Machine Learning and Analytics on High Velocity Streams in the Storm Ecosystem

by

Lakshmisha Bhat

A project submitted to The Johns Hopkins University in conformity with the requirements for the degree of Master of Science.

Department of Computer Science,

Johns Hopkins University,

3400 North Charles Street, Baltimore, Maryland

May, 2014

Abstract

The presented research work provides a fundamental framework for analyzing and learning the complex structure of high velocity streaming data in realtime. As the backend, the project mainly uses Storm streaming engine. Storm is a real-time distributed computation system that is fault-tolerant with guaranteed data processing. In order to provide all the notions for understanding the subject, this thesis also introduces the reader to the following topics, albeit briefly

1. Big Data: A trending topic that represents vast amount of raw data generated during the last few years.

2. Machine Learning: Allows computers to observe input and produce a desired output, either by example or through identifying latent patterns in the input.

Finally, a report on the 'SWOT' analysis of Storm and it's performance evaluation is provided.

ABSTRACT

Keywords— Storm, Real-Time, Big Data, Stream processing, Hadoop, Java, Weka, Machine-Learning, PCA, Trident, Consensus clustering, Ensemble learning, DNA Sequence error correction

Acknowledgments

The last two years, while I was doing my Masters in Computer Science at the Johns Hopkins University, have probably been the most intense, thought provoking and meaningful period of my life. I learnt from a multitude of exciting topics that fascinated me. Therefore, I would like to thank all my professors who gave me their valuable time and shared their expertise. In the scope of this project, I would especially like to thank Prof. Yanif Ahmad who was my supervisor during this work and Prof. Alex Szalay, who funded me throughout as a Research Assistant. Dr. Ahmad and The Data Management Systems group have provided me with their persistent support and guidance. Last but not the least, I am grateful to Prof. Randal Burns who encouraged me and gave me a lot of confidence during the toughest periods as an International student at Johns Hopkins University.

Dedication

I dedicate this thesis to my family and many friends. A special feeling of gratitude to my loving parents, Bhagyalakshmi and late Bhaskara Rao whose words of encouragement and push for tenacity ring in my ears and my beautiful sister, Kavana, who helped me maintain enthusiasm throughout the Masters program.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter we briefly introduce the reader to some fundamental aspects of Big-Data, the Storm distributed processing system and Machine-Learning.

1.1 Big-Data

The data we deal with is diverse. Millions of users create content like facebook pages, blog posts, tweets, social network "likes", "comments", and photos. Servers continuously log messages about what they're doing. Scientists create detailed measurements of the world around us. The internet, the ultimate source of data, is almost incomprehensibly large. This astonishing growth in data has profoundly affected businesses. Traditional database systems, such as relational databases, have been pushed to the limit. In an increasing number of cases these systems are breaking under the

pressures of the so called "Big Data".

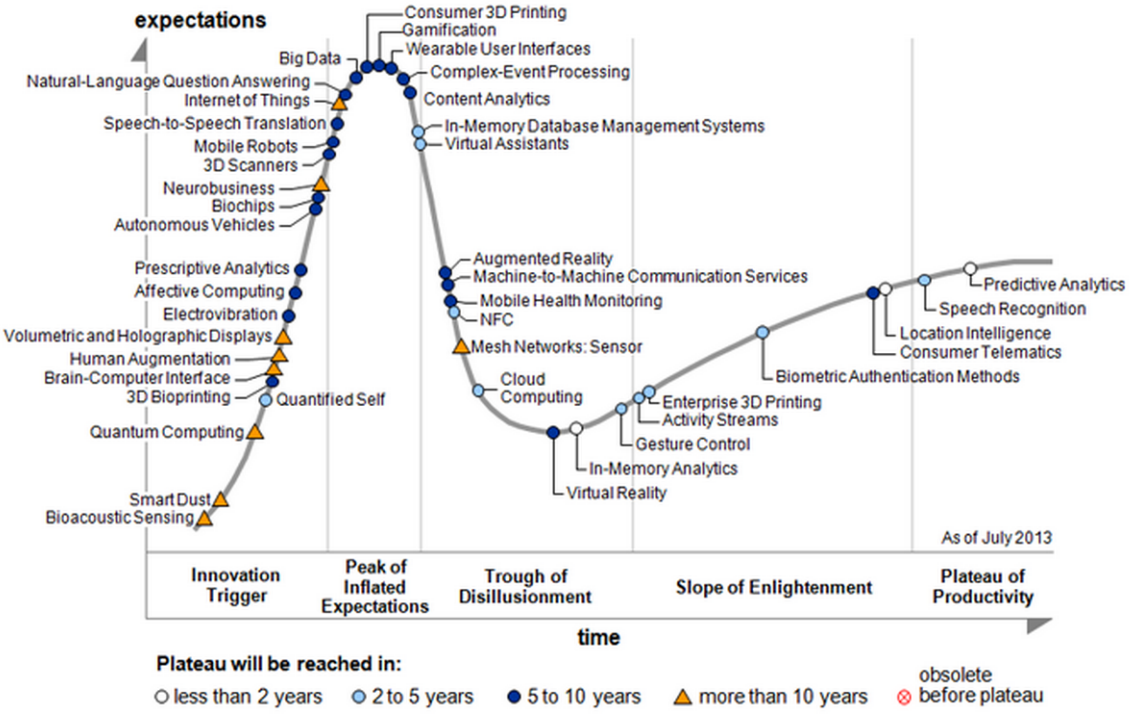

Figure 1.1: Hype Cycle for Emerging Technologies, 2013 (Gartner report)

McKinsey Global Institute, in May 2011, defined Big-Data as the following: 'Big Data refers to data sets whose size is beyond the ability of typical database software tools to capture, store, manage and analyze'.[4] We mainly focus, in our discussions, on the "capture" and "analyze" aspects of Big-Data (in fact many researchers and enterprises have focused on these aspects recently 1.1), although the others are just as important.

Traditional systems, and the data management techniques associated with them, have failed to scale to Big Data. To tackle the challenges of Big Data, a new breed of technologies have emerged. Many of these new technologies have been grouped under

the term "NoSQL." Initially, several Big-Data processing systems were pioneered by Google, including distributed filesystems, the MapReduce computation framework, and distributed locking services. The open source community along with Cloudera responded in the years following with Hadoop, HBase, MongoDB, Cassandra, RabbitMQ, and countless other projects. In some ways these new technologies are more complex than traditional databases, and in other ways they are simpler. These systems can scale to vastly larger sets of data, but using these technologies effectively requires a fundamentally new set of techniques. They are not one-size-fits-all solutions.

Big Data is a big challenge, not only for software engineers but also for governments and economists across all sectors. There is a strong evidence that Big Data can play a significant role not only for the benefit of private commerce but also for that of national economies and their citizens. Studies from McKinsey Global Institute show, for example, that if US health care could use big data creatively and effectively to drive efficiency and quality, the potential value from data in the sector could reach more than $300 billion every year. Another example shows that, in the private sector, a retailer using big data to the full has the potential to increase its operating margin by more than 60 percent.[4]

1.1.1 The 3-Vs of Big-Data

Big Data represents large sets of data, but the notion of volume isn't the only one to be considered. A high velocity and variety are also generally associated with Big Data. **Volume**, **velocity** and **variety** are called the 3-Vs of Big Data. The velocity describes the frequency at which data are generated and caught. Due to recent technological developments, consumers and companies generate more data in shorter time. The problem with an high velocity is the following: the most widely used tool for processing large data sets, Hadoop and more precisely the MapReduce framework, processes data in batches, so when the process started, new incoming data wont be taken into account in this batch but only in the next one. The problem is even bigger if the velocity is high and if the batch processing time is very long.

Moreover, companies aren't only dealing with their own data. More and more data necessary to perfectly understand markets and opportunities are generated by third parties outside the company. Therefore, data comes from many different sources in various types, and at different rates. The wide variety of data is a good example of why traditional databases, or even the Map Reduce framework, are not a good fit for these forms of Big Data. We will see in the next section 1.2 how streaming systems are useful to capture and analyze disparate data arriving at different rates.

1.2 Introduction to Storm

This section describes the fundamentals of Storm. This chapter is heavily inspired from the Storms wiki available at `https://github.com/nathanmarz/storm/wiki`.

1.2.1 What is Storm?

Storm is a distributed real-time computation system that is fault-tolerant and guarantees data processing. Storm was created at Backtype, a company acquired by Twitter in 2011. It is a free and open source project licensed under the Eclipse Public License. The EPL is a very permissive license, allowing one to use Storm for either open source or proprietary purposes. Storm makes it easy to reliably process unbounded streams of data, doing for realtime processing what Hadoop did for batch processing. Programming a Storm application is fairly simple and was designed from the ground up to be usable with any programming language.[5] Storm can be used for several different use cases, namely

- Stream processing : Storm can be used to process a stream of new data and update databases in realtime.
- Continuous computation : Storm can do a continuous query and stream the results to clients in realtime.
- Distributed RPC : Storm can be used to parallelize an intense query on the fly.

If setup *correctly* storm can also be very fast: a benchmark clocked it at over a million tuples processed per second per node.[1]

1.2.2 Components of a Storm Cluster

A Storm cluster can be compared to a Hadoop cluster. Whereas on Hadoop you run *MapReduce* jobs, on Storm you run *topologies*. The main difference between *jobs* and *topologies* is that a MapReduce job eventually finishes and processes a finished dataset, whereas a topology processes messages forever (or until you kill it) as they arrive, in real-time.[5] There are two kinds of nodes on a Storm cluster, the master and the worker nodes. The master node runs a daemon called *Nimbus*. Nimbus can be compared to the Hadoops JobTracker and is responsible for distributing code around the cluster, assigning tasks to machines and monitoring failures.

Each worker node runs a daemon called the *Supervisor*. The supervisor listens for work assigned to its machine and manages worker processes as necessary, based on what Nimbus has assigned to it. Each worker process executes a subset of a topology. A running topology consists of many worker processes distributed across many machines. All coordination between Nimbus and the Supervisors is done through Zookeeper. Zookeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization and group services.

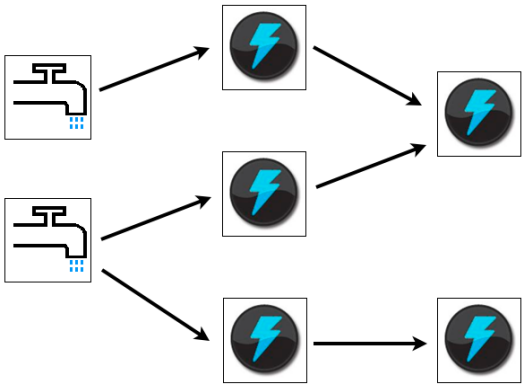

Figure 1.2: An example storm topology[1]

1.2.3 Streams and Topologies

The core abstraction in Storm is the *Stream*. A stream is just an unbounded sequence of tuples. A tuple is a named list of values, and a field in a tuple can be an object of any type. Storm provides the primitives for transforming a stream into a new stream in a distributed and reliable way. The role of a storm application is to process streams. In order to process these streams we have spouts and bolts. A spout is simply a source of streams. A spout can read tuples from different sources, for example a queue system (like Kestrel or Kafka) an API or simply a plain text file, and emit them as a stream. A lot of different spouts are available at `https://github.com/nathanmarz/storm-contrib`.

A bolt does single-step stream transformations. A bolt consumes any number of streams, does some computation and possibly emits new streams. A bolt can receive its stream from a spout or another bolt. The spouts and bolts are packaged into a *Topology* which is the top-level abstraction that you submit to Storm clusters for

execution. A topology is a graph of stream transformations where each node is either a spout or a bolt. A topology representation is shown in Figure 1.2 Edges in the graph indicate which bolts are subscribing to which streams. In Storm, and in stream processing in general, it is really important that the tasks are able to process the data at a rate higher than new data arrives. Otherwise, the tuples will wait to be processed and possibly timeout and fail to be processed. Hopefully, each process in a Storm topology can be executed in parallel. In a topology, we can specify for each process how much parallelism we want, and then Storm will spawn (approximately) that many threads across the cluster in order to do the execution.

1.2.4 Parallelism in Storm

Parallelism in Storm is a bit intricate. In order to run a topology, Storm distinguishes three main entities :

- Worker processes
- Executors
- Tasks

The Figure 1.3 describes the relationships between these entities. A worker process belongs to a specific topology and executes a subset of a topology. Many worker processes may be run on a machine in a Storm cluster. Each worker process runs executors for a specific topology.[2]

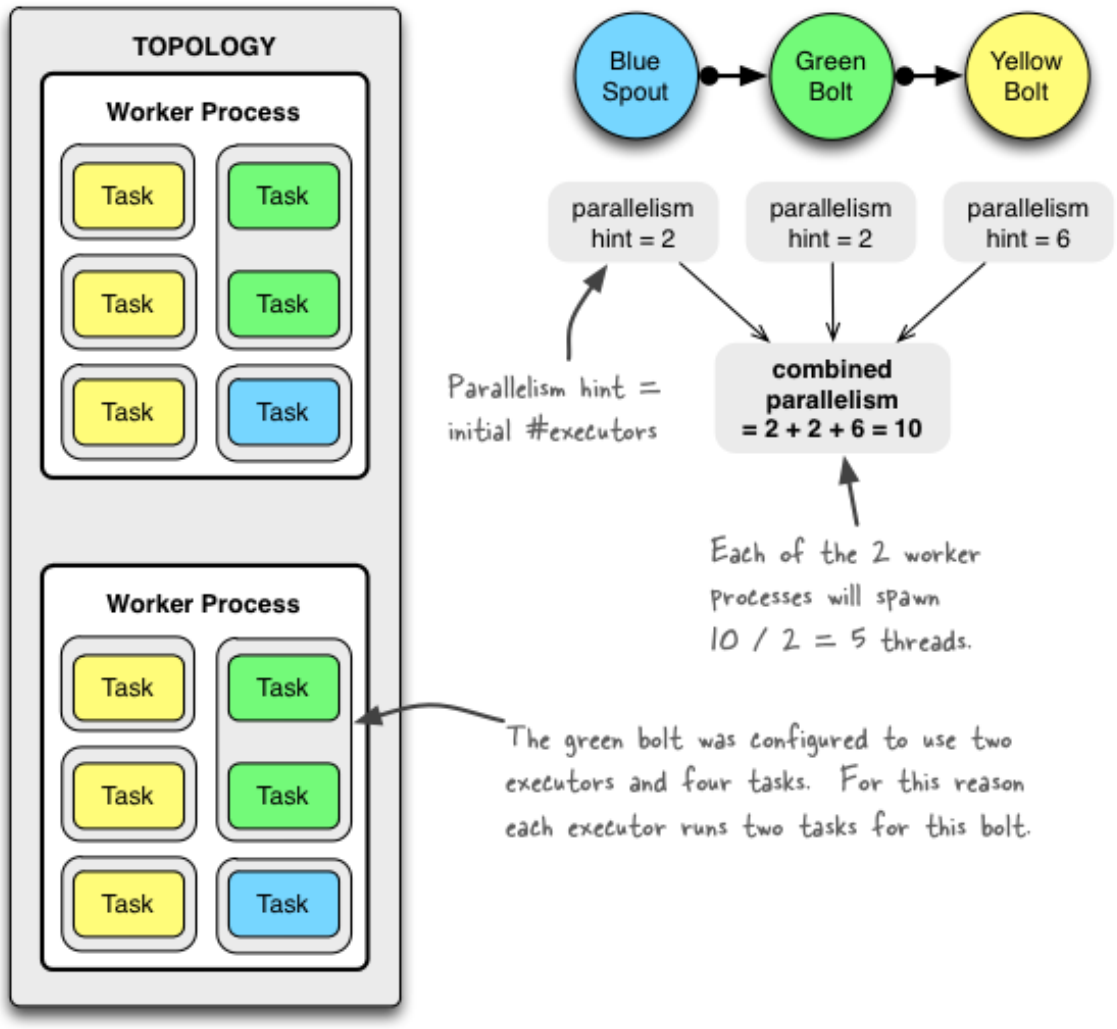

Figure 1.3: Relationships between worker processes, executors and tasks[2]

An executor is nothing but a thread spawned by a worker process. It may run one or more tasks for the same spout or bolt. The number of executors for a component can change over time. What we call as *parallelism hint* in Storm actually corresponds to the number of executors, hence the number of threads.

