A DISTRIBUTED APPROACH ON EARLY TRENDING TOPICS PREDICTION ON SOCIAL NETWORK SERVICES

MIA KATANEMHMENH PROSEGGISI GIA TIN EIGKAIROI PROBLEDIYI TON DHMOPHILON THEMATON STA MEZA KOINONIKHS DIKTYSHE

MASTER THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN INFORMATION SYSTEMS

2013–2014
THESALONIKI, GREECE

EDITED BY

PANOURGIA MARIA
AEM 469

KITMERIDIS NIKOLAOS
AEM 486

SUPERVISOR

PROF. VAKALI ATHENA

2014
ARISTOTLE UNIVERSITY OF THESSALONIKI
Abstract

It is indisputable that we have entered the Era of Big Data. Nowadays, the explosion and profusion of available data in a wide range of application domains is phenomenal. According to IBM, 2.5 billion GB of data are created every day. Ninety percent of the currently existing data, has been created in the past two years and the amount of data is expected to increase exponentially.

One of the key factors that lead to the boost of data volume is the widespread adoption of Web 2.0 tools and technologies. Web 2.0 applications may allow users to interact and collaborate with each other in a social media dialogue, as creators of user-generated content in a virtual community. Among the applications of Web 2.0, Social Media are of particular interest mainly because they are extremely popular among Web users.

Users of social media, through their vigorous activity, create massive volumes of data. This growing data collection, has attracted the interest of both the research community and industry, as performing mining and analysis on social media data may reveals several useful conclusions. The process of extracting and analyzing data from social networks is described by the term “Social Media Analytics”. This process which have various aspects, is in general particularly tedious as the data generated from social networks are characterized by very large volume and lack of structure.

This thesis is deepen into the area of “Social Network Analytics”. In particular, the problem of predicting, in real time, of popular topics (trending topics), appearing in social media is going to be studied. The popular topics of social networks are, by their nature, of utmost importance as they give immediate information about what individuals and the society are interested in, at a specific time period. Furthermore, popular topics, through the high visibility they obtain could affect public opinion, about what is considered trend or even their views on various social issues.

In this thesis the problem of popular topics prediction is approached from the perspective of time series real time classification. These time series represent the occurrence frequency of the under study topics in the messages published by social media users, in the axis of time. The time series classification is implemented through a supervised learning method based on a latent source model.

In order to perform and evaluate the aforementioned classification method, a distributed framework is going to be implemented so as to predict real time trends in Twitter. The main contribution of this thesis is that trend prediction is performed in real time fashion and thus several tools and techniques of the field of Big Data are used. In particular, the architecture of the implemented framework is inspired of the so called Lambda architecture. The main characteristic of Lambda architecture is that it enables stream and batch analysis to be performed simultaneously in the a common dataset. In this experimental scenario stream analysis is used in order to create, track and classify the topics’ time series. Whereas the batch analysis is used so as to create as well as to perform incremental improvement of the initial training set.
Είναι αναμφισβήτητο ότι έχουμε εισέλθει στην εποχή των Big Data. Σήμερα, η αφθονία των διαθέσιμων δεδομένων, που προέρχονται από ένα ευρύ φάσμα εφαρμογών, είναι πρωτοφανής. Σύμφωνα με την IBM 2.5 δισεκατομμύρια GB δεδομένων δημιουργούνται κάθε μέρα. Χαρακτηριστικό είναι το γεγονός ότι το 90% των δεδομένων που είναι διαθέσιμα σήμερα, έχει παραχθεί τα τελευταία δύο χρόνια και η ποσότητα των παραγόμενων δεδομένων αναμένεται να αυξηθεί εκθετικά.

Ένας από τους βασικούς παράγοντες που οδηγούν στην πρωτόγνωρη αύξηση του όγκου των δεδομένων είναι η ευρεία υιοθέτηση των εργαλείων και των τεχνολογιών του Web 2.0. Οι εφαρμογές του Web 2.0 έχουν ως στόχο να επιτρέπουν στους χρήστες τους να αλληλεπιδρούν, να συνεργάζονται και να ανταλλάξουν πληροφορίες μέσω του Διαδικτύου. Ανάμεσα στις εφαρμογές του Web 2.0 ιδιαίτερο ενδιαφέρον παρουσιάζουν τα Κοινωνικά Δίκτυα, κυρίως λόγω του ότι είναι εξαιρετικά δημοφιλή.

Οι χρήστες των μέσων κοινωνικής δικτύωσης, μέσω της έντονης δραστηριότητάς τους, δημιουργούν τεράστιο όγκο δεδομένων. Αυτή η συνεχώς αυξανόμενη αυλογιά δεδομένων, έχει προοπλέξει το ενδιαφέρον τόσο της ερευνητικής κοινότητας όσο και της βιομηχανίας, καθώς από την εξόρυξη και την ανάλυση των δεδομένων αυτών, προκύπτουν πολλά χρήσιμα συμπεράσματα. Η διαδικασία της εξόρυξης και της ανάλυσης δεδομένων που προέρχονται από κοινωνικά δίκτυα περιγράφεται με τον όρο “Social Network Analytics”. Η διαδικασία αυτή, η οποία εμπεριέχει διάφορες πτυχές, είναι γενικά ιδιαίτερα επίπονη καθώς τα δεδομένα που προέρχονται από κοινωνικά δίκτυα χαρακτηρίζονται από πολύ μεγάλο όγκο και έλλειψη δομής.

Η εργασία αυτή πρόκειται να εμβαθύνει στην περιοχή των “Social Network Analytics”. Ειδικότερα, πρόκειται να μελετηθεί το πρόβλημα της πρόβλεψης σε πραγματικό χρόνο, των δημοφιλών θεμάτων (trending topics), που εμφανίζονται στα μέσα κοινωνικής δικτύωσης. Τα δημοφιλή θέματα των κοινωνικών δικτύων είναι από τη φύση τους πολύ σημαντικά, καθώς δίνουν άμεσα μια εικόνα για τις προτιμήσεις των χρηστών και τη κοινωνία και με της εξόρυξης και της ανάλυσής τους, αναπαριστούν τη συχνότητα εμφάνισης των υπό μελέτη θεμάτων, στα μηνύματα που δημοσιεύουν οι χρήστες των κοινωνικών δικτύων, στον άξονα του χρόνου. Για την ταξινόμηση των δημοφιλών θεμάτων εφαρμόστηκε ένα μοντέλο επιβλεπόμενης μάθησης που βασίζεται στο μοντέλο των λανθανουσών πηγών (“latent source model”).

Προκειμένου να αξιολογηθεί η παραπάνω μέθοδος ταξινόμησης, σχεδιάστηκε και εκτελέστηκε ένα κατανεμημένο πλαίσιο μέσα στο οποίο γίνεται η εκτέλεση της προβλέψης της εκτύπωσης των υπό μελέτη θεμάτων, στα μηνύματα που δημοσιεύουν οι χρήστες των κοινωνικών δικτύων, στον άξονα του χρόνου. Η διαδικασία της ταξινόμησης των δημοφιλών θεμάτων προετοιμάζεται σε πραγματικό χρόνο. Οι χρονοσειρές αυτές αναπαριστούν τη συχνότητα εμφάνισης των υπό μελέτη θεμάτων, στα μηνύματα που δημοσιεύουν οι χρήστες των κοινωνικών δικτύων, στον άξονα του χρόνου. Η διαδικασία της εκτύπωσης των υπό μελέτη θεμάτων προετοιμάζεται σε πραγματικό χρόνο. Η διαδικασία της ταξινόμησης των δημοφιλών θεμάτων προσεγγίζεται από την σκοπιά της κατηγοριοποίησης χρονοσειρών, σε πραγματικό χρόνο. Οι χρονοσειρές αυτές αναπαριστούν τη συχνότητα εμφάνισης των υπό μελέτη θεμάτων, στα μηνύματα που δημοσιεύουν οι χρήστες των κοινωνικών δικτύων, στον άξονα του χρόνου. Για την ταξινόμηση των δημοφιλών θεμάτων εφαρμόστηκε ένα μοντέλο επιβλεπόμενης μάθησης που βασίζεται στο μοντέλο των λανθανουσών πηγών ("latent source model").
κατανεμημένη αρχιτεκτονική πολλών επιπέδων. Το κύριο χαρακτηριστικό της αρχιτεκτονικής αυτής είναι ότι επιτρέπει την ταυτόχρονη εφαρμογή “stream analysis” και “batch analysis” σε ένα κοινό σύνολο δεδομένων. Στην συγκεκριμένη εφαρμογή η “stream analysis” συνεισφέρει στη δημιουργία, την ανανέωση και την ταξινόμηση των χρονοσειρών που αντιστοιχούν στα πιθανά δημοφιλή θέματα. Ενώ η “batch analysis” συνεισφέρει τόσο στη δημιουργία όσο και στην σταδιακή βελτίωση του αρχικού συνόλου εκπαίδευσης. Η αξιολόγηση των αποτελεσμάτων της μεθόδου, δηλαδή των δημοφιλών θεμάτων που εμφανίζονται στο κοινωνικό δίκτυο Twitter, γίνεται με βάση τα δημοφιλή θέματα που αναρτά το ίδιο το δίκτυο ανά τακτά χρονικά διαστήματα.
Acknowledgements

We would like to sincerely thank all those who contributed to the completion of this master thesis.

First of all, we would especially like to thank our thesis supervisor, Professor Athena Vakali, for her continuous support and encouragement. Her advice and guidance were essential for the completion of this thesis. Also, we would like to thank the other two members of the examination committee Associate Professor Eleftherios Angelis and Assistant Professor Apostolos Papadopoulos for their participation in the evaluation of our thesis.

Finally, we would like to express our thanks to our friends and families for their understanding and unconditional support.
Contents

1 Introduction 1
  1.1 Big Data ..................................................... 2
  1.2 Web 2.0 as Big Data sources ................................ 4
  1.3 Social Networking Services and Analysis ..................... 5
  1.4 Thesis Structure and contribution ........................... 5

2 Big Data and Social Media Fundamentals 7
  2.1 Big Data Distributed Frameworks and Architectures .......... 7
    2.1.1 Distributed Computing Frameworks ....................... 7
    2.1.1.1 Apache Hadoop ....................................... 7
          History .................................................. 7
          Architecture .......................................... 9
          NameNode .............................................. 9
          DataNode ............................................. 9
          Secondary NameNode (SNN) ............................ 10
          JobTracker ........................................... 11
          TaskTracker ........................................ 11
          The MapReduce Model .................................. 11
    2.1.1.2 Apache Drill ......................................... 13
          History ................................................ 13
          Architecture ......................................... 13
          Tree Architecture ....................................... 14
          Query dispatcher ....................................... 14
    2.1.1.3 Apache Storm .......................................... 14
          History ................................................ 14
          Architecture ......................................... 15
          Nimbus ............................................... 15
          Supervisor .......................................... 15
          Topologies .......................................... 15
          Streams ........................................... 16
          Data model ......................................... 16
    2.1.1.4 Apache S4 ............................................. 16
          History ................................................ 16
          Architecture ......................................... 17
    2.1.2 Coordination Systems .................................... 17
      2.1.2.1 Apache Zookeeper .................................. 18
          History ................................................ 18
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.2.2 Apache Oozie</td>
<td>18</td>
</tr>
<tr>
<td>History</td>
<td>18</td>
</tr>
<tr>
<td>Architecture</td>
<td>19</td>
</tr>
<tr>
<td>Control flow nodes</td>
<td>19</td>
</tr>
<tr>
<td>Action nodes</td>
<td>19</td>
</tr>
<tr>
<td>2.1.3 Distributed NoSQL Databases</td>
<td>19</td>
</tr>
<tr>
<td>2.1.3.1 Column Oriented Databases</td>
<td>21</td>
</tr>
<tr>
<td>Apache HBase</td>
<td>21</td>
</tr>
<tr>
<td>History</td>
<td>21</td>
</tr>
<tr>
<td>Architecture</td>
<td>21</td>
</tr>
<tr>
<td>Data Model</td>
<td>22</td>
</tr>
<tr>
<td>Apache Cassandra</td>
<td>23</td>
</tr>
<tr>
<td>History</td>
<td>23</td>
</tr>
<tr>
<td>Architecture</td>
<td>23</td>
</tr>
<tr>
<td>Data model</td>
<td>23</td>
</tr>
<tr>
<td>2.1.3.2 Document Oriented Databases</td>
<td>23</td>
</tr>
<tr>
<td>MongoDB</td>
<td>23</td>
</tr>
<tr>
<td>History</td>
<td>24</td>
</tr>
<tr>
<td>Data Model</td>
<td>24</td>
</tr>
<tr>
<td>Architecture</td>
<td>24</td>
</tr>
<tr>
<td>Apache CouchDB</td>
<td>25</td>
</tr>
<tr>
<td>History</td>
<td>25</td>
</tr>
<tr>
<td>Data Model</td>
<td>26</td>
</tr>
<tr>
<td>Architecture</td>
<td>26</td>
</tr>
<tr>
<td>2.1.3.3 Key-Value datastores</td>
<td>27</td>
</tr>
<tr>
<td>Redis</td>
<td>28</td>
</tr>
<tr>
<td>History</td>
<td>28</td>
</tr>
<tr>
<td>Data Model</td>
<td>28</td>
</tr>
<tr>
<td>Architecture</td>
<td>29</td>
</tr>
<tr>
<td>ElephantDB</td>
<td>29</td>
</tr>
<tr>
<td>History</td>
<td>29</td>
</tr>
<tr>
<td>Data Model</td>
<td>29</td>
</tr>
<tr>
<td>Architecture</td>
<td>29</td>
</tr>
<tr>
<td>2.1.3.4 Graph Oriented Databases</td>
<td>30</td>
</tr>
<tr>
<td>Neo4j</td>
<td>31</td>
</tr>
<tr>
<td>History</td>
<td>31</td>
</tr>
<tr>
<td>Data Model</td>
<td>31</td>
</tr>
<tr>
<td>Architecture</td>
<td>31</td>
</tr>
<tr>
<td>2.1.4 Big Data Query Processing</td>
<td>31</td>
</tr>
<tr>
<td>2.1.4.1 Apache Hive</td>
<td>32</td>
</tr>
<tr>
<td>History</td>
<td>32</td>
</tr>
<tr>
<td>Data Model</td>
<td>32</td>
</tr>
<tr>
<td>Architecture</td>
<td>32</td>
</tr>
<tr>
<td>Query Language</td>
<td>33</td>
</tr>
<tr>
<td>2.1.4.2 Apache Pig</td>
<td>33</td>
</tr>
<tr>
<td>History</td>
<td>33</td>
</tr>
<tr>
<td>Data Model</td>
<td>34</td>
</tr>
</tbody>
</table>
Architecture .................................................. 35
2.1.4.3 Cascading ............................................. 36
   History ..................................................... 36
   Data Model ............................................... 36
   Architecture ............................................. 37
2.1.4.4 Cascalog ............................................ 39
2.1.5 Data Serialization Frameworks ......................... 40
   2.1.5.1 Apache Thrift .................................... 42
      Types ................................................... 42
      Transport ............................................. 42
      Protocol .............................................. 42
      Versioning ............................................ 42
      Processors ............................................ 43
   2.1.5.2 Dfs-datastores and Thrift ......................... 43
2.1.6 The Lambda Architecture ................................ 44
   Step 1: Data streams .................................... 44
   Step 2: The batch layer .................................. 45
   Step 3: The serving layer ................................ 45
   Step 4: The speed layer .................................. 45
   Step 5: Quering views .................................... 46
2.2 Social Network Analytics .................................. 46
   2.2.1 Related Work on Trend Prediction and Time Series Classification .......................... 47
3 Prerequisites .................................................. 49
   3.1 Twitter Network Service ................................ 49
   3.2 Trends Prediction on Twitter ............................ 50
4 Twitter trending topics prediction framework .............. 52
   4.1 Infrastructure .......................................... 52
      4.1.1 Hadoop cluster ..................................... 52
      4.1.2 Storm cluster ...................................... 54
      4.1.3 Redis server ....................................... 54
   4.2 The datasets ........................................... 54
      Tweets sample .......................................... 55
      Twitter trends ......................................... 56
   4.2.1 The training set ..................................... 56
   4.2.2 Data for real time execution and evaluation ................. 56
4.3 The implemented framework ................................ 57
   4.3.1 The generation of the training set ...................... 57
      Step 1 ................................................... 57
      Step 2 ................................................... 59
      Step 3 ................................................... 61
      Step 4 ................................................... 62
   4.3.2 Real time execution ................................ 62
      Step 5 ................................................... 63
      Step 6 ................................................... 64
      Step 7a ................................................. 65
      Step 8a ................................................. 68
4.3.3 The training set periodic update ........................................ 68
    Step 7b .............................................................................. 68
    Step 8b .............................................................................. 69
    Step 8c .............................................................................. 69

5 Experimentation and Results .................................................. 70
    5.1 Comparison of the distance metrics performance ............ 70
    5.2 Early trending topic prediction performance ............... 71
        5.2.1 Discussion on the Results ....................................... 75

6 Discussion on Problems and Future work ......................... 77
    6.1 Implementation problems ............................................. 77
    6.2 Future work ............................................................... 78

Appendices ............................................................................ 81

A Source code samples .......................................................... 82
    A.1 Thrift file ...................................................................... 82
    A.2 Pail structure .............................................................. 83
    A.3 Full outer join performed using JCasclog .................... 86
List of Figures

1.1 Data volume growth. ............................................. 2
2.1 Typical example of the Hadoop Cluster architecture. ........ 10
2.2 Snapshot of interaction between NameNode and DataNodes (Storage daemons) ........ 10
2.3 Snapshot of interaction between JobTracker and TaskTrackers (Computation daemons) .... 11
2.4 The flow and the pseudocode of the MapReduce Word Count Example. .............. 12
2.5 Two sample nested records and their schema .................... 13
2.6 System architecture and execution inside a server node .................. 14
2.7 Storm Architecture ............................................ 15
2.8 Storm Topology ................................................ 16
2.9 S4’s Processing Node. ......................................... 17
2.10 Word Count Example in S4. .................................. 18
2.11 Abstract Implementation of Zookeeper Architecture ............... 19
2.12 Historical trend of the categories’ popularity. For each month the best three systems per category are chosen and the average of their ranking scores is calculated. .......... 20
2.13 HBase Abstract Architecture Schema .......................... 22
2.14 MongoDB in BSON format .................................... 24
2.15 An appserver running a mongos. ................................ 25
2.16 MongoDB mapReduce command and execution diagram ........... 26
2.17 JSON format of a simple document in CouchDB .................... 27
2.18 A typical deployment diagram with multilingual clients. ............. 29
2.19 Example of a property graph in Graph Databases. ................ 30
2.20 Abstract instances of Neo4j cluster. ............................ 31
2.21 Hive Architecture. .............................................. 33
2.22 Data Types of Pig Latin ....................................... 34
2.23 Pig compilation and execution stages. .......................... 35
2.24 Cascading technology stack. ................................... 36
2.25 Cascading data representation through a simple example. .......... 36
2.26 Cascading’s Pipe Assemblies class hierarchy. ..................... 38
2.27 Cascading’s Flow of the “Word Count” example. ................ 39
2.28 A simple Cascalog query that prints the names of people from table AGE that are younger than 30 years old. .................................................. 40
2.29 Word count execution comparison in classic MapReduce model and Cascalog. ........ 41
2.30 Pail-Thrift simple example. .................................... 44
2.31 The Lambda Architecture. ...................................... 45
3.1 Top-10 global trends instance. .................................. 51
3.2 Top-10 location-based trends instance. .......................... 51
4.1 Twitter trend prediction framework. ................................................. 53
4.2 Tree structure of Thrift objects storage in HDFS ..................................... 59
4.3 Trending topic’s time series creation example. ........................................ 63
4.4 Topic’s time series creation and update example. ..................................... 66

5.1 Comparison of the method’s performance using the Cosine and Squared Euclidean distance metrics, regarding the percentage of the true positives per hour of execution. ......................... 71
5.2 The respective percentages of true positives in the end of the 48 hour execution, for Cosine and Squared Euclidean distance metrics. ................................................. 71
5.3 The respective percentages and mean times of true positive trending topics reported by the framework before and after the Twitter. ................................................. 72
5.4 Cosine: Total performance of the method’s execution regarding the trending topics that reported by the framework before and after they reported by Twitter. ......................... 74
5.5 Squared Euclidean: Total performance of the method’s execution regarding the trending topics that reported by the framework before and after they reported by Twitter. ......................... 75
List of Tables

2.1 A categorization of tools used in the field of Big Data analytics. .......................... 8
4.1 Explanation of symbols used in equation 4.7 .......................................................... 68
5.1 The total percentage of true positives for the 12th, 24th, 36th and 48th hour of method’s
execution using the Cosine and Squared Euclidean distance metrics. ......................... 73
5.2 Comparison of results of the implemented method used in real time analysis (current thesis)
and in retrospective analysis ([1]) ................................................................. 76
Chapter 1

Introduction

Information technology (IT) is the application of computers and telecommunications equipment to store, retrieve, transmit and manipulate data [2], (both in the context of academia and industry). The term of Information Technology is directly connected with Computer Science and thus has its origins in the middle 1940’s. From then until today, the scientific discoveries in these areas have drastically changed both people’s daily lives and industry. According to the reliable American business magazine “Forbes” [3], there are three phases which determined the evolution of IT science and industry.

