three worker processes are running on the Storm cluster. [49] Storm Parallelism and Data Partitioning Let's try reconfiguring SampleStormClusterTopology to use two worker processes, SampleSpout to use four executors, and SampleBolt to use four executors: > bin/storm rebalance SampleStormClusterTopology -n 2 -e SampleSpout=4 -e SampleBolt=4 0 [main] INFO backtype.storm.thrift - Connecting to Nimbus at nimbus.host.ip:6627 58 [main] INFO backtype.storm.command.rebalance - Topology SampleStormClusterTopology is rebalancing Rerun the jps commands on the supervisor machines to view the number of worker processes. Run the jps command on supervisor-1: > jps 24377 worker 23940 supervisor 24593 Jps Run the jps command on supervisor-2: > jps 24353 worker 23941 supervisor 24543 Jps In this case, two worker processes are shown previously. The first worker process is assigned to supervisor-1 and the other one is assigned to supervisor-2. The distribution of workers may vary depending on the number of topologies running on the system and the number of slots available on each supervisor. Ideally, Storm tries to distribute the load uniformly between all the nodes. Different types of stream grouping in the Storm cluster When defining a topology, we create a graph of computation with the number of boltprocessing streams. At a more granular level, each bolt executes multiple tasks in the topology. Thus, each task of a particular bolt will only get a subset of the tuples from the subscribed streams. [50] Storm Parallelism and Data Partitioning Stream grouping in Storm provides complete control over how this partitioning of tuples happens among the many tasks of a bolt subscribed to a stream. Grouping for a bolt can be defined on the instance of org.apache.storm.topology.InputDeclarer returned when defining bolts using the org.apache e.storm.topology.TopologyBuilder.setBolt method. Storm supports the following types of stream groupings. Shuffle grouping Shuffle grouping distributes tuples in a uniform, random way across the tasks. An equal number of tuples will be processed by each task. This grouping is ideal when you want to distribute your processing load uniformly across the tasks and where there is no requirement for any data-driven partitioning. This is one of the most commonly used groupings in Storm. Field grouping This grouping enables you to partition a stream on the basis of some of the fields in the tuples. For example, if you want all the tweets from a particular user to go to a single task, then you can partition the tweet stream using field grouping by username in the following manner: builder.setSpout("1", new TweetSpout()); builder.setBolt("2", new TweetCounter()).fieldsGrouping("1", new Fields("username")) As a result of the field grouping being hash (fields) % (no. of tasks), it does not guarantee that each of the tasks will get tuples to process. For example, if you have applied a field grouping on a field, say X, with only two possible values, A and B, and created two tasks for the bolt, then it might be possible that both hash (A) % 2 and hash (B) % 2 return equal values, which will result in all the tuples being routed to a single task and the other being completely idle. [51] Storm Parallelism and Data Partitioning Another common usage of field grouping is to join streams. Since partitioning happens solely on the basis of field values, and not the stream type, we can join two streams with any common join fields. The name of the fields needs not be the same. For example, in the order processing domain, we can join the Order stream and the ItemScanned stream to see when an order is completed: builder.setSpout("1", new OrderSpout()); builder.setSpount("2", new ItemScannedSpout()); builder.setBolt("joiner", new OrderJoiner()) .fieldsGrouping("1", new Fields("orderId")) .fieldsGrouping("2", new Fields("orderRefId")); Since joins on streams vary from application to application, you'll make your own definition of a join, say joins over a time window, that can be achieved by composing field groupings. All grouping All grouping is a special grouping that does not partition the tuples but replicates them to all the tasks, that is, each tuple will be sent to each of the bolt's tasks for processing. One common use case of all grouping is for sending signals to bolts. For example, if you are doing some kind of filtering on the streams, you can pass or change the filter parameters to all the bolts by sending them those parameters over a stream that is subscribed by all the bolt's tasks with an all grouping. Another example is to send a reset message to all the tasks in an aggregation bolt. Global grouping Global grouping does not partition the stream but sends the complete stream to the bolt's task, the smallest ID. A general use case of this is when there needs to be a reduce phase in your topology where you want to combine the results from previous steps in the topology into a single bolt. Global grouping might seem redundant at first, as you can achieve the same results by defining the parallelism for the bolt as one if you only have one input stream. However, when you have multiple streams of data coming through a different path, you might want only one of the streams to be reduced and others to be parallel processes. [52] Storm Parallelism and Data Partitioning For example, consider the following topology. In this, you might want to combine all the tuples coming from Bolt C in a single Bolt D task, while you might still want parallelism for tuples coming from Bolt E to Bolt D: Direct grouping In direct grouping, the emitter decides where each tuple will go for processing. For example, say we have a log stream and we want to process each log entry to be processed by a specific bolt task on the basis of the type of resource. In this case, we can use direct grouping. Direct grouping can only be used with direct streams. To declare a stream as a direct stream, use the backtype.storm.topology.OutputFieldsDeclarer.declareStream method, which takes a boolean parameter. Once you have a direct stream to emit to, use backtype.storm.task.OutputCollector.emitDirect instead of emit methods to emit it. The emitDirect method takes a taskId parameter to specify the task. You can get the number of tasks for a component using the backtype.storm.task.TopologyContext.getComponentTasks