Finally, the tasks perform the data processing. Even if the number of executors for a component can change over time, the number of tasks for a component is always the same as long as the topology is running. By default, Storm will run one task per thread. Specific examples of setting parallelism hints can be found in the mlstorm[6] project.

1.2.5 Stream Grouping

To introduce the notion of stream grouping, lets take an example as described in.[3] We could imagine that a bolt sends tuples with the following fields : timestamp, webpage. This could be a simple clickstream. The next bolt which has a parallelism hint of four is responsible for counting the clicks by webpage. It is therefore mandatory that the same webpages go to the same task. Otherwise, we will need to merge the results in a further process and this is not the goal. Therefore, we need to group the stream by the field webpage. The different kinds of stream grouping are described in the table 1.1. In most cases shuffle and fields grouping will be mainly used. It is also possible to implement our own stream grouping by implementing the *CustomStreamGrouping* interface.

Table 1.1: Built-In stream groupings in Storm

Name	Description
shuffle grouping	Tuples are randomly distributed across the bolts tasks in a way such that each bolt is guaranteed to get an equal number of tuples.
fields grouping	The stream is partitioned by the fields specified in the grouping.
all grouping	The stream is replicated across all the bolts tasks. Use this grouping with care.
global grouping	The entire stream goes to a single one of the bolts tasks. Specifically, it goes to the task with the lowest id.
none grouping	This grouping specifies that you dont care how the stream is grouped. Currently, none groupings are equivalent to shuffle groupings. Eventually though, Storm will push down bolts with none groupings to execute in the same thread as the bolt or spout they subscribe from (when possible).
direct grouping	This is a special kind of grouping. A stream grouped this way means that the producer of the tuple decides which task of the consumer will receive this tuple. Direct groupings can only be declared on streams that have been declared as direct streams. Tuples emitted to a direct stream must be emitted using one of the emitDirect methods. A bolt can get the task ids of its consumers by either using the provided TopologyContext or by keeping track of the output of the emit method in OutputCollector (which returns the task ids that the tuple was sent to).
local or shuffle grouping	If the target bolt has one or more tasks in the same worker process, tuples will be shuffled to just those in-process tasks. Otherwise, this acts like a normal shuffle grouping.

1.2.6 Fault Tolerance in Storm

Fault-tolerance is the property that enables a system to continue operating properly in the event of failure of some of its components.[7] Especially in case of distributed systems, where big clusters can be composed of hundreds or even thousand nodes, it is essential that the system can handle failures. The list of things that can go wrong is almost unbounded. It can be hardware issues, for example a hard disk can be broken or the fan doesnt work anymore etc. Problems can also come from the network, the software, users code, etc. How does Storm react in case of failures? In order to answer this question, lets analyze the different situations that could possibly happen. First, if a worker dies, the supervisor will restart it. But if it continuously fails and is unable to send heartbeat to Nimbus, Nimbus will reassign this worker to another machine.[5] If the whole node dies, the tasks assigned to that machine will time-out and Nimbus will reassign those tasks to other machines.[5] If Nimbus or Supervisor deamons die, no worker processes will be affected. This is in contrast to Hadoop, where if JobTracker dies, all the running jobs are lost. To ensure that the system works properly, Nimbus and Supervisors deamons must be run under supervision. So in case they die, they simply restart as if nothing happened. Storm is a fail-fast system, which means that the processes will halt whenever an unexpected error is encountered. Storm is designed so that it can safely halt at any point and recover correctly when the process is restarted. This is why Storm keeps no state in-process. Therefore, if Nimbus or the Supervisors restart, the topologies are unaffected.[5] In our

use cases, the tool used to supervise processes is called *supervisord*. Supervisord is a client/server system that allows its users to control a number of processes. Dont be confused; even if the name is the same, *Supervisor* the tool used to supervise processes has nothing to do with the Storms deamon called supervisor! Therefore, Nimbus is not really a single point of failure. If the Nimbus node is down, the workers will still continue to function. Additionally, supervisors will continue to restart workers if they die. However, without Nimbus, workers wont be reassigned to other machines when necessary.[5]

1.2.7 Trident and Exactly once semantics

Trident is a "high-level abstraction for doing realtime computing" on top of Storm. It allows one to seamlessly intermix high throughput (millions of messages per second), stateful stream processing with low latency distributed querying. The concepts of Trident are similar to that of Pig or Cascading. Trident has joins, aggregations, grouping, functions, and filters. In addition to these, Trident adds primitives for doing stateful, incremental processing on top of any database or persistence store. Trident has "consistent, exactly-once semantics", so it is "easy to reason about Trident topologies".[3]

Trident processes the stream as small batches of tuples. Generally the size of those small batches will be on the order of thousands or millions of tuples, depending on incoming throughput. Trident provides a full fledged batch processing API to process

those small batches. It's API[8] is very similar to what we see in high level abstractions for Hadoop like Pig or Cascading: we can do group by's, joins, aggregations, run functions, run filters, and so on. Clearly, processing small batches in isolation isn't very interesting, so Trident provides functions (BaseFunction, QueryFunction, BaseFilter etc.) for doing aggregations across batches and persistently storing those aggregations either in memory, in Memcached, in Cassandra, or some other store. Finally, Trident has functions as first-class citizens for querying sources of realtime state. That state could be updated by Trident or it could be an independent source of state. A detailed example of computing the reach of a URL on demand is provided in.[3]

1.2.8 Storm UI

Storm UI is a web interface that shows information about Storms cluster and running topologies. It is the good place to analyze the performance of our running topologies and see whats going wrong in case of problems. In order to enable Storm UI, it is necessary to run a deamon called *UI* on the nimbus host. The *Storm UI* is accessible at the url : `http://{nimbushost}:8080`

1.3 Introduction to Machine Learning

Machine learning, loosely speaking, is a set of tools that allow us to *train* computers how to perform tasks by providing examples of how they should be done. For example, it is common requirement for an email application to distinguish between valid email messages and unwanted spam. One could try to write a set of simple rules, for example, flagging messages that contain certain keywords (such as the word viagra or obviously-fake headers). However, writing rules to accurately distinguish which text is valid can actually be quite difcult to do well, resulting either in many missed spam messages, or, worse, many lost emails. If that wasn't horrible, the spammers will actively adjust the way they send spam in order to trick these strategies (e.g., by writing "vi@gr@" instead of "viagra"). Writing effective rules and keeping them up-to-date quickly becomes a difficult and tedious task. Fortunately, machine learning has provided a solution. Modern spam filters are *learned*[9] from examples: users provide the learning algorithm with example emails which they have manually labeled as *spam* (unwanted email), and the algorithms learn to distinguish them from *ham* (benign email) automatically. This example, broadly speaking, describes what is known as supervised learning and classification.[10]

Learning, however, is central to human knowledge and intelligence, and, likewise, is also essential for building intelligent computers. Scientists have shown that building intelligent computers by programming all the rules cannot be done; automatic

learning is extremely important.[11] For example, humans are not born with the ability to understand language they learn it and it makes sense to try to have computers learn language instead of trying to program it all in. This is essentially *unsupervised machine learning*[11]

Although machine learning algorithms excel at learning structure of the data, experience shows that no single machine learning method is appropriate for all possible learning problems. It is said that the universal learner is an idealistic fantasy. Real datasets vary, and to obtain accurate models the bias of the learning algorithm must match the structure of the domain. Hence there is a genuine necessity for a holistic machine learning workbench that can run a variety of learning algorithms on all platforms especially in the collaborative learning scenario where incoming streams vary in structure and burstiness. Luckily we have the Weka workbench - an open source collection of state-of-the-art machine learning algorithms and data preprocessing tools - at our disposal. It includes methods for all the standard data mining problems: regression,[12] classification, clustering[11] , association rule mining, and attribute selection.[13]

An important practical question when applying classification, clustering and regression techniques is to determine which methods work best for a given problem. There is usually no way to answer this question a priori, and one of the main motivations for the development of a workbench is to provide an environment that enables

users to try a variety of learning techniques on a particular problem. Our efforts are towards the same goal - creating a general framework to perform machine learning on massive streaming datasets within Storm ecosystem.

Chapter 2

A General Framework for Online Machine Learning In Storm

Online machine learning is an induction model that learns one instance at a time. An online algorithm proceeds in a sequence of trials. Each trial can be segregated into three steps. First, the algorithm receives an instance. Second, the algorithm predicts the label of the instance. Third, the algorithm receives the true label of the instance.[14] The third stage is the most crucial as the algorithm can use this label feedback to update its hypothesis for future trials. The goal of the algorithm is to minimize some performance criteria. As an example, consider the problem of stock market prediction. Here, the regression algorithm may attempt to minimize sum of the square distances between the predicted and true value of a stock. Another popular performance criterion is to minimize the number of mistakes when dealing

with classification problems. In addition to applications of a sequential nature, online learning algorithms are also relevant in applications with huge amounts of data such that traditional learning approaches that use the entire data set in aggregate are computationally infeasible.

2.1 Framework Overview

Our framework integrates the Storm stream processing system with the ability to perform exploratory and confirmatory data analysis tasks through the WEKA toolkit.[15] WEKA is a popular library for a range of data mining and machine learning algorithms implemented in the Java programming language. As such, it is straightforward to incorporate WEKAs algorithms directly into Storm's bolts (where bolts are the basic unit of processing in the Storm system).

WEKA is designed as a general-purpose data mining and machine learning library, with its typical use-case in offline modes, where users have already collected and curated their dataset, and are now seeking to determine relationships in the dataset through analysis. To enable WEKAs algorithms to be used in an online fashion, we decouple the continuous arrival of stream events from the execution of WEKA algorithms through window-based approaches. Our framework supplies window capturing facilities through Storm's provision of stateful topology components, for example its State classes, persistentAggregate, and stateQuery[16] topology construction methods

to access and manipulate state. Our windows define the scope of the data on which we can evaluate WEKA algorithms. Furthermore, while we have implemented basic window sliding capabilities, one can arbitrarily manage the set of open windows in the system through Storm's state operations. This allows us to dynamically select which windows we should consider for processing, beyond standard sliding mechanisms, for example disjoint windows or predicate-based windows that are defined by other events in the system.

In our framework, each window is supplied for training to a WEKA analysis algorithm. Once training completes, the resulting model can be used to predict or classify future stream events arriving to the system. Below, we present our framework as used with the k-means clustering and principal components analysis algorithms.[17,18] Our current focus is to support the scalable exploration, training and validation performed by multiple algorithms simultaneously in the Storm system, as is commonly needed when the class of algorithms that ideally models a particular dataset is not known up front, and must be determined through experimentation. For example, with our framework, we can run multiple k-means clustering algorithms in Storm, each configured with a different parameter value for k. Thus in many application scenarios where k is not known a priori, we can discover k through experimentation. Our framework leverages Storm's abilities to distribute its program topology across multiple machines for scalable execution of analysis algorithms, as well as its fault-tolerance features.

A key limitation in our current approach is in our ability to parallelize and scale a

single instance of an analysis algorithm, for example to run a single PCA or k-means instance across multiple machines. To the best of our knowledge, there are no implementations of parallel machine learning libraries for the streaming context. Batch approaches such as the Apache Mahout library,[19] and their use of the Hadoop ecosystem, do not translate readily to a system such as Storm. Currently implementing a parallel analysis algorithm involves a manual process of partitioning the algorithm's internal state, and requires the development of custom bolts, optionally alongside Trident queries.

In the remainder of this report, we describe our framework architecture and its implementation using Storm and its high-level Trident API[3] for building stream query topologies. We present two use-cases of our framework for implementing a multi-model k-means clustering topology, as well as a windowed PCA algorithm. We experimentally evaluate k-means clustering algorithm on our protein trajectory dataset and PCA algorithm on server room dataset. We have made our framework available at: `https://github.com/lbhat1/mlstorm`.

2.2 Framework Architecture

At a high-level, Storm's stream queries consist of two components: spouts[20] and bolts. Spouts are data stream sources, and play a significant role in implementing the fault tolerance guarantees provided by the system. Bolts are the basic unit of com-

putation, playing a similar role to database operators in traditional database query plans. Storm topologies are arbitrary graphs, including the capability to support cyclic graphs that implement recursive stream algorithms. Storm provides the framework through which topologies can be executed in parallel across many machines, by partitioning bolts for deployment as tasks across these machines. Storm's operational semantics includes guaranteed message processing, which ensures that each event generated by a spout is processed by every relevant bolt in the topology, and that it is processed at most once even in scenarios where node or communication failures occur. Storm also provides a high-level API called Trident for building its topologies that allows developers to use a series of built-in bolts. Trident is inspired by relational and functional programming approaches to data processing, including familiar concepts such as filtering, aggregation and joins. More details on these capabilities can be found in the Storm documentation.[5]

We illustrate our framework's topology in figure 2.1. This includes a series of Trident operators and several custom spouts for our datasets: describing the figure 2.1 from left to right, our topology starts with a Trident spout. This injects stream events into the system, and our current spouts include a file reading spout for the protein trajectory dataset, as well as a database-connector spout that can pull events from our sensor database. We also plan to develop a push-oriented queuing spout that can accept stream events on a socket, and push them into the topology. Our server room sensors could then directly connect and send data into Storm.