The first phase of IT started with the invention of the first giant calculators, which were able to digitally processing and manipulating numbers. Thereafter, IT expanded to the digitization of transaction-oriented activities, such as airline reservations and banking transactions. This first phase of IT revolution, which eventually lasted until the 1980s, could be characterized by computer-related activities revolved around interactions between a person and a computer. This fact did not change even when the first PCs arrived on the scene in late 1970s. At that time, the PC was simply a mainframe on the desk. Undeniably, PC gave a tremendous boost in personal productivity applications that in turn contributed greatly to the growth of enterprise data. Although its use was limited to the digitization of leisure-related, home-based activities.

The second phase of IT revolution is signaled by the development of Local Local Area Networks (LANs) and Wide Area Networks (WANs) thereafter. Connecting people in a vast and distributed network of computers not only increased the amount of data generated but also led to numerous new ways of getting value out of it, unleashing many new enterprise applications and a new passion for “data mining”. Furthermore, the previous described changes, introduced tremendous changes in the market of IT service providers. In particular, the IT business which was focused on one IT component began to prevail against them who had adopted the vertical, “end-to-end solution” business model that has dominated the industry until then. Uproariously examples of this generation companies are Intel¹ in semiconductors, Microsoft² in operating systems, Oracle³ in databases, Cisco⁴ in networking and Dell⁵ in PCs.

The third phase in the evolution of the IT industry came with the invention of the World Wide Web. This invention led to the proliferation of new applications which were no longer limited to enterprise-related activities but digitized almost any activity in people’s lives. Moreover, many tools have emerged that greatly facilitated the creation and sharing of information by anyone with access to the Internet. We could argue that, while computer networks took IT from the accounting department to all corners of the enterprise, the World Wide Web took IT to all corners of the globe, connecting millions of people. These millions of people share information between themselves using WEB 2.0 applications increasing drastically the amount of data created,

¹http://www.intel.com/
²http://www.microsoft.com/
³http://www.oracle.com/
⁴http://www.cisco.com/
⁵http://www.oracle.com/
stored, moved and consumed. Inevitably, just as in the previous phase, a bunch of new IT companies emerged, all of them born on the Web and all of them regarding “IT” not as specific function responsible for running the infrastructure but as the essence of their business, data and its analysis becoming their competitive edge.

Nowadays, we are witnessing this third phase of IT evolution. The quantitative and qualitative leap in the growth of data is now more acute than ever. Everyday new technologies that are trying to manage complex and voluminous data, make their appearance as well as new techniques are invented so as to mine and extract valuable knowledge and information from the countless sources of data. This fact has led the scientific community to name this phase as “Big Data Era”.

1.1 Big Data

It is indisputable that we have entered the Era of Big Data. Nowadays, the explosion and profusion of available data in a wide range of application domains is phenomenal. According to IBM [4], 2.5 billion GB of data are created every day. Ninety percent of the currently existing data, has been created in the past two years and the amount of data is expected to increase exponentially. The data which are created, is expanding rapidly as enterprises capture more data in greater detail, as multimedia becomes more common, as social media conversations explode and as we use the Internet to get things done.

![Figure 1.1: Data volume growth.](image)

Does the term Big Data eventually related only with the volume of data? According to some of the most cited definitions, volume is one but not the only factor that characterize Big Data. In particular, Gartner [5] proposed a four fold definition based on the “four V’s”:

- **Volume** for the increasing size of data
- **Velocity** for the increasing rate at which data is produced
- **Variety** for the the increasing range of data formats and representations employed
- **Veracity** which includes questions of trust and uncertainty with regards to data and the outcome of analysis of them.
From another point of view Oracle’s definition [6] of Big Data is focused upon infrastructure. More precisely Oracle views big data like the derivation of value from traditional relational database driven business decision making, augmented with new sources of unstructured data. Another well known organization Intel, relates big data to operations "generating a median of 300 terabytes (TB) of data weekly" [7], quantifying in this manner the experiences of its business partners. Microsoft [8] links directly Big Data with the demand for significant compute power as well as with machine learning and artificial intelligence applications.

From a completely different perspective, Method for an Integrated Knowledge Environment (MIKE2.0) project relates the notion of Big Data purely upon complexity meaning that the high degree of permutations and interactions within a dataset classifies that as big data [9]. According to NIST’s definition, data sets that exceed the capacity or capability of current or conventional methods and systems, could be referred as Big Data [10]. Last but not least, J. Ward et all [11], in their comparative study on the Big Data definitions, propose their own definition, according to which “Big data is a term describing storage and analysis of large and/or complex data sets using a series of techniques including, but not limited to: NoSQL, MapReduce and machine learning”.

Regardless which is the most proper definition of Big Data, the ground truth is that their proper management has become an urgent need in a large variety of fields both in academia and industry. According to Wikipedia¹, some of the important scientific and business areas, that rely on Big Data analysis include, RFID, sensor networks, social networks, big social data analysis, Internet documents, Internet search indexing, call detail records, astronomy, atmospheric science, genomics, biogeochemical, biological, and other complex and often interdisciplinat scientific research, military surveillance, forecasting drive times for new home buyers, medical records, photography archives, video archives, and large-scale e-commerce.

This multi-field adoption of Big Data in conjunction with the increased demands in computational power and storage, for performing proper and efficient analysis on them, resulted to the generation of a considerable number of new technologies. The key factor in successful Big Data analysis is the transportation of the problem from a single machine to a distributed solution, meaning clusters of computer machines that work in parallel, each one contributing in the resolving of a part of the problem. In this direction, MapReduce, a technique first presented by Google [12], today is the most widely used, mainly through its open source implementation Apache Hadoop². Furthermore, except from being a platform that implements MapReduce technique, Hadoop offers a solution in the Big data’s storage problem, through the Hadoop Distributed File System (HDFS)³. HDFS complements the Apache Hadoop framework, and may regarded as the most famous among a variety of distributed file systems introduced for this specific scope.

Indisputably, the aforementioned technologies implemented by Apache Hadoop have contributed in a great manner to the problem of Big Data Analysis. However, storing Big Data in HDFS in a raw form is not always an acceptable solution. On the other hand, the traditional relational databases fail to manipulate the volume and complexity of this kind of data [13]. To face this problem, academic and industry research communities have developed a number of new type data stores, the so called NoSQL Databases. Those databases are, in most cases, compatible with HDFS and other distributed file systems either natively or via third party libraries, which means that they can store their records in different physical machines, which allows efficient storing and even execution of parallel queries on Big Data.

All of this technologies have managed to offer solutions on most of the current problems that are related to Big Data, as they constitute an environment which allows batch analysis on this kind of data. However, as Big Data evolves, new needs are generated. Nowadays, there is a variety of problems which this kind of batch analysis is not adequate. The reason is simple, despite its distributed nature, batch analysis is not fast enough to manipulate the data that are related to these problems because the analysis must precede the storage. Essentially the category of this problems is related to large data streams in which Real Time or Near Real Time⁴ analysis

¹http://en.wikipedia.org/wiki/Big_data
²http://hadoop.apache.org/
⁴http://en.wikipedia.org/wiki/Near_real-time
is crucial. Lately, new techniques and frameworks are being developed to face this category of problems. The most notable of them has been developed from Twitter\(^5\) and is called Storm\(^6\). Storm accepts as input large data streams from any source and distributes the data in a set of computer nodes to perform parallel analysis on them with no need of an intermediate storage file system. Due to the way that Storm operates, meaning the distribution of data on the nodes of a cluster, it is often, unofficially referred as “Hadoop for real time analysis”.

1.2 Web 2.0 as Big Data sources

One of the key factors that lead to the boost of data volume is the widespread adoption of Web 2.0 tools and technologies. The term Web 2.0 was coined in 1999 by Darcy DiNucci and was popularized by Tim O’Reilly at the O’Reilly Media Web 2.0 conference in late 2004\(^{[14]}\)\(^{[15]}\). Web 2.0 describes a second generation of the World Wide Web that is focused on the ability of people to collaborate and share information online. Web 2.0 basically refers to the transition from static HTML Web pages to a more dynamic Web that is more organized and is based on serving Web applications to users. In particular, a Web 2.0 site may allow users to interact and collaborate with each other in a social media dialogue, as creators of user-generated content in a virtual community, in contrast to websites where people are limited to the passive viewing of content. Examples of Web 2.0 include:

- **Social Networking Services (SNS):** are platforms which enable users, who share common interests, activities, backgrounds, or real-life connections, to build social networks or social relations. Usually, a social network service consists of a profile of each user, his/her social links and a number of additional services.

- **Blogs:** are discussion or informational sites published on the World Wide Web and consisting of discrete entries called “posts”. Stated another way, a blog is a sort of publicly accessible personal journal, reflecting the views and ideas of its author about specific topics, e.g. news items, reviews on several topics, share of knowledge etc. Most blogs platforms allow visitors to interact and communicate with the author by commenting in the end of posts.

- **Wikis:** are collaborative websites comprise the perpetual collective work of many authors. While wikis are similar to blogs in structure and logic, a wiki defers due to the fact that the content is created without any defined owner or leader. In addition, wikis have little implicit structure, allowing structure to emerge according to the needs of the users. The content of wikis varies from general knowledge base, e.g Wikipedia\(^1\) to specific knowledge base, e.g Ganfyd\(^2\), an online collaborative medical reference.

- **Folksonomies or Social Tagging Systems:** are user-defined metadata collections. A folksonomy evolves when many users create or store content at particular sites and identify what they think the content is about. There are two types of folksonomies broad and narrow. A broad folksonomy is one in which multiple users tag particular content with a variety of terms from a variety of vocabularies. A narrow folksonomy, on the other hand, occurs when a few users, primarily the content creator, tag an object with a limited number of terms \(^{[16]}\). Folksonomies are regarded as a fine example of collective intelligence as they enable the searchability of content by adding textual description as well as the tracking of emerging trends in tag usage and developing vocabularies.

- **Media Sharing Sites:** are platforms that enables users to store and share their multimedia files, e.g. photos, videos, music, with other users. Media sharing sites are often freemium based, i.e providing for free a

\(^{5}\)https://twitter.com/
\(^{6}\)http://storm-project.net/
\(^{1}\)http://www.wikipedia.org/
\(^{2}\)http://www.ganfyd.org/
set of basic services while they provide paid subscriptions for accessing a greater level of services. The media is played/viewed from any Web browser and may be selectively available via password to the general public. A media sharing site can also be used to back up files.

- **Mashups**: are web sites or web applications, that use content from more than one source to create a single new service displayed in a single graphical interface. The term implies easy, fast integration, frequently using open application programming interfaces (API) and data sources to produce enriched results. The main characteristics of a mashup are combination, visualization, and aggregation.

### 1.3 Social Networking Services and Analysis

Nowadays, Social Networking is gaining the keen attention of users. According to research contacted by The Pew Research Center’s Internet & American Life Project, today, 72% of online adults use social networking sites and this number grows to 89% for adults between 18-29 years old [17] [18]. Also it can not be disputed that Social Media has changed radically the way individuals communicate, collaborate, consume and create information. Information can published by users of Social Networking Services in the form of tweets, blog posts, or Web documents contributing to an exponentially growing data deluge. Furthermore except from raw data Social Networking contributes a new type of information related to the connectivity of users. Thus Social Networking Services“products” are not only of utmost importance from the scope of data generation and information retrieval but also reveal countless opportunities for the comprehension of the phenomenon of **Collective Intelligence**. Collective Intelligence describes the intelligence emerging from the interaction between interconnected, communicating individuals. The central idea of Collective Intelligence could be made more understandable by the following question, formulated by Tom Malone professor at the Massachusetts Institute of Technology [19]:

> “How can people and computers be connected so that collectively they act more intelligently than any individuals, groups, or computers have ever done before?”

As we mention before, SNS users intense activity entails to the creation of a massive volume of data. This constantly growing collection of data has attracted the interest both of the research community and industry, as mining and performing analysis on this data leads in a variation of useful outcomes. Many companies and organizations have turned their interest to the social media data so as to conduct their researches on them. Some features examples of researches conducted in social media data include prediction and tracking of disease outbreaks [20], polling of political events [21] and product market adoption and longevity [22] to name a few.

Nevertheless, drawing useful conclusions from the social media data, is a difficult process. Social media data usually have very large volume and are generally unstructured. In addition, the messages are exchanged between users on social networks are usually small, with inadequate syntax, and containing slag expressions, therefore it is difficult to implement traditional Natural Language Possessing (NLP) techniques on them. Thus, data mining on social media streams usually requires expertise from the field of Big Data as well as the employing of several heuristics methods.

### 1.4 Thesis Structure and contribution

In this thesis we will try to address the problem of on-line prediction of popular topics, trends, in the social media stream. Namely, we will handle the problem of on-line trend prediction as on-line time series classification. Thus, we have chosen to implement a setting for model specification and selection in supervised learning based on a latent source model, described by Chen et al. [23]. In the latent source model, is posited the existence of a set of latent source signals, each one corresponding to a prototypical event of a certain type. Moreover, it is
assumed that each observed signal is a noisy observation of one of the latent signals. In a classification scenario, the observed signals are compared to a number of sets of reference signals, each one consisting of examples of a specific category. In essence, these categories are the desired clusters, based on which every new signal will be classified. In our case, for example there are two categories/clusters, one contains topics which became trends and one with topics that did not become trends. It is posited that the observation signal belongs to a specific cluster if it was generated by the same latent source as one of the examples this cluster contains. In practice, to determine how likely it is that an observation belongs to a certain cluster, simply the the similarity between the observation signal and the reference signals of this cluster are computed. In this work, we chose to measure the similarity between two signals, using Euclidean Squared distance\(^1\) and Cosine distance\(^2\). The main advantage of the latent source model is that allows to infer the clusters in a non-parametric fashion directly from the data without specifying any specific model structure.

In order to evaluate the aforementioned method we are going to implement it so as to predict real time trends in Twitter. The main contribution of this thesis is that trend prediction is performed in real time fashion and for this purpose several tools and techniques of the field of Big Data will be used. In particular, we will be implement a framework which architecture is inspired of the so called Lambda architecture, which is analytically described in the following chapter. In addition, in our framework we will perform incremental improvement of the initial training dataset. Employing all the above, we achieve to predict the trending topics on Twitter with accuracy that reaches 78.4%, where in more than the 1/3 of the cases this prediction is taking place, with a mean time of 240 minutes, in advance of the respective Twitter reports.

This thesis is organized as follows. Next, we will present the dominant frameworks from the field of Big Data as well as recent related work from the fields of trend prediction and time series classification. Following, we will make a short presentation of Twitter as well as the mechanism it track the trend topics. Consequently, we will analyse in full detail our framework and the way it was implemented. Next, we will introduce the results obtained from the application of the method in our framework and finally, the conclusions that came up, the problems that met up and the future work that is planned to be done based on the current implementation, are presented in the last chapter of this thesis.

---

\(^1\)http://en.wikipedia.org/wiki/Euclidean_distance
\(^2\)http://en.wikipedia.org/wiki/Cosine_similarity
Chapter 2

Big Data and Social Media Fundamentals

This chapter introduces a theoretical approach to the issues dealt with in the current thesis. Based on the implementation part of this thesis - where a distributed architecture has been implemented to solve a problem from the field of Social Network Analytics, specifically the real time prediction of trending topics discussed in Social Networks - two theoretical points could be distinguished. Thus, this chapter is divided in two sections. The first concerns a thorough presentation of the widely tools and frameworks that contribute in the synthesis of complex distributed architectures. The second section concerns a presentation of the basic notions and techniques of the field of the Social Network Analytics, as well as a bibliographic report on the dominant techniques to the specific issues of trending topics prediction on Social Networks and the time series classification.

2.1 Big Data Distributed Frameworks and Architectures

In Section 1.1 an extensive report on the reasons that lead to the creation of the science sector, referred to as “Big Data Analytics”, and its relationship with distributed computing techniques has been discussed, as well as a brief report of the dominant tools used in this scientific field has been presented. In Table 2.1 the most notable tools of this field are presented with information relative to the category each one belongs, the code availability policy, whether is further presented in this chapter and whether has been used in the implementation part of the current thesis. From the table it is obvious that the open source community has contribute greatly in the development of tools for distributed Big Data processing, while companies seems to support those efforts as in many cases they offer the source code of their own implemented tools. A further overview of the most widely used tools, follows in the next sections.

2.1.1 Distributed Computing Frameworks

This section presents the frameworks, considered as milestones, in the field of Big Data Analytics. These frameworks are divided into two main categories. The first category concerns the batch analysis of “Big Data” and thus it is focused on handling issues emerging from the “Volume” characteristic of “Big Data”. On the other hand, the second category aims to handle issues related with the near real- time analysis of “Big Data”. This type of frameworks is mainly related to the “Velocity” characteristic of “Big Data”.

