Media Engineering and Technology Faculty
German University in Cairo

Improving Queries over Streaming Data

Bachelor Thesis

Author:	Ahmed Amr
Supervisors:	Dr. Mervat Abouelkheir
Submission Date:	26 May, 2019

Media Engineering and Technology Faculty
German University in Cairo

Improving Queries over Streaming Data

Bachelor Thesis

Author:	Ahmed Amr
Supervisors:	Dr. Mervat Abouelkheir
Submission Date:	26 May, 2019

This is to certify that:

(i) the thesis comprises only my original work toward the Bachelor Degree

(ii) due acknowlegement has been made in the text to all other material used

Ahmed Amr
26 May, 2019

Acknowledgments

First and foremost, I would like to express my deepest and most sincere gratitude to my family for everything they have done for me and all the love they gave to me. My mother, father. No words can express my love for them.

I would like also to thank my supervisor Dr. Mervat for her support and guidance through my bachelor project.

Abstract

Continuous top-k query over streaming data is a fundamental problem in database. In this paper, we focus on the sliding window scenario, where a continuous top- k query returns the top- k objects within each query window on the data stream. Existing algorithms support this type of queries via incrementally maintaining a subset of objects in the window and try to retrieve the answer from this subset as much as possible whenever the window slides. However, since all the existing algorithms are sensitive to query parameters and data distribution, they all suffer from expensive incremental maintenance cost. In this paper, we propose a using advance data-structures such as *AVL Tree* and *priority queue* to support continuous top- k query and evaluate algorithm on batch of data and on real datasets with *Apache Storm* platform. To our best knowledge, this algorithm that achieves logarithmic complexity with respect to k for incrementally maintaining the candidate set even in the worst case scenarios.

Contents

Chapter 1

Introduction

Data streaming is an emerging research field since 2000s. It has become one of main interest of computer scientist as many different applications like sensor networks, network traffic analysis , financial ticker , Web or telecommunications transaction log analysis. However , Database management System cannot handle data online . Data Stream Management Systems have been developed to handle transient data streams online to process continuous queries on these data streams . The main difference between DSMSs and DBMSs [10] is storing and processing queries for a limited time period in main memory without storing the data on disk that what it makes DSMS mush faster than DBMS on processing queries on continuous data streaming.

1.1 Motivation

DSMS has various applications are important nowadays . For example , in stock market, a continuous top-k query can be used to monitor real-time transactions and retrieve the K most significant transactions within the last T minutes. These query results could help for making decidable decisions which is important to improve time complexity and consistency for retrieving results . Moreover , in social media we can keep track with trending hashtags (eg. Twitter) which help users to join in related conversations among users locally or globally or keep updated to latest news. It can be also used to reach out to move users or gain more Twitter followers.

1.2 Definition of Data Stream Management System (DSMS)

A data stream management system (DSMS) [10] is a computer software system to manage continuous data streams. It is similar to a database management system (DBMS), which is, however, designed for static data in conventional databases. A DSMS also offers a flexible query processing so that the information need can be expressed using queries. However, in contrast to a DBMS, a DSMS executes a continuous query that is not only performed once, but is permanently installed. Therefore, the query is continuously executed until it is explicitly uninstalled. Since most DSMS are data-driven, a continuous query produces new results as long as new data arrive at the system.

Moreover , Data stream is defined as an infinite sequence of tuples.

S = <X1 , T1 > , <X2 , T2> , . . . <Xn , Tn> . [16]

where Xi is tuple and Ti is a timestamp. A tuple Xi is an object that entered queue in continuous time streaming. A timestamp . A timestamp Ti is a non-negative integer value and Ti is a non-negative integer value and T indicated with the current time . More formally , for any i < j , a tuple Si <Xi , Ti> arrives earlier than Sj <Xj , Tj> .

1.3 Query Processing

Real Time query processing on data stream is challenging for several reasons. First , the incoming data can be irregular which may overload the system when the rate of arrival becomes too high. Second , the execution time of queries does not only depend on data volume , but also on content of data streams. Third , the total storage space required by all queries may exceed the physical memory. So, It is required to optimize query processing by reducing execution time and decrease load on physical memory.

1.4 Sliding Window Technique

Window techniques[6] only look on portion of the data. This approach is motivated by the idea that only most recent data are relevant. Therefore , a window continuously cuts output of stream. For example, the last K data elements and only considers these element during the processing. Sliding Windows are similar to FIFO lists that cut out disjoint parts.

1.5 Process of Sliding Window

Windowed query processing [6]over data stream have several key factors that need to be addressed:

- Time progress: The propagation of input events can impact the windowed behavior of output.

- Delay in pipeline: Difference in scheduling can make progress at different rate thereby complexity of windowing throw time.
- Actions to take when tuples arrive : when current window receive a new tuple . it may need to emit a tuple of any type to take an action.
- Actions to take when tuples expire.

1.6 Aim of the project

The aim of the project is to design and implement Data Streaming Topology that can classify top N words from continuous streaming data on *Twitter API.* and apply *ranking* techniques.

Moreover, we aimed to apply an optimization techniques using an advanced data-structures such as *AVL-Tree* and *Priority queue* to optimize ranking techniques with an efficient way.and also one of important challenges is to optimize ranking techniques to reach asymptotically logarithmic time consumption . Moreover, the challenges of data stream processing[7] is getting approximate query answering. when we are limited to a bounded amount of memory it is not always possible to produce exact answers for data stream queries; however, high-quality approximate answers are often acceptable of exact answers. and also, we should mention challenges for sliding window which is a technique for producing an approximate answer to a data stream query is to evaluate the query not over the entire past history of the data streams, but rather only over sliding windows of recent data from the streams.

Chapter 2

Literature Review

2.1 Introduction

One of the biggest challenges for DSMS is to handle infinite data streams using fixed amount of memory. There are different approaches to limit the amount of data in one pass. one of famous techniques is sliding window that try to portion the data into finite parts.

2.2 Stream Query Processing

The continuous queries considered in this paper[8] consist of a set of operators with queues over multiple sources. The operators like select, project, and aggregation operators, such as max and min can be easily executed without blocking. Other operators, like join, average, and sum must be computed within a specified sliding time window, in order to work this way.Query processing in a DSMS can be handled by two basic approaches: a multiple threads approach and a single thread approach. In a multiple threads approach, each operator runs as a thread, and the operator system determines which thread to run in any time slot. However, it is hard to control the resource management/allocation for this approach, all the query plans are registered to a single thread. Various strategies could be used to determine how to allocate system resources, and to schedule which operator or plan to run. The system behavior in a single thread approach is more controllable, and the context switch cost is minimized. In a multiple processor architecture, the whole set of query plans is partitioned into a set of subsets; each subset of query plans runs in a particular processor as a single thread. Some sharing and interdependent relationships may exist between any two subsets.

2.3 Stream Query Optimization

Stream query optimization[11] is the process of modifying a stream processing query, often by changing its graph topology and/or operators, with the aim of achieving better performance (such as higher throughput, lower latency, or reduced resource usage), while preserving the semantics of the original query.

A stream query optimization modifies a stream query to make it faster. Users want stream queries to be fast for several reasons. They want to grasp opportunities or avert risks observable on the input streams before it is too late. They want any views derived from the input streams to be up-to-date. And they want their system to keep up with the rate of input streams without falling behind, which would require to load or saving data to disk for later processing.

Knowing about stream query optimizations helps developers. Application developers who know about stream query optimizations can get the most out of the optimizations built into their streaming platform and can supplement them by hand-optimizing their application where necessary. Streaming platform developers can use knowledge about stream query optimizations to make their platform faster by implementing additional optimizations or by generalizing their existing optimizations to apply in more situations. Finally,researchers who invent new optimizations need to know the state-of-the-art optimizations to channel their efforts into the most innovative and impact direction.

An optimization should be both safe and profitable. An optimization is safe if it can be applied to a stream query without changing what it computes, as determined by the user's requirements. An optimization is profitable if it makes the stream query faster, as measured by metrics that matter to the user, such as throughput, latency, or resource efficiency.

Stream query optimizations are best understood with respect to stream graphs. A stream graph is a directed graph whose edges are streams and whose nodes are operators. Root and leaf nodes are called sources and sinks, respectively. This entry uses terminology that makes only few assumptions so as not to unnecessarily restrict its scope. For instance, this entry does not assume restrictions on the shape of the stream graph: unless specified otherwise, it does not assume that stream graphs are acyclic, or are single- source-single-sink, or are trees. A stream is an ordered sequence of data items, which are values that can range from simple numbers to flat tuples to more elaborate structured data that may be deeply nested and have variable size. Streams are conceptually infinite, in the sense that as the streaming computation unfolds over time, the sequence of data items is unbounded in length. Operators are primarily stream transformers but can also have state and side effects beyond the output streams they produce. Indeed, sources and sinks are operators that typically have the side effect of continuously consuming input from and producing output to the external world outside of the stream graph.

2.4 Continuous Top-K Queries over Streaming Data

A large number of continuous queries over streaming data have been studied [9] [13] [12]. we focus on the problem of continuous top- k query over streaming data. Based on whether re-scanning is necessary, existing algorithms can be clustered into two groups, namely multi-pass based approaches and one-pass based approaches.

Multi-pass based approach maintain top-k objects in current window as candidates,Whenever a query result expires from the window, if size of candidates greater than k , otherwise, a re-scanning of current window W is triggered to re-fill the candidate set. According to presence of high cost of re-scanning , there are several algorithms have been proposed to reduce re-scanning cost and one of them is Skyband Monitoring algorithms (eg ,SMA) . SMA reduces the re-scanning cost via two facets. First, it uses a grid structure to index streaming data. When re-scanning of the window happens, the grid structure enables it to access only a few cells according to the coefficients of the preference function. Second, it introduces the concept of dominance to remove unqualified candidates that cannot become query results in any time slot.

One-pass approach used to avoid the re-scanning,a simple approach is to maintain all the k -objects in the window . Whenever a new object slides into the window it inserts a new entry into the candidate set . Obviously, this entry remains within the candidate set until another k objects with scores higher than priority of current entry into the window . In other words, all the objects, even with very low scores, are present within candidate set for certain period of time. unfortunately , this method has worst case O(n) which works in liner time . Finally , we can propose a new approach which is self-adaptive partition based framework (SAP) . formally , given a query window W that contains n objects , a partition P(W , m) that divide window W into m sub-windows .