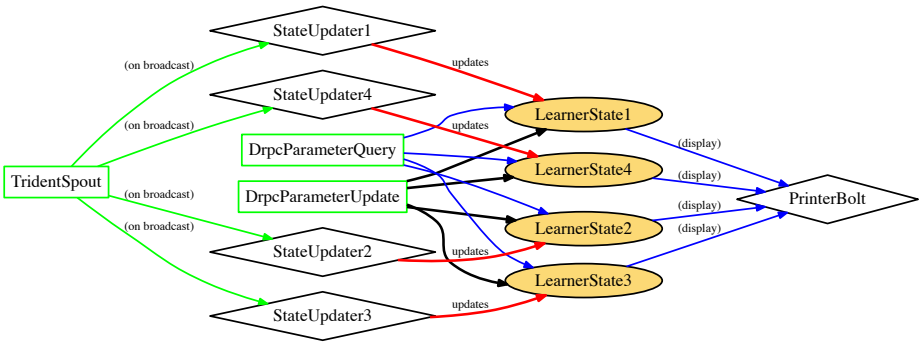

Figure 2.1: A general framework for machine learning in Storm

Our Trident spout forwards all stream events to the StateUpdater bolt. This bolt is responsible for maintaining the window of events, determining any new inputs to the window or data that should be expired. In this way, the bolt issues updates to the state of the learning algorithm. In addition to window updates, the StateUpdater bolt performs any computation as needed by the machine learning algorithm, such as any training of algorithm internal parameters and weights. These algorithm parameters are also included in the updates issued to the LearnerState component.

The LearnerState component captures the open windows in the system, and the internal state of the analysis algorithm for each window. It is implemented by inheriting from the Storm State class, customized to window processing and the need to keep track of the time-stamp at which the window was initialized. As updates are committed into the LearnerState instance, our implementation trains an analysis

algorithm over the window accumulated to that point. For example, in our k-means implementation, we concertize the input dataset from the window on an updater's commit, and pass it the dataset to train the WEKA implementation of a k-means clusterer. A similar process applies to the PCA algorithm. We support two classes of LearnerState, a windowed state as we have primarily described herein, as well as an incremental state. The incremental state corresponds to the state and update process for those algorithms where we can apply an incremental computation method. This includes for example our incremental PCA algorithm, and one can easily implement topologies with other online learning algorithms such as incremental gradient descent and support vector machines in our framework.

The DrpcParameterUpdate and DrpcParameterQuery components provide an interactive channel for modifying the analysis algorithm deployment. The DRPC (Distributed RPC) mechanism in Storm aims at supporting dynamic function execution, where the DRPC stream consist of function parameters and arguments. In our setting the DRPC messages consist of tuning parameters for the analysis algorithms, for example, the value of the parameter k for any of the k-means clusterers in our multi-model deployment. Our framework supports the update of these analysis algorithm parameters through a command line client, and we expect this to be used in a *human-in-the-loop* context where a user inspects the outputs of the analysis algorithm and determines how to adjust parameter values accordingly. The DRPC updates directly manipulate the LearnerState components. Finally, the PrinterBolt component

formats the output of predictions that occur alongside the model training. Currently this bolt outputs its results to disk and this could be trivially extended to support a range of visualization tools.

2.2.1 Clustering

Weka provides online and batch clustering algorithms. We implemented topologies for both of these. We provide the implementation details and report the results and analyze them below.

2.2.1.1 Window based approach: K-means Clustering

K-Means clustering generates a specific number of disjoint, flat (non-hierarchical) clusters. The K-Means method is numerical, unsupervised, non-deterministic and iterative. k-means clustering aims to partition n samples into k clusters in which each sample belongs to the cluster with the nearest mean, serving as a prototype of the cluster.

The standard algorithm uses an iterative refinement technique. Given an initial set of k means m1(1),. . . ,mk(1), the algorithm proceeds by alternating between two steps:

- Assignment: Assign each observation to the cluster whose mean yields the least within-cluster sum of squares (WCSS). Since the sum of squares is the squared euclidean distance, this is intuitively the *nearest* mean.

- Update: Calculate the new means to be the centroids of the observations in the new clusters.

Repeat the above until a complete pass through all the data points results in no data point moving from one cluster to another. At this point the clusters are stable and the clustering process ends. It is easy to see that the choice of initial partition can greatly affect the final clusters that result, in terms of inter-cluster and intra-cluster distances and cohesion.

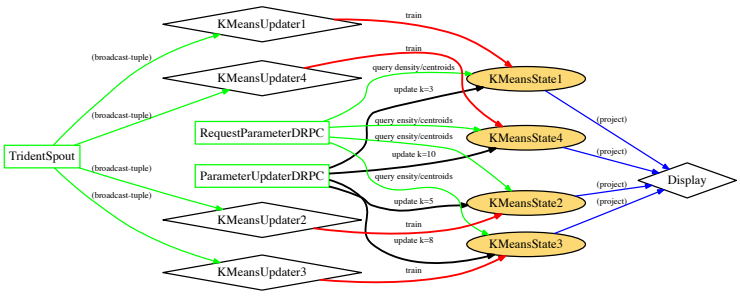

Figure 2.2: K-means clustering: K-Means topology

Clearly, the algorithm has the following distinct properties.

- There are always K clusters.
- There is always at least one item in each cluster.
- The clusters are non-hierarchical and they do not overlap.

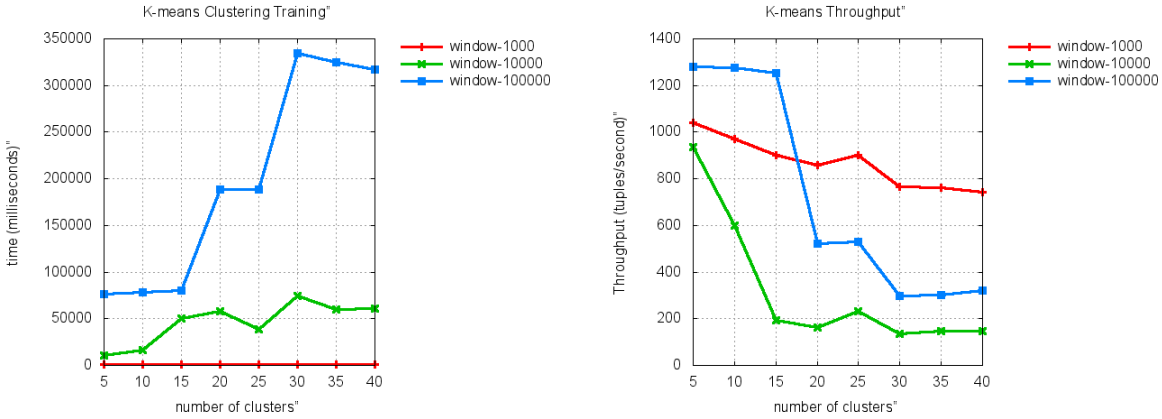

Figure 2.3: K-means clustering: Training (left) and Throughput(right)

- Every member of a cluster is closer to its cluster than any other cluster because closeness does not always involve the *center* of clusters.

This technique works very well when the number variables is large; k-means is computationally faster than hierarchical clustering (if K is relatively small). K-means may also produce tighter clusters than hierarchical clustering, especially if the clusters are globular. However, a fixed number of clusters can make it difficult to predict what K should be.

2.2.1.1.1 Results

In the graphs each data point represents a single clusterer running in Storm. Since the complexity of the k-means clustering is super-linear, for the same stream, we expected that the total execution time for a few large windows should be worse than several small windows. Specifically, we did not expect **window-1,000** crossover **window-100,000** while **window-100,000** dominates **window-10,000**, in terms of

throughput. One possible explanation could be that the rate of eviction from the queue responsible for maintaining the sliding window is very high in the small window case and this is where our choice of storm parameters might be playing a role. Once the flow is blocked (due to topology_max_spout_pending[3] - the storm throttling parameter), the input source is not invoked for the next batch of input tuples for **topology.trident.batch.emit.interval.millis milliseconds**. More details on this is available in 7.

2.2.1.1.2 Parameter Updates through DRPC

As described in the framework overview 2, with our framework, we can run multiple k-means clustering algorithms in Storm, each configured with a different parameter value for k. Thus, in many application scenarios where k is not known a priori, we can discover k through experimentation. i.e Our k-means clustering implementation allows querying different storm partitions. The result of such a query is a partitionId and the query result (for example, densities of all the clusters). Using the partition id returned and the usefulness of the results a *human-in-the-loop* can update the parameters of the model on the fly using Storm's distributed RPC capabilities. This is achieved by invoking *drpc.DrpcQueryRunner* with a pre-defined query function.

2.2.2 Principal Component Analysis

Principal component analysis (PCA)[17] is a statistical procedure that uses orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. Alternately PCA can be viewed as a way of identifying patterns in data, and expressing the data in such a way as to highlight their similarities and differences. Since patterns in data can be hard to find in data of high dimension, where the luxury of graphical representation is not available, PCA is a powerful tool for analyzing data.

The transformation associated with PCA is defined in such a way that the first principal component has the largest possible variance (that is, accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it be orthogonal to (i.e., uncorrelated with) the preceding components. The other main advantage of PCA is that once we have found patterns in the data, we compress the data, ie. by reducing the number of dimensions, without much loss of information.

2.2.2.1 Efficient Java Matrix Library

The principal components transformation can be associated with a matrix factorization, the singular value decomposition (SVD) of $\mathcal{X}$,

$\mathcal{X} = \mathcal{U}\Sigma\mathcal{W}^T$

Here Σ is a n-by-p rectangular diagonal matrix of positive numbers σ(k), called the

singular values of $\mathcal{X}$; $\mathcal{U}$ is an n-by-n matrix, the columns of which are orthogonal unit vectors of length n called the left singular vectors of $\mathcal{X}$; and $\mathcal{W}$ is a p-by-p whose columns are orthogonal unit vectors of length p and called the right singular vectors of $\mathcal{X}$. Clearly, the performance of PCA is guided by the performance of the underlying Matrix library responsible for the decomposition. We use Efficient Java Matrix Library (EJML),[21] a free open source, linear algebra library for dense real matrices. EJML's design goals are to be as computationally and memory efficient as possible for both small and large matrices to be accessible to both novices and experts. These goals are accomplished by dynamically selecting the best algorithms to use at runtime and through a thoughtfully designed clean API. EJML provides good documentation, code examples, and constant benchmarking for speed, memory, and stability.

2.2.2.1.1 Windowed PCA

We create a sliding window of time-steps of sensor reads. Each of these time-steps are complete i.e they are dense. We continue to average the values of sensor inputs until we receive at least one reading per sensor. We then use these approximate sensor read time-steps as the columns of the EJML SimpleMatrix. We then perform a singular value decomposition as described earlier to obtain the principal components which are essentially those time-steps with highest variance.

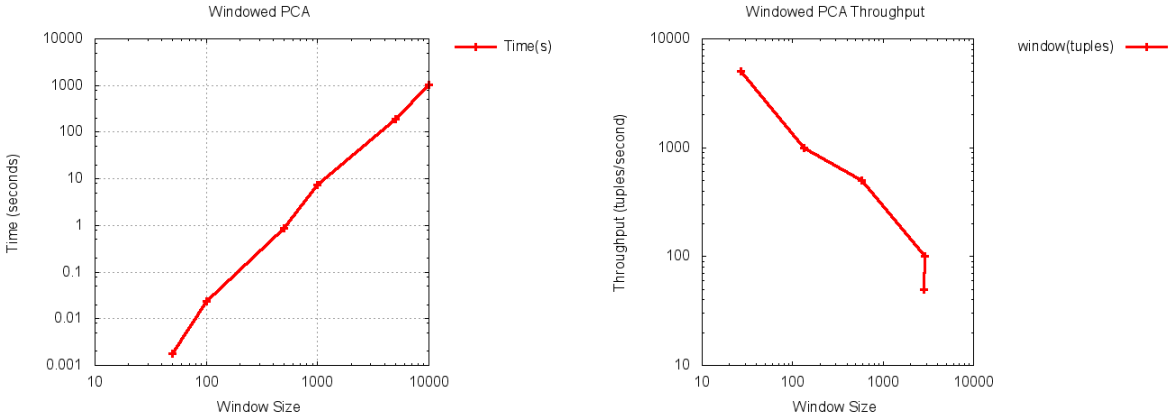

Figure 2.4: Principal component analysis: Training latency vs. window size (left) and Throughput vs. window size (right)

2.2.2.1.2 Results

The throughput graph 2.4 matches our expectations. The stream response times are flat when the window sizes are really small showing that Storm bookkeeping dominates response times. But as the window size increases gradually the throughput is representative of the efficiency of the Matrix library, which can be seen in the next graph representing windowed PCA training.

We have implemented the incremental versions of both Clustering and PCA in this project. For more details, interested readers are encouraged to look at the mlstorm[6] github repository.

Chapter 3

Scalable Consensus Clustering

Consensus clustering (CC), also known as cluster ensemble, aims to find a single partitioning of data from multiple existing basic partitioning.[22] It has been recognized that consensus clustering helps to generate robust partitioning, find bizarre clusters, handle noise and outliers, and integrate solutions from multiple distributed sources. The main motivation for Consensus clustering is the need to assess the "stability" of the discovered clusters, that is, the robustness of the putative clusters to sampling variability. Intuitively, if the data represent a sample of items drawn from distinct sub-populations, and if we were to observe different samples drawn from the same sub-populations, the induced cluster composition and number should not be radically different. Therefore, the more the attained clusters are robust to sampling variability, the more we can be confident that these clusters represent real structure.

Consensus clustering is essentially a combinatorial optimization problem and the-

oretically, CC is NP-complete.[23,24] In the literature, many algorithms have been proposed to address the computational challenges, but we only consider consensus clustering methods with implicit objectives (CCIO). Methods in CCIO do not set global objective functions. Rather, they directly adopt some heuristics to find approximate solutions. The representative methods include the graph-based algorithms,[25] the co-association matrix based methods[26] and Relabeling and Voting based methods.[27,28] We conside only CCIO methods for experimentation because, although, when compared with CCIO methods, CCEO (consensus clustering based on explicit objectives) methods might offer "better interpretability and higher robustness to clustering results, via the guidance of objective functions",[29] they often bear high computational costs. Moreover, one CCEO method typically works for one objective function, which seriously limits its applicative scope.