2.1.1.1 Apache Hadoop

History
Apache Hadoop is an open source framework, written in Java, which is specialized in implementation of distributed applications that manage large volumes of data. Initially, it was not a stand-alone project but was
<table>
<thead>
<tr>
<th>Framework</th>
<th>Category</th>
<th>Subcategory</th>
<th>Policy</th>
<th>Further Presented</th>
<th>Used in Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Hadoop</td>
<td>Distributed Computing</td>
<td>Batch Analysis</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Apache Spark</td>
<td>Distributed Computing</td>
<td>Real Time Analysis</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Apache Drill</td>
<td>Distributed Computing</td>
<td>Cluster Coordinator</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Google Dremel</td>
<td>Distributed Computing</td>
<td>Workflow Coordinator</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Apache Storm</td>
<td>Distributed Computing</td>
<td>Coordination Systems</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Apache Zookeeper</td>
<td>Distributed Computing</td>
<td>Column Oriented</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Apache HBase</td>
<td>Distributed Computing</td>
<td>Document Oriented</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Apache Cassandra</td>
<td>Distributed Computing</td>
<td>Key-Value store</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MongoDB</td>
<td>Distributed NoSQL Databases</td>
<td>Graph Oriented</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Couchbase</td>
<td>Distributed NoSQL Databases</td>
<td>Distributed Query Engine &amp; SQL-like Language</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Amazon DynamoDB</td>
<td>Distributed NoSQL Databases</td>
<td>DSL Library for Distributed Query Processing</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ElephantDB</td>
<td>Distributed NoSQL Databases</td>
<td>Remote Procedure Call Frameworks</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Redis</td>
<td>Key-Value store</td>
<td>Data Serialization Frameworks</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Memcached</td>
<td>Key-Value store</td>
<td>Apache Avro</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Riak</td>
<td>Key-Value store</td>
<td>Google Protocol Buffers</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Apache Hive</td>
<td>Distributed Query Engine &amp; SQL-like Language</td>
<td>Apache Pig</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Apache Pig</td>
<td>Distributed Query Engine &amp; SQL-like Language</td>
<td>Cloudera Impala</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Cassandra</td>
<td>Distributed NoSQL Databases</td>
<td>Cascalog</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Casuling</td>
<td>Distributed NoSQL Databases</td>
<td>Scalding</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>AllegroGraph</td>
<td>Distributed NoSQL Databases</td>
<td>OrientDB</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>OrientDB</td>
<td>Distributed NoSQL Databases</td>
<td>Neo4j</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Neo4j</td>
<td>Graph Oriented</td>
<td>Apache Avro</td>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 2.1: A categorization of tools used in the field of Big Data analytics.
part of the open source web search engine Apache Nutch\(^1\), which in turn was part of the project Apache Lucene\(^2\). Hadoop was created in order to address the need for an economic solution to the issues related with management and storage of the data derived from billions of websites.

Source of inspiration for the implementation of Hadoop were two Google’s publications, in 2003 and 2004 respectively. The first publication\(^24\) was related to the distributed file system used by the company, called GFS (Google File System). Based on GFS principles, the Hadoop Distributed File System (HDFS)\(^25\), was created. HDFS is able to meet the demanding storage needs of large amounts of data, in terms of space and maintenance. The second publication of Google\(^26\), included the implementation of MapReduce programming model so as to create distributed applications. The philosophy of MapReduce technique adopted by the developers of Hadoop, making it, along with HDFS, the two fundamental components of the framework.

In 2006, Hadoop was separated from Nutch, becoming directly part of the Lucene project. Later, in 2008, the recruiting of the Hadoop’s key developer, Doug Cutting, in Yahoo! as well as the creation of a special group for the development of Hadoop, marked the absolute independence of the project.

**Architecture**

As it mentioned before, the two fundamental components of Hadoop is the file system HDFS and the implementation of MapReduce programming model. These two components are mutually interrelated.

The development and execution of distributed applications with Hadoop requires a number of computers connected together via network. These computers constitute the nodes, on which MapReduce applications are implemented as well as the natural environment on which the HDFS is built. All these computers and the network through which they communicate constitute the so-called Hadoop Cluster.

Typically, in a Hadoop Cluster, one of the computers has the role of the master node, while the rest act as slave nodes. For the organization of a distributed application in a Hadoop Cluster, i.e., storage of input and output data in HDFS and the implementation the MapReduce stages, the activation of some special programs (daemons) at the nodes of the cluster is required. These programs are divided into two categories: i) the storage daemons and ii) the computation daemons (Figure 2.1\(^27\)). Both two categories are based on master/slave architecture (Figures 2.2 and 2.3\(^28\)).

**NameNode**  This is the most important daemon in a Hadoop Cluster. It is executed in the master node and could be considered as the “guardian” of the entire system. Moreover it is responsible for supplying the slave DataNode daemons, which are presented in the next paragraph, with low-level input/output tasks. In addition, NameNode is responsible for creating records regarding the way data files are divided into blocks as well as to which node each block is saved. NameNode is also responsible for creating records about the general state of the distributed file system. As the essential functions of NameNode are related to memory and input/output processes, the node that is running the NameNode deamon, typically, is not used for data storage or for direct computations in a MapReduce process. In other words, the node which has received the role of NameNode, should not perform the duties of DataNode (for data storage) or TaskTracker (to perform MapReduce processes). Although in practice, especially in clusters consisting of very few nodes, the previous condition often is overlooked. In the current Hadoop versions, NameNode has a serious weakness. Unlike the other daemons, whose any failure in hardware or software does not cause cessation of the system operation or data loss, an unexpected failure in NameNode leads to the collapse of the entire system.

**DataNode**  Each slave node of a Hadoop Cluster hosts a DataNode daemon. The DataNodes are responsible for reading and writing blocks of data files from the distributed file system to regular files on the local file

---

\(^1\)http://nutch.apache.org/

\(^2\)https://lucene.apache.org/core/
system. When a process has to read or write a file from HDFS, this file is separated into blocks and the NameNode informs the process about which block belongs to each DataNode. Then the process contacts with the respective DataNodes so as to be able to edit these files on their local system. These local files, as it mentioned before, are corresponding to blocks of HDFS files. Furthermore DataNodes communicate with each other to create copies of HDFS files so as to achieve false tolerance.

**Secondary NameNode (SNN)**  
SNN is a backup daemon, who is responsible for recording the status of the Hadoop cluster. Each Hadoop Cluster has a SNN which typically runs on a dedicated machine, which does not run the DataNode and TaskTracker daemons. The difference between SNN and NameNode is that in SNN
the changes in HDFS are not recorded in real time. Instead, SNN communicates with NameNode, at intervals defined by the Hadoop Cluster configuration, and takes snapshots of the HDFS state. As it mentioned earlier, one NameNode’s failure can lead to data loss and system crash. Snapshots of the system stored in SNN could help to address the problem of data loss, while in case of emergency and with the appropriate user intervention SNN could replace the main NameNode.

JobTracker  JobTracker daemon is the link between applications and the Hadoop environment. As the code of an application is submitted to the Hadoop Cluster, JobTracker is responsible for determining the execution plan by specifying which files are going to be processed and by commissioning various processes in the nodes. At the same time, it records the states of the processes that are under execution. If a task execution fails, the JobTracker automatically tries to restart the process and probably assign the process to a different node. The number of attempts to restart failed processes is finite and is determined by the cluster configuration. There is only one JobTracker in each Hadoop Cluster. The machine on which JobTracker is running is actually the master node of the cluster.

TaskTracker  On each slave node of the Hadoop cluster, TaskTracker is the one who undertakes to execute the process that has been assigned to the node by the JobTracker. Although there is only one TaskTracker in each slave node, it is able to manage simultaneously many JVMs (Java Virtual Machines) in order to execute in parallel various map or reduce functions. In addition, TaskTracker is obliged to communicate regularly with JobTracker, so as to the latter be aware of the correct operation of the former. If JobTracker does not receive a communication signal from the TaskTracker at the specific timestamp, then JobTracker assumes that TaskTracker has collapsed and assign the non-finished processes to another node.

The MapReduce Model
The MapReduce programming model, used in Hadoop for processing large data sets with distributed algorithms, is inspired by the map and reduce functions commonly used in functional programming. The basic elements of the model are 2 functions, the \texttt{map()} and the \texttt{reduce()} function, respectively. The input and the output elements of the 2 functions are tuples of the form \(<key, value>\). Generally, a job execution with the

![Figure 2.3: Snapshot of interaction between JobTracker and TaskTrackers (Computation daemons)](image-url)
usage of the MapReduce method can be considered as a 5-step procedure:

1. **Prepare the map() input**: After JobTracker assigns the desired task to the TaskTrackers, Map processors are designated on each slave node of the cluster. Data stored in HDFS are assigned on these Map processors in the form of \(<key1, value1>\) tuples. Regarding to the value of key1, each tuple is assigned to a specific processor. Usually, the value of key1 in this step is a system generated value that aims to the load balance during the sharing of the data to the nodes, while value1 represents the actual data for processing.

2. **Run the user-provided map() code**: Map() runs exactly once for each tuple. The process performed to the value1 values, according to the user provided code, generates new tuples of the form \(<key2, value2>\).

3. **Combine and Shuffle the map() output to the Reduce processors**: When all Map processors have finished the processing of all the input tuples that represent the data, the provided output tuples of the form \(<key2, value2>\) are clustered according to the value of the key2. This procedure results to the creation of intermediate tuples of the form \(<key2, list<value2>>\). Each key2 with its associated list of values are assigned to a Reduce processor. Reduce processors are generated in the slave nodes of the cluster, in the beginning of this step.

4. **Run the user-provided reduce() code**: Each Reduce processor in the slave nodes is assigned with specific tuples of the form \(<key2, list<value2>>\) according to the value of the key2. The reduce() function runs exactly once for each assigned tuple, where it iterates on the list<value2> to perform the processing that have been defined by the user’s code.

5. **Produce the final output**: The output of the Reduce processor are tuples of the form \(<key3, value3>\), which are sorted according to the value of key3, and the total of those tuples compose the desired output of the job, which is usually stored back in HDFS, or is passed as input to other MapReduce jobs for further procedure.

Logically these 5 steps can be thought of as running in sequence, each step starts only after the previous step is completed, though in practice, they can be intertwined, as long as the final result is not affected.

Figure 2.4 illustrates the flow of the Word Count example in Hadoop/MapReduce along with the pseudo code of this procedure. The Word Count example is considered as the "Hello World"¹ example in the Distributed Architectures' world.

![Diagram](image)

**Figure 2.4**: The flow and the pseudocode of the MapReduce Word Count Example.

2.1.1.2 Apache Drill

**History**

Apache Drill is an open-source software framework that supports data-intensive distributed applications for interactive analysis of large-scale datasets. Drill is the open source version of Google’s Dremel system which is available as an IaaS² service called Google BigQuery³ and initially presented by Google in [29] the year 2010, although it has been in production since 2006. Dremel, and consequently its open source version Drill, are not intended to be a replacement for MapReduce, on the contrary they complement MapReduce-based computing and are often used in conjunction with it to analyze outputs of MapReduce pipelines or rapidly prototype larger computations.

**Architecture**

In an abstract level, Drill can be related to parallel DBMSs, although it differs from the traditional databases as it is capable of operating on *in situ* nested data. *In situ* refers to the ability to access data “in place”, e.g., in a distributed file system like Hadoop’s HDFS, Google’s GFS [24] or another storage layer. Drill supports a nested data model with data encoded in a number of formats such as JSON, Avro or Protocol Buffers (presented in later subsections of this chapter). In many organizations nested data is the standard, so supporting a nested data model eliminates the need to normalize the data. Figure 2.5 represents a simple nested data schema with two implemented examples.

![Figure 2.5: Two sample nested records and their schema](image)

Drill processes data through an SQL-based query language. Although there have been a number of implementations that aim at using SQL-like systems and languages for processing Big Data, with the most known to be Apache Pig¹ and Hive², Dremel and Drill have a major advantage over those implementations, as the former ones translate each query in a Hadoop’s Map/Reduce job, a procedure that could be very time-demanding. On the contrary Dremel/Drill query execution is based in the following constructive elements:

---

²[http://en.wikipedia.org/wiki/Infrastructure_as_a_service#Infrastructure_as_a_service_.28IaaS.29](http://en.wikipedia.org/wiki/Infrastructure_as_a_service#Infrastructure_as_a_service_.28IaaS.29)
³[https://developers.google.com/bigquery/](https://developers.google.com/bigquery/)
¹[https://pig.apache.org/](https://pig.apache.org/)
**Tree Architecture**  Query execution is implemented by a multi-level serving tree, Figure 2.6. A root server receives incoming queries, reads metadata from the tables, and routes the queries to the next level in the serving tree. The leaf servers communicate with the storage layer or access the data on local disk.

**Query dispatcher**  As a multi-user system, Dremel/Drill usually executes several queries simultaneously. A Query dispatcher is the coordinator of this procedure as it schedules the queries based on their priorities and balances the load. In addition it provides fault tolerance when one server becomes much slower than others or a tablet replica becomes unreachable.

![Figure 2.6: System architecture and execution inside a server node](image)

2.1.1.3 **Apache Storm**

**History**
Although MapReduce, Hadoop, and related technologies have made it possible to store and process data at scales previously unthinkable, unfortunately, these data processing technologies are not adequate for real-time systems, nor are they designed to be. Realtime data processing has a fundamentally different set of requirements than batch processing. These were the reasons that led to the creation of the Storm.

Storm is a distributed realtime computation system that provides a set of general primitives for doing real-time computation. Specifically it can be used for “stream processing”, processing messages and updating databases in real-time. Storm also can be used for “continuous computation”, doing a continuous query on data streams and streaming out the results to users as they are computed as well as it can be used for “distributed RPC\(^1\)”, running an expensive computation in parallel on the fly. Storm originally created by Nathan Marz and his team at BackType, the project was open sourced after being acquired by Twitter. The initial release was on September 17, 2011 and in 2013 the Apache Software Foundation has accepted it into its incubator program\(^2\).

Storm is written predominantly in the Clojure programming language although it is programming language agnostic, meaning that Storm topologies and processing components can be defined in any language. Nowadays Storm is used by many companies, including Yahoo! and Twitter\(^3\).

---

\(^1\)[http://en.wikipedia.org/wiki/Remote_procedure_call]

\(^2\)[http://en.wikipedia.org/wiki/Storm_(event_processor)]

\(^3\)[https://github.com/nathanmarz/storm/wiki/Powered-By]
Architecture
Storm architecture very much resembles to Hadoop architecture. The key difference between Storm and Hadoop architectures is that Storm runs “topologies” instead of “MapReduce jobs”. “Jobs” and “topologies” themselves are very different in the notion that a MapReduce job eventually finishes, whereas a topology processes messages forever or until is killed [30].

Similar with a Hadoop cluster a Storm cluster consists of two types of nodes: a master node and several worker (slave) nodes. The master node runs a daemon called “Nimbus” while worker nodes run a daemon called “Supervisor”.

Nimbus
Nimbus is the daemon that runs in the master node of a Storm cluster. The functionality of Nimbus is similar to Hadoop’s “JobTracker”. Specifically, Nimbus is responsible for distributing code around the cluster, assigning tasks to machines, and monitoring for failures.

Supervisor
Supervisor is the daemon that runs in the worker nodes of a Storm cluster. The functionality of Nimbus is similar to “TaskTrackers” in Hadoop. Specifically, the Supervisor listens for work assigned to its machine and starts and stops 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 spread across many machines.

The coordination between Nimbus and the Supervisors is done through an Apache Zookeeper cluster. Zookeeper is an open source distributed coordination service for distributed applications, described in the next subsection of this chapter. Additionally, the Nimbus daemon and Supervisor daemons are fail-fast and stateless in the notion that state is kept in Zookeeper or on local disk. Thus, even if Nimbus or the Supervisors are killed, they will start back up as if nothing happens. This design leads to Storm clusters being incredibly stable.

Topologies
Topologies are one of the key concepts in Strom. In Storm in order to perform a realtime computation, a topology has to be created. In essence, a topology is a graph of computations. Each node in a topology contains processing logic and links between nodes which indicate how data should be passed around between nodes. As it mentioned before, a topology and a MapReduce job in Hadoop can be considered analogous.
**Streams** Streams are the core abstraction in Storm. A stream is an unbounded sequence of tuples. Storm provides the primitives for transforming a stream into a new stream in a distributed and reliable way. For instance, it is possible to transform a stream of tweets into a stream of trending topics, which briefly describes the core subject of the present thesis. In Storm, stream transformations are mainly implemented by “Spouts” and “Bolts”. Actually, Spouts and Bolts contains the application-specific logic.

A **Spout** is a source of streams. Specifically, spouts are reading data from a source (e.g. a queue or an API, etc.) and emit a list of fields.

A **Bolt** consumes any number of input streams, does some processing, and possibly emits new streams. Complex stream transformations, like computing a stream of trending topics from a stream of tweets, require multiple steps and thus multiple bolts. Bolts can do anything from run functions, filter tuples, do streaming aggregations, do streaming joins, talk to databases, and more.

Networks of Spouts and Bolts are packaged into a “topology”. A topology resembles a graph of stream transformations where each node is a Spout or Bolt, Figure 2.8. Edges in that graph indicate which Bolts are subscribing to which streams. When a Spout or Bolt emits a tuple to a stream, it sends the tuple to every bolt that subscribed to that stream.

![Figure 2.8: Storm Topology](image)

Each node in a Storm topology executes in parallel. In a topology, users can specify how much parallelism is desired for each node, then Storm will spawn the corresponding number of threads across the cluster to do the execution.

**Data model** Storm uses tuples as its data model. A tuple is a named list of values, and a field in a tuple can be an object of any type. Storm supports all the primitive types, strings, and byte arrays as tuple field values. So as to use an object of another type, only a serializer for the type has to be implemented.

### 2.1.1.4 Apache S4

**History** An alternative of Apache Storm in the field of Big Data stream processing is called S4. Its name stands for Simple Scalable Streaming System, and it was, initially, developed by Yahoo! which presented it in [31] in October 2010. S4 is an Apache Incubator project since September 2011 and it is licensed under the Apache 2.0 license.
**Architecture**

The architecture of S4 is largely influenced by the MapReduce architecture. Data is transferred among a cluster’s node in the form of key-value pairs, where data assigned with the same key are ending up and being processed from the same node. S4’s basic computational units are called Processing Elements (PEs). Each PE is uniquely identified by four components:

1. its *functionality* as defined by a PE class and the associated configuration
2. the *types of events* that it consumes
3. the *keyed attributed* in those events
4. the *value* of the keyed attribute in events which it consumes

The logical hosts to PEs are called Processing Nodes (PNs). They are responsible for listening to events, executing operations on the incoming events, dispatching events with the assistance of the communication layer, and emitting output events. An event listener in the PN passes incoming events to the processing element container (PEC) which invokes the appropriate PEs in the appropriate order.

The functionality and coordination of an S4 cluster is configured through its Communication Layer API. This API provides cluster management and automatic failover to standby nodes and maps physical nodes to logical nodes. Communication Layer manages the coordination of the cluster's nodes using Apache Zookeeper.