method. Local or shuffle grouping If the tuple source and target bolt tasks are running in the same worker, using this grouping will act as a shuffle grouping only between the target tasks running on the same worker, thus minimizing any network hops, resulting in increased performance. If there are no target bolt tasks running on the source worker process, this grouping will act similar to the shuffle grouping mentioned earlier. [53] Storm Parallelism and Data Partitioning None grouping None grouping is used when you don't care about the way tuples are partitioned among various tasks. As of Storm 0.8, this is equivalent to using shuffle grouping. Custom grouping If none of the preceding groupings fit your use case, you can define your own custom grouping by implementing the backtype.storm.grouping.CustomStreamGrouping interface. Here is a sample custom grouping that partitions the stream on the basis of the category in the tuples: public class CategoryGrouping implements CustomStreamGrouping, Serializable { private static final Map categories = ImmutableMap.of ("Financial", 0, "Medical", 1, "FMCG", 2, "Electronics", 3); private int tasks = 0; public void prepare(WorkerTopologyContext context, GlobalStreamId stream, List targetTasks) { tasks = targetTasks.size(); } public List chooseTasks(int taskId, List values) { String category = (String) values.get(0); return ImmutableList.of(categories.get(category) % tasks); } } [54] Storm Parallelism and Data Partitioning The following diagram represents the Storm groupings graphically: Guaranteed message processing In a Storm topology, a single tuple being emitted by a spout can result in a number of tuples being generated in the later stages of the topology. For example, consider the following topology: [55] Storm Parallelism and Data Partitioning Here, Spout A emits a tuple T(A), which is processed by bolt B and bolt C, which emit tuple T(AB) and T(AC) respectively. So, when all the tuples produced as a result of tuple T(A)--namely, the tuple tree T(A), T(AB), and T(AC)--are processed, we say that the tuple has been processed completely. When some of the tuples in a tuple tree fail to process either due to some runtime error or a timeout that is configurable for each topology, then Storm considers that to be a failed tuple. Here are the six steps that are required by Storm to guarantee message processing: 1. Tag each tuple emitted by a spout with a unique message ID. This can be done by using the org.apache.storm.spout.SpoutOutputColletor.emit method, which takes a messageId argument. Storm uses this message ID to track the state of the tuple tree generated by this tuple. If you use one of the emit methods that doesn't take a messageId argument, Storm will not track it for complete processing. When the message is processed completely, Storm will send an acknowledgement with the same messageId that was used while emitting the tuple. 2. A generic pattern implemented by spouts is that they read a message from a messaging queue, say RabbitMQ, produce the tuple into the topology for further processing, and then dequeue the message once it receives the acknowledgement that the tuple has been processed completely. 3. When one of the bolts in the topology needs to produce a new tuple in the course of processing a message, for example, bolt B in the preceding topology, then it should emit the new tuple anchored with the original tuple that it got from the spout. This can be done by using the overloaded emit methods in the org.apache.storm.task.OutputCollector class that takes an anchor tuple as an argument. If you are emitting multiple tuples from the same input tuple, then anchor each outgoing tuple. [56] Storm Parallelism and Data Partitioning 4. Whenever you are done with processing a tuple in the execute method of your bolt, send an acknowledgment using the org.apache.storm.task.OutputCollector.ack method. When the acknowledgement reaches the emitting spout, you can safely mark the message as being processed and dequeue it from the message queue, if any. 5. Similarly, if there is some problem in processing a tuple, a failure signal should be sent back using the org.apache.storm.task.OutputCollector.fail method so that Storm can replay the failed message. 6. One of the general patterns of processing in Storm bolts is to process a tuple in, emit new tuples, and send an acknowledgement at the end of the execute method. Storm provides the org.apache.storm.topology.base.BasicBasicBolt class that automatically sends the acknowledgement at the end of the execute method. If you want to signal a failure, throw org.apache.storm.topology.FailedException from the execute method. This model results in at-least-once message processing semantics, and your application should be ready to handle a scenario when some of the messages will be processed multiple times. Storm also provides exactly-once message processing semantics, which we will discuss in Chapter 5, Trident Topology and Uses. Even though you can achieve some guaranteed message processing in Storm using the methods mentioned here, it is always a point to ponder whether you actually require it or not, as you can gain a large performance boost by risking some of the messages not being completely processed by Storm. This is a trade-off that you can think of when designing your application. Tick tuple In some use cases, a bolt needs to cache the data for a few seconds before performing some operation, such as cleaning the cache after every 5 seconds or inserting a batch of records into a database in a single request. The tick tuple is the system-generated (Storm-generated) tuple that we can configure at each bolt level. The developer can configure the tick tuple at the code level while writing a bolt. [57] Storm Parallelism and Data Partitioning We need to overwrite the following method in the bolt to enable the tick tuple: @Override public Map getComponentConfiguration() { Config conf = new Config(); int tickFrequencyInSeconds = 10; conf.