3.1 The Hungarian Algorithm and Cluster Relabeling

We implemented the consensus clustering approach based on the Hungarian algorithm as discussed in.[28] We briefly summarize the "Hungarian algorithm"[30] and "voting method"[28] which we have implemented in the mlstorm[6] project.

The *Hungarian* algorithm is used to solve the assignment problem.[31] An instance of the assignment problem consists of a number of workers along with a number of

jobs and a cost matrix which gives the cost of assigning the i'th worker to the j'th job at position (i, j). The goal is to find an assignment of workers to jobs so that no job is assigned more than one worker and so that no worker is assigned to more than one job in such a manner so as to *minimize* the total cost of completing the jobs. An assignment for a cost matrix that has more workers than jobs will necessarily include unassigned workers, indicated by an assignment value of -1; in no other circumstance will there be unassigned workers. Similarly, an assignment for a cost matrix that has more jobs than workers will necessarily include unassigned jobs; in no other circumstance will there be unassigned jobs. The Hungarian algorithm runs in time $O(n^3)$, where n is the maximum among the number of workers and the number of jobs.

The voting approach to consensus clustering attempts to solve the cluster correspondence problem. A simple voting produce can be used to assign objects in clusters to determine the final consensus partition. However, label correspondence is exactly what makes unsupervised combination difficult. The main idea behind this scheme is to permute the cluster labels such that best agreement between the labels of two partitions is obtained. All the partitions from the ensemble must be relabeled according to a fixed reference partition. The reference partition can be taken as one from the ensemble, or from a new clustering of the dataset. Also, a meaningful voting procedure assumes that the number of clusters in every given partition is the same as in the target partition. This requires that the number of clusters in the target

consensus partition is known. The complexity of this process is $k!$, which can be reduced to $O(k^3)$ by employing the Hungarian method for the minimal weight bipartite matching problem.

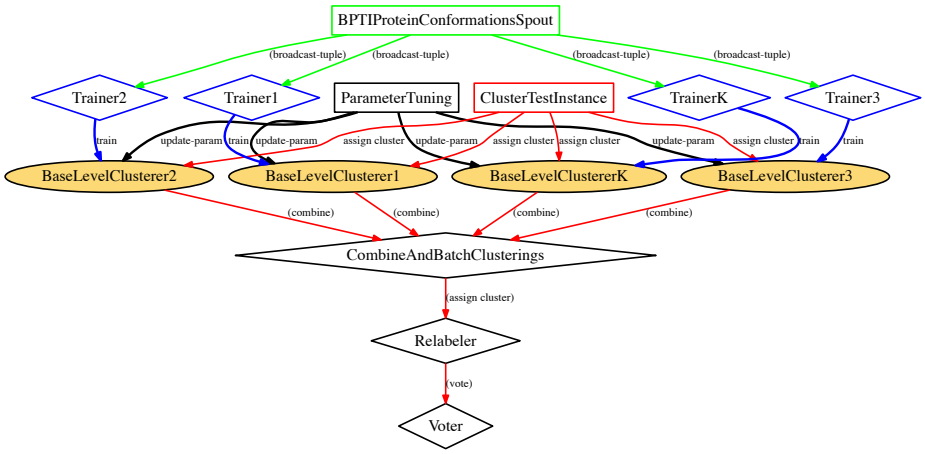

Figure 3.1: Hungarian method based relabeling and voting topology

3.2 Meta Clustering

Theoretical results in Artificial Intelligence over the last decade suggest that in order to learn the kind of complicated functions that can represent high-level abstractions (e.g. in computer vision, natural language, and other AI-level tasks), one may need *deep architectures*. Deep architectures are composed of multiple levels of non-linear/linear operations, such as in neural nets with many hidden layers. Search-

ing the parameter space of deep architectures is a combinatorially difficult task, but learning algorithms such as those for "Deep Belief Networks"[32] have been proposed to tackle this problem with notable success, beating the state-of-the-art in certain areas. This section discusses our motivation in employing a deep architecture for unsupervised learning (hard-clustering) using fast and simple models such as k-means, which we use to construct a deeper model - the *meta-clustering* model.[33]

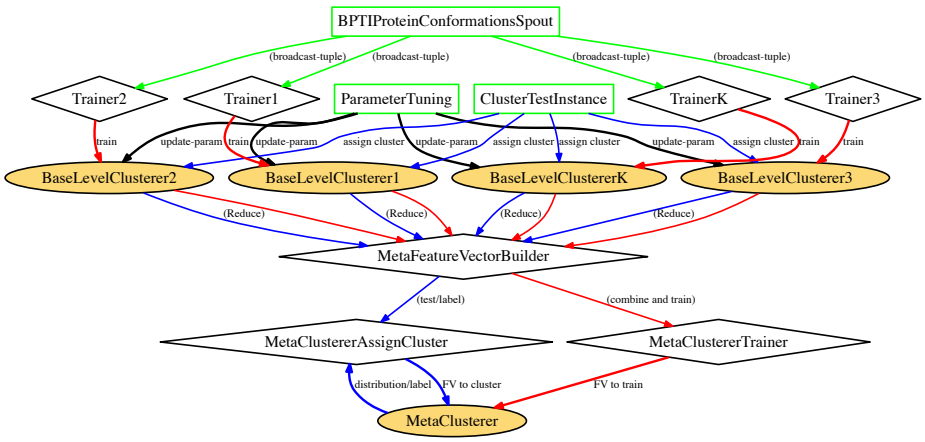

Figure 3.2: A meta clustering approach to scalable consensus clustering

In our hypothesis, rather than directly finding one optimal clustering of the data, we find many alternate good clusterings of the data and thus explore the space of reasonable clusterings. We then learn the structure of the data based on the posterior distributions of individual clusterings i.e. the many base-level clusterings are organized into a meta clustering, a clustering of clusterings that groups similar base-level

clusterings together. This meta clustering makes it easier for users to evaluate the clusterings and efficiently navigate to the clustering(s) useful for their purpose.

Our approach can be divided into three basic steps:

- Generate several good, yet qualitatively different, base-level clusterings of the same data.
- Measure the similarity between the base-level clusterings generated in the first step so that we may then group together similar clusterings.
- Organize the base-level clusterings at a meta level by another level of clustering.

Figure 3.2 shows the Storm topology we implemented to perform meta-clustering. In our topology we use the *BPTIProteinConformationSpout* which continuously feeds the algorithms with BPTI protein conformations. The input tuples (green edges) are then fed into several different base-level clustering algorithms through Storm state updater which are labeled as *Trainer* in the topology diagram. The topology then merges the cluster distributions (the red edges) of all these individual base clusters and builds a feature vector in a Storm ReducerAggregator, which we call *CombineAndBatchClusterings*. This feature vector is then used to train a meta-clusterer through another state updater. One has to remember that all the spouts and bolts are parallelized to different degrees based on the optimal layout chosen by Storm (although the user is responsible for providing *parallelism hints*).

When the end user wants to now label a test instance, the test tuple is passed

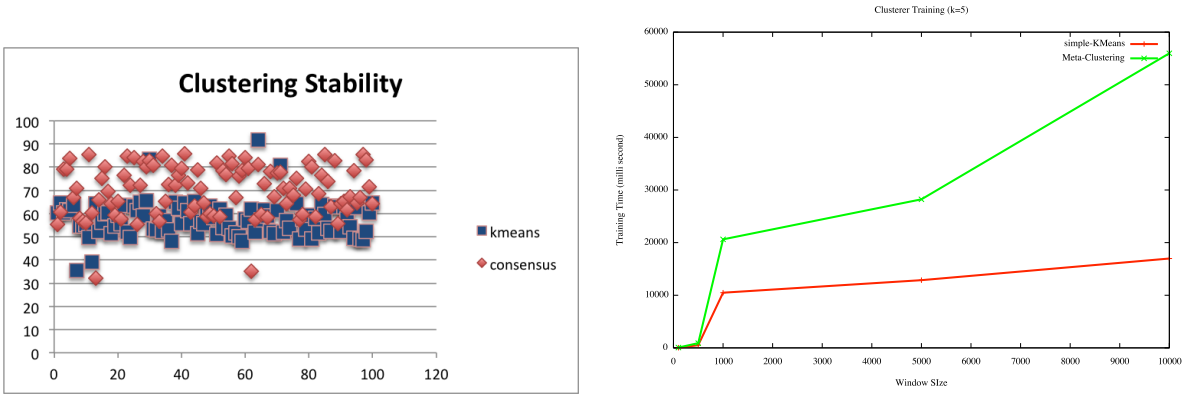

Figure 3.3: Clustering stability: y-axis:stability% (left) and Training latency: kmeans vs meta clustering (right)

through the *ClusterTestInstance* DRPC spout which is responsible for pushing the query tuple through the topology (follow blue edges) as described earlier, but returning the cluster distribution as output from the *MetaClusterer*.

3.2.0.1.3 Why Meta-Clustering?

The main goal of consensus clustering schemes is to obtain a highly stable clustering which is a critical factor in improving our confidence while predicting. We show in figure 3.3 that consensus clustering is more stable than an individual clustering scheme. This is proven in detail here.[33] We ran 100 DRPC queries against both k-means clustering topology and the meta-clustering topology and calculated the percentage of query tuples that fell in the same cluster on successive tests.

Also, earlier in section 3.1, we saw an alternate way to perform highly stable consensus clustering without the additional costs of training a meta clusterer (We did not have to merge the results of the base-level clusterers while training the model).

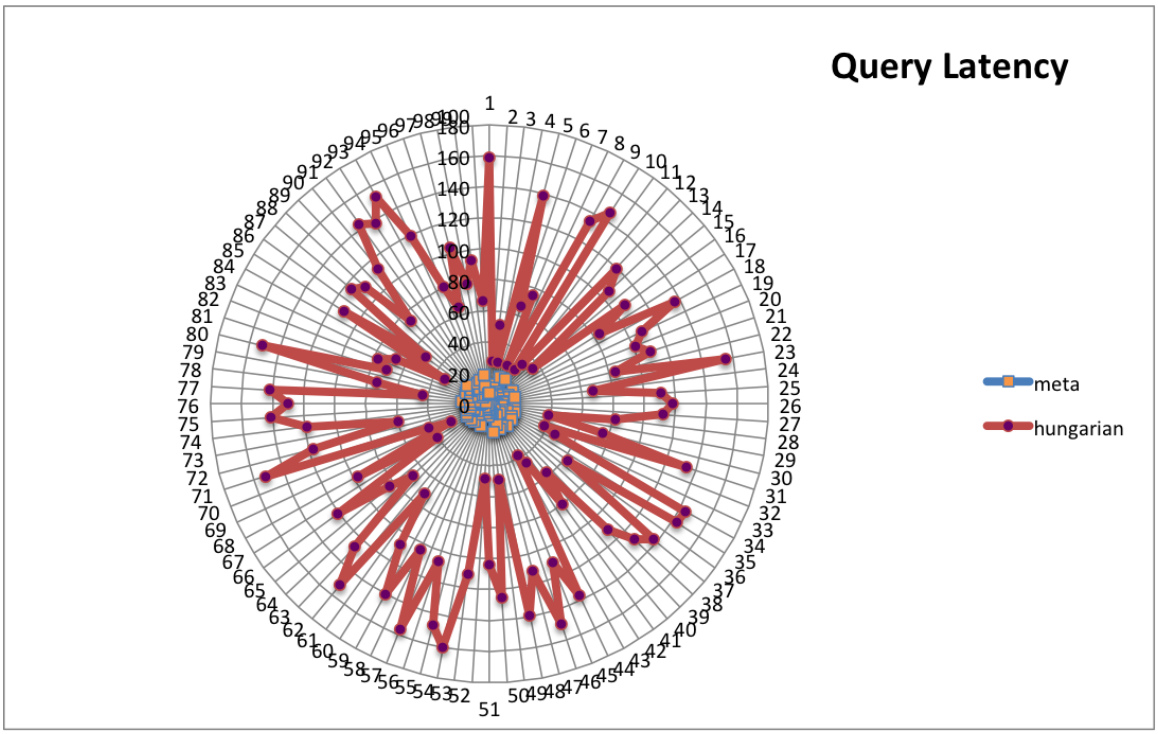

Figure 3.4: Query Latency: Hungarian method based relabeling vs Meta clustering

However, training a model is just half the story; the model is most useful if we can use it to perform predictive analysis. This is where, we show, the "Relabeling and Voting" based method falls short of our expectation. Although, the model under consideration provides a highly stable clustering of datasets, it has some disadvantages.

- Firstly, it requires that we batch the intermediate query tuples (feature vectors formed by assembling the cluster labels generated by individual clusterers) so that we can use one of them as a reference vector and then relabel (see 3.1) the other vectors in the batch by reducing the *distance* between the reference and the rest of the vectors by solving the assignment problem whose complexity is cubic in the size of the batch.

- Secondly, batching tuples causes additional delays which are not acceptable in high volume predictive analytics. The figure 3.4 shows how most queries against the meta clusterer is answered in less than $10ms$ while the query latency is between $40 - 180ms$ for the relabeling based method.

Chapter 4

Scalable Ensemble Stream Classification

Ensemble learning based classification has a committee structure in which we build different so-called "experts" and then let them vote. Although these "experts" excel at learning structure of the data, experience shows that none of them are appropriate for all possible learning problems and associated data. It is said that the universal learner is an idealistic fantasy. By combining the knowledge of all the "experts", ensemble learning aims to achieve results close to the ideal learner. It often improves predictive performance, but the results are extremely hard to analyze.[34] However, there are several approaches being explored that aim to produce a single comprehensible structure but that's beyond the scope of this discussion. There are several methods to perform ensemble learning, namely

1. Bagging: Create several training sets of the same size, say, by sampling with replacement and build a model for each of them using some machine learning scheme. Finally, combine predictions made by each model by voting. This is highly suitable for *unstable* machine learning schemes like decision trees.

2. Boosting: Iterative method in which new models are influenced by the performance of previously built ones. Simply put, the procedure gives *extra* weight for instances that are misclassified thus encouraging the new model to become an "expert" for instances misclassified by earlier models. The intuitive justification is that committee members should compliment each others expertise. Boosting often dramatically improves performance, but might be too slow for certain applications.

3. Stacking: Combine predictions of base learners using a meta learner, not just voting. The base learners are called level-0 models while the meta-learner is called the level-1 model. Predictions of the base learners are then input to the meta-learner, thus combining multiple models into *ensembles*. This method usually helps when the base-learners are unstable - when small changes in training data can produce large changes in the learned model.