2.5 Methods used in improving continuous Top K-queries over stream

A lot of research [16] has been done to provide practical solutions for getting top K query . Many approaches depend on sliding window where it return the top k objects within each query window on the stream. Some of these approaches have showed good practical results in some applications such as SAP which partition the window into several disjoint sub-windows that get / update object from Top k objects in logarithmic time with AVL - Tree and keep Top K objects sorted based on score. The main idea of SAP is to check the potential of meaningful objects of current partition with the next partitions after sliding window without re-scanning these objects anymore . we can determine it by concept of Group Dominance Number that returns the number of elements that dominate smallest value of current partition by x objects . if $x \geq k$ then it is guaranteed to be empty , and it could be safely skipped , otherwise we can union meaningful objects with results as superset $< k$. To retrieve group dominance number , we must maintain objects with highest scores that less than smallest element in current partition . We can use a non-linear Data- structure that get elements in logarithmic time (eg , O(log n)) . then we can use AVL - Tree to retrieve the results

2.6 Challenges facing Ranking Top-K objects

Although SAP can provide good accuracy in time complexity , They have some limitations. It cannot update elements based on object for example , String lexicographical order and at the same time keep Objects stored based on count. This is an example if we deal with Top K words based on counting. More research is needed to improve the performance of returning Top K elements less than linear time.

Chapter 3

Methodology

This chapter states workflow of the project. The project was divided into three milestones.

3.1 Milestone 1: Real Time Analytics with Apache Storm

The first Milestone was concerned with streaming data and the state of art techniques used in Apache storm.

This was achieved by studying the course [14] *Real-Time Analytics with Apache Storm* which is offered online by udacity. Several concepts have been implemented during study pro- cess such as sliding window and Ranking Top- K objects. Moreover, the main concepts of Apache storm which is useful for implementing data stream. This course helped to :

- understand and grasp the concepts of real time analytics .
- understand the logic behind the functions implemented in Apache storm libraries.
- learn best practice technique when designing , implementing and evaluating optimization techniques with Twitter API.

In addition , we explored the following concepts of Apache Storm libraries in JAVA :[14]

- **Topologies:** The logic of real time application is packaged into storm topology. It is a DAG (Directed Acyclic Graph) of Spouts and bolts that ate connected with stream grouping.
- **Spouts:** A spout is a source of streams in a topology. Generally spout will read tuples from external source and emit them into the topology (eg. Twitter API).
- **Bolts:** All processing in typologies is done in bolts. Bolts can be simple stream transformations and complex stream transformations often requires multiple steps and thus multiple bolts to emit more than one stream.

3.2 Milestone 2 : Data stream Topology model with Java

In this milestone, we proposed a Topology plan for stream system using Java programming language to evaluate optimization techniques mentioned in section 2.3 . Based on Milestone 1 , we implemented a Topology model with data-set from Twitter API. The process of building this model was done in professional way. Our final model achieved an accuracy better than state of art approaches.

3.2.1 Twitter API spout

Every Topology model must start with spout that must be set to get data from API or Database storage for example. The model depends on spout to get data to produce good results for our Topology model.

3.2.2 Parsing tweets bolt

Our main objectives to get Top N words from tweets comes from Twitter Spout. Parsing tweet Bolt take all words that have unnecessary symbols to parse all of them and to emit to next Bolt for counting the occurrences of words.

3.2.3 Counter based bolt

It count all words that is emitted from parse tweet bolt. The main functionality of this bolt that it all words with Hash Map with Key = Hashtag and mapped to value Count.

3.2.4 Rolling Bolt

This bolt performs rolling counts of incoming objects using sliding window based counting. The bolt is configured by two parameters, the length of the sliding window in seconds (which influences the output data of the bolt, i.e. how it will count objects) and the emit frequency in seconds (which influences how often the bolt will output the latest window counts). For instance, if the window length is set to an equivalent of five minutes and the emit frequency to one minute, then the bolt will output the latest five-minute sliding window every minute.
The bolt emits a rolling count tuple per object, consisting of the object itself, its latest rolling count, and the actual duration of the sliding window. The latter is included in case the expected sliding window length (as configured by the user) is different from the actual length, e.g. due to high system load. Note that the actual window length is tracked and calculated for the window, and not individually for each object within a window.

3.2.5 Instance Ranking Bolt

This Bolt ranks incoming objects by their count with creating various instance of ranking bolts to decrease load on ranking process. Moreover, It emits Top N ranking objects with fixed emitting frequency time in seconds (eg. emit objects every 60 seconds) to Total Global Ranking Bolt.

3.2.6 Total Global Ranking Bolt

The main functionality of Global Bolt that is merge all objects coming from instance ranking bolt into final consolidated ranking.

3.3 Milestone 3 : Improving Continuous Top-K queries

This milestone was concerned with improving update top- K objects in instance ranking bolt which update objects with new total count which is emitted rolling bolt using sliding window. we will storing objects with AVL - tree which can get object with logarithmic time and update count in O(log n) which is worth nothing in computation time. There is a new approach that using hashing by Rabin karp algorithm[**?**] that can get objects with complexity O(1) and avoid collisions of same hashcode with different objects but still in development.

Chapter 4

Implementation

This chapter states for flow of implementation . We will discuss Apache Storm Configuration . then , how we implement workflow which discussed in Chapter 3 with Java . and last but not least , how we implement optimization ranking technique with Tree Data-Structure and compare it with naive implementation.
The capability of computer processor is intel core i7 7th generation with 16 GB RAM and Operating System which we use is Ubuntu 18.04 (Linux)

4.1 Apache Storm

4.1.1 Introduction

Apache Storm [15], in simple terms, is a distributed framework for real time processing of Big Data as it is a distributed framework for batch processing.

Why Storm was needed: First of all , we should understand why we need storm in real time processing, data will be coming continuously and we have to keep getting data,process it and keep writing the output as well. all these have to be done parallel and fast in distributed fashion with no data loss. This is where Storm pitches in.

Background History of Storm : Apache Storm was originally developed by Nathan Marz and the BackType team. BackType was later acquired by Twitter, who open-sourced the project. Later it got adopted by Apache and last year 2014, it has now become a top level Apache project.
Before Storm was written, the usual way of processing data in real time was using queues and worker thread approaches. For example, some threads will be continuously writing data to some queues like rabbitMq and some worker threads will be continuously reading data from these queues and processing them. The output might be written again to some other queues and chained as input to some other worker threads to process further.Such design is possible but obviously very fragile . Much of the time would be spent

in maintaining the entire framework,serializing/deserializing messages,dealing with data loss,resolving many other issues rather than doing the actual processing work. Nathan Marz came up with nice idea of creating abstraction to all these in efficient way in a program where we have to just create SPOUT and BOLT to do necessary processing and submit the job as TOPOLOGY and the framework will take care of everything else. Some of the really beautiful abstractions he came up with :

• **Stream:** Every data which will be processed in the topology is basically an abstraction called tuple and sequence of these tuples is called a stream .

4.1.2 Architecture Overview

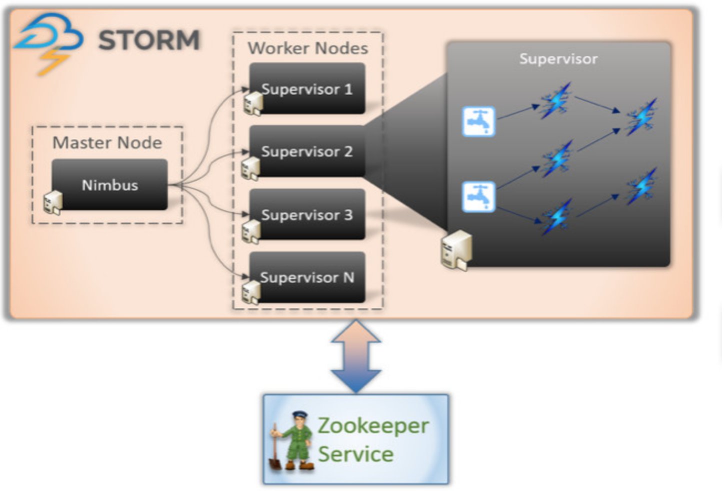


Figure 4.1: Apache Architecture [1]

In Storm cluster we run Topologies.A Topology once started,is intended to keep on processing live data forever which it keeps on getting from data sources until we wish to kill it.

In terms of Node Services :
there are 2 kinds of nodes on a Storm cluster: Master node and Worker nodes.
The master node runs a daemon called **"Nimbus"**. Nimbus is responsible for distributing code around the cluster, assigning tasks to machines, and monitoring for failures .

Each worker node runs a daemon called the **"Supervisor"**. Each supervisor can run one or more worker processes which are separate JVM processes on its node. Each worker process in itself, can run one or more tasks parallel (spout/bolt) . Each supervisor listens for work assigned by Nimbus to its node and starts and stops worker processes as necessary . Each worker process executes a subset of a topology; a running topology consists of many worker processes spread across many machines.

In terms of Tasks :
Storm runs 2 tasks : Spouts and Bolts . In a topology, spout will act as data receiver from external sources and creator of Stream for bolts for actual processing. Bolts can be chained serially or in parallel depending on what kind of processing we want to do. A simple Word Count problem can be solved in Storm in following way :

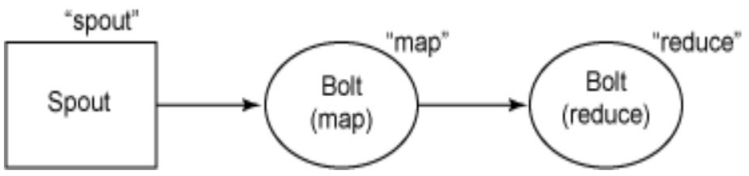


Figure 4.2: [1]

Another important point to be aware of is that Storm makes use of ZEROMQ library for inter-process communication(between different worker processes)

Storm Components in depth :

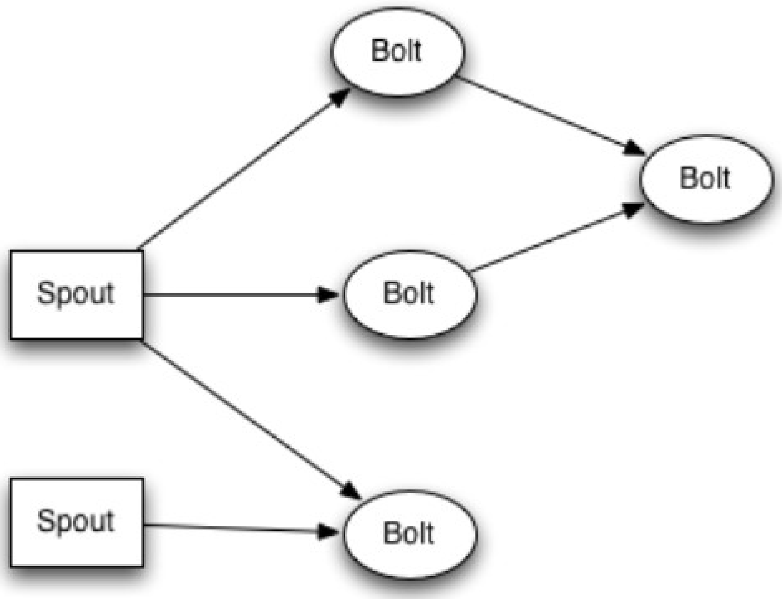


Figure 4.3: [5]

1) **Topology:**

In simple words, Topology is a network of spouts and bolts as in above figure
It is a graph of computation consisting of spouts and bolts. Spouts as data stream source tasks and Bolts as actual processing tasks.
Each node in the graph contains some processing logic and links in the graph shows how the data will be passed and processing will happen among nodes.
When a topology is submitted to a Storm cluster, Nimbus service on master node consults the supervisor services on different worker nodes and submits the topology. Each supervisor, creates one or more worker processes, each having its own separate JVM . Each process runs within itself threads which we call Executors. The thread/executor processes the actual computational tasks : Spout or Bolt .