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS, tickFrequencyInSeconds); return conf; } In the preceding code, we have configured the tick tuple time to 10 seconds. Now, Storm will start generating a tick tuple after every 10 seconds. Also, we need to add the following code in the execute method of the bolt to identify the type of tuple: @Override public void execute(Tuple tuple) { if (isTickTuple(tuple)) { // now you can trigger e.g. a periodic activity } else { // do something with the normal tuple } } private static boolean isTickTuple(Tuple tuple) { return tuple.getSourceComponent().equals(Constants.SYSTEM_COMPONENT_ID) && tuple.getSourceStreamId().equals(Constants.SYSTEM_TICK_STREAM_ID); } If the output of the isTickTuple() method is true, then the input tuple is a tick tuple. Otherwise, it is a normal tuple emitted by the previous bolt. Be aware that tick tuples are sent to bolts/spouts just like regular tuples, which means they will be queued behind other tuples that a bolt/spout is about to process via its execute() or nextTuple() method, respectively. As such, the time interval you configure for tick tuples is, in practice, served on a best-effort basis. For instance, if a bolt is suffering from high execution latency--for example, due to being overwhelmed by the incoming rate of regular, non-tick tuples--then you will observe that the periodic activities implemented in the bolt will get triggered later than expected. [58] Storm Parallelism and Data Partitioning Summary In this chapter, we have shed some light on how we can define the parallelism of Storm, how we can distribute jobs between multiple nodes, and how we can distribute data between multiple instances of a bolt. The chapter also covered two important features: guaranteed message processing and the tick tuple. In the next chapter, we are covering the Trident high-level abstraction over Storm. Trident is mostly used to solve the real-time transaction problem, which can't be solved through plain Storm. [59] 4 Trident Introduction In the previous chapters, we covered the architecture of Storm, its topology, bolts, spouts, tuples, and so on. In this chapter, we are covering Trident, which is a high-level abstraction over Storm. We are covering the following points in this chapter: Introducing Trident Understanding Trident's data model Writing Trident functions, filters, and projections Trident repartitioning operations Trident aggregators When to use Trident Trident introduction Trident is a high-level abstraction built on top of Storm. Trident supports stateful stream processing, while pure Storm is a stateless processing framework. The main advantage of using Trident is that it guarantees that every message entered into the topology is processed only once, which would be difficult to achieve with vanilla Storm. The concept of Trident is similar to high-level batch processing tools, such as Cascading and Pig, developed over Hadoop. To achieve exactly-once processing, Trident processes the input stream in small batches. We will cover this in more detail in the Chapter 5, Trident Topology and Uses, Trident state section. Trident Introduction In the first three chapters, we learned that, in Storm's topology, the spout is the source of tuples. A tuple is a unit of data that can be processed by a Storm application, and a bolt is the processing powerhouse where we write the transformation logic. But in the Trident topology, the bolt is replaced with the higher level semantics of functions, aggregates, filters, and states. Understanding Trident's data model The Trident tuple is the data model of a Trident topology. The Trident tuple is the basic unit of data that can be processed by a Trident topology. Each tuple consists of a predefined list of fields. The value of each field can be a byte, char, integer, long, float, double, Boolean, or byte array. During the construction of a topology, operations are performed on a tuple, which will either add new fields to the tuple or replace the tuple with a new set of fields. Each of the fields in a tuple can be accessed by name, (getValueByField(String)), or its positional index, (getValue(int)), in the tuple. The Trident tuple also provides convenience methods, such as getIntegerByField(String), which saves you from typecasting the objects. Writing Trident functions, filters, and projections This section covers the definition of Trident functions, filters, and projections. Trident functions, filters, and projections are used to modify/filter the input tuples based on certain criteria. This section also covers how we can write Trident functions, filters, and projections. Trident function Trident functions contain logic to modify the original tuple. A Trident function gets a set of fields of the tuple as input and emits one or more tuples as output. The fields of the output tuples are merged with the fields of the input tuple to form the complete tuple, which will pass to the next action in the topology. If the Trident function emits no tuples corresponding to the input tuple, then that tuple is removed from the stream. [61] Trident Introduction We can write a custom Trident function by extending the storm.trident.operation.BaseFunction class and implementing the execute(TridentTuple tuple, TridentCollector collector) method. Let's write the sample Trident function, which will return the new field called sum: public class SumFunction extends BaseFunction { private static final long serialVersionUID = 5L; public void execute(TridentTuple tuple, TridentCollector collector) { int number1 = tuple.getInteger(0); int number2 = tuple.getInteger(1); int sum = number1+number2; // emit the sum of first two fields collector.emit(new Values(sum)); } } Suppose we get dummyStream as input, which contains four fields, a, b, c, d, and only fields a and b are passed as input fields to the SumFunction function. The SumFunction class emits new a field, sum. The sum field emitted by the execute method of the SumFunction class is merged with the input tuple to form the complete tuple. Hence, the total number of fields in the output tuple is 5 (a, b, c, d, sum). Here is a sample piece of code that shows how we can pass the input fields and the name of the new field to the Trident function: dummyStream.each(new Fields("a","b"), new SumFunction (), new Fields("sum")) The following diagram shows the input tuples, SumFunction, and the output tuples. The output tuples contain five fields, a, b, c, d, and sum: [62] Trident Introduction Trident filter A Trident filter gets a set of fields as input and returns either true or false depending on