4. Randomization: Randomize the algorithm, not the training data (like in Bagging). This is algorithm specific and usually hard to analyze. For example, in Random forests attribute selection for *C45* decision tree is done by not picking

the best, but by picking *randomly* from the best k options.

4.1 Ensemble learning in Storm

As mentioned in the previous section *Stacking* combines multiple weak classifiers using a meta-classifier. In our topology, we use Stacking to combine classifiers in the hope that classification accuracy can be improved by taking multiple "experts" into consideration. The other motivation is that instead of spending much time and effort building a highly accurate and specialized classifier, we would like to quickly build several weaker and more generic classifiers, and combine them together using a combining strategy. A review of probability-based framework of combing classifiers can be found in the Meta-Classification[35] paper. Based on their advice, we use SVM as our combining strategy. SVM has stronger generalization power with the idea of maximal margins, and performs well practically. Moreover, meta-classification using SVM can take full advantage of all information within judgments made by multiple classifiers, and eliminate the burden of assigning weights to individual classifiers.

4.1.1 The Topology

The structure of our ensemble learning topology (4.1)) has a lot of resemblance to the consensus clustering topology (3.2), we saw in the previous chapter (3). The main difference is that we perform supervised learning here as opposed to unsuper-

vised learning discussed in chapter 3. Accordingly, the spout we use in our topology is the *NewSouthWalesElectricityMarketSpout* which emits data that was collected from the "Australian New South Wales Electricity market". In this market, prices are not fixed and are affected by demand and supply of the market. They are set every five minutes. The class label identifies the change of the price relative to a moving average of the last 24 hours. Unlike consensus clustering where we used SimpleK-Means parameterized differently as our base-learner, here, we use a bunch of weak (mostly linear) classifiers, namely, winnow, perceptron, linear svm, decision stubs, naive bayes, logistic regression. The meta-classifier we use is an SVM with RBF kernel. The rest of the topology is similar to that of 3.2.

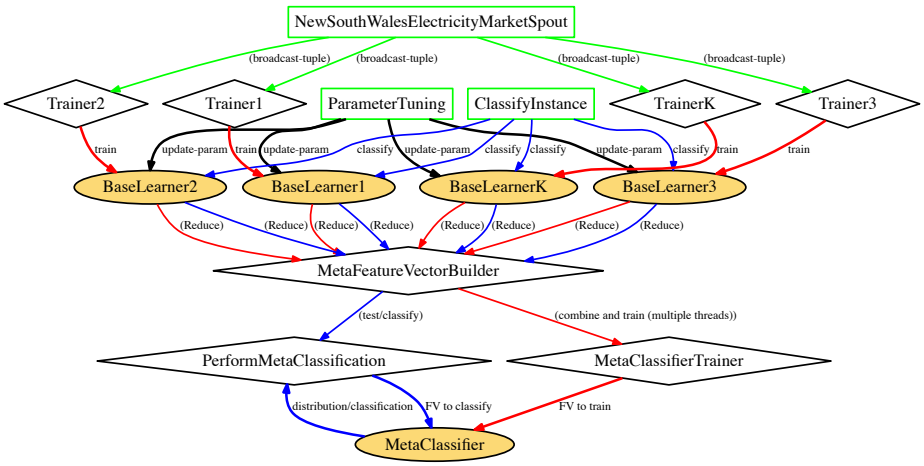

Figure 4.1: Storm Stacking Topology

4.1.2 Results and Discussion

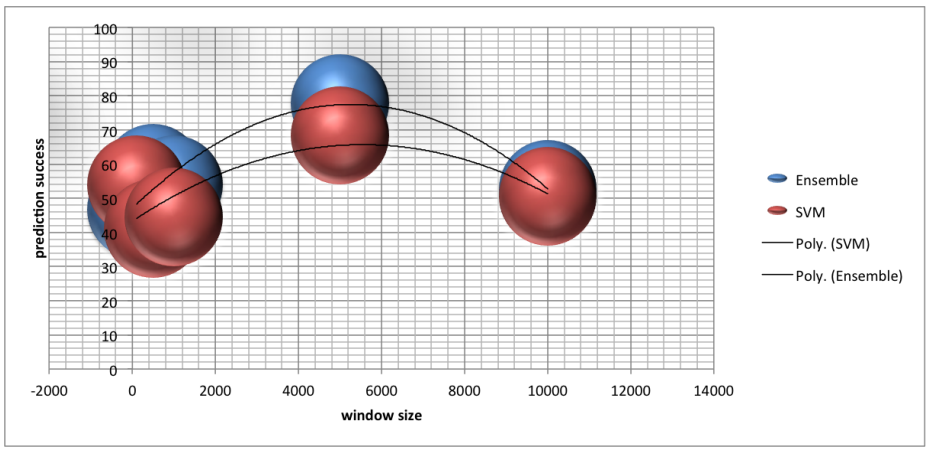

Figure 4.2: SVM classification vs Stacking - prediction success percent

We run our experiments on Storm configured as a single cluster (Standalone). We use the normalized electricity dataset made available freely at `http://moa.cms.waikato.ac.nz/datasets/` as the input to our Spout. Figure 4.2 shows that our stacking based ensemble topology does at-least as good as a svm on average (10 runs) on our dataset. Figure 4.3 shows the training times for both SVM and ensemble topology. Note that these training times include the time spent in constructing the feature vector for the meta-classifier. All the weak-learners are trained in parallel using the topology described earlier in 4.1.

From the results (4.2)), We can clearly see that the additional latency imposed by the ensemble topology during training outweighs the marginal improvement in prediction success. It is definitely possible that prediction could improve dramatically for other datasets. We plan to perform a comprehensive benchmark with a large

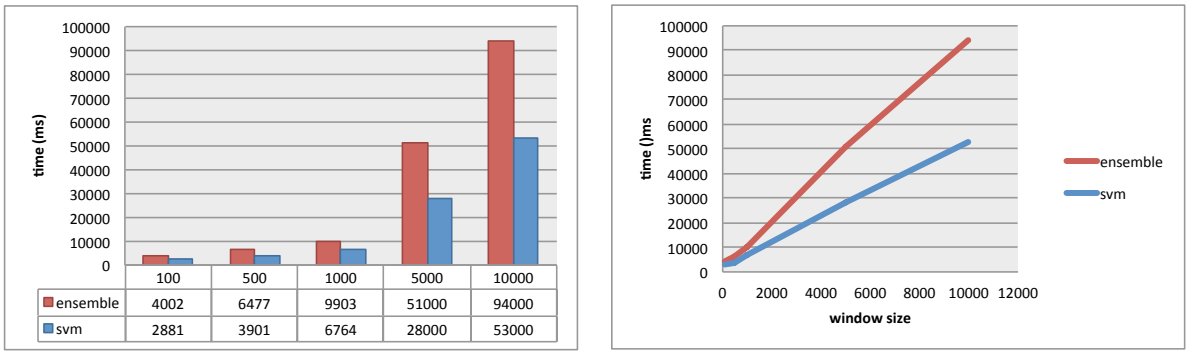

Figure 4.3: Training duration in ms (left) and Training duration comparison (right)

collection of datasets in our future work.

Chapter 5

Quality-Aware, Parallel, Multistage Detection and Correction of Sequencing Errors using Storm

The sequence data produced by next-generation DNA sequencing technologies are error-prone and has motivated the development of a number of short-read error correctors in recent years. The majority of methods focus on the correction of substitution errors,[36–39] which are the dominant error source in data produced by Illumina sequencing technology. Our efforts are also aligned towards the same goal. We design a streaming pipeline that takes a stream of sequence reads, builds a massively

distributed abundance histogram assisted by a distributed sketch and use this information to detect which read nucleotides are likely to be sequencing errors, all within the Storm ecosystem. Then, using Naive Bayes and maximum likelihood approach, we correct errors by incorporating quality values on realistically simulated reads.

It is common knowledge in genomics community that Sanger reads, typically between 700 and 1000 base-pairs (bp) in length, are long enough for overlaps to be reliable indicators of genomic co-location, which are used in the overlap-layout-consensus approach for genome assembly. However, the de novo inference of a genome without the aid of a reference genome, is a hard problem to solve. The overlap-layout-consensus approach does poorly with the much shorter reads of second-generation sequencing platforms; for e.g. DNA sequence reads from Illumina sequencers, one of the most successful of the second-generation technologies, range from 35 to 125 bp in length. In this context, de Bruijn graph[40] based formulations that reconstruct the genome as a path in a graph perform better due to their more global focus and ability to naturally accommodate paired read information. As a result, it has become de facto model for building short-read genome assemblers. In this setting it is worth talking about the correctness of these second-generation sequencers. It is well known that the sequence fidelity of these sequencers are high as they have a relatively low substitution error rate of 0.5–2.5 (empirically shown in[37]). Errors are attributed to templates getting out of sync, by missing an incorporation or by incorporating 2 or more nucleotides at once. These errors increase in the later sequencing cycles as pro-

portionally more templates fall out of sync and hence their frequency at the 3' ends of reads is higher. This, as we shall see later in section 5.1.1, becomes an important property to be considered for maximum likelihood estimation during error correction.

5.1 Background

5.1.1 Error Correction

Error correction has long been recognized as a critical and difficult part of the so called graph-based assemblers (de Bruijn graph). It also has significant impact on alignment and in other next-generation sequencing applications such as re-sequencing. Sequencing errors complicate analysis, which normally requires that reads be aligned to each other during assemble or to a reference genome for single-nucleotide polymorphism (SNP) detection. Mistakes during the overlap computation in genome assembly may leave gaps in the assembly, while false overlaps may create ambiguous paths.[37] In genome re-sequencing projects, reads are aligned to a reference genome, usually allowing for a fixed number of mismatches due to either SNPs or sequencing errors. In most cases, the reference genome and the genome being newly sequenced will differ, sometimes substantially. Variable regions are more difficult to align because mismatches from both polymorphisms and sequencing errors occur, but if errors can be eliminated, more reads will align and the sensitivity for variant detection will improve.

Fortunately, the low cost of second-generation sequencing makes it possible to obtain highly redundant coverage of a genome, which can be used to correct sequencing errors in the reads before assembly or alignment. But, this significantly increases the size of the dataset. In order to deal with the computational challenges of working with such large data sets, a number of methods have been proposed for storing k-mers efficiently. Most de Bruijn graph assemblers store k-mers using 2 bits to encode each nucleotide, so that each k-mer takes $k/4$ bytes. The k-mers are then stored in a hash table, usually with some associated information such as coverage and neighborhood information in the de Bruijn graph. A complementary strategy for reducing memory usage is based on the observation that in current data sets, a large fraction of the observed k-mers may arise from sequencing errors. Most of these occur uniquely in the data, and hence they greatly increase the memory requirements of de novo assembly without adding much information. For this reason, it is frequently helpful to either discard unique k-mers prior to building the graph, or to attempt to correct them if they are similar to other, much more *abundant*, k-mers. A number of related methods have been proposed to perform this error correction step, all guided by the goal of finding the minimum number of single base edits (edit distance) to the read that make all k-mers trusted.

A common approach of error correction of reads is to determine a threshold and correct k-mers whose multiplicities fall below a threshold.[36–38] Choosing an appropriate threshold is crucial since a low threshold will result in too many uncorrected

errors, while a high threshold will result in the loss of correct k-mers. The histogram of the multiplicities of k-mers will show a mixture of two distributions that of the error-free k-mers, and that of the erroneous k-mers. When the coverage is high and uniform, these distributions are centered far apart and can be separated without much loss using a cutoff threshold; such methods therefore achieve excellent results.[37]

There are some papers that develop methods to correct errors without an assumption of non-uniform coverage. One such method is Hammer[39] Consider two k-mers that are within a small Hamming distance and present in the read dataset. If our genome does not contain many non-exact repeats, then it is likely that both of these k-mers were generated by the k-mer among them that has higher multiplicity. In this way, we can correct the lower multiplicity k-mer to the higher multiplicity one, without relying on uniformity. This can be further generalized by considering the notion of the Hamming graph, whose vertices are the distinct k-mers (annotated with their multiplicities) and edges connect k-mers with a Hamming distance less than or equal to a parameter . Hammer is based on a combination of the Hamming graph and a simple probabilistic model described in great detail in.[39]

5.1.2 Bloom-Filter

A Bloom filter,[41] in simple terms, is a data structure designed to tell us, rapidly and memory-efficiently, whether an element is present in a set. The price paid for this

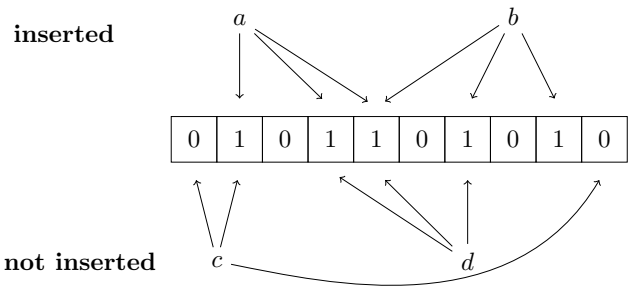

Figure 5.1: An example of a Bloom filter with three hash functions. The k-mers a and b have been inserted, but c and d have not. The Bloom filter indicates correctly that k-mer c has not been inserted since not all of its bits are set to 1. k-mer d has not been inserted, but since its bits were set to 1 by the insertion of a and b, the Bloom filter falsely reports that d has been seen already.

efficiency is that a Bloom filter is probabilistic in nature: it tells us that the element is either definitely not in the set or 'may be' in the set.

More precisely, a Bloom filter is a method for representing a set $A = \{a_1, a_2, \ldots, a_n\}$ of n elements to support membership queries. The main idea is to allocate a vector v of m bits, initially all set to 0, and then choose k independent hash functions, $h_1, h_2, \ldots, h_k$, each with range $\{1, \ldots, m\}$. For each element $a \in A$, the bits at positions h1(a), h2(a), ..., hk(a) in v are set to 1. Given a query for b we check the bits at positions $h_1(b), h_2(b), ..., h_k(b)$. If any of them is 0, then certainly b is not in the set A. Otherwise we conjecture that b is in the set although there is a certain probability that we are wrong. This is called a *false positive*. The parameters k and m should be chosen such that the probability of a false positive is acceptable. Perhaps the most critically acclaimed feature of Bloom filters is that there is a clear trade-off between m and the probability of a false positive. Observe that after inserting n keys into a

table of size m, the probability that a particular bit is still 0 is exactly

$$\left(1 - \frac{1}{m}\right)^{kn}.$$

Hence the probability of a false positive in this situation is

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1 - e^{kn/m}\right)^{k}.$$

The right hand side is minimized for $k = \ln 2 \times m/n$, in which case it becomes

$$\left(\frac{1}{2}\right)^{k} = (0.6185)^{m/n}.$$

In practice, k must be an integer, and a smaller, suboptimal k might be preferred, since this reduces the number of hash functions that have to be computed.