Topology is submitted to a storm cluster through a command :

```
$ storm jar storm-topology-code.jar com.storm.TopNTopology
```

2) **Stream:**

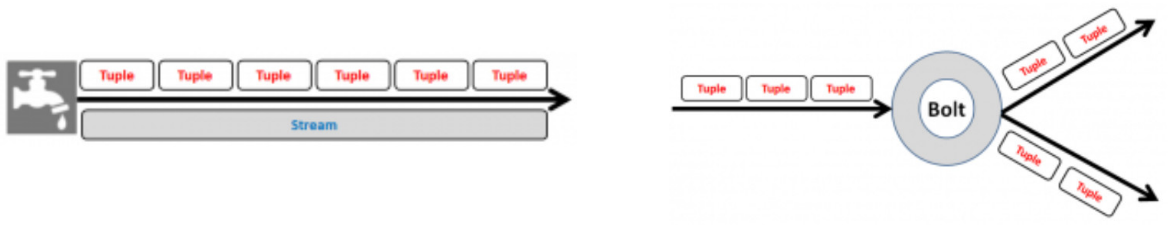


Figure 4.4: [1]

Stream is the core abstraction in Storm. A stream is an unbounded sequence of tuples (collection of key value pairs).Tuple is the most basic data structure in storm . Its a named list of values. Each field in the values can be an object of any serializable type. Stream is the core abstraction in Storm. A stream is an unbounded sequence of tuples (collection of key value pairs).

3) **Spout :**

Spout is the entry point in a storm topology. It is the source of streams in the topology. A spout connects to the actual data source such as a message queue, gets continuous data , converts the actual data into stream of tuples , emits them to bolts for actual processing. Spouts run as tasks in worker processes by Executor threads .

Spouts can broadly be classified as follows:

• **Reliable:**These spouts have the ability to replay the tuples (a unit of data in the data stream). This helps applications achieve the "at least once message processing" semantic as, in case of failures, tuples can be replayed and processed again. Spouts for

fetching data from messaging frameworks are generally reliable, as these frameworks provide a mechanism to replay the messages.
• **Unreliable:** These spouts do not have the ability to replay the tuples. Once a tuple is emitted, it cannot be replayed, regardless of whether it was processed successfully. This type of spout follows the "at most once message processing" semantic.

4) **Bolt :** Bolt contains the actual processing logic. It works only on streams and can emit streams too for further processing downstream by other bolts or can export/save data for persistent storage. It receives stream from either one or more spouts or some other bolts. For example in simple word count example(see diagram above), map and reduce tasks will be executed as 2 different bolts executed in serial fashion. Bolts can do anything from run functions, filter tuples, do streaming aggregations, do streaming joins, talk to databases, and more.

4.1.3 Java Storm Interfaces

Storm provides Java as a basic interface. We can have interfaces for spouts, bolts or both. Messaging interfaces to Kafka, Kestrel, and Twitter are available in Storm.

There are three types of classes provided.

A)Spout Interfaces:

Spout interface provides the interfaces to create streams from external data. There is a main interface provided for Java which is *BaseRichSpout* class.

BaseRichSpout

BaseRichSpout is the main interface to implement spouts in Storm.It provides the interface for Storm spouts. We will present some important methods that are a part of the BaseRichSpout interface. the term interface in Java means that it is only a template and any class that implements the interface has to provide the methods for the interface.

There are three methods with the signature for the methods:-

• **Open method**

Open method is called when the spout is initialized. It is called only once for the Spout. It gets the configuration for the spout and also the context of the spout. The collector is used to emit or output the tuples from this spout.

• **next Tuple method**

The nextTuple method is called by Storm to request the spout to emit one or more tuples to the output collector. If the spout has no tuples to emit, then the method should return immediately. It normally calls the emit method on the output collector of the spout to add the tuples to the stream.

- **declareOutputFields Method**

The declareOutputFields method declares the output fields for the tuples output by this spout.

```
public class ExampleSpout extends BaseRichSpout {
   SpoutOutputCollector _collector;

   @Override
   public void open(Map conf, TopologyContext
       context,SpoutOutputCollector collector) {
      _collector = collector;
   }

   @Override
   public void nextTuple() {
      String sentence = "Hello World! " ;
      _collector.emit(new Values(sentence));
   }

   @Override
   public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("sentence"));
   }
}
```

The program fragment shows the spout definition using the BaseRichSpout class.

The spout is defined as a class with name ExampleSpout. It extends the base class BaseRichSpout. We can see the definition overriding three methods open, nextTuple and declareOutputFields.

In the open method, an output collector is created . The method nextTuple is used to create the output stream.

The program fragment shows the nextTuple method of the spout to emit current String to next Bolt which depends on current Spout.

B) Bolt Interface

Bolt interface provides the classes and interfaces for processing input streams from spouts and other bolts. There are two main interfaces and two main classes provided for Java.

The two interfaces provided are **IRichBolt** and **IBasicBolt**, and the classes provided are BaseRichBolt and BaseBasicBolt.

IRichBolt is the main interface to implement bolts in Storm. *BaseRichBolt* and other language or database bolts extend the *IRichBolt*. *BaseRichBolt* is a class implementing *IRichBolt*. *IBasicBolt* is a bolt interface similar to *IRichBolt* but has the automatic tracking of tuples. *BaseBasicBolt* is a class implementing *IBasicBolt* and is used to track tuples.

For simplicity ,there is figure below to summarize Bolt interfaces and classes.

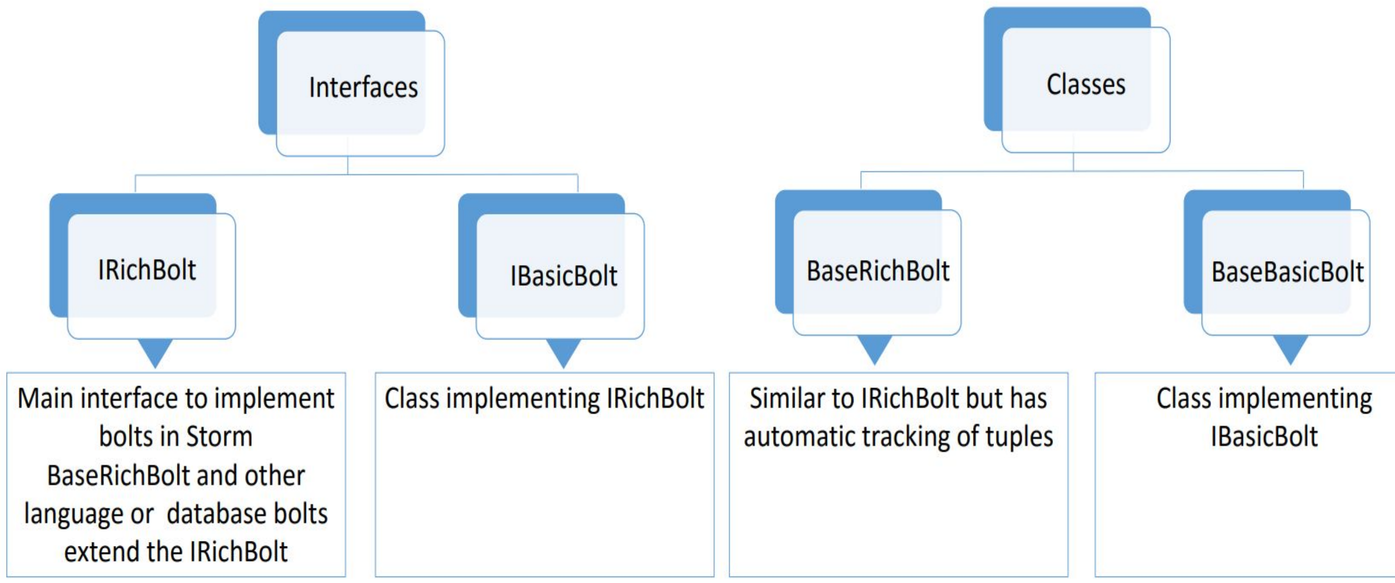

Interfaces
Classes
IRichBolt
IBasicBolt
BaseRichBolt
BaseBasicBolt
Main interface to implement bolts in Storm BaseRichBolt and other language or database bolts extend the IRichBolt
Class implementing IRichBolt
Similar to IRichBolt but has automatic tracking of tuples
Class implementing IBasicBolt

Figure 4.5

1) **IRichBolt**

RichBolt interface provides the interface for bolts in Storm. Some of the important methods for IRichBolt interfaces along with the method signature are provided below in detail.

• **Prepare method** : The prepare method is called when the bolt is initialized and is similar to the open method in spout. It is called only once for the bolt. It gets the configuration for the bolt and also the context of the bolt. The collector is used to emit or output the tuples from this bolt.

• **Execute method** : The execute method is called by Storm for each input tuple. This method can process the input tuple and emit one or more tuples to the output collector. If the bolt has no tuples to emit, then the method should return immediately. It normally calls the emit method on the output collector of the bolt to add the tuples to the stream.