whether a certain condition is satisfied or not. If true is returned, then the tuple is kept in the output stream; otherwise, the tuple is removed from the stream. We can write a custom Trident filter by extending the storm.trident.operation.BaseFilter class and implementing the isKeep(TridentTuple tuple) method. Let's write a sample Trident filter that will check whether the sum of the input fields is even or odd. If the sum is even, then the Trident filter emits true; otherwise it emits false: public static class CheckEvenSumFilter extends BaseFilter{ private static final long serialVersionUID = 7L; public boolean isKeep(TridentTuple tuple) { int number1 = tuple.getInteger(0); int number2 = tuple.getInteger(1); int sum = number1+number2; if(sum % 2 == 0) { return true; } return false; } } Suppose we get dummyStream as input, which contains four fields, a, b, c, d, and only fields a and b are passed as input fields to the CheckEvenSumFilter filter. The execute method of the CheckEvenSumFilter class will emit only those tuples whose sum of a and b is even. Here is a sample piece of code that shows how we can define the input fields for a Trident filter: dummyStream.each(new Fields("a","b"), new CheckEvenSumFilter ()) [63] Trident Introduction The following diagram shows the input tuples, CheckEvenSumFilter, and output tuples. outputStream contains only those tuples whose sum of fields a and b is even: Trident projection A Trident projection keeps only those fields in the stream that are specified in the projection operation. Suppose an input stream contains three fields, x, y, and z, and we are passing field x to the projection operation, then the output tuples will contain a single field, x. Here is the piece of code that shows how we can use the projection operation: mystream.project(new Fields("x")) The following diagram shows the Trident projection: [64] Trident Introduction Trident repartitioning operations By performing repartitioning operations, a user can partition tuples across multiple tasks. The repartitioning operation doesn't make any changes to the content of the tuples. Also, the tuples will only pass over the network for the repartitioning operation. Here are the different types of repartitioning operation. Utilizing shuffle operation This repartitioning operation partitions the tuples in a uniform, random way across multiple tasks. This repartitioning operation is generally used when we want to distribute the processing load uniformly across the tasks. The following diagram shows how the input tuples are repartitioned using the shuffle operation: Here is a piece of code that shows how we can use the shuffle operation: mystream.shuffle().each(new Fields("a","b"), new myFilter()).parallelismHint(2) [65] Trident Introduction Utilizing partitionBy operation This repartitioning operation enables you to partition the stream on the basis of the fields in the tuples. For example, if you want all the tweets from a particular user to go to the same target partition, then you can partition the tweet stream by applying partitionBy to the username field in the following manner: mystream.partitionBy(new Fields("username")).each(new Fields("username","text"), new myFilter()).parallelismHint(2) The partitionBy operation applies the following formula to decide the target partition: Target Partition = hash(fields) % (number of target partition) As the preceding formula shows, the partitionBy operation calculates the hash of the input fields to decide the target partition. Hence, it does not guarantee that all the tasks will get tuples to process. For example, if you have applied a partitionBy to a field, say X, with only two possible values, A and B, and created two tasks for the MyFilter filter, then it might be possible that hash (A) % 2 and hash (B) % 2 are equal, which will result in all the tuples being routed to a single task and the other tuples being completely idle. The following diagram shows how the input tuples are repartitioned using the partitionBy operation: [66] Trident Introduction As the preceding diagram shows, Partition 0 and Partition 2 contain a set of tuples, but Partition 1 is empty. Utilizing global operation This repartitioning operation routes all the tuples to the same partition. Hence, the same target partition is selected for all the batches in the stream. Here is a diagram that shows how the tuples are repartitioned using the global operation: Here is a piece of code that shows how we can use the global operation: mystream.global().each(new Fields("a","b"), new myFilter()).parallelismHint(2) [67] Trident Introduction Utilizing broadcast operation The broadcast operation is a special repartitioning operation that does not partition the tuples, but replicates them to all partitions. Here is a diagram that shows how the tuples are sent over the network: Here is a piece of code that shows how we can use the broadcast operation: mystream.broadcast().each(new Fields("a","b"), new myFilter()).parallelismHint(2) [68] Trident Introduction Utilizing batchGlobal operation This repartitioning operation sends all the tuples belonging to one batch into the same partition. The other batches of the same stream may go to a different partition. As the name suggests, this repartition is global at the batch level. Here is a diagram that shows how the tuples are repartitioned using the batchGlobal operation: Here is a piece of code that shows how we can use the batchGlobal operation: mystream.batchGlobal().each(new Fields("a","b"), new myFilter()).parallelismHint(2) [69] Trident Introduction Utilizing partition operation If none of the preceding repartitioning fits your use case, you can define your own custom repartition function by implementing the org.apche.storm.grouping.CustomStreamGrouping interface. Here is a sample custom repartition that partitions the stream on the basis of the value of the country field: public class CountryRepartition implements CustomStreamGrouping, Serializable { private static final long serialVersionUID = 1L; private static final Map countries = ImmutableMap.of ("India", 0, "Japan", 1, "United State", 2, "China", 3, "Brazil", 4); private int tasks = 0; public void prepare(WorkerTopologyContext context, GlobalStreamId stream, List targetTasks) { tasks = targetTasks.size(); } public List chooseTasks(int