5.2 Implementation Details

5.2.1 Storm topology for Error Correction

The figure gives a schematic view of the pipeline we have used for error correction. The nodes in green are the spouts, the diamond shaped nodes are the bolts which were described earlier in 1.2. Streams are represented by arrows and are annotated with the source and destination cardinalities. These cardinalities correspond to the number of

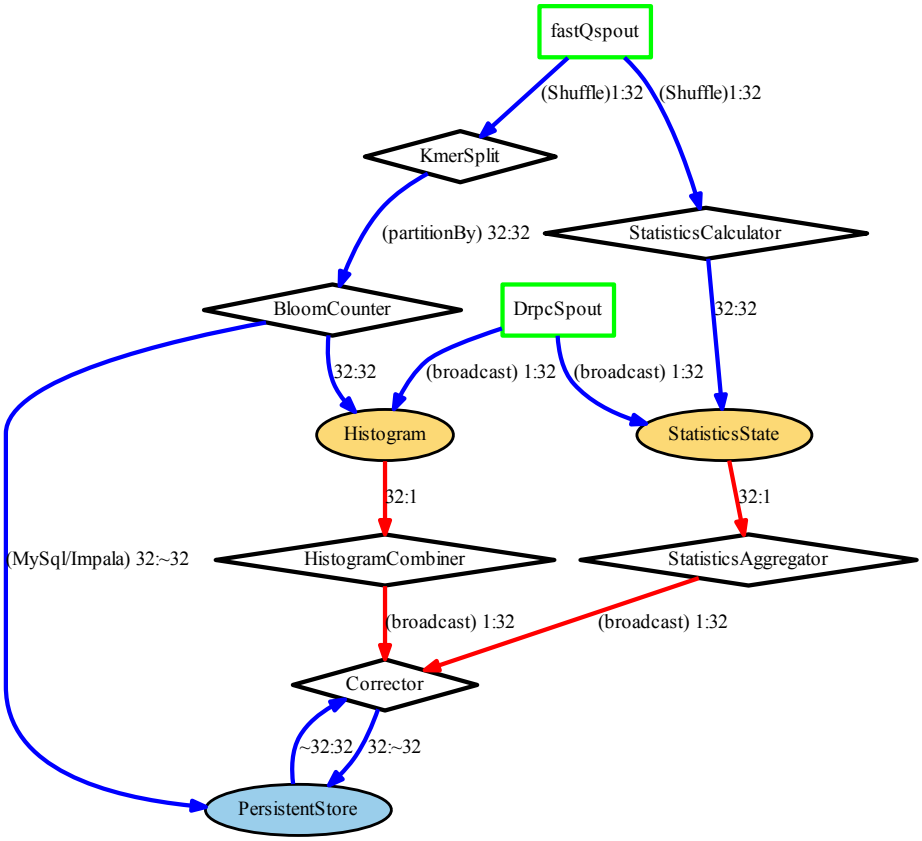

Figure 5.2: The storm pipeline used for error correction. The nodes in green are the spouts, the diamond shaped nodes are the bolts. Streams are represented by arrows and are annotated with the source and destination cardinalities.

executors performing a task in parallel. The process of detecting and correcting errors using this topology is described below and the complete source code has been shared freely at `https://github.com/lbhat1/compgenomics/tree/master/src/project`.

5.2.2 Storing and counting k-mers using Bloom-Filter

To count all non-unique k-mers we use a Bloom filter B and a simple hash table T to store k-mers. The Bloom filter keeps track of k-mers we have encountered so far and acts as a *staging area*, while the hash table stores all the k-mers seen at least twice so far. The idea is to use the memory-efficient Bloom filter to store implicitly all k-mers seen so far, while only inserting non-unique k-mers into the hash table. Initially both the Bloom filter and the hash table are empty. All k-mers are generated sequentially from the sequencing reads. For each k-mer, x, we check if x is in the Bloom filter B. If it is not in B then we update the appropriate bits in B to indicate that it has now been observed. If x is in B, then we check if it is in T and bump its count, and if not, we add it to T.

All these k-mer statistics (the sketch, some nucleotide counts per position in the read and the HashTable) are stored within an in-memory state and continuously updated as the stream arrives. The reads are then persisted in a back-end store (single node MySql, 3-node Impala). The stream rate was configured at 100,000 reads per second. We were mostly able to keep up with the stream with 32/64 executors spread across 4 storm nodes (The scale-up wasn't great when using 64 executors due to MySql connection sharing). Occasionally, the network glitches caused storm to replay the messages from the transactional spouts because of which we had to make our state

updater idempotent. Not everything was perfect with storm - storm is built to be fail fast - and worker nodes restart as soon as they have a problem. This increases the chances of losing the memory mapped state unless we persist them, perhaps to a fast column store. This can be done easily in storm $BaseStateUpdater < State >$'s commit() method. It should also be noted that configuring storm batches appropriately is extremely important. When we chose a small batch size, the cost of acknowledgments was much higher and thus did not allow us to throttle the spouts as well as we wanted.

This scheme guarantees that all k-mers with a coverage of 2 or more are inserted into T. However a small proportion of unique k-mers will be inserted into T due to false positive queries to B. After the first pass through the sequence data, one can re-iterate over T to obtain exact counts of the k-mers in T and then simply delete all unique k-mers or perhaps all the k-mers with $multiplicity < m$. The time spent on the second round is at most $O(G)$, and tends to be less since hash table lookups are generally faster than insertions.

In our implementation, we use a distributed Bloom Filter that only counts a unique *partition* of the so-called k-mer stream. This means that the Bloom Filter false positives are reduced by a factor of *paralleism* specified for the state updaters.

5.2.3 Localizing errors

We choose a cutoff (multiplicity = 5, in our experiments) to separate trusted and untrusted k-mers based on our intuition and the knowledge of the dataset. But in general this approach is flawed because the datasets need not have uniform coverage or our genome may contain many non-exact repeats. A highly sophisticated approach to choosing this cutoff is discussed in.[36,37] Once we choose the cutoff, all reads containing an untrusted k-mer become candidates for correction. In most cases the pattern of untrusted k-mers will localize the sequencing error to a small region. The figure 5.3 from[37] makes this notion very clear. The reader is advised to look it up to find more details.

5.2.4 Naive Bayes: Sequencing error probability model

The Naive Bayes Classifier[9] technique is based on the so-called Bayesian theorem and is particularly suited when the dimensionality of the inputs is high, often the case in Natural Language Processing. Despite its simplicity, Naive Bayes can often outperform more sophisticated classification methods. However, despite its simplicity it is very easy to fall into certain pitfalls due to the poor understanding of the central assumption behind Naive Bayes, namely conditional independence.[42] We take our chances and jump on the bandwagon and throw this model at the sequencing error

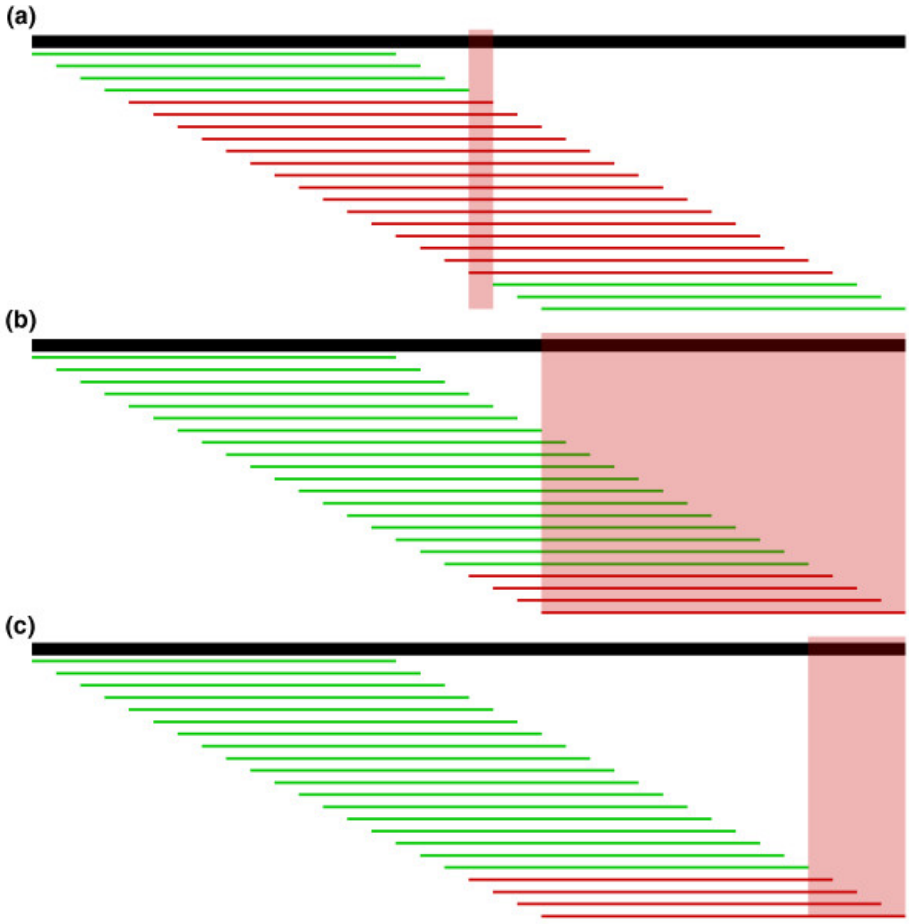

Figure 5.3: Trusted (green) and untrusted (red) 15-mers are drawn against a 36 bp read. In (a), the intersection of the untrusted k-mers localizes the sequencing error to the highlighted column. In (b), the untrusted k-mers reach the edge of the read, so we must consider the bases at the edge in addition to the intersection of the untrusted k-mers. However, in most cases, we can further localize the error by considering all bases covered by the right-most trusted k-mer to be correct and removing them from the error region as shown in (c).

correction problem. Of course, we make some highly inappropriate assumptions to ensure our correction model is slick and fast. First, we must define the likelihood of a set of corrections. Let the dataset contain n reads. Let O $= O_1, O_2, ..., O_n$ represent the observed nucleotides at a given position in reads $1, 2, ..., n$ respectively, and A $= A_1, A_2, ..., A_n$ the actual nucleotides at the same position in reads $1, 2, ..., n$ of the

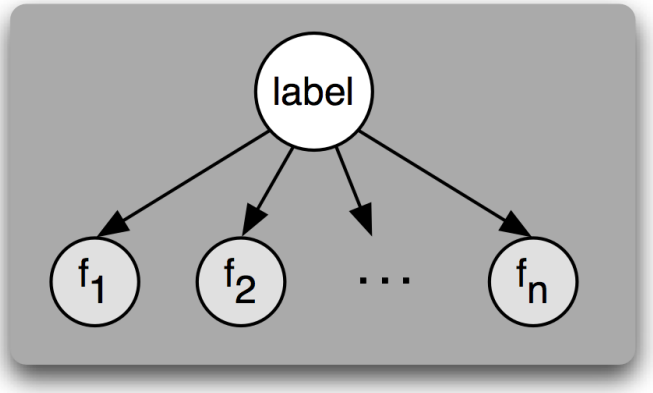

Figure 5.4: Naive bayes classifier - features are assumed to be conditionally independent given class label

DNA. Then

$$P(A = a_k|O_1, \ldots, O_n) = \frac{P(A = a_k)\ p(O_1, \ldots, O_n|A)}{p(O_1, \ldots, O_n)}.$$

In plain English the above equation can be written as

$$\text{posterior} = \frac{\text{prior} \times \text{likelihood}}{\text{evidence}}.$$

Now we make the assumption that position p_k in read r_i is conditionally independent of position p_j in read r_j given A. So, now

$$P(A = a_k|O_1, \ldots, O_n) = \frac{1}{Z} P(A = a_k) \prod_{i=1}^{n} p(O_i|A = a_k)$$

where Z (the evidence) is a scaling factor dependent only on $O_1, \ldots, O_n$, that is, a constant if the values of the observed variables O_i are known. Because we compare likelihoods, $P(O_i = o_i)$ is the same for all assignments to A and is ignored. $P(A_i = a_k)$ is defined by the GC% of the genome, which we estimate by counting Gs and Cs in the sequencing reads or by referring to the reference genome. In case of E.Coli, the GC% is 51% which we hard-code for our testing. The $P(O_i|A = a_k)$ is estimated from the *trusted* portions of the read. In machine learning terminology, the trusted k-mers (we can probably relax this to contain all k-mers to get a better generalization) act as our *labeled training set*. That is, to estimate $P(O_i|A_i)$ for all values A_i can take namely A, C, T, G, we set the value $P(O_i = o_i|A_i = o_i) = p_i$ in the conditional probability table. We distribute the rest of the probability i.e. $(1 - p_i)$ among the other $A_i \neq o_i$ according to their prior probabilities (estimated from their GC content).

5.3 Results

In this chapter, we pose a question - is sequencing error correction a streaming application? We argue (and empirically prove) that Storm, a continuous event processing system, is more than capable of handling these kind of loads and may pave a way to solve several more problems in computational genomics in real time. With Storm's capability to parallelize different parts to the topology to different degrees, we can fine tune our computation to get *optimal* results. We also present a method

that identifies all the k-mers that occur more than once in a DNA sequence data set. Our method does this using a Bloom filter, a probabilistic data structure that stores all the observed k-mers implicitly in memory with greatly reduced memory requirements. For our test data sets (E. coli K12 substrain MG1655 [SRA:SRX000429]), we see up to **50%** savings in memory usage compared to naive approach, with modest costs in computational speed. This approach may reduce memory requirements for any algorithm that starts by counting k-mers in sequence data with errors. We also propose to use a straight forward but highly competitive probabilistic model in Naive Bayes to model sequencing errors. We were able to correct **98%** of sequencing errors in the EColi K12 dataset. We leave the testing of our model and the topology on human genome dataset for future work.

5.4 Conclusions

Counting k-mers from sequencing data is an essential component of many recent methods for genome assembly from short read sequence data. They are also found to be useful in alignment and variant detection. However, in current data sets, it is frequently the case that more than half of the reads contain errors and are observed just once. Since these error-containing k-mers are so large in number, they can overwhelm the memory capacity of available high-performance machines, and they increase the computational complexity of downstream analysis. In this thesis, we

describe a straightforward application of the Bloom filter to help identify and store the reads that are present more than more than m times in a data set, and hence likely to be trusted. Although one might claim that such methods trades off reduced memory usage for an increase in processing time, our use of fast, simple but sufficiently strong hash function aka. murmur alleviates any such concerns.