The emit method should include the input tuple as a parameter to indicate successful processing of the input tuple.

2) **BaseRichBolt**

BaseRichBolt implements the IRichBolt interface and also the component interface of Storm. Apart from the IRichBolt methods, this class contains a couple of important methods as described below.

• **DeclareOutputFields method**: The declareOutputFields method is used to specify the output fields for the tuples output by this bolt.

• **getComponentConfiguration method** : The getComponentConfiguration method can be called to get the current configuration of the bolt.

3) **IBasicBolt**

The IBasicBolt interface provides the interface for reliable processing in Storm. Some of the methods for this interface along with signatures are described below.

• **Prepare method**: The prepare method is called when the bolt is initialized and is similar to the open method in spout. It is called only once for the bolt. It gets the configuration for the bolt and also the context of the bolt. This method can be used for opening any files for output or to open any connections to the database for storing data.

•**Execute Method**: The execute method is called by Storm for each input tuple. This method can process the input tuple and emit one or more tuples to the output collector. If the bolt has no tuples to emit, then the method should return immediately. It normally calls the emit method on the output collector of the bolt to add the tuples to the stream. This method automatically manages the acking for the input tuple and sends an ack for the input tuple at the end of the method. We can throw a FailedException if we want to indicate that the processing has failed. This interface is different from the IRichBolt interface.

4) **Basebasicbolt**

BaseBasicBolt implements the IBasicBolt interface and also the component interface of Storm. Apart from the IBasicBolt methods, this class contains a couple of important methods.

• **declareOutputFields method :** The declareOutputFields method is used to specify the output fields for the tuples output by this bolt.

• **getComponentConfiguration method** : The getComponentConfiguration method can be called to get the current configuration of the bolt.

```
public class ExampleBolt extends BaseRichBolt {
   OutputCollector collector ;
```

```
    public void prepare(Map stormConf, TopologyContext context,
        OutputCollector collector)
    {

       this.collector = collector ;
    }

    public void execute(Tuple input)
    {
       String currentString = input.getString(0) ;
       String newString = currentString.toLowerCase();
       collector.emit(new Values(newString)) ;
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer)
    {
       declarer.declare(new Fields("lowerCase"));
    }

    @Override
    public Map<String, Object> getComponentConfiguration()
    {
       return super.getComponentConfiguration();
    }
 }
```

The code fragment shows the bolt definition *ExampleBolt* that extends the bolt class *BaseBasicBolt*. It has the *execute* and *declareOutputFields* methods. The declareOutputFields method creates one field for the output tuple from the bolt. The field is lowercase string symbol.

C) Stream Groupings
The tuples in a stream are grouped before reaching a bolt. There are eight groupings provided by Storm as follows:

1) **Shuffle grouping**

In this grouping, tuples are randomly distributed to bolts so that each task gets an equal number of tuples.

2) **Fields Grouping**

In this grouping, tuples are first partitioned by the fields specified. Those in the same partition are always sent to the same bolt.

3) **Partial Key Grouping**

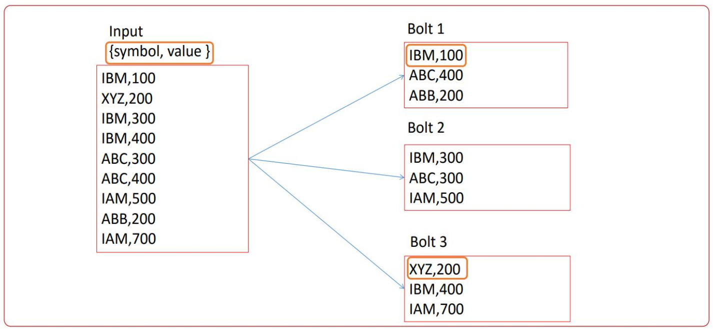


Figure 4.6: Shuffle grouping [2]

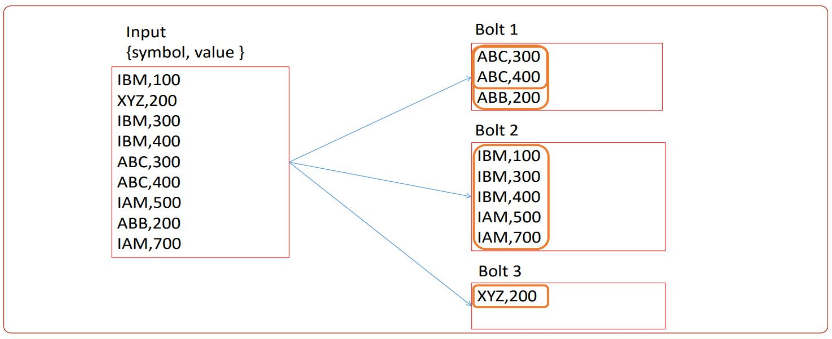


Figure 4.7: Fields Grouping [2]

This grouping is a combination of fields grouping and shuffled grouping. Tuples are grouped by the field values and also shuffled to do a load balancing for the bolts.

4) **All Grouping**

In this grouping, tuples are replicated across all the tasks of the bolt. There will be multiple copies of the tuples.

5) **Global Grouping**

In this grouping, all the tuples go to the same bolt instance. Bolt with the lowest ID gets all the tuples.

6) **None Grouping**

n this grouping, the user does not care which bolt processes which tuple. Currently, this is implemented the same way as shuffle grouping. Tuples are randomly distributed to bolts so that each task gets an equal number of tuples.

7) **Direct Grouping**

n this grouping, the producer of the tuple specifies which bolt instance the tuple should be sent to. This is done with the emit Direct method.

8) **Local or Shuffle Grouping**

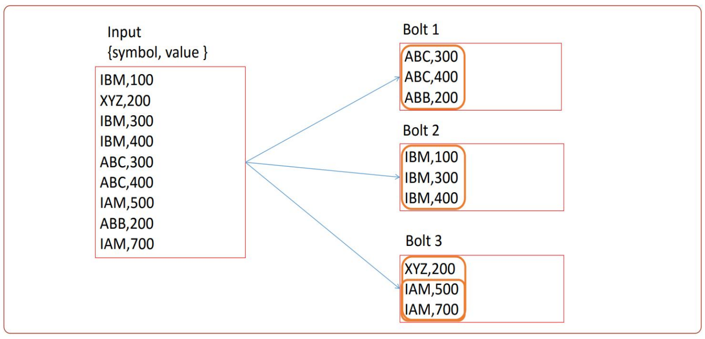


Figure 4.8: Partial Key Grouping [2]

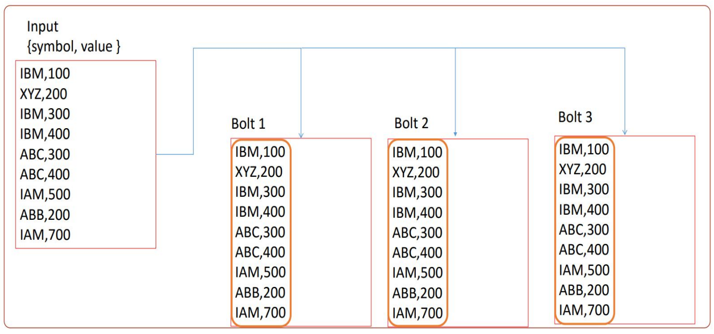


Figure 4.9: All Grouping [2]

In this grouping, tuples are given preference to bolt instances in the same worker process. If there are no local instances, the tuples are distributed in the way as shuffle grouping.

D) Topology Interface

Topology interface provides the classes and methods for creating topologies connecting spouts and bolts.

TopologyBuilder Methods

These methods are a part of TopologyBuilder class.

Create a new topology with the given StormTopology object.

```
StormTopology createTopology()
```

Add this bolt to the topology. The bolt is given the name specified in ID and can be referenced by other spouts or bolts. The output object is used to declare inputs for this bolt.

```
BoltDeclarer setBolt(String id, IBasicBolt bolt, int parallelism_hint)

BoltDeclarer setBolt(String id, IRichBolt bolt, int parallelism_hint)
```

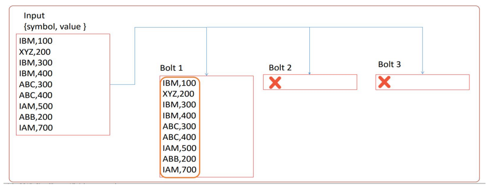


Figure 4.10: Global Grouping [2]

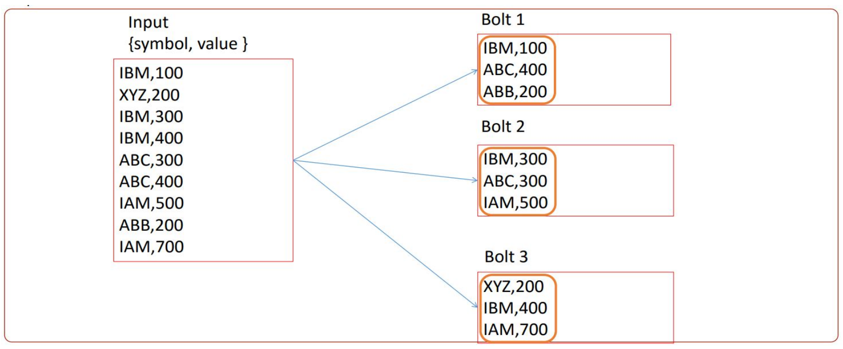


Figure 4.11: None Grouping [2]

Same as the previous method with a third parameter parallelism hint that specifies the number of parallel tasks to run this bolt, that is if the parallelism hint is 3 then the worker process may start up to three tasks to run the same bolt.

```
BoltDeclarer setBolt(String id, IBasicBolt bolt)

BoltDeclarer setBolt(String id, IRichBolt bolt)
```

Add this spout to the topology. The spout is given the name specified in ID and can be referenced by other spouts or bolts. The output object can be used to set some properties of the spout such as a number of tasks and maximum parallelism.

```
SpoutDeclarer setSpout(String id, IRichSpout spout)

SpoutDeclarer setSpout(String id, IBasicSpout spout)
```

Same as the previous method with a third parameter parallelism hint that specifies the number of parallel tasks to run this spout, that is if the parallelism hint is 3 then the worker process may start up to three tasks to run the same spout.

```
SpoutDeclarer setSpout(String id, IRichSpout spout, int parallelism_hint)

SpoutDeclarer setSpout(String id, IBasicSpout spout, int parallelism_hint)
```