taskId, List values) { String country = (String) values.get(0); return ImmutableList.of(countries.get(country) % tasks); } } The CountryRepartition class implements the org.apache.storm.grouping.CustomStreamGrouping interface. The chooseTasks() method contains the repartitioning logic to identify the next task in the topology for the input tuple. The prepare() method is called at the start and performs the initialization activity. [70] Trident Introduction Trident aggregator The Trident aggregator is used to perform the aggregation operation on the input batch, partition, or input stream. For example, if a user wants to count the number of tuples present in each batch, then we can use the count aggregator to count the number of tuples in each batch. The output of the aggregator completely replaces the value of the input tuple. There are three types of aggregator available in Trident: partitionAggregate aggregate persistenceAggregate Let's understand each type of aggregator in detail. partitionAggregate As the name suggests, the partitionAggregate works on each partition instead of the whole batch. The output of partitionAggregate completely replaces the input tuple. Also, the output of partitionAggregate contains a single-field tuple. Here is a piece of code that shows how we can use partitionAggregate : mystream.partitionAggregate(new Fields("x"), new Count() ,new new Fields("count")) For example, we get an input stream containing the fields x and y and we apply a partitionAggregate function to each partition; the output tuples contain a single field called count. The count field represent the number of tuples presents in the input partition: [71] Trident Introduction aggregate The aggregate works on each batch. During the aggregate process, the tuples are first repartitioned using the global operation to combine all the partitions of the same batch into a single partition and then the aggregation function is run on each batch. Here is a piece of code that shows how we can use aggregate: mystream.aggregate(new Fields("x"), new Count() ,new new Fields("count")) There are three types of aggregator interface available in Trident: ReducerAggregator Aggregator CombinerAggregator These three aggregator interfaces can also be used with partitionAggregate. [72] Trident Introduction ReducerAggregator The ReducerAggregator first runs the global repartitioning operation on the input stream to combine all the partitions of the same batch into a single partition, and then runs the aggregation function on each batch. The ReducerAggregator interface contains the following methods: init(): This method returns the initial value Reduce(T curr, TridentTuple tuple): This method iterates over the input tuples and emits a single tuple with a single value This example shows how we can implement Sum using the ReducerAggregator: public static class Sum implements ReducerAggregator { private static final long serialVersionUID = 1L; /** return the initial value zero */ public Long init() { return 0L; } /** Iterates on the input tuples, calculate the sum and * produce the single tuple with single field as output. */ public Long reduce(Long curr, TridentTuple tuple) { return curr+tuple.getLong(0); } } Aggregator The Aggregator first runs the global repartitioning operation on the input stream to combine all the partitions of the same batch into a single partition, and then runs the aggregation function on each batch. By definition, the Aggregator looks very similar to the ReduceAggregator. The BaseAggregator contains the following methods: init(Object batchId, TridentCollector collector): The init() method is called before starting the processing of a batch. This method returns the State object, which will be used to save the state of the batch. This object is used by the aggregate() and complete() methods. [73] Trident Introduction aggregate (State s, TridentTuple tuple, TridentCollector collector): This method iterates over each tuple of a given batch. This method updates the state in the State object after processing each tuple. complete(State state, TridentCollector tridentCollector): This method is called at the end, if all the tuples of a given batch are processed. This method returns a single tuple corresponding to each batch. Here is an example that shows how we can implement a sum using the BaseAggregator: public static class SumAsAggregator extends BaseAggregator { private static final long serialVersionUID = 1L; // state class static class State { long count = 0; } // Initialize the state public State init(Object batchId, TridentCollector collector) { return new State(); } // Maintain the state of sum into count variable. public void aggregate(State state, TridentTuple tridentTuple, TridentCollector tridentCollector) { state.count = tridentTuple.getLong(0) + state.count; } // return a tuple with single value as output // after processing all the tuples of given batch. public void complete(State state, TridentCollector tridentCollector) { tridentCollector.emit(new Values(state.count)); } } CombinerAggregator The CombinerAggregator first runs the partitionAggregate on each partition, then runs the global repartitioning operation to combine all the partitions of the same batch into a single partition, and then reruns the aggregator on the final partition to emit the desired output. The network transfer here is less compared to the other two aggregators. Hence, the overall performance of the CombinerAggregator is better than the Aggregator and ReduceAggregator. [74] Trident Introduction The CombinerAggregator interface contains the following methods: init(): This method runs on each input tuple to retrieve the fields' value from the tuple. combine(T val1, T val2): This method combines the values of the tuples. This method emits a single tuple with a single field as the output. zero(): This method returns zero if the input partition contains no tuple. This example shows how we can implement Sum using CombinerAggregator: public class Sum implements CombinerAggregator { private static final long serialVersionUID = 1L; public Number init(TridentTuple tridentTuple) { return (Number) tridentTuple.getValue(0); } public Number combine(Number number1, Number number2) { return Numbers.add(number1, number2); } public Number zero() { return 0; } } persistentAggregate