Chapter 6

Storm Performance Evaluation

Evaluating Storm's performance is quite tricky. It is highlighted on Storm's website[1] that Storm is extremely fast; "a benchmark clocked it at over a million tuples processed per second per node". When benchmarking a distributed system like Storm it is important to take the following into account:

1. The kind of processing being done: of course, heavy computation like the analytics workload we've seen in earlier chapters will be slower than some light computation, like computing *max* value seen in the stream thus far.

2. The hardware of the machines and the physical layout (for a pipeline) chosen by the system.

We now describe the measured performance of Storm for some specific use-cases detailed below.

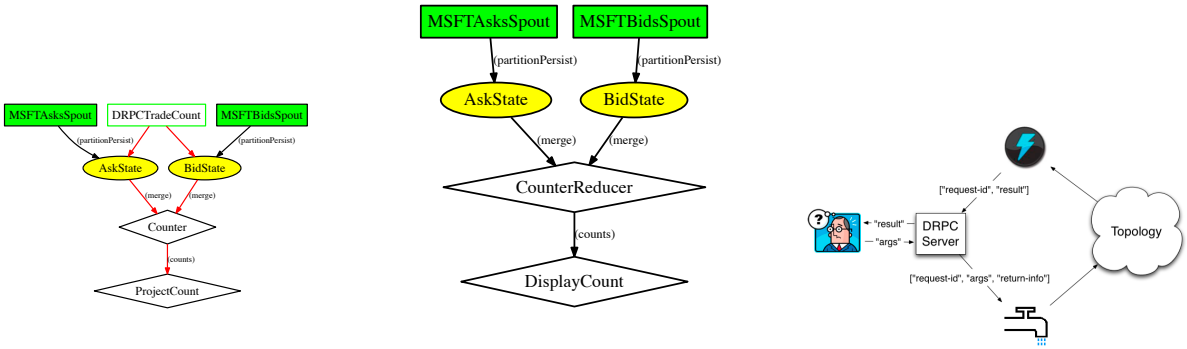

Figure 6.1: [a] FinancialTransaction topology with DRPC queries (left). [b] Financial-Transaction topology with continuous aggregation (center). [c] typical DRPC workflow (right)[3]

DRPC queries and continuous aggregations

Distributed Remote Procedure Calls (DRPC) come packaged with Storm. Fig 6.1 [c] shows an illustration of a DRPC workflow. As an overview, a DRPC client sends the DRPC server the name of the *function* to execute and the arguments to that function. The topology implementing that function uses a *DRPCSpout* to receive a function invocation stream from the DRPC server. Each function invocation is tagged with a unique identifier by the DRPC server. The topology then computes the result and at the end of the topology a bolt called *ReturnResults* connects to the DRPC server and gives it the result for the function invocation identifier. The DRPC server then uses the identifier to match up that result with which client is waiting, unblocks the waiting client, and sends it the result.

It's quite evident that the DRPC querying capabilities are extremely useful to make dynamic requests and parameter updates to the state as we saw in 2.2.1.1.2.

Intuitively, the idea behind DRPC is to parallelize the computation of really intense query functions on the fly such that the Storm topology takes in as input a stream of function arguments, and emits an output stream of the results for each of those function calls. However, the cost of encoding the input arguments and results into JSON strings -which is quite annoying given that such encodings of large objects add significantly to the overall query latency - and the constraint that limits the results of the query to less than 64 kilobytes, makes DRPC one of the most frustrating features of Storm from a programmer's perspective. This is backed up by the results of an experiment in which we use a simple *FinanceTopology* to replay 24 hours worth of trades of MSFT shares and then compute aggregate trade during the day; results show that continuous aggregation 6.1[b] is about 16 times faster than the equivalent DRPC based implementation 6.1[a]. Note that there are mainly two other factors coupled with JSON encoding that contribute to this slowness

1. DRPC query implementations create a snapshot of the state per query, leading to the stalling of state updates and

2. micro-stragglers such as garbage collection initiated frequently.

The source code for these tests are freely available at `https://github.com/lbhat1/BDConsistency`.

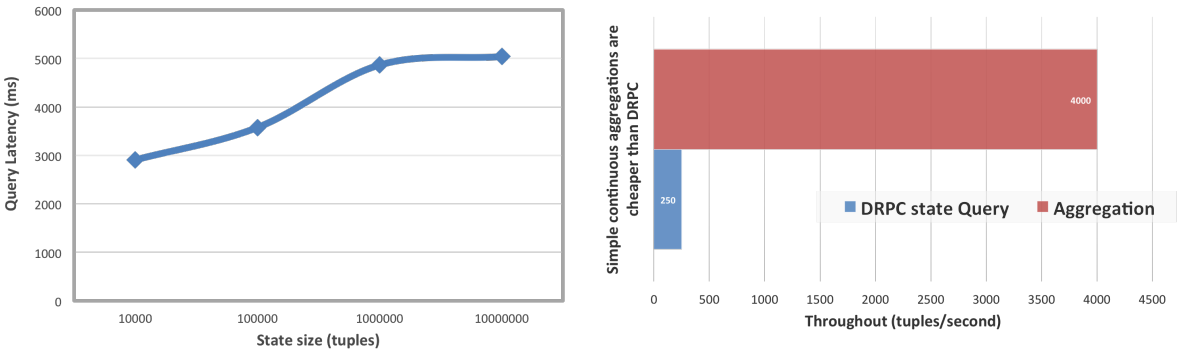

Figure 6.2: Query latency vs. State size (left) and Continuous aggregations vs. DRPC state queries (right)

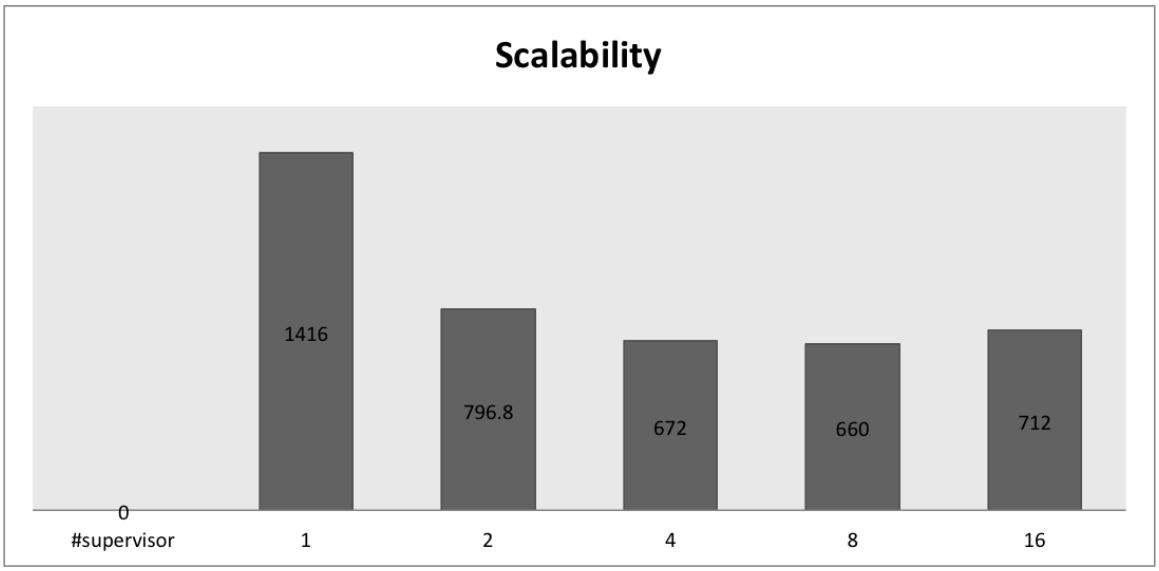

Figure 6.3: Storm scalability

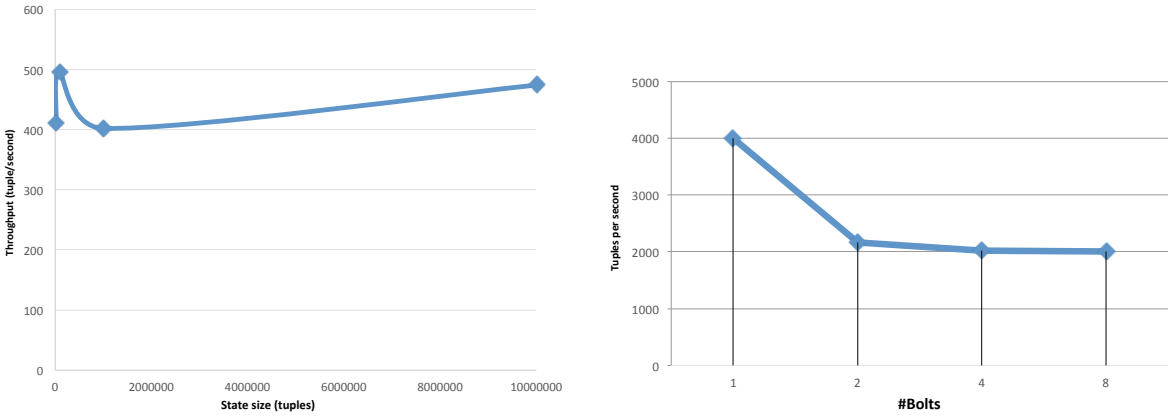

Figure 6.4: AxFinder: state vs throughput (left). chained filters: throughput vs. num bolts (right)

Throughput and Scalability

In order to analyze the scalability of Storm we performed the following experiment : 15 GB of BPTI protein conformations encoded as JSON objects were loaded into the topology. The processing done on this data was similar to what we described in 2.2.1 without the DRPC updates. Therefore the topology was mainly parsing the data encoded in JSON and learning the structure of the data using SimpleKMeans. The topology has been run five times with the same data and the same configuration; once each with 1,2,4,8 and 16 supervisors. The goal of this experiment was to analyze the performance based on the number of worker nodes. It is clear from the figure 6.3 that Storm is picking suboptimal layouts when we add more supervisors to the cluster. We could not find any parameter that we could use to physically anoint bolts per node.

Figure 6.4 shows how Storm scales when we chain a number of filters in our

topology. The aim was to analyze how the cost of book-keeping and routing tuples through bolts affect throughput. We know from experience that Storm uses only a subset of supervisors to run a topology and group as many bolts as possible on a local node. The advantage of this scheme is reflected by the throughput numbers seen in the in the graph 6.4[b]. Figure [a] shows how throughput varied with increase in state size for the AxFinder financial query implemented in Storm. As per the graph the throughput increases with the size of the state, but at some point we should hit the memory limit per node and thus we expect throughput to fall sharply due to stragglers such as garbage collection and stream replay.

Chapter 7

Storm Experience

In this section we present an experience report on the use of Storm for online machine learning based applications, describing the implementation and deployment challenges that we overcame in designing and evaluating our framework [2].

Storm is an open-source system that was initially developed by BackType before its acquisition by Twitter, Inc. The documentation and support for Storm primarily arises from the open-source user community that continues to develop its capabilities through the github project repository. Storm has been incubated as an Apache project as of September 2013. In this section we provide an initial report on our experiences in developing with the Storm framework, particularly as we deploy it onto a cluster environment and develop online learning, consensus clustering and event detection algorithms.

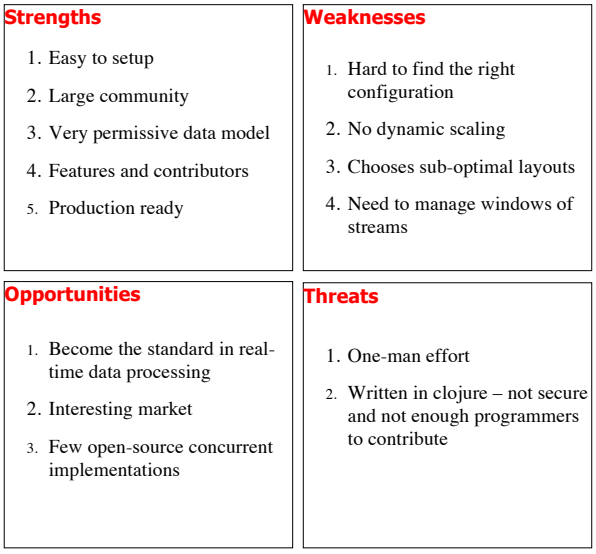

Figure 7.1: SWOT analysis of Storm streaming system

7.1 Debugging a Storm topology

Here, we briefly describe our understanding of the internals of the Storm system that was useful in scaling the Storm cluster and debugging production issues. In some cases, we also make parameter recommendations for Trident topologies.

7.1.1 Supervisor: The node governor

A supervisor is a JVM process launched on each storm worker machine. This does not execute your code but just supervises it. The number of workers is set by number of supervisor.slots.ports on each machine. One can limit the no. of JVMs spun off

per machine by just adding only one entry to this list of ports.

7.1.2 Worker: The power-horse

- A worker is the power-horse of the Storm ecosystem. It is a JVM process launched by the supervisor. Our experience shows that intra-worker transport is more efficient than inter-worker communication and so we decided to run one worker per topology per machine.

- If worker dies, supervisor will restart it. However, workers die for any minor failure (eg. network glitches). Storm is a fail-fast system which means the processes will halt whenever an unexpected error is encountered. Storm is designed so that it can safely halt at any point and recover correctly when the process is restarted. Storm obediently restarts the worker processes immediately after a failure. This means that if you are going to store critical, uncommitted data in-memory, then you have to cover all the failure scenarios including temporary network glitches. That makes storm in-memory state highly volatile and brittle.

7.1.3 Coordinator: The mastermind

The coordinator is responsible for handling batching, aid guaranteed processing and strict ordering of committed data. Based on our perusal of the Storm code-base,

- Coordinator generates new transaction identifier, sends tuple, which influences

spout to dispatch a new batch.

- Each transaction identifier corresponds identically to single trident batch and vice-versa
- Transaction identifiers for a given "topo_launch" are serially incremented globally.
- Coordinator knows about Zoo-keeper (transactional); so it recovers the transaction in case of failures.

7.1.4 Executor: The core storm agent

An executor is responsible for executing one bolt or spout. This means, if there are 3 *BPTIProtienConformationSpout* spouts on a worker, there are three executors that are continuously spouting. We found it quite useful to monitor the executor threads under a profiler (*YourKit*) by running Storm in a single cluster (standalone) mode. We were able to identify several queue buffering issues that were related to throttling (7.1.6) via this process.