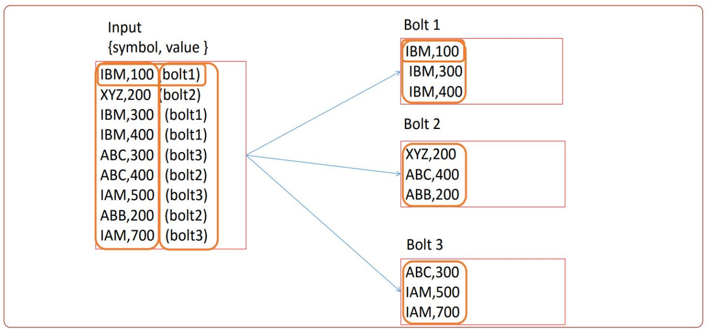


Figure 4.12: Direct Grouping [2]

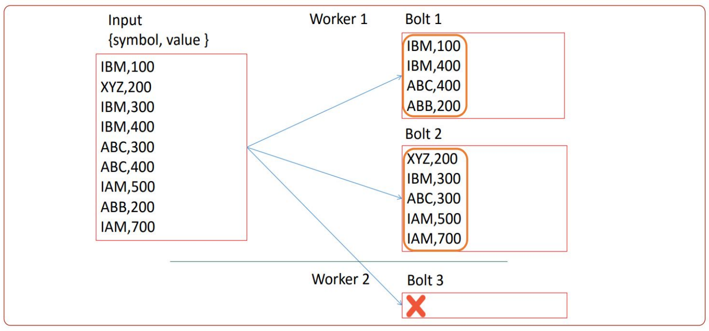


Figure 4.13: Local or Shuffle Grouping [2]

BoltDeclarer Methods

BoltDeclarer class contains the methods for grouping the input tuples for the bolt and also to specify which spout/bolt to use for input. This class contains the following important methods.

StormSubmitter Methods

StormSubmitter class is used to submit a topology to Storm cluster when we use the command Storm jar to submit a topology class to a cluster. The given table lists some of the methods of this class including their signature.

• *submitTopology Method :* The submitTopology method is used to submit a topology to run on the cluster. It takes three parameters.

```
void submitTopology(String name, Map stormConf, StormTopology topology)
```

The first parameter name is the name given to the topology and will be shown when we use the command storm list. stormConf is used to specify configuration parameters such as the number of workers and debug flag.

• **Topology Builder Example**

The main function that connects the spouts and bolts using TopologyBuilder

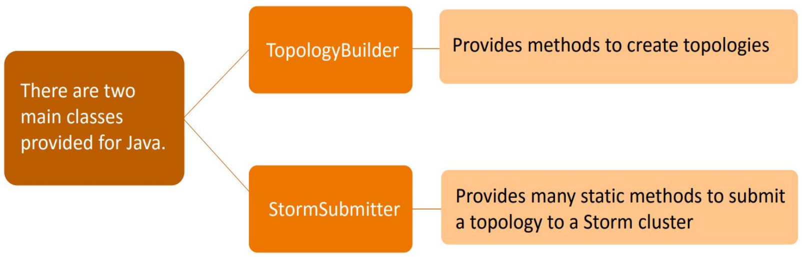

There are two main classes provided for Java.
TopologyBuilder
Provides methods to create topologies
StormSubmitter
Provides many static methods to submit a topology to a Storm cluster

Figure 4.14: Topology Interface [2]

```java
public class ExampleTopology
{
   public static void main (String [] args)
   {
      TopologyBuilder builder = new TopologyBuilder() ;
      ExampleSpout spout = new ExampleSpout() ;

      builder.setSpout("spout", spout, 1);

      builder.setBolt("bolt", new
         ExampleBolt(),10).shuffleGrouping("spout");

      Config config = new Config() ;
      config.setDebug(true);

      LocalCluster cluster = new LocalCluster() ;

      cluster.submitTopology("ExampleTopology" , config ,
         builder.createTopology());

      cluster.killTopology("ExampleTopology");

      cluster.shutdown();

   }
}
```

The code fragment for the main function that connects the spout and bolts to create the topology is shown here. The main function first gets a handle for the TopologyBuilder object.

Then, it calls the setSpout method to set the ExampleSpout as the spout. This component is given the name spout. The third parameter 1 to setSpout indicates to run only one spout in parallel. Since we are reading from a single file, we want only one instance to read the file. Next, it calls the setBolt method to set the ExampleBolt bolt. This bolt is given the name bolt. Here, the third parameter is 10 indicating ten bolt instances to run in parallel. The output of this method is stored as a ExampleBolt object. Next, the bolt is linked to the spout using shuffle grouping. So, the ten bolt instances will get the tuples from the spout in random order. Finally, the main function submits the topology with the given configuration to Storm. It uses ExampleTopology as the name of the topology. The createTopology method is called to create the topology, and the handle is sent to Storm submit.

4.2 Topology Storm Design

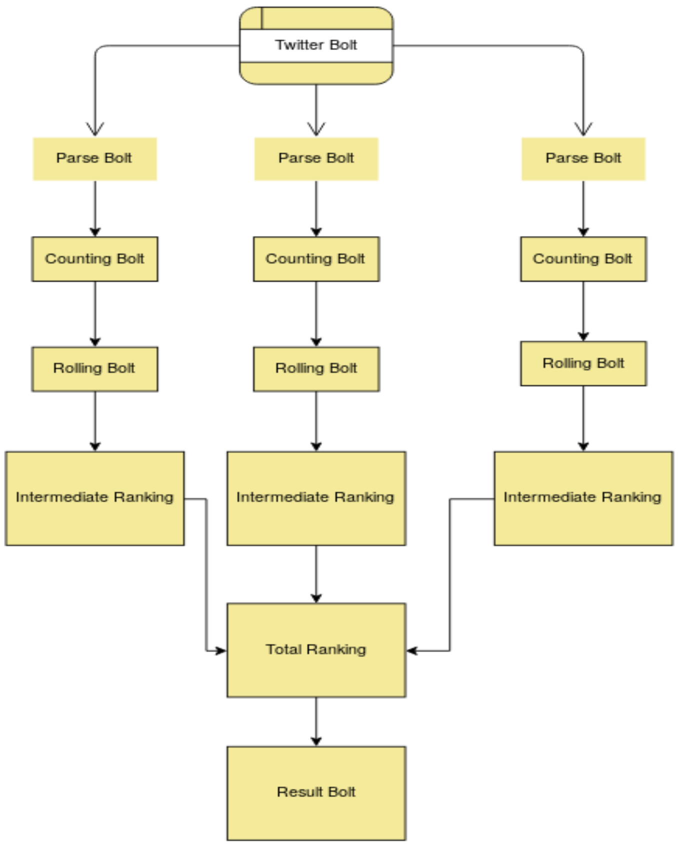


Figure 4.15: Storm Topology Architecture [3]

This section states workflow for Topology design that will be used for evaluating optimization technique . Topology design is like DAG graph which means that any entity will not work unless all dependencies are finished . First of all , we start topology design with *Spout* to take data from *Twitter API.* then , we will create some bolts to make come computations after getting data from Spout . For more elaboration , Spout class emit data to *Parse Bolt* which parse all characters that insignificant such as , ! etc . then

, Parse Bolt emit to *Count Bolt* which used to count all words . Furthermore , Count Bolt emit to *Rolling Bolt* which used Sliding window Technique . Moreover, Data will be emitted to *Intermediate Ranking* which used to update all words to Intermediate Ranking Bolt using Data-Structure *AVL Tree* . Finally , all data from Intermediate Ranking Bolts are merged to *Total Ranking Bolt* which put all data in *Priority queue* where It is already predefined in Java. We used Priority queue to return all values sorted based on Word's count.

4.2.1 Twitter Spout

Basically , Twitter Spout is used for getting all words from tweets in a continuous streaming data. First of all , We extends TwitterSpout class with BaseRichSpout to override all methods of BaseRichSpout as we mentioned before in section 4.1.3. Moreover, We create class which implements an interface *StatusListener*. StatusListener is used to add all tweets into the queue buffer.

```
private class TweetListener implements StatusListener
{

   public void onStatus(Status status)
   {
      queue.offer(status.getText());
   }
}
```

. We add an external library called *twitter4j* to build configuration of Spout and get all data from twitter API .

Basically , We start with adding configuration with credentials for twitter 4j . we build config with Consumer key , Consumer secret key , Access token and Access secret token .

```
public TweetSpout(String key, String secret, String token,String
    tokensecret)
{
   custkey = key;
   custsecret = secret;
   accesstoken = token;
   accesssecret = tokensecret;
}
```

. Then, we override five methods which extends *BaseRichSpout*

- **Open Method**

When open methods declared. queue will be created it add tweets in queue buffer. then, provide *spout collector* to emit data to nextBolt . Moreover, we build the configuration with credentials for twitter 4j as we mentioned before. Finally , we provide the handler *TwitterListener* for twitter stream to add tweets in queue buffer.

```java
@Override
public void open(Map map,TopologyContext topologyContext,
    SpoutOutputCollector spoutOutputCollector)
{
   queue = new LinkedBlockingQueue<String>(1000);

   collector = spoutOutputCollector;

   ConfigurationBuilder config = new ConfigurationBuilder().
   setOAuthConsumerKey(custkey).
   setOAuthConsumerSecret(custsecret).
   setOAuthAccessToken(accesstoken).
   setOAuthAccessTokenSecret(accesssecret);

   TwitterStreamFactory fact = new TwitterStreamFactory(config.build());

   twitterStream = fact.getInstance();

   twitterStream.addListener(new TweetListener());

   twitterStream.sample();
}
```

- **nextTuple method**

Basically , It is used to get data from queue buffer from the peek and remove it . then , we have two possibilities. if there is no element in the queue . therefore , program will sleep for 50 millis to wait for API to add tweets in queue buffer. Otherwise , *Spout Collector* emit tweet to *Parse Bolt.*

```java
@Override
public void nextTuple()
{
   String ret = queue.poll() ;

   if(ret == null)
```

```
        {
           try {
              sleep(50);
           } catch (InterruptedException e) {
              e.printStackTrace();
           }

           return;
        }
        collector.emit(new Values(ret));
     }
```

- **Component Configuration method**

Component Configuration is used for setting maximum tasks parallelism equals to one. Formally , We set configuration to create only one instance for Twitter Spout.

```
        public Map<String, Object> getComponentConfiguration()
        {
           Config ret = new Config();
           ret.setMaxTaskParallelism(1);
           return ret;
        }
```

- **declareOutputFields method**

Basically , It is used of naming output fields which is emitted by *nextTuple* method with **tweet**.

```
        public void declareOutputFields(OutputFieldsDeclarer declarer)
        {
           declarer.declare(new Fields("tweet"));
        }
```

- **Close method**

It is used for shutdown twitter stream when we are going to exit.

```
        public void close()
        {
           twitterStream.shutdown();
        }
```

4.2.2 Parse Tweet Bolt

Parse tweet bolt is used to parse all unused characters from all tweets that came from *Twitter Spout* . First of all , We extends ParseTweetBolt class with BaseRichBolt to override all methods of BaseRichBolt. we override two methods which extends BaseRichBolt.