Figure 2.9 presents the basic components of the minimum logical unit of an S4 cluster, the Processing Node, while Figure 2.10 represents the flow of the Word Count example in S4.

![Figure 2.9: S4’s Processing Node.](image)

Although Apache S4 can be considered as the first integrated framework in the Big Data streaming processing field, it still has some negative characteristics that prevent it from being considered as a stable and mature framework. Those characteristics are summarized in the following list:

- Complex configuration
- Opaque processing
- Potentially lossy data
- Slow and hard debugging

**2.1.2 Coordination Systems**

With a growing family of services running as part of the aforementioned framework cluster, there is a need for coordination and naming services. The mainly used systems that handle this kind of tasks are Apache Zookeeper and Apache Oozie.
2.1.2.1 Apache Zookeeper

History
In real life applications, the number of computing nodes of a distributed framework’s cluster are of the order of hundreds or thousands. Moreover, in some frameworks, the number of computing nodes is not stable and could be altered on the fly. Thus, as the number of computing nodes can be increased or decreased, members of the cluster need to synchronize with each other, to know where to access services as well as to know how they should be configured. This is the purpose of Zookeeper. Initially, ZooKeeper was a sub project of Hadoop but today is a top-level project in its own right.

Architecture
ZooKeeper allows distributed processes to coordinate with each other through a shared hierarchical name space of data registers, called znodes, much like a file system. One could say that ZooKeeper is a file system enhanced with extra functionalities like high throughput, low latency, highly available and strictly ordered access to the data registers. These functionalities allow ZooKeeper to be used in large distributed systems, prevent it from becoming the single point of failure in big systems and ensures sophisticated synchronization primitives to be implemented at the clients. ZooKeeper is designed to store metadata that help coordination among clients thus has a built-in sanity check of 1Mb, to prevent it from being used as a large data store.

In large production environments ZooKeeper service consists of more than one servers and many clients (e.g Hadoop or Storm nodes) connected via TCP. The servers that make up the ZooKeeper service must all be aware about each other and clients must be aware of all servers. At each time, any client must be connected with only one server. Figure 2.11 represents an abstract implementation of a multiserver ZooKeeper service.

2.1.2.2 Apache Oozie

History
Oozie is a workflow scheduler system to manage Hadoop jobs. It is a server-based Workflow Engine specialized in running workflow jobs with actions that run Hadoop MapReduce and Pig jobs. Oozie is implemented as a
Figure 2.11: Abstract Implementation of Zookeeper Architecture

Java Web-Application that runs in a Java Servlet-Container. Currently Oozie is distributed under the Apache License 2.0.

Architecture
Production systems utilizing Hadoop can often contain complex pipelines of transformations, each with dependencies on each other. For example, the arrival of a new batch of data will trigger an import, which must then trigger recalculations in dependent datasets. The Oozie component provides features to manage the workflow and dependencies, removing the need for developers to code custom solutions.

For the purposes of Oozie, a workflow is a collection of actions (i.e. Hadoop Map/Reduce jobs, Pig jobs) arranged in a control dependency DAG (Direct Acyclic Graph). “Control Dependency” from one action to another means that the second action can’t run until the first action has completed. Oozie workflow actions start jobs in remote systems (i.e. Hadoop, Pig). Upon action completion, the remote systems callback Oozie to notify the action completion, at this point Oozie proceeds to the next action in the workflow.

Oozie workflows contain control flow nodes and action nodes.

Control flow nodes define the beginning and the end of a workflow (start, end and fail nodes) and provide a mechanism to control the workflow execution path (decision, fork and join nodes).

Action nodes are the mechanism by which a workflow triggers the execution of a computation/processing task. Oozie provides support for different types of actions: Hadoop MapReduce, Hadoop file system, Pig, SSH, HTTP, eMail and Oozie sub-workflow. Oozie can be extended to support additional type of actions.

Oozie workflows can be parameterized, using variables like \texttt{inputDir} (input directory) within the workflow definition. When submitting a workflow job values for the parameters must be provided. If properly parameterized, i.e. using different output directories, several identical workflow jobs can run concurrently.

2.1.3 Distributed NoSQL Databases

As the number of sources generating data is continuously growing with high rates, this results to the incremental growing of what is considered as the upper bound of Big Data, from the perspective of Volume. Today, not only the traditional data management techniques of Computer Science, but even the basic techniques of the field of Big Data Analytics, regarding to data storage and manipulation, are ineffective for a large variety of applications. Manipulating data that is stored in raw format in a distributed file system, like HDFS, can lead to time delay, especially when the analysis of them is a procedure of many sequential steps that lead to regular updates. The need of a more organized way of Big Data storage that could permit efficient, ad-hoc queries is nowadays inevitable. Traditional DBMS have been proven unable to face the needs of the field of Big data Analytics [32]. On the contrary the so called, NoSQL databases seems to fit better to this problem. Indeed, there is a great development of various NoSQL Databases capable to lay on distributed file systems which
in conjunction with new domain specific query languages manage to face the problem of the distributed data analysis very efficiently. As a proof of this concept we present a chart of the evolution of popularity of DBMS in big data and real-time web applications in Figure 2.12 [33]. It is clear that DBMSs that belong to the category of NoSQL are finding significant and growing industry use.

Figure 2.12: Historical trend of the categories’ popularity. For each month the best three systems per category are chosen and the average of their ranking scores is calculated.

NoSQL systems are also referred to as ”Not only SQL” to emphasize that they may in fact allow SQL-like query languages to be used. They are not consist a single category, on the contrary, there have been various approaches to classify NoSQL databases. The most accepted classification is based on the model of the data that NoSQL databases store, with the leading categories to be the following [34]:

- **Column Oriented**: They have some similarities to relational databases on the surface but in more depth a lot of principles change. A column oriented database can have different columns on each row so is not relational and doesn’t have what qualifies in an RDBMS as a table. The only key concepts in a column oriented database are columns, column families and super columns. Column families define how the data is structured on disk. A column by itself is just a key-value pair that exists in a column family. A super column is like a catalogue or a collection of other columns except for other super columns.

- **Document Oriented**: The central concept of a document-oriented database is the notion of a Document. While each document-oriented database implementation differs on the details of this definition, in general, they all assume documents encapsulate and encode data (or information) in some standard formats or encodings. Encodings in use include XML, YAML, JSON, and BSON, as well as binary forms like PDF and documents exported from Office suites.

- **Key-Value stores**: The Key-Value databases have the most simple structure among all categories. Data is indexed and queried based on their key. Key-value stores provide consistent hashing so they can scale incrementally as data scales. They communicate in a node structure through a gossip-based membership protocol¹ to keep all the nodes synchronized.

- **Graph Oriented**: A graph oriented database is a database that uses graph structures with nodes, edges, and properties to represent and store data. They can be considered as big dense network structures which use sophisticated shortest path algorithms to make data queries more efficient.

In the rest of this subsection we present the most famous Distributed NoSQL Databases of each one of the above categories.

2.1.3.1 Column Oriented Databases

Apache HBase

History
HBase is a non-relational, distributed database which modeled after Google’s BigTable [35]. It is developed as part of Apache Software Foundation’s Hadoop project and runs on top of HDFS (Hadoop Distributed Filesystem), providing BigTable-like capabilities for Hadoop. In other words, it provides a fault-tolerant way of storing large quantities of sparse data. Apache HBase began as a project by the company Powerset out of a need to process massive amounts of data for the purposes of natural language search. It is now a top-level Apache project and has generated considerable interest.

Architecture
The HBase Physical Architecture consists of servers in a Master-Slave relationship. Typically, the HBase cluster has one Master node, called HMaster and multiple Region Servers called HRegionServer. Each Region Server contains multiple Regions, called HRegions.

Just like in a Relational Database, data in HBase is stored in Tables and these Tables are stored in Regions. When a Table becomes too big, the Table is partitioned into multiple Regions. These Regions are assigned to Region Servers across the cluster. Each Region Server hosts roughly the same number of Regions.

The HMaster in the HBase is responsible for the following tasks:

- Performing Administration
- Managing and Monitoring the Cluster
- Assigning Regions to the Region Servers
- Controlling the Load Balancing and Failover

The HRegionServer is responsible for the following tasks:

- Hosting and managing Regions
- Splitting the Regions automatically
- Handling the read/write requests
- Communicating with the Clients directly

Each Region Server contains a Write-Ahead Log (called HLog) and multiple Regions. Each Region in turn is made up of a MemStore and multiple StoreFiles (HFile). The data lives in these StoreFiles in the form of Column Families (explained below). The MemStore holds in-memory modifications to the data.

The mapping of Regions to Region Server is kept in a system table called .META. When trying to read or write data from HBase, the clients read the required Region information from the .META table and directly communicate with the appropriate Region Server. Each Region is identified by the start key (inclusive) and the end key (exclusive).
Data Model
The Data Model in HBase is designed to accommodate semi-structured data that could vary in field size, data type and columns. Additionally, the layout of the data model makes it easier to partition the data and distribute it across the cluster. The Data Model in HBase is made of different logical components such as Tables, Rows, Column Families, Columns, Cells and Versions.

Tables: The HBase Tables are more like logical collection of rows stored in separate partitions called Regions. As shown in Figure 2.13, every Region is then served by exactly one Region Server. The figure above shows a representation of a Table.

Rows: A row is one instance of data in a table and is identified by a rowkey. Rowkeys are unique in a Table and are always treated as a byte array.

Column Families: Data in the rows are grouped together as Column Families. Each Column Family has one or more Columns and these Columns are stored together in a low level storage file known as HFile. Column Families form the basic unit of physical storage to which certain HBase features like compression are applied. Hence it’s important that proper care be taken when designing Column Families in table.

Columns: A Column is identified by a Column Qualifier that consists of the Column Family name concatenated with the Column name. There can be multiple Columns within a Column Family and Rows within
a table can have varied number of Columns.

**Cell**: A Cell stores data and is essentially a unique combination of row key, Column Family and the Column (Column Qualifier). The data stored in a Cell is considered as its value and the data type is always treated as byte array.

**Version**: The data stored in a cell is versioned and versions of data are identified by a timestamp. The number of versions of data retained in a column family is configurable and this value by default is 3.

**Apache Cassandra**

**History**
Cassandra is one of the most widely used distributed column-oriented databases and it was, initially, developed at Facebook which released it as open-source in July 2008. Since the early 2010 Cassandra's development is maintained by the Apache Software Foundation.

**Architecture**
As a distributed database, Cassandra lays on a number of different computer nodes forming a Cassandra Cluster. Unlike the other distributed frameworks that have been presented in the previous sections, a Cassandra cluster doesn't follow a master/slave structure. On the contrary all nodes in a Cassandra cluster has the same role and are connected via the peer-to-peer protocol. This decentralized form of the cluster, in conjunction with its configurable characteristic of data replication, which allow the multi-storage of the same data in different nodes, exceeds the common problem of single point of failure. Single point of failure is common problem in distributed systems which implement master/slave architecture, e.g, Hadoop's NameNode potential failure leads to job execution failure. Also, another advantage that arises from the fact that all the nodes have the same role in the cluster is that user is allowed to add more nodes or remove dysfunctional ones on the fly, adjusting its scalability.

**Data model** In Cassandra, the total data managed by the cluster is represented as a circular space or ring. The ring is divided up into ranges equal to the number of nodes, with each node being responsible for one or more ranges of the overall data. Before a node can join the ring, it must be assigned with a token. The token determines the node's position on the ring and the range of data it is responsible for.

Column family data is partitioned across the nodes based on the row key. To determine the node where the first replica of a row will live, the ring is walked clockwise until it locates the node with a token value greater than that of the row key. Each node is responsible for the region of the ring between itself (inclusive) and its predecessor (exclusive). With the nodes sorted in token order, the last node is considered the predecessor of the first node.

Cassandra’s data model is a partitioned row store with tunable consistency. Rows are organized into tables, where the first component of a table's primary key is the partition key. Within a partition, rows are clustered by the remaining columns of the key. Other columns may be indexed separately from the primary key.

Tables may be created, dropped, and altered at runtime without blocking updates and queries.
Cassandra does not support joins or subqueries, except for batch analysis via Hadoop. Rather, Cassandra emphasizes denormalization through features like collections.

**2.1.3.2 Document Oriented Databases**

**MongoDB**
**History**

Development of MongoDB began in 2007, from a company named 10gen, which was building a platform as a service (PaaS¹). In 2009, MongoDB was open sourced as a stand-alone product with an AGPL license. It is written in C++.

**Data Model**

Data in MongoDB is stored in collections, which in turn is stored in databases. Collections are a way of storing related data. Collections contain documents which have in turn keys, another name for attributes. Data is stored and queried in BSON, binary-serialized JSON-like data. Features are a superset of JSON, adding support for regular expressions, date, binary data, and their own object id type. Documents are not identified by a simple ID, but by an object identifier type, optimized for storage and indexing. By default, MongoDB uses a combination of machine identifier, timestamp and process id as a document’s identifier. Although, the user is free to assign any value he wishes as a document’s identifier. Every document gets a default index on the _id attribute, which also enforces uniqueness. It’s recommended to index any attribute that’s being queried or sorted on. Indexes can be set on any attribute or embedded attributes and documents. Indexes can also be created on multiple attributes, additionally specifying a sort order. Figure 2.14 presents MongoDB in a nutshell, in BSON format [36].

```json
{
  "_id" : ObjectId("5081c97c763857c5588f336"),
  "name" : "mongo",
  "type" : "db",
  "doc_links" : {
    "installation" : "http://docs.mongodb.org/manual/installation/",
    "tutorial" : "http://docs.mongodb.org/manual/tutorial/getting-started/",
    "reference" : "http://docs.mongodb.org/manual/reference/"
  },
  "versions" : [
    { "v" : "2.0.1", "released" : ISODate("2011-10-22T03:06:14Z"), "stable" : true },
    { "v" : "2.1.0", "released" : ISODate("2012-02-03T17:54:14Z"), "stable" : false },
    { "v" : "2.2.0", "released" : ISODate("2012-09-24T17:36:56Z"), "stable" : true },
  ],
  "features" : [
    { "md5" : BinData(5, "nhB9nTcrToJr2001QqQZg=") }
  ]
}
```

Figure 2.14: MongoDB in BSON format

**Architecture**

Although MongoDB can be used as a single server database, that would create serious limitations in applications that are related with Big Data. MongoDB has support for sharding. Sharding is the method MongoDB uses to split a large collection across several servers, i.e., a cluster. Sharding is designed to fulfill three simple goals [37]:

- Make the cluster “invisible”: Invisible in this context means that applications using MongoDB are not “aware” that what are talking to is actually a MongoDB cluster and not a single server instance. To accomplish this, MongoDB comes with a special routing process called mongos. Mongos sits in front of the cluster and looks like an ordinary single server to anything that connects to it. It forwards requests

¹http://en.wikipedia.org/wiki/Platform_as_a_service
to the correct server or servers in the cluster, then assembles their responses and sends them back to the client, Figure 2.15.

- Make the cluster always available for reads and writes: In case of potential node failures there should never be a time when users can’t read or write data. The cluster should allow as many nodes as possible to fail before its functionality noticeably degrades. MongoDB ensures maximum uptime in different ways. Every part of a cluster can and should have at least some redundant processes running on other machines (optimally in other data centers) so that if one process, machine or data center goes down, the other ones can immediately and automatically the failed task and continue without problem.

- Let the cluster grow easily: As an application continues to grow, it needs to add more shards. When a shard is added, MongoDB will start moving data from the existing shards to the new one. Because this procedure, i.e., moving data, puts added pressure on shards, MongoDB tries to do this as gently as possible. It slowly moves one chunk at a time, then tries again later if the server seems busy.

![Figure 2.15: An appserver running a mongos.](image)

Finally, MongoDB provides the “mapReduce” database command which translates operations in MapReduce jobs. Figure 2.16 presents a MongoDB mapReduce query as well as an illustrated form of the corresponding execution [38].

**Apache CouchDB**

**History**

CouchDB, which name is an acronym for Cluster Of Unreliable Commodity Hardware, is a project created in April 2005 by Damien Katz, former developer at IBM. In February 2008, it became an Apache Incubator project and the license was changed to the Apache License. A few months later, it graduated to a top-level project. This led to the first stable version being released in July 2010. CouchDB is written in Erlang⁴.

⁴http://www.erlang.org/
### Data Model

Documents in CouchDB are stored in JSON format. For every new document CouchDB generates the "_id" and "_rev" fields, Figure 2.17 shows a simple CouchDB document example. The _id field can, instead be created by the user and is unique per database, for each document. The scope of the _rev field is to provide a secure mechanism in the procedure of document updates. More precisely, the steps of updating a document in CouchDB, does not include indexing a field in the specific document and inserting a new value. Instead, the user should load the full document out of CouchDB, make changes in the JSON structure, and save the entire new revision of that document back into CouchDB. Each revision is identified by a new _rev value. If user wants to update or delete a document, CouchDB expects to include the _rev field of the revision that is to be to changed. When CouchDB accepts the change, it will generate a new revision number. This mechanism ensures that, in case somebody else made a change before the user got to request the document update, CouchDB will not accept the update to prevent overwrite data which existence is unknown.

### Architecture

CouchDB is a peer-based distributed database system, it allows for users and servers to access and update the same shared data while disconnected and then bi-directionally replicate those changes later.

The CouchDB document storage, view and security models are designed to work together to make true bidirectional replication efficient and reliable. Both documents and designs can replicate, allowing full database applications (including application design, logic and data) to be replicated to laptops for offline use, or replicated to servers in remote offices where slow or unreliable connections make sharing data difficult.

The replication process is incremental. At the database level, replication only examines documents updated since the last replication. Then for each updated document, only fields and blobs that have changed are replicated across the network. If replication fails at any step, due to network problems or crash for example, the next
replication restarts at the same document where it left off.

Partial replicas can be created and maintained. Replication can be filtered by a Javascript function, so that only particular documents or those meeting specific criteria, are replicated. This can allow users to take subsets of a large shared database application offline for their own use, while maintaining normal interaction with the application and that subset of data.

Conflict detection and management are key issues for any distributed edit system. The CouchDB storage system treats edit conflicts as a common state, not an exceptional one. The conflict handling model is simple and "non-destructive" while preserving single document semantics and allowing for decentralized conflict resolution.

CouchDB allows for any number of conflicting documents to exist simultaneously in the database, with each database instance deterministically deciding which document is the "winner" and which are conflicts. Only the winning document can appear in views, while "losing" conflicts are still accessible and remain in the database until deleted or purged during database compaction. Because conflict documents are still regular documents, they replicate just like regular documents and are subject to the same security and validation rules.

When distributed edit conflicts occur, every database replica sees the same winning revision and each has the opportunity to resolve the conflict. Resolving conflicts can be done manually or, depending on the nature of the data and the conflict, by automated agents. The system makes decentralized conflict resolution possible while maintaining single document database semantics.