7.1.5 Ack'ing: Guaranteed message processing

An Acker is just a regular bolt and all the interesting action takes place in its execute method. The following details might help one identify an issue in the ack'ing framework.

- The acker holds a single O(1) lookup table which is actually a collection of lookup tables: *current, old and dead*. New tuple trees are added to the current bucket; after every *timeout* seconds, current becomes old, and old becomes dead- they are declared failed and their records retried.

- An acker always knows a given tuples stream-id (*id == tuple[0]*)

- There is a time-expiring data structure, the RotatingHashMap; when you go to update or add to it, it performs the operation on the right component of HashMap. Periodically (when it receives a tick tuple in Storm8.2+), it pulls off the oldest component of the HashMap and mark it as dead; invoke the expire callback for each element in that HashMap.

Based on our experience we make the following recommendation for ack'ing parameters.

- We set number of ackers equal to number of workers. (default is 1 per topology)

- It is gainful, in terms of debugging, to consider tuning the parameters

  1. **topology.trident.batch.emit.interval.millis** and

  2. **topology.trident.topology.max.spout.pending**

  in your *Storm* configuration as they are directly related to ack'ing performance.

7.1.6 Throttling a Storm topology

- Max spout pending (TOPOLOGY_MAX_SPOUT_PENDING) configuration parameter sets the number of tuple trees live in the system at any point in time.

- Trident batch emit interval (**topology.trident.batch.emit.interval.millis**) sets the maximum pace at which the trident master batch coordinator issues new seed tuples. For example, if batch delay is 500ms and the most recent batch was released 486ms, the spout coordinator will wait 14ms before dispensing a new seed tuple. If the next pending entry isn't cleared for 523ms, it will be dispensed immediately.

- Trident batch emit interval is extremely useful to prevent congestion, especially around startup/rebalance. As opposed to a traditional Storm spout, a Trident spout will likely dispatch hundreds of records with each batch. For instance, if max-spout-pending is 20, and the spout releases 500 records per batch, the spout will try to cram $10,000$ records into its send queue.

- As a rule of thumb, set the **topology.trident.batch.emit.interval.millis** to be comfortably larger than the *end-to-end* latency – there should be a short additional delay after each batch completes.

7.1.7 Tuning Batch Size

A batch in Storm is an epoch in the life-cycle of a streaming application. A pure streaming system can be seen as a batch processing system with batch size 1. The batch size has direct consequence on the throughput of a streaming application and hence it is prudent to consider the following before a decision is made.

- Trident *each* functions do not care about batch size; *partitionAggregate, persistentAggregate, partitionPersist, partitionQuery* etc. do.

- Set the batch size to optimize the throughput of the most expensive batch operation - a bulk database operation, network request, or large aggregation. When the batch size is too small, bookkeeping dominates response time i.e response time is constant.

- Execution times increase slowly and we get better and better records-per-second throughput with increase in batch size. However, at some point, we start overwhelming some resource and execution time increases sharply (usually due to network failures and replays, in our case)

- Trident promptly replays an entire batch in case of failures. In case one is working with a non-transactional system, for example, updating an in-memory machine learning model in a State's commit method, one may have to re-train the model on the same input all over again. This is a classic problem in log-based rollback recovery fault tolerance models where one has to reconstruct

state by replaying the logged updates. Hence, keeping the batch size small and check-pointing during the synchronous check-pointing phase would be beneficial if the state is considerably smaller (usually the case in streaming applications). This is an example of co-ordinated check-pointing and message-logging protocol. However, if the state is large, one may have to revert to pure message-logging based protocols since check-pointing may be prohibitively expensive.

7.2 Blocking Spout: A common mistake

Transactional and Opaque transactional spouts are discussed in detail in Trident tutorial.[20] A spout must never block when emitting.If it blocks, critical book-keeping tuples will get trapped, and the topology will hang. Hence, it's emitter keeps an *overflow buffer*, and publishes as follows:

- if there are tuples in the overflow buffer add the tuple to itthe queue is certainly full.
- otherwise, publish the tuple to the flow with a non-blocking call. That call will either succeed immediately or fail with an *InsufficientCapacityException*, in which case, add the tuple to the overflow buffer.
- The spout's async-loop won't call nextTuple method if overflow is present, so the overflow buffer only has to accommodate the maximum number of tuples emitted in a single nextTuple call.

7.3 Trident: Functional style programming

Trident is a high-level abstraction for doing real-time computing on top of Storm. It has great features[3] and makes stream pipelining very easy. However, we focus only on it's quirky features here.

- If you don't need stateful processing, then using Trident would be a waste of resources (CPU, RAM etc.)

- Storm is stateless stream processing framework and Trident provides stateful stream processing. However, Trident adds complexity to a Storm topology, lowers performance (when using smaller batches) and generates state. So, one needs to ask the following question before jumping to use Trident – Do we need the *exactly once* processing semantics of Trident or can we live with the *at least once* processing semantics of Storm? If exactly once semantics is a necessity, use Trident, otherwise avoid it.

- Trident doesn't have support for cyclic bolts. In case you inadvertently create a cyclic topology you are greeted with bizarre, inexplicable errors emanating from low level *thrift* code.

7.4 Load Balancing

During our experiments we saw that Storm consistently chose sub-optimal node assignments (physical layout). Since we don't have any control (through configuration parameters) to manually assign spouts and bolts to supervisor nodes, we were at the mercy of Storm to choose the *optimal* configuration. We recognized that Storm was choosing only a minimal subset of all the supervisors in the system to process a topology, as though it was anticipating the submission of more topologies. Since we were running only a couple of topologies simultaneously, a lot of cluster bandwidth was under utilized. Moreover, this also meant we could not set the no. of executors per node based on the number of available cores on that node. This resulted in micro-stragglers (Java garbage collection and network bottlenecks).

7.5 Cyclic topologies

Storm supports cyclic topologies. However, without clear documentation and examples, it is extremely difficult to get a cyclic topology to run without bugs. Note that *Trident* doesn't support cyclic topologies because Storm has no any way to associate a batch of input stream with a merged stream. Precisely, Trident has no means to distinguish a batch of input tuples from the batch emitted by intermediate bolts within a cyclic context which *merge* with the original input stream. To support cyclic topologies, Trident would have to implement a structured "pointstamp"[43] instead of

a *batch identifier*. More details on this is available in detail in Naiad paper.[43]

Bibliography

[1] N. Marz, "Website of the storm project." [Online]. Available: http://www.storm-project.net

[2] N. Michael, "Understanding the parallelism of a storm topology." May 2012. [Online]. Available: http://www.michael-noll.com/blog/2012/10/16/understanding-the-parallelism-of-a-storm-topology/

[3] N. Marz, "Trident tutorial." [Online]. Available: http://storm.incubator.apache.org/documentation/Trident-tutorial.html

[4] H. Chen, R. H. L. Chiang, and V. C. Storey, "Business intelligence and analytics: From big data to big impact," *MIS Q.*, vol. 36, no. 4, pp. 1165–1188, Dec. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2481674.2481683

[5] N. Marz, "Storm, distributed and fault-tolerant realtime computation." 2012. [Online]. Available: http://storm.incubator.apache.org/documentation/Home.html

BIBLIOGRAPHY

[6] B. Lakshmisha, "Machine learning in storm," May 2014. [Online]. Available: https://github.com/lbhat1/mlstorm

[7] "Fault tolerance." [Online]. Available: http://en.wikipedia.org/wiki/Fault-tolerant_system

[8] N. Marz, "Storm spouts tutorial." [Online]. Available: http://storm.incubator.apache.org/documentation/Trident-API-Overview.html

[9] N. Friedman, D. Geiger, M. Goldszmidt, G. Provan, P. Langley, and P. Smyth, "Bayesian network classifiers," in *Machine Learning*, 1997, pp. 131–163.

[10] G. V. Cormack, "Email spam filtering: A systematic review," *Found. Trends Inf. Retr.*, vol. 1, no. 4, pp. 335–455, Apr. 2008. [Online]. Available: http://dx.doi.org/10.1561/1500000006

[11] Z. Ghahramani, "Unsupervised learning," in *Advanced Lectures on Machine Learning*. Springer-Verlag, 2004, pp. 72–112.

[12] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*, no, Ed. Belmont, CA: Wadsworth International Group, 1984.

[13] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *J. Mach. Learn. Res.*, vol. 3, pp. 1157–1182, Mar. 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=944919.944968

BIBLIOGRAPHY

[14] N. Littlestone, "Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm," in *Machine Learning*, 1988, pp. 285–318.

[15] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1656274.1656278

[16] N. Marz, "Trident state tutorial." [Online]. Available: http://storm.incubator.apache.org/documentation/Trident-state.html

[17] I. Jolliffe, *Principal Component Analysis*. Springer Verlag, 1986.

[18] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '07. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035. [Online]. Available: http://dl.acm.org/citation.cfm?id=1283383.1283494

[19] S. Owen, R. Anil, T. Dunning, and E. Friedman, *Mahout in Action*, 1st ed. Manning Publications Co. 20 Baldwin Road PO Box 261 Shelter Island, NY 11964: Manning Publications Co., 2011. [Online]. Available: http://manning.com/owen/

BIBLIOGRAPHY

[20] N. Marz, "Storm spouts tutorial." [Online]. Available: http://storm.incubator.apache.org/documentation/Trident-spouts.html

[21] A. Peter, "Efficient java matrix library." [Online]. Available: https://code.google.com/p/efficient-java-matrix-library

[22] S. Monti, P. Tamayo, J. Mesirov, and T. Golub, "Consensus clustering – a resampling-based method for class discovery and visualization of gene expression microarray data," in *MACHINE LEARNING 52 (2003) 91118 FUNCTIONAL GENOMICS SPECIAL ISSUE*, 2003.

[23] V. Filkov and S. Skiena, "Integrating microarray data by consensus clustering." *International Journal on Artificial Intelligence Tools*, vol. 13, no. 4, pp. 863–880, 2004. [Online]. Available: http://dblp.uni-trier.de/db/journals/ijait/ijait13.html#FilkovS04

[24] A. Topchy, A. K. Jain, and W. Punch, "Clustering ensembles: Models of consensus and weak partitions," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 12, pp. 1866–1881, 2005.

[25] A. Strehl and J. Ghosh, "Cluster ensembles — a knowledge reuse framework for combining multiple partitions," *J. Mach. Learn. Res.*, vol. 3, pp. 583–617, Mar. 2003. [Online]. Available: http://dx.doi.org/10.1162/153244303321897735

[26] A. L. N. Fred and A. K. Jain, "Combining multiple clusterings using evidence

accumulation," *IEEE Transaction on Pattern Analysis and Machine Intelligence*, vol. 27, pp. 835–850, 2005.

[27] B. Fischer and J. M. Buhmann, "Bagging for path-based clustering," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 25, no. 11, pp. 1411–1415, Nov. 2003. [Online]. Available: http://dx.doi.org/10.1109/TPAMI.2003.1240115

[28] H. G. Ayad and M. S. Kamel, "Cumulative voting consensus method for partitions with variable number of clusters," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 30, no. 1, pp. 160–173, Jan. 2008. [Online]. Available: http://dx.doi.org/10.1109/TPAMI.2007.1138

[29] J. Wu, H. Liu, H. Xiong, and J. Cao, "A theoretic framework of k-means-based consensus clustering," in *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, ser. IJCAI'13. AAAI Press, 2013, pp. 1799–1805. [Online]. Available: http://dl.acm.org/citation.cfm?id=2540128.2540386

[30] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, pp. 83–97, 1955.

[31] A. Balakrishnan, T. L. Magnanti, and P. Mirchandani, "Network design," in *Annotated bibliographies in combinatorial optimization*, M. Dell'Amico, F. Maffioli, and S. Martello, Eds. Chichester: John Wiley, 1997, pp. 311–334.

BIBLIOGRAPHY

[32] G. E. Hinton and S. Osindero, "A fast learning algorithm for deep belief nets," *Neural Computation*, vol. 18, p. 2006, 2006.

[33] Y. Zhang and T. Li, "Consensus clustering + meta clustering = multiple consensus clustering." in *FLAIRS Conference*, R. C. Murray and P. M. McCarthy, Eds. AAAI Press, 2011. [Online]. Available: http://dblp.uni-trier.de/db/conf/flairs/flairs2011.html#ZhangL11

[34] R. E. Schapire, "A brief introduction to boosting," in *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI'99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 1401–1406. [Online]. Available: http://dl.acm.org/citation.cfm?id=1624312.1624417

[35] W.-H. Lin and A. G. Hauptmann, "Meta-classification: Combining multimodal classifiers." in *Revised Papers from MDM/KDD and PAKDD/KDMCD*, ser. Lecture Notes in Computer Science, O. R. Zaane, S. J. Simoff, and C. Djeraba, Eds., vol. 2797. Springer, 2002, pp. 217–231. [Online]. Available: http://dblp.uni-trier.de/db/conf/kdd/mdm-kdmcd2002.html#LinH02a

[36] R. Chikhi and P. Medvedev, "Informed and automated k-mer size selection for genome assembly," *BioInformatics*, pp. 1–7, 2013.

[37] S. L. S. David R Kelley, Michael C Schatz, "Quake: quality-aware detection and correction of sequencing errors," *Genome Biology*, vol. 11, no. 11, p. 13, 2010.

BIBLIOGRAPHY

[38] J. S. Yongchao Liu and B. Schmidt, "Musket: a multistage k-mer spectrum based error corrector for illumina sequence data," *BioInformatics*, 2012.

[39] P. Medvedev, E. Scott, B. Kakaradov, and P. Pevzner, "Error correction of high-throughput sequencing datasets with non-uniform coverage," *Bioinformatics*, vol. 27, no. 13, pp. i137–i141, 2011.

[40] W. M. Idury RM, "A new algorithm for dna sequence assembly." *PubMed Central*, vol. 2, no. 2, pp. 291–306, 1995.

[41] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[42] A. P. Dawid, "Conditional Independence in Statistical Theory," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 41, no. 1, pp. 1–31, 1979. [Online]. Available: http://dx.doi.org/10.2307/2984718

[43] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 439–455. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522738

Vita

N. Lakshmisha Bhat received the Engg. B. degree in Computer Science and Engineering from The National Institute of Engineering, India in 2007, and enrolled in the Computer Science Master's program at Johns Hopkins University in 2012. His interests are in high-volume data processing, streaming analytics and scientific databases. His current research focuses on machine learning in streaming systems like Apache Storm and Yahoo! S4.

Starting in June 2014, Lakshmisha will work on the 'Content Analytics for Salesforce Communities' project at Salesforce.com in San Fransisco, CA.