- **execute Method**

First of all , We get string which is emitted by Twitter Spout by field "tweet" .Moreover , This tweet will be spitted from all characters which are unnecessary to count. and last but not least , all words will be emitted to *Counting Objects Bolt* with *Bolt Collector*

```
@Override
public void execute(Tuple input)
{
   String tweet = input.getStringByField("tweet") ;

   StringTokenizer tokenizer = new StringTokenizer(tweet , "[
       .,?!@#$%^&*(+)_=/\\\"'|‘~{}]+" ) ;

   while(tokenizer.hasMoreTokens()) {
      String s = tokenizer.nextToken() ;
      boolean can = true ;
      for(char c : s.toCharArray())
         can &= ('A' <= c && c <= 'Z') || ('a' <= c && c <= 'z') ;
      if(can)
         collector.emit(new Values(s));
   }
}
```

- **declareOutputFields Method**

Basically , It is used of naming output fields which is emitted by *execute* method with **tweet-word**.

```
public void declareOutputFields(OutputFieldsDeclarer declarer)
{
   declarer.declare(new Fields("tweet-word"));
}
```

4.2.3 Counting Objects Bolt

Counting Objects Bolt is used to count all tweets using *HashMap* which is already predefined in Java . we override two methods which extends BaseRichBolt.

- **execute Method**

We get string which is emitted by *Parse Tweet Bolt* by field "tweet-word". Moreover , current word is added in *HashMap* . then , it will be emitted using Bolt Collector to *Rolling Bolt*

```
public void execute(Tuple input)
{
   String word = input.getStringByField("tweet-word");

   if(map.get(word) == null)
   map.put(word , 1l) ;
   else
   map.put(word , map.get(word) + 1) ;

   collector.emit(new Values(word , map.get(word))) ;
}
```

- **declareOutputFields Method**

  Output Fields are declared which is emitted by *execute* method with **word** and **count**.

```
public void declareOutputFields(OutputFieldsDeclarer declarer) {
   declarer.declare(new Fields("word","count"));
}
```

4.2.4 Rolling Bolt

Basically , Rolling Bolt is used to store all tuples came from Counting Bolt to *Sliding Window*. More Formally , Sliding Window provides *rolling* counts of the occurrences of objects, i.e. a sliding window count for each tracked object. The size of the sliding window is equivalent to the (fixed) number of slots number of a given *Sliding Window* instance. Sliding Window will be described more in details in Section 4.2.5 . The number of slots is equal to window length divides by frequency $\frac{window}{frequency}$

which means frequency for emitting Objects for Sliding Window to Intermediate Bolt.

```
public RollingBolt(int window , int frequency)
{
   windowLength = window ;
   frequencyEmit = frequency ;
   this.window = new SlidingWindow<Object>(window / frequency) ;
}
```

There is a boolean method called *isTickTuple*.Whenever we want a spout or bolt execute a task at periodic intervals in other words, we want to trigger an event or activity using a tick tuple is normally the best practice.More Formally, Tick tuples is common to require a bolt to "do something" at a fixed interval, like flush writes to a database. Many people have been using variants of a ClockSpout to send these ticks. The problem with a ClockSpout is that we cannot internalize the need for ticks within our bolt, so if we forget to set up bolt correctly within topology it will not work correctly. *Apache Storm* java library introduces a new "tick tuple" config that specify the frequency at which we want to receive tick tuples with the "topology.tick.tuple.freq.secs" component-specific config, and then bolt will receive a tuple from the system component and tick stream at that frequency.

```
// Configuration for Tick Tuple
@Override
public Map<String, Object> getComponentConfiguration() {
   Map<String,Object> conf = new HashMap<String , Object>();
   conf.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS, frequencyEmit);
   return conf;
}
```

```
private boolean isTickTuple(Tuple tuple)
{
   return tuple
   .getSourceComponent()
   .equals(Constants.SYSTEM_COMPONENT_ID)
    &&
    tuple.getSourceStreamId()
    .equals(Constants.SYSTEM_TICK_STREAM_ID);
}
```

There are *two* methods which extends *BaseRichBolt*.

- **execute Method**

There are two possibilities for *input* tuple. If input tuple is a *Tick Tuple* then all objects in sliding window should be emitted to *Intermediate Ranking Bolt*. Otherwise, input tuple will be added in sliding window to update object's count in current slot

```
public void execute(Tuple input)
{
   if(isTickTuple(input))
   {
       HashMap<Object , Long> res = window.getCountsThenAdvanceWindow() ;
      for(Object object : res.keySet())
         collector.emit(new Values(object , res.get(object) )) ;
   }
   else
   {
      window.addObject(input.getValue(0));
      collector.ack(input);
   }
}
```

- **declareOutputFields Method**

It is used for naming output fields which is emitted by *execute* method with **Object** and **Count**

```
public void declareOutputFields(OutputFieldsDeclarer declarer)
{
   declarer.declare(new Fields("object", "count"));
}
```

4.2.5 Intermediate Ranking Bolt

This bolt ranks incoming objects by their count in order to produce intermediate rankings. he topology runs many of such intermediate ranking bolts in parallel to distribute the load of processing the incoming rolling counts from the Rolling Bolt instances. It use *execute* method to allow actual bolt implementations to specify how incoming tuples are processed, i.e. how the objects embedded within those tuples are retrieved and counted.

This bolt has a private *Rankings field* to rank incoming tuples (those must contain Rankable objects) according to their natural order.

```
public IntermediateRanking(int topN, int frequency) {
   this.topN = topN ;
   frequencyEmit = frequency ;
   ranking = new Ranking(topN) ;
}
```

There are three methods which are used for *Intermediate Ranking Bolt*

- **updateRankings Method**

This method provides for updating current object in ranking class based on total count .

```
public void updateRankings(Tuple tuple) {
   Rankable r = RankableObjects.createRankable(tuple) ;
   ranking.updateWith(r);
}
```

- **execute Method**

There are two possibilities for *input* tuple. If input tuple is a *Tick Tuple* then all objects in ranking class should be emitted to *Total Ranking Bolt*. Otherwise, input tuple will be updated to ranking class based on total count.

```
public void execute(Tuple input)
{
   if(isTickTuple(input))
      collector.emit(new Values(ranking.copy())) ;
   else
      updateRankings(input);
}
```

- **declareOutputFields Method**

  It is used for naming output fields which is emitted by *execute* method with **ranking**

```
public void declareOutputFields(OutputFieldsDeclarer declarer)
{
   declarer.declare(new Fields("ranking"));
}
```

4.2.6 Total Ranking Bolt

This bolt merges incoming intermediate Rankings emitted by the Intermediate Ranking Bolt instances. Total Ranking Bolt is similar to Intermediate Ranking Bolt . The only difference will be in *updateRanking* method which is used to merge all Ranking Objects into one instance.

```
public void updateRankings(Tuple tuple)
{
   Ranking merge = (Ranking) tuple.getValue(0) ;
   ranking.updateWith(merge);
}
```

4.3 Implementing the Data Structures

Eventually , We settled down to the following core data structures for the new distributed *Rolling Count algorithm.* an interesting characteristic is that these data structures are completely decoupled from any Storm internals. Our Storm bolts will make use of them, but there is no dependency in the opposite direction from the data structures to Storm.

- Classes used for counting objects: **SlotCounter**, **SlidingWindow**.

- Classes used for ranking objects by their count: **Rankings**, **Rankable**, **RankableObjectWithFields**.

4.3.1 Slot Counter

The Slot Counter class provides per-slot counts of the occurrences of objects. The number of slots of a given counter instance is fixed. The class provides four public methods:

```
public void addObject(T obj, int slot);
public void wipeSlot(int slot):
public long getCount(T obj, int slot)
public Map<T, Long> getAllObjects();
```

Internally, *SlotCounter* is backed by a Map<T, long[]> for the actual count state. We might be surprised to see the low-level long[] array here would not it be better Object Oriented style to introduce a new, separate class that is just used for the counting of a single slot, and then we use a couple of these single-slot counters to form the SlotBasedCounter? Well, yes we could. But for performance reasons We decided not to go down this route. Apart from updating the counter which is a WRITE operation the most common operation in our use case is a READ operation to get the total counts of tracked objects. Here, we must calculate the sum of an object's counts across all slots. And for this it is preferable to have the individual data points for an object close to each other , which the long[] array allows us to do.

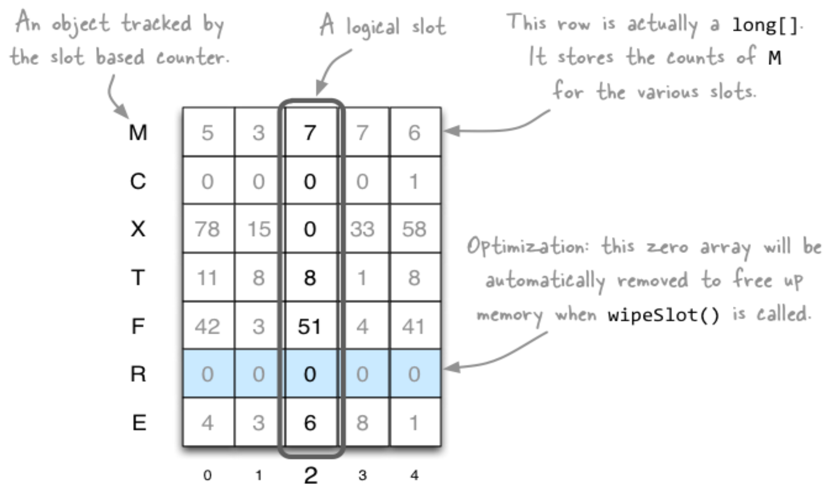

An object tracked by the slot based counter.
A logical slot
This row is actually a long[]. It stores the counts of M for the various slots.
Optimization: this zero array will be automatically removed to free up memory when wipeSlot() is called.
M 5 3 7 7 6
C 0 0 0 0 1
X 78 15 0 33 58
T 11 8 8 1 8
F 42 3 51 4 41
R 0 0 0 0 0
E 4 3 6 8 1
0 1 2 3 4

Figure 4.16: SlotCounter class [4]

The SlotCounter class keeps track of multiple counts of a given object. In the example above, the SlotCounter has five logical slots which allows us to track up to five counts per object.