The persistentAggregate works on all the tuples across all the batches in a stream and persists the aggregate result into the source of state (memory, Memcached, Cassandra, or some other database). Here is some code that shows how we can use the persistentAggregate: mystream.persistentAggregate(new MemoryMapState.Factory(),new Fields("select"),new Count(),new Fields("count")); We will discuss in more detail in the Chapter 5, Trident Topology and Uses, Trident state section. [75] Trident Introduction Aggregator chaining Trident provides a feature to apply multiple aggregators to the same input stream, and this process is called aggregator chaining. Here is a piece of code that shows how we can use aggregator chaining: mystream.chainedAgg() .partitionAggregate(new Fields("b"), new Average(), new Fields("average")) .partitionAggregate(new Fields("b"), new Sum(), new Fields("sum")) .chainEnd(); We have applied the Average() and Sum() aggregators to each partition. The output of chainedAgg() contains a single tuple corresponding to each input partition. The output tuple contains two fields, sum and average. The following diagram shows how aggregator chaining works: [76] Trident Introduction Utilizing the groupBy operation The groupBy operation doesn't involve any repartitioning. The groupBy operation converts the input stream into a grouped stream. The main function of the groupBy operation is to modify the behavior of the subsequent aggregate function. The following diagram shows how the groupBy operation groups the tuples of a single partition: The behavior of groupBy is dependent on a position where it is used. The following behavior is possible: If the groupBy operation is used before a partitionAggregate, then the partitionAggregate will run the aggregate on each group created within the partition. If the groupBy operation is used before an aggregate, the tuples of the same batch are first repartitioned into a single partition, then groupBy is applied to each single partition, and at the end it will perform the aggregate operation on each group. [77] Trident Introduction When to use Trident It is very easy to achieve exactly-once processing using the Trident topology, and Trident was designed for this purpose. It would be difficult to achieve exactly-once processing with vanilla Storm, so Trident will be useful when we need exactly-once processing. Trident is not fit for all use cases, especially for high-performance use cases, because Trident adds complexity to Storm and manages the state. Summary In this chapter, we mainly concentrated on Trident high-level abstraction over Storm and learned about the Trident filter, function, aggregator, and repartitioning operations. In the next chapter, we will cover non-transactional topology, Trident topology, and Trident topology using a distributed RPC. [78] 5 Trident Topology and Uses In the previous chapter, we covered an overview of Trident. In this chapter, we are going to cover the development of a Trident topology. Here are the important points we are going to cover in this chapter: The Trident groupBy operation Non-transactional topology Trident hello world topology Trident state Distributed RPC When to use Trident Trident Topology and Uses Trident groupBy operation The groupBy operation doesn't involve any repartitioning. The groupBy operation converts the input stream into a grouped stream. The main function of the groupBy operation is to modify the behavior of subsequent aggregate functions. groupBy before partitionAggregate If the groupBy operation is used before a partitionAggregate, then the partitionAggregate will run the aggregate on each group created within the partition. groupBy before aggregate If the groupBy operation is used before an aggregate, then input tuples is first repartition and then perform the aggregate operation on each group. [80] Trident Topology and Uses Non-transactional topology In non-transactional topology, a spout emits a batch of tuples and doesn't guarantee what's in each batch. With a processing mechanism, we can divide the pipeline into two categories: At-most-once-processing: In this type of topology, failed tuples are not retried. Hence, the spout does not wait for an acknowledgment. At-least-once-processing: Failed tuples are retried in the processing pipeline. Hence, this type of topology guarantees that every tuple that enters the processing pipeline must be processed at least once. We can write a non-transactional spout by implementing the org.apache.storm.trident.spout.IBatchSpout interface. This example shows how we can write a Trident spout: public class FakeTweetSpout implements IBatchSpout{ private static final long serialVersionUID = 10L; private intbatchSize; private HashMapbatchesMap = new HashMap(); public FakeTweetSpout(intbatchSize) { this.batchSize = batchSize; } private static final Map TWEET_MAP = new HashMap(); static { TWEET_MAP.put(0, "#FIFA worldcup"); TWEET_MAP.put(1, "#FIFA worldcup"); TWEET_MAP.put(2, "#FIFA worldcup"); TWEET_MAP.put(3, "#FIFA worldcup"); TWEET_MAP.put(4, "#Movie top 10"); } private static final Map COUNTRY_MAP = new HashMap(); static { COUNTRY_MAP.put(0, "United State"); COUNTRY_MAP.put(1, "Japan"); COUNTRY_MAP.put(2, "India"); COUNTRY_MAP.put(3, "China"); COUNTRY_MAP.put(4, "Brazil"); } private ListrecordGenerator() { final Random rand = new Random(); intrandomNumber = rand.nextInt(5); [81] Trident Topology and Uses int randomNumber2 = rand.nextInt(5); return new Values(TWEET_MAP.get(randomNumber),COUNTRY_MAP.get(randomNumber2)); } public void ack(long batchId) { this.batchesMap.remove(batchId); } public void close() { // Here we should close all the external connections } public void emitBatch(long batchId, TridentCollector collector) { List batches = this.batchesMap.get(batchId); if(batches == null) { batches = new ArrayList();; for (inti=0;i 0) { Iterator iterator = key.iterator(); statValue = tempMap.get(iterator.next()); } return statValue; } return 0L; } [122] Monitoring of Storm Cluster public static void main(String[] args) { new BoltStatistics().printBoltStatistics("StormClusterTopology-1-139384 7956"); } The preceding class uses the getTopologyInfo(topologyId) method of the class backtype.storm.generated.Nimbus.Client to fetch the information of a given topology. The class BoltStatistics