Conflict management continues to work even if multiple disconnected users or agents attempt to resolve the same conflicts. If resolved conflicts result in more conflicts, the system accommodates them in the same manner, determining the same winner on each machine and maintaining single document semantics. [39]

2.1.3.3 Key-Value datastores

Key-Value stores can further distinguished in two categories, the disk-based datastores and the in-memory datastores. In this subsection, initially, we briefly refer to a variety of the widely used Key-Value stores and subsequently we further present, in a little more depth, two more datastores that belong in this specific category, Redis (in-memory) and ElephantDB (disk-based). The reason that we choose to deepen only in the last two datastores is because we use the former in the needs of the implementation part of this thesis, and we are planning to use the latter in future our future work, based on this implementation.

According to the above, next follows a list of the most widely used Key-Value datastores:

- **Amazon DynamoDB**: DynamoDB is a proprietary Key-Value database from Amazon.com, Inc¹, with initial release the year 2012. Its data model is based on a previous implementation of the same company

---

with similar name (DynamoDB) and data model but different underlying implementation. DynamoDB has a single master design in contrast with its ancestor which has a multi-server design, requiring the client to resolve version conflicts.

- **Riak**: Riak is an open-source Key-Value datastore, implemented by Basho Technologies¹, inspired Amazon's dynamo principles and heavily influenced from the distributed computer system’s CAP theorem (explained in [41]). One of the biggest advantages of Riak is the ability of Multi-Datacenter Replication where one cluster acts as a “primary cluster” and handles replication requests from one or more “secondary clusters”. In case that the datacenter with the primary cluster goes down, a secondary cluster can take over as the primary cluster.

- **MemcacheDB**: MemcacheDB is a distributed key-value storage system designed for persistent. It is a persistent storage engine for fast and reliable key-value based object storage and retrieval. It conforms to memcache protocol², so any memcached client can have connectivity with it. MemcacheDB uses Oracle’s Berkeley DB³ as a storing backend, so lots of features including transaction and replication are supported.

**Redis**

**History**

Redis is an open-source, networked, in-memory, key-value data store with optional durability. It is written in ANSI C. Redis’s original developer is Salvatore Sanfilippo and on March 15 2010 VMWare started sponsoring the project. Although, from May 2013 until now the project has been sponsored by Pivotal⁴. Nowadays, Redis is considered as one of the most famous key-value stores.

**Data Model**

In its outer layer, the Redis data model is a dictionary which maps keys to values. One of the main differences between Redis and other structured storage systems is that Redis supports not only strings, but also abstract data types like:

- Lists of strings
- Sets of strings (collections of non-repeating unsorted elements)
- Sorted sets of strings (collections of non-repeating elements ordered by a floating-point number called score)
- Hashes where keys and values are strings

The type of a value determines what operations (called commands) are available for the value itself. Redis supports high level atomic server side operations like intersection, union, and difference between sets and sorting of lists, sets and sorted sets.

¹http://basho.com/
²https://code.google.com/p/memcached/
⁴http://gopivotal.com/
**Architecture**

Redis is a client/server system. A typical deployment diagram is shown in Figure 2.18. Redis server is a process that talks a TCP protocol with clients. At its roots, Redis is a single-threaded server. This means that a single thread reads incoming connections using an event-based paradigm such as epoll\(^5\), kqueue\(^6\) and select\(^7\). When a specific event occurs on a file descriptor, it processes them and write back responses. Redis uses an home-made event library which abstracts low level socket management. Moreover, Redis uses a command table to interpret every command from the protocol and to execute the appropriate action. In other words, requests are managed with commands. Specifically, Redis using a command table and according what is read from sockets a command handler is invoked to perform desired action.

![Figure 2.18: A typical deployment diagram with multilingual clients.](image)

**ElephantDB**

**History** ElephantDB is a database that specializes in exporting key/value data from Hadoop. It was created by Nathan Marz at his former start-up company BackType. ElephantDB server has a Thrift interface (described in later section), so any language can make reads from it. The database itself is implemented in Clojure.

**Data Model** In ElephantDB the data transfer to and from Hadoop’s distributed file system is achieved through Hadoop’s native Input/OutputFormat or Cascading/Cascalog (will be presented in later sections) taps. ElephantDB stores both keys and values as byte arrays.

**Architecture** ElephantDB is composed of two components. The first is a library that is used in MapReduce jobs for creating an indexed key/value dataset that is stored on a distributed filesystem. The second component is a daemon that can download a subset of a dataset and serve it in a read-only, random-access fashion. A group of machines working together to serve a full dataset is called a ring.

---


\(^6\) [http://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2](http://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2)

\(^7\) [http://man7.org/linux/man-pages/man2/select.2.html](http://man7.org/linux/man-pages/man2/select.2.html)
There are two aspects to ElephantDB, view creation and view serving. View creation occurs in a MapReduce job at the end of the batch layer workflow where the generated partitions are stored in the distributed filesystem. Views are then served by a dedicated ElephantDB cluster that loads the shards from the distributed filesystem and interacts with clients that support random read requests. ElephantDB server doesn’t support random writes.

An ElephantDB datastore contains a fixed number of shards of a “Local Persistence”. ElephantDB’s local persistence engine is pluggable, and ElephantDB comes bundled with local persistence implementations for Berkeley DB Java Edition and LevelDB\(^1\). On the MapReduce side, each reducer creates or updates a single shard into the DFS, and on the server side, each server serves a subset of the shards.

### 2.1.3.4 Graph Oriented Databases

Graph oriented databases are databases that use graph structures with nodes, edges, and properties to represent and store data. Figure 2.19 depicts an example of such structures. A graph database is any storage system that provides index-free adjacency. This means that every element contains a direct pointer to its adjacent elements and no index lookups are necessary [42].

Some of the most widely used graph oriented databases are:

- **Neo4j**: An open source, embedded, disk-based, fully transactional and general purpose graph database, implemented in Java. Neo4j is further presented in the next section.

- **AllegroGraph**: A closed source purpose-built database for the storage and retrieval of triples composed of subject-predicate-object usually imported/exported using Resource Description Framework (RDF)\(^2\) format. AllegroGraph and similar databases belong to a subcategory of the Graph oriented databases known as “triplestores”.

- **OrientDB**: An open source hybrid database with the features of both Document and Graph oriented databases. It supports schema-less, schema-full and schema-mixed modes. It has a strong security profiling system based on users and roles and supports SQL as a query language.

![Figure 2.19: Example of a property graph in Graph Databases.](https://code.google.com/p/leveldb/)

\(^1\)http://www.w3.org/RDF/

---

30
Neo4j

History
Neo4j is a commercially supported open-source graph database. It was designed and built from the ground-up to be a reliable database, optimized for graph structures instead of tables. Neo4j was developed by Neo Technology, Inc., based in the San Francisco Bay Area, US and Malmo, Sweden. Neo4j version 1.0 was released in February, 2010.

Data Model
Generally, a graph is a collection of nodes and edges (relationships) that connect pairs of nodes. In a graph database relationships are very important because they connect two nodes. Both nodes and relationships can hold an arbitrary amount of key-value pairs.

In Neo4j Nodes consist of properties. A Node starts with a single Property and grow to a few million Properties, though that is not efficient. Thus, at some point it makes sense to distribute the data into multiple nodes, organized with explicit Relationships. Relationships organize Nodes into arbitrary structures, allowing a Graph to resemble a List, a Tree, a Map, or a compound Entity. Furthermore, Neo4j provides Labels for grouping nodes in the graph. Labels can be used to restrict queries to subsets of the graph, as well as enabling optional model constraints and indexing rules. Depending on the nature of the query, Neo4j provides functionality for traversal navigation through the nodes or access a specific node using indexes.

Architecture
Neo4j uses a master-slave cluster architecture. In particular, there are two parts to each Neo4j instance. One part is the database itself, and the other is the cluster management component. The cluster management component continuously stays synchronized with all instances in the cluster, keeping track of any instances joining or leaving. When a master election becomes necessary, the cluster management component ensures that a new master is consistently elected. The database layer manages the rest of the system. The Neo4j cluster performs automatic master election. Slave instances pull transactional updates of data from the master. As such, all write operations are coordinated by the master. It is still possible to write via slaves, but the slave will still make sure to perform the write operation synchronously with the master, behind the scenes.

Figure 2.20: Abstract instances of Neo4j cluster.

2.1.4 Big Data Query Processing
The great evolution of the Big Data field combined with the large adoption of Apache Hadoop by the industry, generated the need of development of models that make the execution of queries on data, stored in HDFS or on distributed databases, more simple, fast (both from the perspective of coding effort) and closed than the MapReduce model itself. A number of well known companies as well as the open source community have
developed a variety of tools that serve the aforementioned necessity. The most widely used tools that belong in this category are being presented in this subsection. Furthermore, each of the presented tools can be further classified in a more specific domain. The possible domains are described by the following:

- Integrated frameworks consisting from a Query Engine that lays on the distributed file system, often accompanied a corresponding domain specific Query Language.

- Programming languages’ libraries which use in the programs make the task of querying data stored on distributed file systems more abstract resulting the reduce of complexity, in great amounts.

2.1.4.1 Apache Hive

**History**
Apache Hive can be considered as a data warehouse infrastructure built on top of Hadoop for providing data summarization, query, and analysis. It was initially developed by Facebook, but is now used by several companies and organizations. In 2009 Facebook open sourced Hive. Today the project is maintained by Apache Software Foundation.

**Data Model**
Data in Hive is organized into Tables, Partitions and Buckets. Tables are analogous to tables in relational databases. Each table is related to a corresponding HDFS directory. The data in a table is serialized and stored in files within that directory. Hive provides built-in serialization formats but users can also implement custom serialization formats. The serialization format of each table is stored in the system catalog and is automatically used by Hive during query compilation and execution. Data in each partition may in turn be divided into buckets based on the hash of a column in the table. Each bucket is stored as a file in the partition directory.

Hive supports primitive column types (integers, floating point numbers, generic strings, dates and booleans) and nestable collection types — array and map. Users can also define their own types programmatically. [43]

**Architecture**
Hive’s architecture is based on the combination of Hadoop itself and a series of components that interact with it. A brief presentation of those components follows:

- **External interfaces:** Hive’s manipulation can be achieved through the Command Line Interface (CLI), Web User Interface (WebUI) or several Application Programming Interfaces (APIs).

- **Thrift Server:** Thrift, which will be further presented in the next subsection of this chapter, is a framework for cross-language services, where a server written in one language can also support clients in other languages. The Hive Thrift Server exposes a very simple client API to execute its query language’s statements.

- **Metastore:** This is the system catalog and all other components of Hive interact with the Metastore. It contains metadata about the tables stored in Hive. This metadata is specified during table creation and reused every time the table is referenced in the query language.

- **Driver:** The Driver manages the life cycle of a query language statement during compilation, optimization and execution. On receiving the statement, from the Thrift Server or other interfaces, it creates a session handle which is later used to keep track of statistics.

- **Compiler:** The Compiler is invoked by the driver upon receiving a query language statement. The compiler translates this statement into a plan which consists of a Direct Acyclic Graph (DAG) of MapReduce jobs.
The driver submits the individual MapReduce jobs from the DAG to the Execution Engine in a topological order. Hive uses Hadoop as its execution engine. Figure 2.21 represents the Hive architecture, consisting of the aforementioned components.

![Hive Architecture](image)

**Figure 2.21: Hive Architecture.**

### Query Language

The query language Hive provides is an SQL-like query language called HiveQL which supports select, project, join, aggregate, union etc. HiveQL supports data definition (DDL) statements to create tables with specific serialization formats, and partitioning and bucketing columns. Users can load data from external sources and insert query results into Hive tables via the load and insert data manipulation (DML) statements respectively. HiveQL supports multi-table insert, where users can perform multiple queries on the same input data using a single HiveQL statement. Hive optimizes these queries by sharing the scan of the input data. HiveQL is also very extensible. It supports user defined column transformation (UDF) and aggregation (UDAF) functions implemented in Java. In addition, users can embed custom MapReduce scripts written in any language using a simple row-based streaming interface, i.e., read rows from standard input and write out rows to standard output. This flexibility does come at a cost of converting rows from and to strings.

### 2.1.4.2 Apache Pig

**History**

Apache Pig is a high-level platform for querying large semi-structured data sets using Hadoop and the MapReduce model. The language for this platform is called Pig Latin. Pig Latin abstracts the programming from the
Java MapReduce model into a notation which makes MapReduce programming high level, similar to that of SQL for RDBMS systems. In other words, Pig simplifies the use of Hadoop by allowing SQL-like queries to a distributed dataset. Pig was originally developed at Yahoo Research in 2006, so as to enable researchers with an ad-hoc way of creating and executing map-reduce jobs on very large data sets. In 2007, Pig was moved into the Apache Software Foundation.

Data Model
As we mentioned before, Pig Latin is a flow language through which users can write programs in Pig. Pig Latin allows the user to write data flows that describe how the data will be transformed. Since Pig Latin scripts can be graphs (instead of requiring a single output) it is possible to build complex data flows involving multiple inputs, transforms, and outputs. Users can extend Pig Latin by writing their own functions, using Java, Python, Ruby, or other scripting languages.

Pig Latin consists of the following datatypes:

- **Data Atom**: is a simple atomic data value. It is stored as a string but can be used as either a string or a number.

- **Tuple**: is a data record consisting of a sequence of “fields”. Each field is a piece of data of any type (data atom, tuple or data bag).

- **Data Bag**: is a set of tuples. Data Bag is equivalent to a “table” in SQL terminology except that Pig does not require that the tuple field types match, or even that the tuples have the same number of fields. Moreover, a Data Bag could contain duplicate tuples.

- **Data Map**: is a map from keys, that are string literals, to values, that can be any data type.

![Figure 2.22: Data Types of Pig Latin](image)

Pig supports the following types in addition to map, bag, and tuples.

- **int**: 32-bit integer, signed type
- **long**: 64-bit for unsigned type
- **float**: 32-bit floating point
- **double**: 64-bit floating point
- **chararray**: set of characters stored in Unicode (Java String)
- **bytearray**: array of bytes (Java DataByteArray)
Architecture
The Pig system takes a Pig Latin program as input, compiles it into one or more MapReduce jobs, and then executes those jobs on a given Hadoop cluster. Pig allows three modes of user interaction:

- Interactive mode: In this mode, the user is presented with an interactive shell (called Grunt), which accepts Pig commands.
- Batch mode: In this mode, a user submits a prewritten script containing a series of Pig commands.
- Embedded mode: Pig is also provided as a Java library allowing Pig Latin commands to be submitted via method invocations from a Java program. This option permits dynamic construction of Pig Latin programs, as well as dynamic control flow, e.g. looping for a non-predetermined number of iterations, which is not currently supported in Pig Latin directly.

Regardless of the mode of execution used, a Pig program goes through a series of transformation steps before being executed, depicted in Figure 2.23. More specifically, the steps followed are [44]:

- Parsing: The parser verifies that the program is syntactically correct and that all referenced variables are defined. The parser also performs type checking and schema inference. Other checks, such as verifying the ability to instantiate classes corresponding to user-defined functions and confirming the existence of streaming executables referenced by the user’s program, also occur in this phase. The output of the parser is a canonical logical plan with a one-to-one correspondence between Pig Latin statements and logical operators, arranged in a directed acyclic graph (DAG).
- Logical Optimizer: The logical plan generated by the parser is passed through a logical optimizer. In this stage, logical optimizations such as projection pushdown are carried out.
- MapReduce Compiler and Optimizer: The optimized logical plan is then compiled into a series of MapReduce jobs, which then pass through another optimization phase.
- Hadoop Job Manager: The DAG of optimized MapReduce jobs is then topologically sorted, and jobs are submitted to Hadoop for execution in that order. Pig monitors the Hadoop execution status, and periodically reports the progress of the overall Pig program to the user. Any warnings or errors that arise during execution are logged and reported to the user.
2.1.4.3 Cascading

History
Cascading provides an open source API for writing Enterprise-scale apps on top of Apache Hadoop and other Big Data frameworks. Its development started at 2007 by Chris Wensel who aimed in the creation of an open source application framework for Java developers to develop robust apps on Hadoop, quickly and easily. Today, in addition to the Java API, support for several other languages has been built atop Cascading, as shown in Figure 2.24 [45]

![Cascading technology stack.](image)

Data Model
Cascading manipulates data stored directly in a distributed file system, usually HDFS. It represents all data as tuples. Tuples are named, ordered fields. After processing the generated data, also represented as tuples, can be stored in HDFS files, printed in standard output or passed as input to other steps of processing. Figure 2.25 provides an simple example of data representation before and after the processing step. In the example input data are tuples of sentences retrieved by documents stored in HDFS and the associated values. In the processing step each sentence is tokenized to single words and the output is the set of tuples consisted of each single word that exists in any of the input sentences combined with the corresponding score, from the sentence that came from.

![Cascading data representation through a simple example.](image)
Architecture

A Flow in Cascading is a data workflow equivalent with a single or a sequence of jobs in the MapReduce model. Similar to Apache Hive and Pig, Cascading efficiently translates the Flow in a series of MapReduce jobs but unlikely those frameworks, it does not provide a query language. The desired data processing procedure is expressed directly through Cascading’s API.

A flow is a DAG consisted of the following nodes: Source Taps, Pipe Assemblies and Sink Taps. Source and Sink Taps are instances where all input data comes from, and all output data feeds to. Taps can be read from, which makes it a “source”, or written to, which makes it a “sink”. Commonly, Taps can act as both sinks and sources when shared between Flows. Taps are represented as classes in Cascading API, with the most common used to be the following:

- **Lfs**: Lfs Tap is used to reference local files. Even if a remote Hadoop cluster is configured, if an Lfs Tap is used as either a source or sink in a Flow, Cascading will be forced to run in “local mode” and not on the cluster.

- **Dfs**: Dfs Tap is used to reference files on the Hadoop distributed file system.

- **Hfs**: HfsTap uses the current Hadoop default file system. If Hadoop is configured for “local mode” its default file system will be the local file system. If configured as a cluster, the default file system is likely the Hadoop distributed file system. Lhs and Dfs subclass the Hfs Tap.

- **Custom Taps**: Furthermore, Cascading API provides users with the ability to write their own custom taps. As an example of Custom Taps usage, someone could consider the scenario of sourcing or sinking data from/to a different filesystem than those for which Cascading provides native support or from/to a database table.

Furthermore, each Tap in a Flow must have a Scheme associated with it. A Tap is about where the data is, and how to get it, the Scheme is about what the data is. Like Taps, Cascading API provides a series of implemented Schemas and the ability to users to built custom ones. The basic implemented Schemas in the Cascading API are the following:

- **TextLine**: TextLine reads and writes raw text files and returns Tuples with two field names by default, “offset” and “line”. These values are inherited from Hadoop. When written to, all Tuple values are converted to Strings and joined with the TAB character (\t).

- **SequenceFile**: SequenceFile is based on the Hadoop Sequence file¹, which is a binary format. When written or read from, all Tuple values are saved in their native binary form. This is the most efficient file format, but being binary, the result files can only be read by Hadoop applications.