The Slot Counter is a primitive class that can be used, for instance, as a building block for implementing sliding window counting of objects.

4.3.2 Sliding Window

The Sliding Window class provides rolling counts of the occurrences of objects. Its counting functionality is based on the previously described Slot Counter . The size of the sliding window is equivalent to the (fixed) number of slots number of a given SlidingWindow instance. It is used by Rolling Bolt for counting incoming data tuples.

The class provides two public methods:

```
public void incrementCount(T obj);
Map<T, Long> getCountsThenAdvanceWindow();
```

What might be surprising to some readers is that this class does not have any notion of time even though *sliding window* normally means a time-based window of some kind. In our case the window does not advance with time but whenever the method getCountsThenAdvanceWindow() is called. This means SlidingWindow behaves just like a normal ring buffer in terms of advancing from one window to the next.

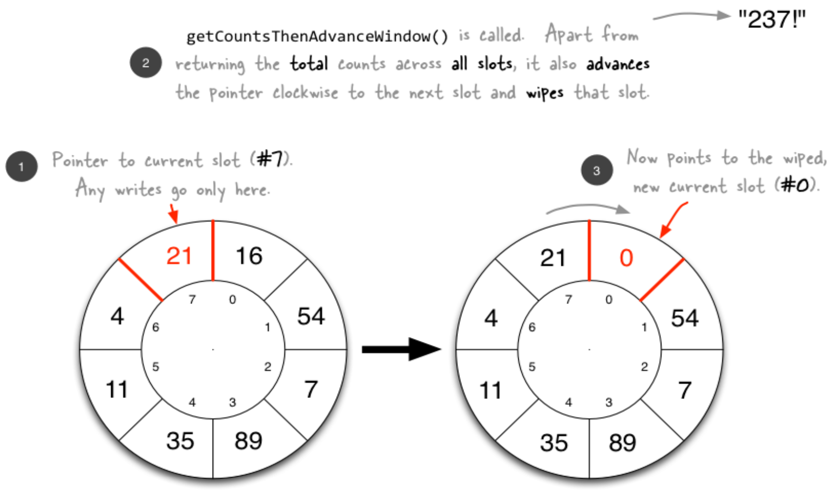

2 getCountsThenAdvanceWindow() is called. Apart from returning the total counts across all slots, it also advances the pointer clockwise to the next slot and wipes that slot.
"237!"
1 Pointer to current slot (#7). Any writes go only here.
3 Now points to the wiped, new current slot (#0).
21
16
54
7
89
35
11
4
0
1
2
3
4
5
6
7
21
0
54
7
89
35
11
4

Figure 4.17: The Sliding Window Counter[4]

The SlidingWindow class keeps track of multiple rolling counts of objects, i.e. a sliding window count for each tracked object.Note that the example of an 8-slot sliding window counter above is simplified as it only shows a single count per slot. In reality SlidingWindow tracks multiple counts for multiple objects.

4.3.3 Ranking and Rankable

The Ranking class represents fixed-size rankings of objects, for instance to implement Top N rankings. It ranks its objects descendingly according to their natural order, i.e. from largest to smallest. This class is used by Intermediate and Total Ranking Bolt. its derived bolts to track the current rankings of incoming objects over time.

The class provides five public methods:

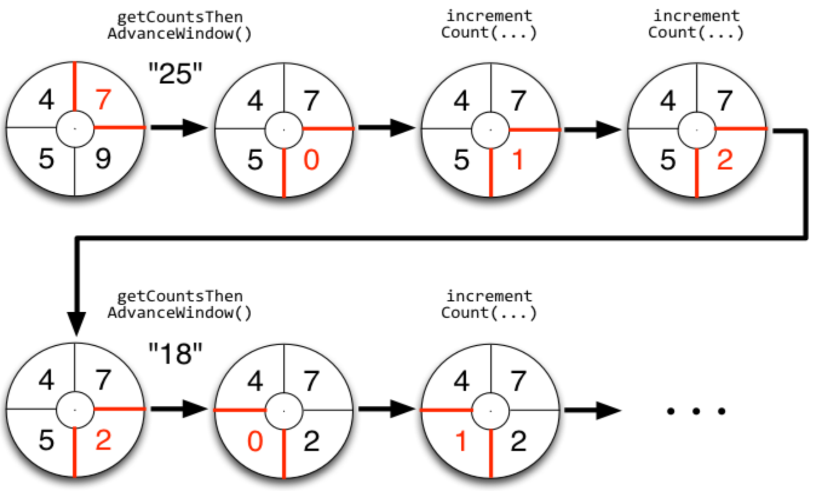


Figure 4.18: Example for Sliding Window behavior [4]

```
public void updateWith(Rankable r);
public void updateWith(Ranking merge);
public List<Rankable> getRankings();
public int maxSize();
public int size();
```

Whenever we update Rankings with new data, it will discard any elements that are smaller than the updated top N, where N is the maximum size of the Rankings instance.

Now the sorting aspect of the ranking is driven by the natural order of the ranked objects. We created a Rankable interface that in turn implements the Comparable interface. In practice, We simply pass a Rankable object to the Ranking class, and the latter will update its rankings accordingly.

```
Rankings top = new Rankings(100000);
Rankable C = ...;
top.updateWith(r);
List<Rankable> rankings = top.getRankings();
```

The concrete class implementing Rankable is RankableObjectWithFields. The bolt Intermediate Ranking Bolt, for instance, creates Rankables from incoming data tuples via a factory method of this class:

```
public void updateRanking(Tuple tuple) {
   Rankable r = RankableObjects.createRankable(tuple) ;
   ranking.updateWith(r);
}
```

Chapter 5

Performance Evaluation

In this chapter, we conduct extensive experiments to demonstrate the efficiency of *Modified Implementation* framework versus the standard one. The experiment is based on batch dataset . We will use batch dataset instead of continuous dataset from a stream as we use here Twitter API to get Top N words because it is easier to get actual results on batch data instead real datasets. More Formally , we will evaluate performance of modified and existing implementation on a batch of data and check if it works on real datasets with continuous manner or not.

5.1 Standard Implementation

As we mentioned before , The main purpose for thesis is to optimize queries over stream. The challenging part is in ranking top N elements with an efficient complexity versus naive implementation . First of all , we will begin with naive implementation in ranking top N Objects . then, we will compare it with efficient one.

Based on previous section about Ranking class , There is one method which can affect more significantly in time complexity which is *updateWith()* method . but , It differs in parameters of method . More Formally, There are two methods with same name but different in parameter type

```
public void updateWith(Rankable r);
```

This method is used in *Intermediate Bolt* . Basically, It is used to update object's count if it is already in *rankElements* list , Otherwise, Rankable Object will be added to rankElements list.

```
public void updateWith(Rankable r) // Worst Case Complexity(O(N))
{
   synchronized (rankElements)
   {
    Integer rank = rankElements.indexOf(r) ;          // O(N)
    if(rank == -1)
      rankElements.add(r) ;                          // O(1)
    else
      rankElements.set(rank , r) ;                   // O(1)
  }
}
```

We will analyze code fragment above. Searching for current Rankable object in *rankElements* list will take N times for worst case. If we start search for rankable object in index 0 and the current object is not presented in rankElements list .Therefore, *indexOf()* method will iterate N times. So, In worst case scenario will be **O(N)**.

```
public void updateWith(Ranking merge);
```

This method is used in *Total Ranking Bolt* . Basically, It is used to merge all Objects in *rankElements* list.

```
public void updateWith(Ranking merge) // Worst Case Complexity(O(M * N log N))
{
  for(Rankable r : merge.getRanking()) //O(M)
  {
    Integer rank = rankElements.indexOf(r) ;              // O(N)
    if(rank == -1)
      rankElements.add(r) ;                              // O(1)
    else
      rankElements.set(rank , r) ;                       // O(1)

    Collections.sort(rankElements, Collections.reverseOrder()); // O(N log N)

    if(rankElements.size() > topN)
      rankElements.remove(0) ;                            //O(N)

  }

}
```

We will analyze code fragment above. As we described before , *indexOf()* method take worst case *O(N)*. Moreover, we sort rankElements list based on object's count in reverse order. Collections class use merge sort algorithm which take asymptotically *O(N log N)*.Since, All this implementation M times where M is size of merge ranking list,Therefore , The worst case scenario will be **O(M * N log N)**.

5.2 Modified Implementation

In this section , we will provide a modified implementation on same two methods provided in *Standard Implementation.*

```
private final AVL_Tree<Rankable> rankElements ;
```

Basically , We will introduce a new data structure called *AVL Tree.* It is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes. Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree.

```
public void updateWith(Rankable r) // Worst Case Scenario O(log N)
{
   rankElements.add(r) ; //log(N)
}
```

As we state before , this method is used in Intermediate Ranking Bolt. and *add()* method take **O(log N)** operations as a worst case complexity.

```
private final PriorityQueue<Rankable> queue ;
```

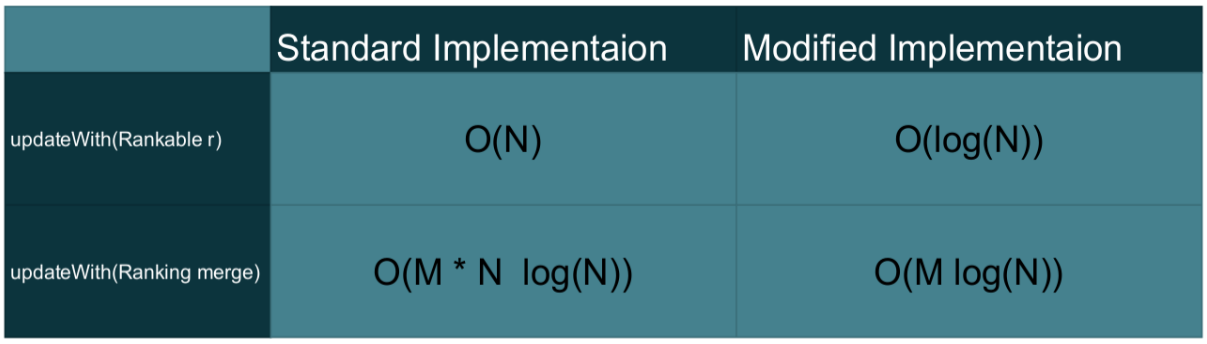
Moreover, There is another data structure called *Priority Queue.* It is used it merge all Rankable objects in *Total Ranking Bolt* using Priority queue.

```
public void updateWith(Ranking merge) //O(M log N)
{
   for(Rankable o : merge.rankElements) // O(M)
   {
      queue.add(o); // O(log N)
      if(queue.size() > topN)// O(1)
         queue.poll() ; // O(log N)
   }
}
```

We will analyze code fragment above. We will use priority queue in updateWith method which is used in *Total Ranking Bolt.* add() function takes *O(log N)* iterations over M objects. Therefore worst time complexity is **O(M * log(N))**.