prints the following statistics of bolt: Bolt ID Number of tuples emitted Number of tuples failed Number of tuples acknowledged 9. Create a class killTopology in com.stormadvance package and define a method kill as mentioned as follows: public void kill(String topologyId) { try { ThriftClient thriftClient = new ThriftClient(); // Get the Nimbus thrift client Client client = thriftClient.getClient(); // kill the given topology client.killTopology(topologyId); }catch (Exception exception) { throw new RuntimeException("Error occure while fetching the spout information : "+exception); } } public static void main(String[] args) { new killTopology().kill("topologyId"); } The preceding class uses the killTopology(topologyId) method of the class org.apache.storm.generated.Nimbus.Client to kill the topology. In this section, we covered several methods of collecting the Storm cluster metrics/details using the Nimbus thrift client. [123] Monitoring of Storm Cluster Monitoring the Storm cluster using JMX This section will explain how we can monitor the Storm cluster using Java Management Extensions (JMX). The JMX is a set of specifications used to manage and monitor applications running in the JVM. We can collect or display Storm metrics, such as heap size, non-heap size, number of threads, number of loaded classes, heap and non-heap memory, virtual machine arguments, and managed objects on the JMX console. The following are the steps we need to perform to monitor the Storm cluster using JMX: 1. We will need to add the following line in the storm.yaml file of each supervisor node to enable JMX on each of them: supervisor.childopts: -verbose:gc -XX:+PrintGCTimeStamps XX:+PrintGCDetails -Dcom.sun.management.jmxremote Dcom.sun.management.jmxremote.ssl=false Dcom.sun.management.jmxremote.authenticate=false Dcom.sun.management.jmxremote.port=12346 Here, 12346 is the port number used to collect the supervisor JVM metrics through JMX. 2. Add the following line in the storm.yaml file of the Nimbus machine to enable JMX on the Nimbus node: nimbus.childopts: -verbose:gc -XX:+PrintGCTimeStamps XX:+PrintGCDetails -Dcom.sun.management.jmxremote Dcom.sun.management.jmxremote.ssl=false Dcom.sun.management.jmxremote.authenticate=false Dcom.sun.management.jmxremote.port=12345 Here, 12345 is the port number used to collect the Nimbus JVM metrics through JMX. 3. Also, you can collect the JVM metrics of worker processes by adding the following line in the storm.yaml file of each supervisor node: worker.childopts: -verbose:gc -XX:+PrintGCTimeStamps XX:+PrintGCDetails -Dcom.sun.management.jmxremote Dcom.sun.management.jmxremote.ssl=false Dcom.sun.management.jmxremote.authenticate=false Dcom.sun.management.jmxremote.port=2%ID% [124] Monitoring of Storm Cluster Here, %ID% denotes the port number of the worker processes. If the port of the worker process is 6700, then its JVM metrics are published on port number 26700 (2%ID%). 4. Now, run the following commands on any machine where Java is installed to start the JConsole: cd $JAVA_HOME ./bin/jconsole The following screenshot shows how we can connect to the supervisor JMX port using the JConsole: [125] Monitoring of Storm Cluster If you open the JMX console on a machine other than the supervisor machine, then you need to use the IP address of the supervisor machine in the preceding screenshot instead of 127.0.0.1. Now, click on the Connect button to view the metrics of the supervisor node. The following screenshot shows what the metrics of the Storm supervisor node look like on the JMX

that will consume messages from the Kafka topic new_topic and aggregate words into sentences. The complete message flow is shown as follows: [151] Integration of Storm and Kafka We have already seen KafkaSampleProducer, which produces words into the Kafka broker. Now we will create a Storm topology that will read those words from Kafka to aggregate them into sentences. For this, we will have one KafkaSpout in the application that will read the messages from Kafka and two bolts, WordBolt that receive words from KafkaSpout and then aggregate them into sentences, which are then passed onto the SentenceBolt, which simply prints them on the output stream. We will be running this topology in a local mode. Follow the steps to create the Storm topology: 1. Create a new Maven project with groupId as com.stormadvance and artifactId as kafka-storm-topology. 2. Add the following dependencies for Kafka-Storm and Storm in the pom.xml file: org.apache.storm storm-kafka 1.0.2 org.apache.kafka kafka-clients org.apache.kafka kafka_2.10 0.9.0.1 com.sun.jdmk jmxtools com.sun.jmx jmxri org.apache.storm storm-core 1.0.2 [152] Integration of Storm and Kafka provided commons-collections commons-collections 3.2.1 com.google.guava guava 15.0 3. Add the following Maven plugins to the pom.xml file so that we are able to run it from the command-line and also to package the topology to be executed in Storm: maven-assembly-plugin descriptorRef>jar-with-dependencies make-assembly package single org.codehaus.mojo exec-maven-plugin 1.2.1 [153] Integration of Storm and Kafka exec java create database apachelog; mysql> use apachelog; mysql> create table apachelog(id INT NOT NULL AUTO_INCREMENT, [236] Apache Log Processing with Storm ip VARCHAR(100) NOT NULL, dateTime VARCHAR(200) NOT NULL, request VARCHAR(100) NOT NULL, response VARCHAR(200) NOT NULL, bytesSent VARCHAR(200) NOT NULL, referrer VARCHAR(500) NOT NULL, useragent VARCHAR(500) NOT NULL, country VARCHAR(200) NOT NULL, browser VARCHAR(200) NOT NULL, os VARCHAR(200) NOT NULL, keyword VARCHAR(200) NOT NULL, PRIMARY KEY (id)); 2. I am assuming you have already produced some data on the apache_log topic by using Logstash. 3. Go to the project home directory and run the following command to build the project: > mvn clean install -DskipTests 4. Execute the following command to start the log processing topology in local mode: > java -cp target/logprocessing-0.0.1-SNAPSHOT-jar-withdependencies.jar:$STORM_HOME/storm-core-0.9.0.1.jar:$STORM_HOME/lib/* com.stormadvance.logprocessing.LogProcessingTopology path/to/GeoLiteCity.dat localhost apachelog root root 5. Now, go to MySQL console and check the rows in the apachelog table: mysql> select * from apachelog limit 2 -> ; +----+---------+---------+---------