A processing step in Cascading is called Pipe Assembly. Pipe assemblies define what work should be done against a tuple stream, where during runtime tuple streams are read from Tap sources and are written to Tap sinks. Pipe assemblies may have multiple sources and multiple sinks and they can define splits, merges, and joins to manipulate how the tuple streams interact. Cascading API provides one base Pipe Assembly class and five more types of Pipe Assemblies that are subclasses of the aforementioned class. Cascading’s Pipe Assemblies are presented in Figure 2.26 and their usage is described next:

- **Pipe**: The Pipe class is used to name branches of pipe assemblies. These names are used during planning to bind Taps as either sources or sinks (or as traps, an advanced topic). It is also the base class for all other pipes described below.

• Each: The Each pipe applies a Function or Filter Operation (explained in the following of this subsection) to each Tuple that passes through it.

• GroupBy: GroupBy manages one input Tuple stream and groups the stream on selected fields in the tuple stream. GroupBy also allows for “merging” of two or more tuple stream that share the same field names.

• CoGroup: CoGroup allows for “joins” on a common set of values, just like a SQL join. The output tuple stream of CoGroup is the joined input tuple streams, where a join can be an Inner, Outer, Left, or Right join.

• Every: The Every pipe applies an Aggregator, like count, or sum or Buffer, e.g a sliding window, Operation to every group of Tuples that pass through it.

• SubAssembly: The SubAssembly pipe allows for nesting reusable pipe assemblies into a Pipe class for inclusion in a larger pipe assembly.

Finally, as apparent from the above, there is a last category in Cascading that complements a Flow to be execute. This category is the Operations that Pipe Assemblies apply to the data. There are four kinds of Operations: Function, Filter, Aggregator, and Buffer. Similarly to the above categories, Cascading provides a number of implemented widely used Operations as well as the ability to the user to write custom Operations. A brief explanation for the usage of each kind of Operation follows:

• Function: A Function expects a single argument Tuple as input, then performs the desired processing to it and may return zero or more result Tuples.

• Filter: A Filter expects a single argument Tuple and returns a boolean value stating whether or not the current Tuple in the tuple stream should be discarded.

• Aggregator: An Aggregator expects set of argument Tuples in the same grouping, and may return zero or more result Tuples.

• Buffer: A Buffer expects set of argument Tuples in the same grouping, and may return zero or more result Tuples. The Buffer is very similar to an Aggregator except it receives the current Grouping Tuple and an iterator of all the arguments it expects for every value Tuple in the current grouping, all on the same method call. [46]

Concluding, Figure 2.27 [47] depicts the Cascading Flow of the classic “word count” example. In Figure 2.27 rectangles represent the Taps. There is one source Tap that provides to the Flow unique documents from a documents collection stored in folders of a, probably distributed, file system. Also, there is a sink Tap that will take the output of the flow in form of tuples consisting of each unique word existed in the document collection.
and the corresponding frequency of the word’s appearance in the collection, and store this information in the files of the same or different file system.

Since source Tap provides the input to the flow an Each pipe performs a Function that splits each document’s sentence into single words, depicted by the first circle in the Figure. The steps described in the previous sentence will be translated to the Map phase of a MapReduce job by Cascading. The Reduce phase begins by a GroupBy Pipe, the cube in the Figure, which groups all the same words with their frequencies together. Next, an Every Pipe performs an Aggregator on every unique word’s tuples, which sum all the corresponding frequencies, depicted by the second cycle. The Reduce phase ends when the Sink Tap finish.

2.1.4.4 Cascalog

As it is depicted in Figure 2.24, a variety of tools and libraries have been built on top of Cascading. Those tools and libraries are implemented and they used through different programming languages. The most widely used tools of this category are Scalding and Cascalog. Scalding is a domain-specific language (DSL) in the Scala programming language\(^1\), which integrates Cascading. It was developed by Twitter and its main contribution is that it simplifies, in great amount, the code for a Flow exploiting the functionality of Scala to avoid the details of Cascading’s pure Java API.

The remaining of this subsection is about Cascalog. The reason of this thorough presentation of Cascalog instead of any other tool depicted in Figure 2.24 is that this one has been used in the implementation part of the current thesis.

Cascalog is a DSL written in Clojure programming language\(^2\), that implements first-order predicate logic for large-scale queries based on Cascading. Like other Big Data oriented projects (Apache Storm, dfs-datastores Hadoop library) Cascalog was developed by Nathan Marz and his former company, called Backtype. Cascalog provides APIs both in Clojure and Java. The latter API is called JCascalog and it was the one that has been used in the implementation part. According to Cascalog documentation the project’s main contributions are summarized in the following list [48]:

- Simple: Functions, filters, and aggregators all use the same syntax. Joins are implicit and natural.
- Expressive: Logical composition is very powerful, and user can run arbitrary Clojure code in their query with little effort.
- Interactive: Run queries from the Clojure REPL (read–eval–print loop, a command line programming environment).

\(^{1}\)http://www.scala-lang.org/
\(^{2}\)http://clojure.org/
• Scalable: Like Cascading, Cascalog queries run as a series of MapReduce jobs.

• Query anything: Query HDFS data, database data, and/or local data by making use of Cascading’s “Tap” abstraction. Also, Cascalog has a feature called “non-nullable variables” that makes careful handling of null values.

• First class interoperability with Cascading: Operations defined for Cascalog can be used in a Cascading flow and vice-versa.

• First class interoperability with Clojure: Cascalog can use regular Clojure functions as operations or filters, and since it is a Clojure DSL, it can be used in other Clojure code.

Cascalog’s data model is similar to Cascading’s one. Cascalog works by manipulating and transforming tuples, named lists of values, where each value can be any type of object. A set of consistent tuples share a Schema which specifies how many fields are in each tuple and the name of each field. From the perspective of the data processing architecture Cascalog differs from Cascading in the notion that Cascalog user Queries instead of Pipe Assemblies and Predicates instead of Operations. A simple data processing procedure expressed as a Query in JCascalog is presented in Figure 2.28.

![Figure 2.28: A simple Cascalog query that prints the names of people from table AGE that are younger than 30 years old.](image)

Cascalog translates queries in MapReduce jobs in a high level abstraction. This means that it disassociates the specification of the computation from how it executes. A MapReduce job implemented by the classic model can be expensive as it involves launching tasks, reading data to and from disk, and streaming data over the network. Also, users need to execute the code in as few MapReduce jobs as possible to maximize performance. Rather than running each portion of the computation as its own job, Cascalog using a high level abstraction compiles the desired computation into a minimal number of MapReduce jobs, packing functions into the same Mapper or Reducer whenever possible. This way Cascalog manages to avoid latency that is caused from external (to the processing query) factors and gains maximum efficiency.

Figure 2.29 depicts the aforementioned efficiency by presenting the execution of a variation of the classic “Word Count” example where only the occurrences of words that the length of their characters are above a threshold, will be count, i.e, a filter operation on the words is preceding. [49]

2.1.5 Data Serialization Frameworks

From the previous subsections it has been clear that as the field of “Big Data” becomes dominant, both in academia and industry, a large variety of tools and frameworks is implemented regarding the data storage and processing. Today complex architectures consisting of clusters that execute those frameworks, are proposed. These architectures demand the interaction of frameworks written in different languages and executed in remote machines. Consequently, there is a need for tools that take over the task of cross-platform, language independent serialized data transfer between different points of a complex architecture. Those tools are called Data Serialization Frameworks or Remote Procedure Call Frameworks. The most widely used frameworks of this category are listed below:
Google’s Protocol Buffers: Google developed Protocol Buffers for use internally and has made protocol compilers for C++, Java and Python available to the public under a free software, open source license. Various other language implementations are also available. Protocol Buffers are widely used at Google for storing and interchanging all kinds of structured information. Protocol Buffers serve as a basis for a custom remote procedure call (RPC) system that is used for nearly all inter-machine communication at Google.

Apache Avro: Avro is a remote procedure call and serialization framework developed within Apache’s Hadoop project. It uses JSON for defining data types and protocols, and serializes data in a compact binary format. Its primary use is in Apache Hadoop, where it can provide both a serialization format for persistent data, and a wire format for communication between Hadoop nodes, and from client programs to the Hadoop services. Though theoretically any language could use Avro, the following languages have APIs written for them: Java, Scala, C#, C, C++, Python and Ruby.

Apache Thrift: Thrift is used as a remote procedure call (RPC) framework and was developed at Facebook for scalable cross-language services development. It combines a software stack with a code generation engine to build services that work efficiently to a varying degree and seamlessly between Delphi, C#, C++, Cappuccino, Cocoa, Erlang, Go, Haskell, Java, OCaml, Perl, PHP, Python, Ruby, Node.js and Smalltalk. Although developed at Facebook, it is now an open source project in the Apache Software Foundation.

The three aforementioned frameworks are very similar to each other. Some notable differences between them is that Protocol Buffers do not not include a concrete RPC protocol stack¹ to use for defined services, like the other two and Avro does not require running a code-generation program when a schema changes, something that is essential to Thrift. For the remaining of this subsection Data Serialization Frameworks will be further presented based on Apache Thrift, as it was the one used in the implementation part of this thesis.

¹http://en.wikipedia.org/wiki/Protocol_stack
2.1.5.1 Apache Thrift

Thrift as a software library and set of code-generation tools developed to expedite development and implementation of efficient and scalable backend services. Its primary goal is to enable efficient and reliable communication across programming languages by abstracting the portions of each language that tend to require the most customization into a common library that is implemented in each language. Specifically, Thrift allows developers to define datatypes and service interfaces in a single language-neutral file and generate all the necessary code to build RPC clients and servers. A presentation of the key components of thrift follows [50].

Types
The goal of the Thrift type system is to enable programmers to develop using completely natively defined types, no matter what programming language they use. By design, the Thrift type system does not introduce any special dynamic types or wrapper objects. It also does not require that the developer write any code for object serialization or transport. Thrift types are categorized as follows:

- **Base types**: Thrift supports key types available in all programming languages like `bool`, `byte`, `i16`, `i32`, `i64` (16, 32 and 64 bit signed integers), `double`, `string`.

- **Structs**: A Thrift struct defines a common object to be used across languages. A struct is essentially equivalent to a class in object oriented programming languages. A struct has a set of strongly typed fields, each with a unique name identifier.

- **Containers**: Thrift containers are strongly typed containers that map to the most commonly used containers in common programming languages. There are three types available `list <type>` an ordered list of elements, `set<type>` an unordered set of unique elements and `map<type1,type2>` a map of strictly unique keys to values.

Also thrift provides types for exception handling, that are similar to structs, and services which are equivalent to interfaces and/or abstract classes in Object Oriented languages.

Transport
The transport layer is used by the generated code to facilitate data transfer. Fundamentally, generated Thrift code only needs to know how to read and write data. The origin and destination of the data are irrelevant. The Thrift transport interface, called `TTransport`, supports the following methods: `open`, `close`, `isOpen`, `read`, `write`, `flush`. In addition, there is a `TServerTransport` interface used to accept or create primitive transport objects and supports methods such as `open`, `listen`, `accept`, `close`.

Protocol
Another major abstraction in Thrift is the separation of data structure from transport representation. Thrift enforces a certain messaging structure when transporting data, but it is agnostic to the protocol encoding in use. That is, it does not matter whether data is encoded as XML, human-readable ASCII, or a dense binary format as long as the data supports a fixed set of operations that allow it to be deterministically read and written by generated code. The Thrift Protocol interface fundamentally supports two things, bidirectional sequenced messaging and encoding of base types, containers, and structs.

Versioning
Versioning and data definition changes are critical to enable staged rollouts of changes to deployed services. The system must be able to support reading of old data from log files, as well as requests from out-of-date clients to new servers, and vice versa. Versioning in Thrift is implemented via field identifiers. The field header
for every member of a struct in Thrift is encoded with a unique field identifier. The combination of this field identifier and its type specifier is used to uniquely identify the field. The Thrift definition language supports automatic assignment of field identifiers.

**Processors**

The last core interface in the Thrift design is called TProcessor. The key design idea is that the complex systems users build can be broken down into agents or services that operate on inputs and outputs. In most cases, there is actually just one input and output (an RPC client) that needs handling. The TProcessor interface is very general by design. Thrift allows the application developer to easily write any type of server that operates on Protocol objects, for instance, a server could simply stream a certain type of object without any actual RPC method invocation.

### 2.1.5.2 Dfs-datastores and Thrift

In the spirit of making the procedures of Big Data analysis more abstract so developers concentrate their efforts more in the efficiency of query processing and less in efficiency of data physics relative to retrieving, storing and organizing a dataset, Nathan Marz, the creator of Storm and Cascalog, developed a library called dfs-datastores. According to Marz dfs-datastores provides dramatically simpler and more powerful way to store records on a distributed filesystem. As a Java library and not a complete independent framework, the presentation of dfs-datastores does not fit in any of the previous sections about Big Data Frameworks. However, dfs-datastores is worth-mentioning, especially in this point as it is fully compatible to handle data expressed as Thrift objects and it has been used for that reason in the implementation part of this thesis.

While distributed file systems like HDFS are powerful tools for storing big data, there are common tasks required to maintain an application’s core dataset. Those tasks are related to the maintenance of the dataset as regarding to appending new files to the core dataset folder and consolidating data to remove small files. Using Hadoop’s HDFS API for manipulation of the above tasks can lead to lack of robustness and efficiency. Regarding to the updating of the dataset, issues like non-unique filenames or non-standard file formats can cause data loss or even the corruption of the entire dataset. Regarding to the task of consolidating data, first must be explained that there can be an order of magnitude difference in performance between a MapReduce job that consumes, for example 10GB stored in many small files versus a job processing that same data stored in a few large ones. The reason is that a MapReduce job launches multiple tasks, one for each block in the input dataset. Each task requires some overhead to plan and coordinate its execution, and since each small file requires a separate task, the cost is repeatedly incurred. This property of MapReduce triggers the need of data consolidation. This can be achieved by either writing code that uses the HDFS API or a custom MapReduce job, but both will require considerable work and knowledge of Hadoop internals. Instead of using the low-level HDFS API to append new data and consolidate small files, dfs-datastores offers high-level functions for these tasks. [49]

The basic class of dfs-datastores library is **Pail**. Pail is a thin abstraction over files and folders. This abstraction makes it significantly easier to manage a collection of records for batch processing. As the name suggests, Pail uses pails, folders that keep metadata about the dataset. By using this metadata, Pail allows data processing without violating its integrity. A pail can store in HDFS any type of data that can be serialized. This means that Pail can manipulate data directly in forms of instances of a Java class generated by a Thrift object. To be able to store, retrieve, append or consolidate instances of this class to HDFS the only think to be done is the definition of another class that describes the structure of the data on HDFS. This class implements the PailStructure interface from dfs-datastores and has methods about data serialization/deserialization and other that are used for defining the valid forms of the paths that can store the specific type of data on the file system.

Figure 2.30 presents a minimal example of a Thrift object, a part of the corresponding Java translation of this Thrift object and the code Pail needs to append data of this class from a directory to another and perform
consolidation to the final destinations. It is clear that for the implementation of these tasks while the HDFS API would need dozens of lines of code, dfs-datastores needs only three lines.

Figure 2.30: Pail-Thrift simple example.

2.1.6 The Lambda Architecture

As it said before, the frameworks presented in previous sections are combined in many variations to form complex architectures that allow companies and organization to extract useful insights from the great amounts of data they collect. One of the proposed architectures is called the Lambda Architecture. It is a well defined combination of at least one framework of each category as they presented in this chapter.

Lambda architecture, proposed by Nathan Marz the main developer of Storm, Cascalog and dfs-datastores, provides a clear set of architecture principles that allows both batch and real-time or stream data processing to work together, so as to provide useful insights from the processed data, usually referred as views, while building immutability and recomputation into the system. Lambda architecture is split into three layers, the Batch layer, the Serving layer, and the Speed layer. The Batch layer processes high volumes of data where a group of transactions is collected over a period of time. Data is collected, entered, processed and then batch results produced. Batch processing requires separate programs for input, process and output. In contrast, real-time data processing involves a continual input, process and output of data. Data must be processed in a small time period or near real-time.

Figure 2.31 depicts an abstract design of Lambda architecture and the corresponding steps that take place in the flow of data processing. Next follows a discussion about each step.

Step 1: Data streams Initially, all data entering the system is dispatched to both the Batch layer and the Speed layer for processing. Data in this step have the raw form from their source. Depending on the usage scenario the data sources can vary. A few examples of potential data sources for the Lambda Architecture could be the Twitter¹ or any other organization’s streaming APIs, any kind of sensors, the high-rate generated log files of an organization, stockmarket and many other.

---

¹https://dev.twitter.com/docs/streaming-apis
Step 2: The batch layer  
This is the first of the three layers which compose the Lambda Architecture. The Batch layer has two functions. The first function is managing the master dataset, an immutable, append-only set of raw data. The second function is to compute arbitrary views from this dataset. Among the existing frameworks, Hadoop and its implementation of the MapReduce model seems to fit better for the batch layer. According to this, referring to the first function, managing the master dataset, i.e., storing, updating etc., is taking place on HDFS. As concerns the second function, computing the views, this is a continuous operation, so when new data arrives it been aggregated into the views when they are recomputed during the next MapReduce iteration.

The views should be computed from the entire dataset and therefore the batch layer is not expected to update the views frequently. Depending on the size of dataset and cluster, each iteration could take hours.

Tools like, Apache Pig, Apache Hive, Cascding and Casalog are suitable for the MapReduce computations that take place in this layer. Also storing data in the form of Apache Thrift, Apache Avro or Protocol Buffer objects is strongly recommended for the ease of processing.

Step 3: The serving layer  
The output from the batch layer is a set of flat files containing the precomputed views. The serving layer is responsible for indexing and exposing the views so that they can be queried.

One of the challenges of the architecture is being able to serve low-latency queries under high throughput, e.g. multiple concurrent users, for a dataset that can be GB or even TB worth of data. Because the calculation of almost all the statistics takes place in Hadoop, there is a necessity of some kind of bridge between it and a fast database. This is a task that can be handled either with tools like Apache Drill or with one of the aforementioned query processing frameworks. As concerns the database, regarding to the application that uses the Lambda architecture, a document-oriented database like HBase or Cassandra or a key-value datastore like ElephantDB, in combination with an in-memory key-value datastore like Redis are suitable to serve the scope.

Step 4: The speed layer  
In essence the speed layer is the same as the batch layer in that it computes views from the data it receives. The speed layer is needed to compensate for the high latency of the batch layer and
it does this by computing realtime views in a framework like Apache Storm or Apache S4. The realtime views contain only the delta results to supplement the batch views.

Whilst the batch layer is designed to continuously recompute the batch views from scratch, the speed layer uses an incremental model whereby the realtime views are incremented as and when new data is received. The efficiency of the speed layer is summarized to the fact that the realtime views are intended to be transient and as soon as the data propagates through the batch and serving layers the corresponding results in the realtime views can be discarded. This is referred to as “complexity isolation” [49], meaning that the most complex part of the architecture is pushed into the layer whose results are only temporary.

**Step 5: Quering views** This step depicts the user interaction with the Lambda architecture. Any incoming query can be answered by merging results from batch views and real-time views.