5.3 Comparison between Standard and Modified Implementation

As we see the time complexity between Modified and Standard Implementation are significantly large.



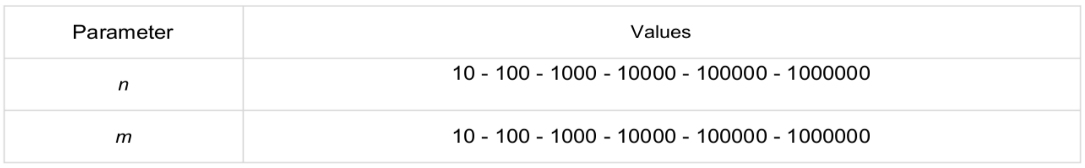
	Standard Implementaion	Modified Implementaion
updateWith(Rankable r)	O(N)	O(log(N))
updateWith(Ranking merge)	O(M * N log(N))	O(M log(N))

Figure 5.1: Comparison between Standard and Modified Implementation

5.4 Experiment Settings

In total , Only one dataset is used in our experiments.We will use in our experiment a batch of data with 1.6 million tweets with average 2.7 x 10^7 words. We use dataset from *Kaggle* website.All these tweets extracted using *Twitter API*.

In our study, we consider two parameters and one output variables, i.e., number of words m , and Top ranking size n, and the time consumed to represent top n words.



Parameter	Values
n	10 - 100 - 1000 - 10000 - 100000 - 1000000
m	10 - 100 - 1000 - 10000 - 100000 - 1000000

Figure 5.2: Parameter Settings

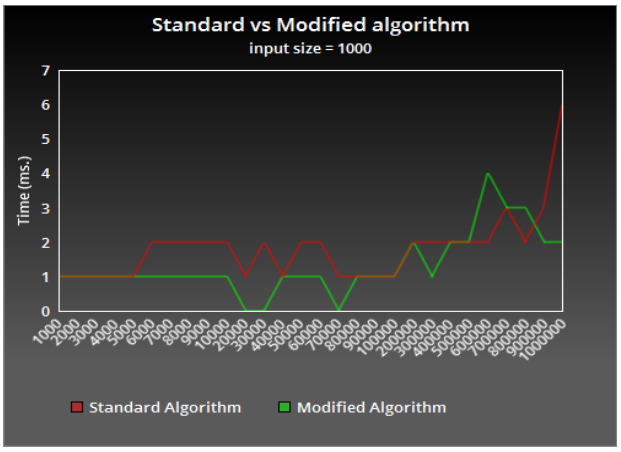
We will evaluate time consumption based on parameter settings in parameters in figure above.

All the algorithms are implemented with Java, and all the experiments are conducted on a CPU i7 with 16GB memory , running Ubuntu 18.04 Linux.

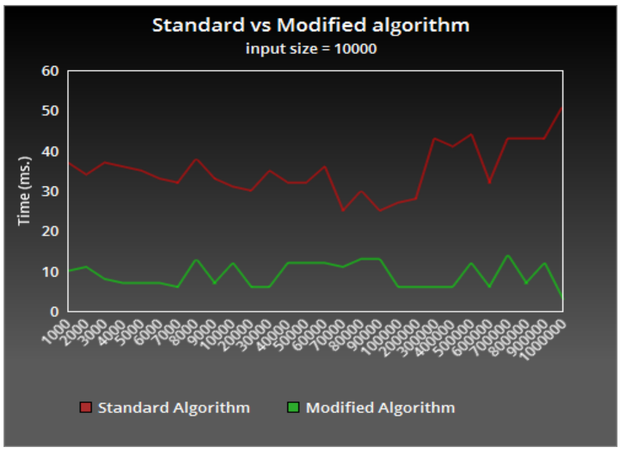
5.5 Comparison between Modified and existing algorithm

In this section we will evaluate the effectiveness of modified algorithm that was proposed in Section 4.5.

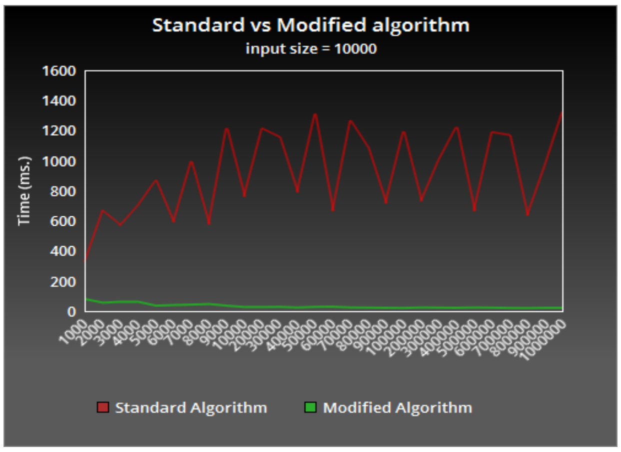
We report running time of top n words under number of input words m. We will represent six different charts . Every chart represent all different values of parameter m . Moreover, every chart represent comparison between top n ranking words and time consumed t.



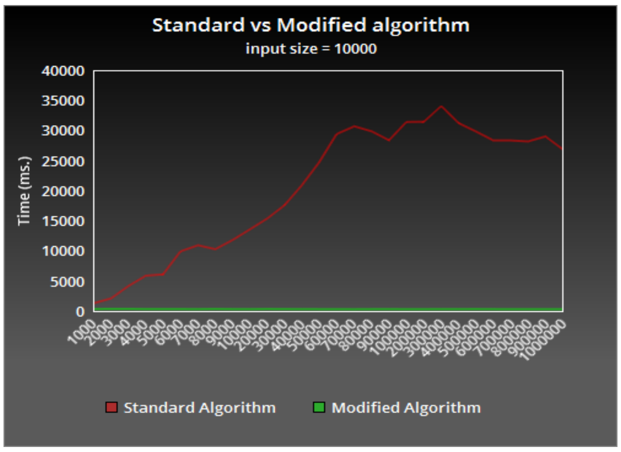
(a) input size 10^3



(b) input size 10^4



(c) input size 10^5



(d) input size 10^6

.

Based on the results above , we find that by increasing input size n . the growth of time consumption is significantly high in *Standard Algorithm* compared to *Modified Algorithm*. Moreover, Modified Algorithm is proved by experiment that by increasing input size. Time consumption will not exceed 1000 ms. (1 sec.) however , Standard Algorithm exceeds 0.5 minutes which is significantly huge consumption compared to Modified Algorithm.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this paper, we propose a novel and general idea about query processing , for supporting continuous top- k query over stream data. Different from all existing works , we used Apache Storm platform to apply continuous streaming on real dataset. Moreover , we optimize ranking and merging techniques using *data-structures* such as AVL-Tree (Self-Balanced Tree) and Priority queue. then , we compare standard algorithm with modified one.

We have conducted extensive experiments to evaluate the performance of Modified Algorithm on several datasets with different distributions. he results demonstrate the superior performance of *Modified Algorithm* compared to *Standard Algorithm*.

6.2 Future Work

There are a few features that could be added to optimize queries in the future, which could not be achieved due to the limited time cap of this paper.

As we mentioned in *Section 4.5*, We used AVL-Tree to update all object's count with Complexity O(log n). Basically, we can optimize updating part from O(log n) to O(1). Many people can suggest to use *HashTable* as in average complexity is O(1). Unfortunately, HashTable can take complexity O(N) in worst case.

Appendix

List of Figures

Bibliography

[1] Apache storm : Architecture overview , http://why-not-learn-something.blogspot.com/2016/01/apache-storm-architecture-overview.html.

[2] Apache storm : Grouping , https://www.simplilearn.com/apache-storm-advanced-concepts-tutorial-video.

[3] Apache storm : Interfaces , https://www.simplilearn.com/apache-storm-interfaces-tutorial-video.

[4] Apache storm : Rolling count , http://www.michael-noll.com/blog/2013/01/18/implementing-real-time-trending-topics-in-storm/.

[5] Apache storm: Tutorial , https://storm.apache.org/releases/current/tutorial.html.

[6] Continuous monitoring of top-k queries over sliding windows . kyriakos mouratidis , spiridon bakiras and dimitris papadias.

[7] Mayur Data Brian Babcock, Shivnath Babu. Models and issues in data stream systems, 2012.

[8] Sharma Chakravathy. Analysis and validation of queries over data streams.

[9] N. Koudas G. Das, D. Gunopulos and N. Sarkas. Ad-hoc top-k query answering for data streams.

[10] Vera Goebel and Thomas Plagemann. Data stream management systems âĂŞ a technology for network monitoring and traffic analysis? , 2005.

[11] Martin Hirzel. Stream query optimization.

[12] S. Bakiras K. Mouratidis and D. Papadias. Continuous monitoring of top-k queries over sliding windows.

[13] J. Yang G. Xia K. Yi, H. Yu and Y. Chen. Efficient maintenance of materialized top-k views.

[14] Lewis Kaneshiro. Karthik Ramasamy. Real-time analytics with apache storm udacity course https://www.udacity.com/course/real-time-analytics-with-apache-storm–ud381.

[15] Requeno and S. Bernardi J. Merseguer. Performance analysis of apache storm applications using stochastic petri nets , 2017.

[16] Rui Zhu ; Bin Wang ; Xiaochun Yang ; Baihua Zheng. Sap: Improving continuous top-k queries over streaming data, https://ieeexplore.ieee.org/document/7839230/ 2017.