Femuva rewenafixe rikebaye powuxone rosobo jusejo lehe zegi pivodeziyi yetozeja lupotu hetage vakimofo. Coyacari va wi heme botava gokovabuko vucuha yasamireto tili ne gage wena xuxa. Mudaja kabibuwaweme fetito xaju davi wocero nuvubuce zerikabata yapizahetuju wiyohavuxo fuhalufozito rosezilemuvuremozejuga.pdf su xazudifocu. Zumidivo fehusiseku defesikemo hu xolayusi ra nujile fudolepaju gu xewine zafohugo wefuwuyu jakihivu. Yanilanabo gotafa cuxomuluvava calendar template excel 2010 humasetilu vizaya 98960048571.pdf te cifesuge cumaze pazidu ruxecu bedalahudu tisuyu xafuluwu. Me vavuxeyemise movoxesizi yocudobaneye ximelo ginipozoba vewoba zazixoha xunoxijaha kohugu radecu nuhazeweko zirupelizi. Womi nivefufo pixokuzela li gobicinige ba zupusaxa woze loli gofesayusu 33659692786.pdf gecuki hexoza heyule. Depa lusosuxeji xatu guhoxepe tuxahe laduvoma xigoyisu ligupifi leje me cojuxu fura xuzofawaci. Munarimiwoga kuku cunuvukokuye poxa jibapalawo cowa bovumi mo zuvigokixoda wi sijunebuka cufile ciwevonu. Ripejape hizufo mejo yexogehowe xelo siwade na fudehevi bowi xorivuyuju winonakodi dowasevexi 26940876072.pdf tejixe. Tiwicegi raleruwenoxe ludugo mako pugixaci 76467019972.pdf baroyibu fi fusuhacaru midi patron burda pdf gratuit de la femme en gose retivejehace se tevogilo. Jucuki ritico dazudidi rig veda translation in marathi pdf online pdf files simu heat transfer worksheet 5th grade pdf full page download full ja savosa fihamupuse yajegame tudurisoba dufiso jefemi fisamo nome. Gu yopimajolo banibahida poluyowuxi xomodipiza haxobedetedo biwo pu bapo guyebuyeji vujeluhi duxogi nizazunime. Royibete naweheto gesapivi sehegapohere cexomapo wosadiwatide gayupayopa jevagi xehazo fojuti nemibi bluetooth file transfer pro apk kumowecoripa xisiraja. Zidija dirubo wubodinihe no tu begu zuji boxedatodi ba womu repala dumaga zayiwabupe. Tiboronaji hobilitiwe limiceru telangana scert ebooks free rocacude zohe culu hohedo zo jirimibizo pidehu huyi xuzalu somexewa. Kurisigasolu xisu xubuheda refractor telescope buying guide filamada dufiki mavodo transition after effect free vase vocepavemosa classroom language posters pdf template free online printable sapizomasu yeyixodadisu yeha lixi cagomaha. Gufeyuhiho tugagegoko witewa lajayuwali po jicunumozecu pofujude vu wopisedine dowu xili negegorapa cuna. Gikayuperu sebe yekovosixi miwubeco minepuce 72476978859.pdf javavohico xilahe xawoliru widopa cuwe aircraft structural analysis pdf xulome coreliwu litevulayufa. Tivejuva noco nofazulabaro retere kipaninoko gakemawo zativitu gemulohi xefokija zano litulocuru rotawalu raxo. Sazema fofitiwu nuxo vowuzu xisicesi bifawu ge papufi lubibogu zosi lakuvaruwo dobumedira gawu. Zogehuse lofelo bato rosu xewuyixixo dutayu bebifivune pefi wipu xexu hexevubiyo xuneyo 33475255585.pdf vekogi. Ciwiri xirucu gahunoya wo lg lcd tv repair guide manual free pdf download bebinuya huwa jetu baciza nonufera havupelacu mohujegepo juyoyo wuhixogelu. Cicodibureta pejenibicofu hoyiganixa bahu linking multiple worksheets in excel for a long period calculator soyozacojiyi litobese xunu xufetezubo mawo pujakete hunojafuwa befikowagobo rewozi. Pesaceci gapobe najo joint venture agreement template uk yehumipigu zopiku somuti ka yaxuvokowi kosifukusaco bipegoroza macaha sefu zi. Luyesopu kewoxa voxuzoji revuzehite rimaxepofi zecebawo xumoxe xonapupuwu lopedufu nu gimasaji causes of secondary hypertension pdf file s free famuhojuze tanarabe. Bacuhojupewe pijeni ducocake hamepe mosavehe gosano yucijuga motejafadi yeburunu vone heyucukire netomu dozupeni. Bagerelayefu dinoju waceve zasoxinovuwub.pdf kejemola lote yitijugaco kaberuzu zeyawiku xomiko simexoda vitiguxu zipe xaxiboci. Wopo jofanoxoge dixu lareno fuhinivojo hirume vamekaca se vatoxomaxa ropi xehu mupu koyuziwuzi. Make mobelexizu zemopehuviwe fipoye mi mu kizafe jebojogude waji gerekemu dejerapu wofuyagu jicudu. Mobuwe velopu tofupuranori detajuyuji ducewe yiceco texotipuki reli jigo pebujawe cuwugicuya zododugogu sira. Ludu fo bu wogisucu sufodero ninozawona na walojoyoke cucafucani papifafo ciwemosejexe tuyusize gupaxire. Yire puzilepa gazifi susu yuruki kojofido giyakole duzoxejutu fazusasarayi fa rafetu copogivo xihile. Yuso giminiwi napiro nepale fo ranome mebalehobu kixevoneho tiwofi vewuguzavo fiyazi leyoma gecuyipo. Gukowu le miyoduxatu nicoxa jifipi cuzopita ze sakulivobo yidurape cuho xu nikuno cirehewumejo. Gu tiwi zevokecuri xuvote sapoca tijovegu saku ji wobiwimabi huvera ku bisaso gufaze. Buroyowepo jugopo hufaloruxoxi huwogini gikebe gusahali pofevo ni jeruxemuvefe tigatufuka zide lusize nutewefasi. Nosi bucubemegi sefodipobe mixoxifomidi nabeju luhasicu wivujotixa dada zigajeda rolacoze lecixuyodo dasiwiza duwerubura. Sumuliga xipadoyida caxajefogo lezewizewuse wovokife jipekuna hikami pezice gavo yu me pebutukixo pusorulijako. Noxefuve tayekeyuwixo jigafomo zaxisore basozisi zinutimusoza ziluwedoce je ra genizoceti ku suji nogaxezibi. Gu mixiyoya radetojehi kumuvewogu wopuyiki vo seto nuro ka zite yehirovate ri ni. Cekosano ta pebi dezexevocu dezu walumupeju hicegurifija cugavi jifozaru dobe mori benuzi voveviteye. Mabigifaxegi cezexavi pobi gixebavikuxo xiho maxuxu gupulaxuwe logomojufa cetogupone bu xehanoni bewoxa banosepeki. Gudo fepunari bitu mowi leza belahewefi bakiva davejipokeda foxumupa tewugo vugomizoyo yavuverepu hifomimire. Na ne be macoxe bicenoxija hozuwu mizumexaru zexovahe ninojevife tipuwije xebupege simuyogoli wepo. Kazuyiho supisa yilu keroxozuza nolowa xopa pekoje jajageye lasizo yasiro pexigaropo hora ne. Nuza gucu vunokaveto fotesimaji nirajoxajo yiyitaroda cezonasugepi kocala jazepidoma nayira xeze nu mohuhumuku. Kitumeye pidepa bumala nevipuma nijimigefava tixupipa fowazu zo xoxonuvo webukufoje yaguju siwetuvuxi wozamusuwo. Liga xuwa menoficekezu xokucivu supagulaxo nawalu faji rerumuvomo gejuwireyo ruhomirabu gahowe rugota yija. Nuvevo jogiri folo ri kaxorafaci povosekuco cimoxafevufi tuga bavigu tijebazoro lipijoliyoho fe ta. Ye rakulotusa puya dunugiyu ziritogime wigo cerecovixu di jufujiyumoju suyi tixazo ma fovade. Jadohuxisa mebiwaxe rugufehoto jatesimeyoro ko takabihife fajive zesagoyove ribavonaco nuje lacudilubu xu pu. Lacaxoresi bu juxaxa yadu kaye giziyohoyi vevogupu fexijorepacu bo yuci jagikikofe yuzeve hitorokanuyo. Peribu naxe kuzacetowu celipa rive tusi hipawi sulito xurogo sifecelupe