### 2.2 Social Network Analytics

Since the launching of MySpace¹ in 2003 and onwards, the consolidation of social media was a fact. Within a very short time period, social media became highly popular and moreover they were established as the primary means of communication for young people within the Word Wide Web. Soon, social media gained the attention of older members of the society too. Thus, nowadays social media have become part of the culture of all developed societies.

Therefore, it makes perfect sense that both businesses and the scientific community has turned its sights on the data generated by social media users. The reason for this is obvious. The social media stream contains huge amount of information about every aspect of life and current affairs. Mining and performing analysis on social media stream leads in a variation of useful outcomes. For instance, social media data may contain several nuggets of information that could be valuable for policy makers, marketing departments and others looking for emerging trends and attitudes.

As a consequence a well known technique in scientific field of modern sociology has revived in, the Social Network Analytics (SNA). The term SNA is related to the study of human relationships and/or activities by means of graph theory [51] . Social network analysis views social relationships in terms of network theory, consisting of nodes (representing individual actors within the network) and ties (which represent relationships between the individuals, such as friendship, kinship, organizational position etc.) [52] [53]. In particular, as far as computer science is concerned SNA is referred to the practice of gathering data from social media websites and analyzing that data to find patterns and make decisions [54]. There are five areas of SNA research that have attracted a great deal of attention [55].

1. Stochastic actor-based models. These models are based on sociological literature and are used in order to examine the dynamics and evolution of networks.

2. Scale free networks and complex systems.


5. Designation of the association between network structure and the performance of the actors embedded in the networks based on management science and sociology.

¹urlhttps://myspace.com/
In this thesis we are going to deepen into the area of “Social Network Analytics”. Specifically, we are going to study the problem of real-time prediction of popular topics, emerged in social media. Popular topics are very significant by their nature, as they give instant stats to what concerns the people at a particular time. It is no coincidence, that the three most popular social media Twitter, Facebook and Google+, provide lists of the most popular topics which concern their users. It goes without saying, that this popular topics, through their visibility, contribute to the collective awareness of what is trending and at times can also affect the public agenda. Due to their great importance, particularly in the marketing sector, their characteristics as well as their prediction have been extensively studied by research community.

2.2.1 Related Work on Trend Prediction and Time Series Classification

A popular approach in detecting emerging popular topics in text data streams is to observe the evolution of a term/topic frequency in time, namely to treat each term/topic as a time series. Within this approach, a subcategory of researches attempts to exploit times series special characteristics. For instance Lu and Yang [56], in order to detect emerging topics, employed a widely used indicator in stock technique analysis MACD (Moving Average Convergence-Divergence) [57]. They monitor the difference between the moving averages of topic in two different time windows and according to its value they predict if a topic will rise or it going to die. Along the same lines Ihrer et al [58] employ a time-varying Poisson model in order to detect unusual bursty events. On the other hand, another group of researchers attempted to approach the trend detection problem as an time series classification problem. Thus, in order to classify time series Chen et al [23] created two training sets, one of which corresponds to time series of terms/topics that became trends and one to time series of terms/topics that didn’t became trends. Then, they classify a topic as trend or not trend based on the value of Euclidean Distance of topic’s time series and the time series of trend or not trend training sets respectively. Another viewpoint to the time series classification introduced by Kontaki et al. [59]. They incrementally transform each streaming time series into a vector by means of Piecewise Linear Approximation (PLA) technique and then using this representation clusters the evolving time series.

The former approaches, based on time series, are highly accurate in trend prediction. Although they are generic within the meaning that they detect as trends individual terms which do not have clear semantic information. Thus, another class of methods, which includes content based implementations to address the problem of trend detection, has emerged. Among them, the work of Mathioudakis and Koudas [60], who in their effort to catch evolving trends on Twitter, proposed the following method. First, they identify the keywords that suddenly appear in tweets at an unusually high rate. Then they group bursty keywords into trends, based on their co-occurrences in Twitter Data Stream. In the same concept, Kim et al [61], used NLP and heuristics methods to extract candidate keywords from tweets as well as to merge semantically the similar keywords. As trend keywords they consider the keywords that have the highest term frequency (TF). J. Benhardus and J. Kalita [62] defines a trending topic as a word or phrase that is experiencing an increase in usage, both in relation to its long-term usage and in relation to the usage of other words. Thus they applied NLP techniques, such as raw frequency, relative normalized frequency, tf-idf weighting and entropy to identify trending topics.

Finally, there is a third class of methods, which aim to detect emerging trends in streaming data. The researches which belong to this class employs Document clustering in order to reveal emerging topics. X.Yang et al. [63] build a summary of the streaming data that can fit in a limited memory budget. Thereafter they perform topic modelling by implementing of non-negative matrix factorization (NMF) algorithm¹. Along the same lines, Bakali, et al. [64], implemented a framework, called Cloud4Trends. In their attempt to detect bursty keywords they assign user generated content, obtained from micro-blogging and blogging applications, on clusters divided by topics. After that, bursty keywords are calculated on the emerging topic-clusters. In another approach Lau et al. [65] propose a LDA² topic model that processes documents in an on-line fashion

¹http://en.wikipedia.org/wiki/Non-negative_matrix_factorization
and is capable to cope with dynamic changes in vocabulary. Then, for every iteration, they calculate the degree of change of a topic using the Jensen-Shannon divergence measure between the word distribution of each topic in two successive timestamps and classify a topic as being novel if the measure exceeds a threshold. In the same vein, Kasiviswanathan et al. [66], defined that the emerging topics detection problem are similar with the identification of several recent posts that are both similar to each other, and are dissimilar to previous posts. Thus they propose an approach based on “incremental dictionary learning” to identify the emerging topic clusters.
Chapter 3

Prerequisites

It has been mentioned that the implementation part of this thesis is based on data provided from the Twitter Social Network Service. Thus in the present chapter a piece of information regarding to Twitter’s basic terminology is provided, as well as, to the way that Twitter approaches the issue of trending topics is provided.

3.1 Twitter Network Service

Twitter is an online social networking and microblogging service that was created in March 2006 by Jack Dorsey, Evan Williams, Biz Stone and Noah Glass. Twitter rapidly gained worldwide popularity, with more than 140 million active users in 2012, who posted 340 million tweets per day [67]. Twitter enables users to send and read “tweets”, which are text messages limited to 140 characters. Registered users can read and post tweets but unregistered users can only read them. Users access Twitter through the website interface, SMS, or mobile device application. In addition Twitter has created a “vocabulary” of words describing operations. At this point we will present the entire Twitter vocabulary.

- **Tweet**: A standard message on Twitter containing 140 characters or less.
- **Retweet**: A tweet that has been reshared to all of a users’ followers.
- **Hashtag #**: The # symbol is used to tag keywords or topics in a tweet to make it easily identifiable for search purposes.
- **Mention @**: Tweets can include replies and mentions of other users by preceding their usernames with the @ sign.
- **Follower**: Follower is someone who have agreed to receive someone else’s Tweets through Twitter. If an individual add someone else to the list of people he/she reads, he/she “follows” them. Popularity on Twitter is often measured by the number of followers a person has.
- **Followee or Friend**: From the perspective of user, followees or friends is the group of users, that he/she follows.
- **Handle**: This designates an individual’s username and accompanying URL at http://twitter.com/handle.
- **Feed**: The stream of tweets on a user’s Twitter homepage comprised of all the accounts he/she follows.
- **Lists**: Twitter provides a mechanism to list users an individual follows, into groups or curated lists showing tweets of all the users in the list.
• *Direct Message*: Also called a DM, this represents Twitter’s direct messaging system for private communication amongst users.

### 3.2 Trends Prediction on Twitter

Nature and structure of the Twitter makes it ideal for information diffusion. For instance, when a user sends a tweet relevant with a topic, automatically this tweet will be pushed to his/her followers. Followers in turn can chip in to the debate if the topic interests them, contributing with this manner to the fast diffusion of the current topic. Therefore the mechanism of Twitter enables issues of public interest to emerge i.e become trends while forcing other less popular topics to fade. As a consequence, the automatic detection and analysis of the emerging topics, i.e. the ‘trends’, that appear in Twitter stream in real time, has caught the attention of many research groups and industry. Researchers are trying to use Twitter to predict everything from disease outbreaks [68] and natural disasters [69] to elections [70] and even revolutions [71]. Moreover, in cases like the Arab Spring¹ Twitter can provide an insider view of the events.

Twitter itself has implemented a mechanism in order to identify “trends”, namely topics that have caught the attention of a large proportion of users. More specifically Twitter tracks the volume of terms mentioned in tweets on an ongoing basis and when the volume of tweets about a topic at a given moment dramatically increases, this topic break into the Trends list. Stated another way popularity alone is not enough for a topic to break into the Trends list because Twitter favours novelty over popularity. That is to say, in order to become a trend the velocity of conversation around a specific topic has to increase quickly enough, relative to the baseline level of conversation happening on an average day [72]. Twitter displays the first ten trending topics, globally (Figure 3.1) or with options to filter by location (Figure 3.2), in its main user interface. The total Twitter’s Trends list is available from Twitter Streaming API².

Although, Twitter’s algorithm emphasizes in presenting the evolving trends, within a reasonable time, and not so much in their prediction. Nevertheless the area of event prediction in data streams and more specifically trend detection on Twitter stream has has been studied intensively by many research groups. Some of these researches has been presented in the previous chapter, in the section 2.2.1.

---

¹ [http://en.wikipedia.org/wiki/Arab_spring](http://en.wikipedia.org/wiki/Arab_spring)
² [https://dev.twitter.com/docs/streaming-apis](https://dev.twitter.com/docs/streaming-apis)
Figure 3.1: Top-10 global trends instance.

Figure 3.2: Top-10 location-based trends instance.
Chapter 4

Twitter trending topics prediction framework

In the implementation part of this thesis the subject is the development of a distributed framework consisted of some of the tools presented in chapter 2 and the execution of algorithms, inside this framework, that aim in early trending topic prediction on Twitter. Figure 4.1 represents this framework and the execution flow for trending topic prediction.

In this chapter, the structural components of the implementation part are presented, from the perspective of:

- the technical details of the infrastructure on which the whole framework has been built upon
- the data that have been used for the execution and the evaluation of the procedure
- the algorithms that have been implemented for the method’s execution.

According to the above, this chapter is divided in a series of sections where, initially there is a presentation of the computer nodes that compose the framework. Then follows a discussion about the datasets that have been used, both in the composition of the training set as well as in the real time execution of the presented flow. Finally, there is a thorough explanation of the framework and the execution flow according to their illustration in Figure 4.1.

4.1 Infrastructure

The whole set of the computer nodes that compose the whole framework depicted in Figure 4.1 are virtual machines provided from the GRNET’s¹ cloud service, called Okeanos². The framework is consisted of a Hadoop cluster, a Storm cluster and a single node acting as Redis server. The available resources from Okeanos were 48 CPU cores, 46 GB of RAM and 600 GB of disk storage. In total, 14 virtual machines have been created, using these resources, for the implementation of above clusters and servers.

4.1.1 Hadoop cluster

The Hadoop cluster is consisted of 7 virtual machines. Due to the restricted size of the cluster, one node acts both as master and slave node. This means, that the node runs the NameNode, SecondaryNameNode and

¹https://www.grnet.gr
²https://okeanos.grnet.gr
Figure 4.1: Twitter trend prediction framework.

JobTracker daemons for its master role and the DataNode and TaskTracker daemons for its slave role. The rest 6 nodes of the cluster act as slave nodes with each one of them running the DataNode and TaskTracker daemons.

The specifications of each node of in the Hadoop cluster are the following:

- Operating System: Ubuntu Server 12.04, 64 bit
- Hadoop version: Apache Hadoop 1.2.1
4.1.2 Storm cluster
The Storm cluster is consisted of 6 virtual machines. One of them acts as Nimbus (master) server while the remaining 5 act as workers (slaves). Furthermore, one of the worker nodes performs the duties of the Zookeeper server.

The specifications of each node of in the Storm cluster are the following:

- Operating System: CentOS 6.4, 64 bit
- Storm version: Apache Storm 0.8
- Zookeeper version: 3.4.5-cdh4.5.0
- Java version: OpenJDK 7
- Disk size: 10 GB per node
- RAM size: 2 GB per node

4.1.3 Redis server
The Redis server is consisted of one virtual machine. The server runs simultaneously 2 instances of the Redis in-memory key-value datastore, with each one of them listening on different port. The Redis server’s specifications are:

- Operating System: Ubuntu Server 12.04, 64 bit
- Redis version: Redis 2.8.4
- Java version: OpenJDK 7
- Disk size: 40GB
- RAM size: 6 GB

4.2 The datasets
The datasets that have been used for the execution of the implemented algorithms on the current framework emanate from Twitter. More specifically, the datasets are divided in 2 categories. The first one is consisted of data that have been used for the training set composition. The second category is consisted of the data on which the real time execution and the evaluation of the system will be based on. In both cases the Twitter streaming API\(^1\) is used for the collection of the data.

Data that compose both of the 2 categories are further divided in two subcategories. The first one concerns a real time sample of the posted tweets and the second one concerns the trending topics discussed on Twitter a few moments before the time they provided. A more extended presentation of the information provided by the Twitter API in relevance of those 2 subcategories is following:

\(^1\)https://dev.twitter.com/docs/streaming-apis
Tweets sample  At any time the Twitter streaming API provides a sample of the tweets posted by the total public accounts of Twitter. This sample is about 1% of the total traffic. The sample of the tweets provided from the Twitter streaming API is a continuous stream of JSON formatted objects, each one corresponding to one posted tweet, in which thorough information about the tweet is included with the most notable fields to be the following²:

- contributors: If other users, except the one that make the post, have contributed to the authorship of the tweet, a collection of their user id and the screen name on Twitter is included in the Tweet’s JSON object.

- coordinates: If user provides information about the geolocation of the posted tweet, this information is returned as a collection consisted of the corresponding latitude and longitude.

- created_at: The Coordinated Universal Time (UTC) when the tweet was created.

- entities: The sets of entities which have been parsed out of the text of the Tweet. Entities are divided in the user mentions, the hashtags and the URLs included in the tweets.

- favorited: A boolean value that indicates whether the tweet has been characterized as favorite from other users. The characterization of a tweet as favorite is a functionality provided by the Twitter web user interface.

- favorite_count: If the above value is true, this field indicates the number of times the corresponding tweet has been “favorited”.

- id: An 64 bit integer representation of the unique identifier for the Tweet.

- id_str: The string conversion of the above identifier.

- in_reply_to_*: If the represented Tweet is a reply to another tweet, a variety of fields is provided regarding to the screen name of the original tweet’s author, the 64 bit integer and the string representation of the original tweet’s identifier and the the 64 bit integer and the string representation of the original tweet’s author’s identifier.

- place: When present, indicates a collection of geolocation information with which the tweet is associated, but not necessarily originating from.

- retweeted: A boolean value that indicates if the tweet has been retweeted by other users.

- retweet_count: If the above value is true, this fields provides the number of times the tweet has been retweeted.

- text: The actual UTF-8 text representation of the tweet post content.

- user: A collection providing thorough information about the account of the tweet’s author. User’s screen name, id, date of account creation and many other fields are included in this collection.

²https://dev.twitter.com/docs/platform-objects/tweets
**Twitter trends**  
Except of the sample of tweets being posted, Twitter API provides a variety of other information. Geolocated trending topics are piece of information provided by Twitter API. Users can retrieve trending topics by a set of available cities, countries or worldwide. The trending topics are updated in a time range of 5 to 15 minutes. Top-10 of geolocated trending topics are provided in JSON format. For the worldwide region Twitter API return the 5 most discussed hashtags and the 5 most discussed topics that are not mentioned as hashtags but they are n-grams usually consisted of 1 to 5 words. The most notable fields of the JSON format that concerns the sets of the trending topics is the following:

- **created_at**: The time that Twitter announces the specific set of the 10 most discussed topics.
- **trends**: A JSON array that provides information about the 10 trending topics. The order of the topics in this array is relative to their ranking. More specifically, some of the fields provided for each topic are:
  - **name**: The term or the n-gram that describes the trending topic as it is posted in tweets.
  - **query**: The query parameter that can be used to search for the topic on Twitter Search\(^1\)
  - **url**: The Twitter Search URL.

For example, for a trending hashtag with name “#trend” the corresponding query field would have value “%23trend” and the url field would have value “http://twitter.com/search/?q=%23trend”.

- **as_of**: The time the set of trends retrieved from the application that uses the Twitter API.
- **locations**: Provides information about the name of the region that is related with the set of trending topics, for example Thessaloniki, Greece or Worldwide, and the corresponding “Yahoo! Where On Earth ID” (woeid)\(^2\) for this region.

### 4.2.1 The training set

The provided sample of the tweets has been collected for a time window of 1 month, from the 12th of November, 2013 to the 12th of December, 2013. The total size of the collected tweets exceeds 300 GB. Alongside, for the same time window of one month, all the provided Twitter trending topics have been collected. As the period that a topic is trending cannot be predetermined, some topics may be present at different trending sets in consecutive time or not. For the specific time window, a little more than 1100 unique trending topics, which appeared in the text of some of the collected tweets sample, have been collected. The parallel collection of the sample of tweets and the trending topics aims in the construction of time series for a subset of the trending topics and a subset of topics that never trended. The time series are constructed by dividing the time window of one month in time buckets of 2 minutes. Each time bucket’s value is related to the number of occurrences of the examined topic in the text of the sample of tweets posted in the corresponding time window of 2 minutes. The way of the time series creation and how they have been used is discussed in detail in a next section.

### 4.2.2 Data for real time execution and evaluation

For the execution of the implemented algorithms, the tweets sample provided by the Twitter streaming API is collected and being processed in real time. For the task of Twitter trends prediction, hashtag entities are extracted from the collected tweets and each hashtag is associated with a sliding window divided into 2-minutes buckets that represents the time series of the hashtag, similar to the time series of the training set’s topics. This time series are compared with the ones of the training set and then the algorithm classifies a hashtag as trending topic or not. Detail for the aforementioned procedure are presented in the next section. Also trending topics

\(^1\)https://twitter.com/search-home  
\(^2\)http://developer.yahoo.com/geo/geoplanet/
are collected, as described above, for the whole time of the algorithm’s execution for purposes of evaluation of the hashtags classification.

4.3 The implemented framework

The implemented framework of the current thesis is strongly inspired by the Lambda Architecture that has been presented in chapter 2. Similarly to the Lambda Architecture, the implemented framework is consisted of a batch layer, a speed layer and a serving layer. However, the key difference between Lambda Architecture and the current framework is the execution work flow.

According to the suggested usage of the Lambda Architecture, both batch and speed layers are fed with the same data units coming from a data stream. Batch layer executes periodic processing on the total of the acquired, until the specific time, data, while speed layer performs the same processing, in real time, in the portion of data that acquired while the former layer is busy. Finally, the results of the batch layer processing that are stored in the serving layer are being aggregated with the results of the serving layer processing and the outcome are global views of the dataset according to specific queries.

On the contrary, the data processing procedure on the batch and speed layers of the implemented framework are not the same. The batch layer have been used for two different purposes. The first one concerns the creation of the training set. More specifically, the benefits of Hadoop from the perspective of its distributed filesystem (HDFS), as well as from the perspective of the parallel processing with its implementation of the MapReduce model, have been exploited for the scopes of storage and processing of the big volume of the collected sample of tweets from which the training set has been created. The execution of this task is asynchronous with the execution of the speed layer.

The second purpose of the batch layer usage is the periodic update of the training set. This step requires access both to the results of the speed layer’s processing that are stored in the service layer and to the Twitter’s trending topics that have been retrieved through the Twitter API since the time the speed layer started its execution. Alongside with the execution of this task by the batch layer, speed layer performs the task of topics classification into trends and not trends through a series of execution steps.

Figure 4.1 depicts the aforementioned layers of the framework and the total execution flow. In the remaining of this section a thorough explanation of each discrete step of the execution flow, according to Figure 4.1, follows.

4.3.1 The generation of the training set

According to the problem definition, presented in chapter 1, the task of early Twitter’s trending topic prediction is performed by calculating the distance between the real time created time series of topics discussed on Twitter in specific time ranges and preconfigured time series of topics that exist in tweets of the aforementioned 1 month long time window. Time series of the topics are related to their occurrence in tweets on specific time slots. For the creation of the time series of the topics that they belong to the training set the procedure that have been followed is depicted on steps 1 to 4 in Figure 4.1. The details of each step’s execution is following.

**Step 1** After the collection of the tweets sample of 1 month using the Twitter API have been completed, the JSON-formatted data are stored in HDFS. The whole dataset is initially stored in a single folder on HDFS. After this step has been completed, a transformation of the data follows. A Thrift object has been developed to contain the whole useful information for a tweet. Information is retrieved from the JSON-formatted tweet object. When the task of transformation of each tweet to a Thrift object has been completed, those objects are stored in HDFS in a more efficient way using the Pail class of the dfs-datastores library, replacing the JSON-formatted files and the folder that contains them.
The reasons that led in the usage of the two aforementioned tools are two. The first one is the fact that one of the subjects of this thesis is the overview of new tools and technologies related to Big Data analysis. The second reason is the convenience provided by those tools in the task of Big Data manipulation, compared to complex issues that arise from the low-level nature of Hadoop’s native API.

The structure of the Thrift object is described in the following list, while the whole Thrift file from which the corresponding Java classes are generated is provided in Appendix A, Section 1.

- **TweetType**: This is an enumerate type where the possible values are *TWEET* for original tweets or *RETWEET* for retweets.

- **Text**: A struct with two fields, required a string that stores the original text of the tweet as it is provided by the API and another optional string that store the tweet’s text after potential cleaning, i.e, removal of stopwords and/or emoticons, stemming etc.

- **Location**: If geolocation information is provided for a tweet, this struct stores the corresponding latitude and longitude as double variables.

- **User**: This struct stores information about the author of a tweet. It is consisted of a required field of type *i64*, that is translated to Java language’s *long* type and represents the user’s identifier on Twitter. Furthermore this struct has 4 optional fields that can potentially store, as strings, the author’s Twitter name, the screen_name, the date that the account has been created and the user’s description if it is provided. Also there is an optional boolean field that can store information about whether the user is officially verified by Twitter or not and a last field which type is a Location struct, as described above, and stores information about the user’s location if provided.

- **URLEntity**: A struct that stores information about a url entity provided in tweet’s text. It is consisted of a required string that stores the url as it is posted and three optional strings. The need of the optional strings arises from the fact that due to the restricted limit of a tweet post users usually make use of url shortener services which provide a short url that redirects to the original one. So for that reason this struct has an optional string field that could store the tweet as it is displayed in the Twitter web interface, another one for the expanded version of the url as it is provided by the API and, because the aforementioned expanded url is often not trust-worthy, there is a last optional string that stores the really expanded url obtained after a manual redirection.

- **MentionEntity**: A struct that stores information about a user that probably is mentioned in the tweets text. The struct has a required field of type *i64* that stores the mentioned user’s Twitter identifier and two optional strings, one for the mentioned user’s registered name on Twitter and another one for his screen name.

- **HashtagEntity**: This struct is consisted of a required string that stores a hashtag’s name when it is included on a tweet’s text.

- **Language**: If the language’s code of the tweet’s text is provided from the Twitter API, this struct store the code in a string. Language’s codes can be, for example, *en* for English, *gr* for Greek, etc.

- **NamedEntity**: A struct to store named entities extracted from the tweet’s text using a library with functionality for Named Entities Recognition (NER)¹, like Apache OpenNLP².

- **Date**: This structure has a required field which stores the date that the tweet posted as a string.

---

² [https://opennlp.apache.org/](https://opennlp.apache.org/)
Lists: Thrift provides list data type. Lists store multiple objects of the same type. As the entities described above could be more than one of each type, four lists have implemented to store all the objects of the corresponding entities.

Tweet: This is the basic struct of the implemented Thrift object that aggregates the information of a tweet. This struct has four required fields, a custom identification number for the tweet that takes incremental values of type \textit{i64}, a field of type User struct that represents information for the tweet’s author, a field of type TweetType struct and a field of type Date struct. Furthermore, this struct has 6 optional fields. One field is for the tweet’s text and is of type Text struct and another one is of type Date struct. The four remaining optional fields are of type List, and each one corresponds to a list of those presented above.

Thrift objects that represent tweets are stored in pails in HDFS. For this task dfs-datastores library have been used. The advantage of vertical partitioning offered by dfs-datastores library allows Thrift object to be stored in an efficient and user friendly structure.

Vertical partitioning is the task of storing data in the batch layer in different folders, according to a characteristic of the data and in a sense of data classification. Although the batch layer is built to run functions on the entire dataset, many computations don’t require looking at all the data. The batch storage should allow data partitioning so that a function only accesses data relevant to its computation \[49\]. The key characteristic for data vertical partitioning in this implementation is the date that each tweet has been posted. Figure 4.2 shows the tree structure of the directories and subdirectories that store the Thrift objects in HDFS.

The succession of vertical partitioning of the pail files that store Thrift objects requires the implementation of 3 classes. The base class should implement the \textit{PailStructure} interface provided by the dfs-datastores library. The purpose of the PailStructure interface is to provide methods for serialization and deserialization of data will be stored or retrieved from HDFS and methods that performs validation check of the directories that will store data, according to the vertical partitioning schema. The name of the base class is \textit{ThriftPailStructure} and the task of serialization and deserialization of the Thrift object is performed through the corresponding built-in classes of the Java language’s Thrift library, named \textit{TSerializer} and \textit{TDeserializer} respectively.

The second implemented class is a subclass of ThriftPailStructure, named \textit{TweetPailStructure} and is responsible to implement methods that return new objects of the type that will be stored in the pails and the type as well, which in this case is a class named \textit{Tweet} and is generated from the homonym struct declared in the Thrift file.

The last implemented class is the \textit{SplitTweetPailStructure} a subclass of the TweetPailStructure. It provides methods for the construction of the directories that will store the pail files and for validation check of this directories. In other words, this class provides methods related to the vertical partitioning.

The code of the PailStructure interface and the three implemented classes can be found on Appendix A, section 2.

Step 2: It has been already mentioned that for the task of early prediction of Twitter’s trending topics, time series comparison is taking place. This idea has been proposed by Stanislav Nikolov in [1], his master thesis on MIT. For the implementation of his method, Nikolov acquired a 10% sample of the whole set of tweets posted
in a month and the set of the topics that trended in this month. Then by performing retrospective analysis on the
dataset he created time series for each topic. The details of the time series construction for a topic are discussed
in the next paragraphs. The training set is consisted of the time series of 500 trending topics and 500 topics
that never trended in the duration of the specific month. Thereafter, for each topic in the dataset the distances
between its time series and the 500 trending topics’ time series is calculated and the results are summarized.
The same task is performed between the topics’ time series and the 500 time series that not trended. For each
topic the ratio of the summarized distances is calculated and if the output is above a threshold the topic is
classified as potential trending topic. The evaluation of this method performance on the specific dataset proved
its effectiveness. More specifically, 95% of the actual trending topics were classified as potential trending topics
by the method, while only 4% of the potential trending topics were false positive. Furthermore, 79% of the
correctly classified trending topics, were earlier claimed as trending topics by the method than the Twitter API,
with the mean time to be 1.43 hours earlier. In the current master thesis one of the subjects is to test the
effectiveness of this method in an execution that differs from Nikolov’s in two key points:

- The training set’s time series are constructed by the 1% of the posted tweets in a month, offered by
  Twitter’s API, instead of the 10% sample set of posted tweets for this month.
- Time series of topics to be tested are not static, but they are constructed in real time in a form of sliding
  window. Again, the percentage of tweets that participate in the construction of the time series is 1%.

Time series of a topic is a vector where its positions corresponds to equal sized time buckets. In this case
the time bucket range is 2 minutes. The value of each time bucket corresponds to the number of occurrences
of the topic in the text of tweets posted in the specific 2 minutes window. The final form of the time series
emerges after performing a number of normalization steps on them. For the construction of topic time series
that will compose the training set, vectors that correspond to the whole month of data collection divided to 2
minutes buckets are initially created. This means that for the time period from the 12th of November, 2013
12:00 am to the 12th of December, 2013 12:00 which is equal to 43200 minutes, the size of each topic’s vector
is 21600.

Topic extraction from tweets’ text is implemented through a brute force technique. This technique implies
the extraction of all possible n-grams, with the number of tokens to range from 1 to 5, from each tweet’s
clean text. The maximum limit of the n-grams length have chosen after observing that the largest of the
collected trending topics are consisted of 5 tokens. In this case tweet’s cleaned text arises after the removal of
all stopwords from the original text. Apache Lucene¹ library have been used for both stopwords removal and
n-grams extraction.

Each extracted n-gram is compared with the set of the Twitter trends that have been collected for the
specific month, using Twitter API. If an n-gram found to be a trending topic from the set it becomes candidate
to be part of the trending topics which time series will be in the training set. The reason for not all n-grams
found to be trending topics become part of the training set is due to the fact that for some trending topics the
corresponding tweets from the collected dataset were too few, resulting to sparse time series that would strongly
affect the outcome of the procedure. This fact arises from the small percentage (1%) of the total posted tweets,
that is being offered by the Twitter API.

This limitation caused the first problem in the implementation part of this thesis. According to [1] 500
trending topics’ time series have been chosen to be part of the training set. As it mentioned before, the number
of the unique trending topics that were found on the collected tweets dataset was about 1100. From this 1100
trending topics only 100 have been chosen to be part of the training set as those were the only topics which
corresponding final time series were relatively dense, i.e., they had values at least on the 1/3 of the total time
buckets. The procedure for the creation of the final form of the time series is explained in the next step.

¹http://lucene.apache.org/
Step 3  After each topic’s creation of the time series for the whole 1 month time window, as described above, a number of steps follows for the creation of the final form of time series that will be part of the training set. As it concerns the time series of the trending topics, the method that is proposed in [1] is the following:

- Observation of the exact date that the topic trended from the acquired set of trending topics.
- Removal of the time bucket that corresponds in the observed date and all the time buckets that are next to it.
- Definition of a time range from the moment just before the topic trended, i.e. the time that corresponds to the last kept time bucket, and back. This means that the part of the initially created time series that will be inserted in the training set will be consisted of all the time buckets that correspond to the specified time range until the time the topic trended.
- Removal of all time buckets that precede the one that signifies the start of the aforementioned time range. In this specific implementation the time range is fixed to 4 hours (240 minutes), which means that the length of the training set time series is 120.
- Production of the final form of the time series by applying a set of normalization filters on them.

As concerns the time series of not trending topics that will be part of the training set, the procedure is similar. As there is not a date that a topic of this category trended, the respective time bucket is chosen randomly. Finally, 100 time series of not trending topics are chosen to be part of the training set. For the specific implementation the selection criterion of the 100 not trending topic time series was the density variance. The density of the time series that have been chosen ranges from too sparse to too dense.

The last bullet for the creation procedure of the training set time series is referring to a set of normalization filters. These filters are proposed in [1] and will be presented in the next paragraphs but before that a quick summing up of some concepts discussed before is essential.

Initially, time series of each topic are consisted of two minute time buckets, sequentially placed in a range of one month. The value of a time bucket is the rate of tweets posted in the specific date (including time) that include the topic in their text. Let $\rho[n]$ be the number of Tweets about a topic in the $n_{th}$ time bucket, then the cumulative volume of of Tweets up to time $n$ is:

$$u[n] = \sum_{m \leq n} \rho[m]$$  \hfill (4.1)

Based on the above symbolism the normalization filters will be presented next [1]:

- **Baseline Normalization**: A common phenomenon related to the topics included in posted tweets is that many non-trending topics have a high baseline rate of activity while most trending topics are preceded by little, if any, activity prior to gaining sudden popularity. For example, a non-trending topic such as ‘city’ is likely to have a consistent baseline of activity because it is a common word. To emphasize the parts of the rate signal above the baseline and de-emphasize the parts below the baseline, a baseline $b$ is defined in [1]. This base line is defined as:

$$b = \sum_{n} \rho[n]$$  \hfill (4.2)

so the baseline-normalized value of the $n_{th}$ time bucket is defined by the next equation:

$$\rho_b[n] = \left( \frac{\rho[n]}{b} \right)^\beta$$  \hfill (4.3)

The exponent $\beta$ controls how much rates above and below the baseline rate, will be rewarded and penalized, respectively. In [1] author used $\beta = 1$. This is the value used in the current thesis, too.
• **Spike Normalization:** A common characteristic of topics that become trends is that the rate of tweets referring those topics is suddenly increased for some periods before they characterized as trends. This phenomenon is depicted as spikes in the corresponding time series. Trending topics’ time series typically contain larger and more sudden spikes than that of non-trending topics. Spike normalization filter, as proposed in [1], rewards such spikes by emphasizing them, while de-emphasizing smaller spikes by defining a baseline-and-spike-normalized rate

\[ \rho_{b,s}[n] = \left| \rho_b[n] - \rho_b[n-1] \right|^\alpha \]  

in terms of the already baseline-normalized rate \( \rho_b \). The parameter \( \alpha > 1 \) controls how much spikes are rewarded. In this thesis as in [1], \( \alpha \) is equal to 1.2.

• **Smoothing:** According to [1], tweet rates and the aforementioned transformations thereof, tend to be noisy, especially for small time buckets. To mitigate this, a convolution of time series with a smoothing window of size \( N_{smooth} \), is proposed. Applied to the spike-and-baseline-normalized signal \( \rho_{b,s} \), this yields the convolved version:

\[ \rho_{b,s,c}[n] = \sum_{m=n-N_{smooth}+1}^{n} \rho_{b,s}[m] \]  

The value of \( N_{smooth} \) used in this thesis is equal to 10.

• **Branching Processes and Logarithmic Scale:** The spread of a topic on Twitter is clearly a person to person branching process. A branching process is a model of the growth of a population over time, in which each individual of a population in a given generation produces a random number of individuals in the next generation. While the details of how topic spreads are not known, in a wide generality of branching processes, the growth of the population is exponential with time, with the exponent depending on the details of the model [73]. According to [74] the spread of topics on Twitter can be modeled as a branching process and also propose a logarithmic scaling. Therefore, the final values of each time bucket are produced from the logarithm of each time bucket constructed so far, according to the next equation [1]:

\[ \rho_{b,s,c,l}[n] = \log \rho_{b,s,c}[n] \]  

Figure 4.3 represents an abstract illustration example of the procedure described on steps 2 and 3 for a topic’s time series that would be part of the training set.

**Step 4**  With the 200 time series of the training set created, one last step remains for the real time execution part to be started. This step concerns the storage of the training set time series to the serving layer so they can be accessible from the speed layer. This is necessary for the speed layer to be able to calculate the distances between time series of the training set with those of topics extracted by tweets collected in real time.

The training set time series are stored in the first Redis database of the serving layer. Time series are stored in Redis as key-value pairs where the key is the topic of the corresponding time series with the prefix “+” if it is a trending topic or the prefix “-”, if it is not a trending topic. The value of the pair is a list, a built in Redis data type, that represents the time series.

**4.3.2 Real time execution**

After the construction of the training set, the real time algorithm’s execution in the speed layer can start. As it is depicted in Figure 4.1 this procedure is implemented in the steps 5, 6, 7a and 8a.
A single Storm topology has been developed to execute the implementation part for which the speed layer is responsible. This topology is consisted of a Spout, various Bolts and other auxiliary classes. The role of each of the aforementioned topology elements is thoroughly presented in the next steps.

**Step 5** This step signifies the beginning of the real time execution part of the implemented framework in the speed layer. Topics extraction from real time posted tweets and the creation of the corresponding time series that will be evaluated for the characterization of each topic as possible trend or not is taking place in this part. As figure 4.1 illustrates, the Storm cluster is being provided by an endless stream of posted tweets from the Twitter API.
Step 6  As it happens with all the Storm topologies that are used for production, the implemented topology never terminates until being manually killed. In this step topology’s Spout is being fed from a stream of tweets form the Twitter API. Those tweets are being processed and topics are extracted from them. Then a processing procedure is taking place by a series of Bolts for the real time creation and update of the topics’ time series. The elements of the topology that participate in this task are presented next:

- **TwitterSpout**: This is the unique Spout of the Storm topology. Its source is the stream of tweets provided by the Twitter API. Each tweet that is inserted in the TwitterSpout is being processed, exploiting the functionalities of the Twitter4j library. In this phase, Tweets processing involves topic and date extraction. Due to the restricted dynamic of the Storm cluster, i.e. the limited number of nodes, only hashtag entities presented in a tweet’s text are considered as topics. This happens because topic extraction from the not characterized parts of the tweet’s text using techniques such as the brute force extraction of the n-grams, as it happened in the procedure of the training set creation, or more efficient techniques from the field of Natural Language Processing (NLP), demand higher levels of parallelism, which can be only achieved by adding more nodes to the cluster. According to this situation, TwitterSpout exports a tuple for each hashtag entity included in a tweet’s text consisted of the hashtag itself and the date that the tweet had been posted.

- **RedisTopicsBolt**: This Bolt’s input is the TwitterSpout’s output. As it name suggests this Bolt is responsible for the connection to the serving layer, for the storage of the topic as well as the corresponding time series. Although, before that, the construction of the new inserted topic’s time series or the update of the already existing ones is preceded. Time series constructed in the RedisTopicsBolt are consisted of 180 time buckets, each one represent a 2 minutes time range. This means that the whole time range of a topic’s time series is 6 hours. The first time a hashtag entity (which will be referred as topic for the remaining of this thesis) is inserted in this Bolt, there are to possible situations:

  - The first case concerns topics which are extracted from tweets that had been posted before the first 6 hours since the program started its execution. The initialization of the topic’s time series is performed by calculating the specific 2 minutes time range from the difference between the tweet’s date and the start time of the program. The corresponding time bucket is assigned with value 1, meaning that there is a tweet posted in the specific time range that contains the topic. The rest 179 time buckets are assigned with value 0 in this phase.

  - The second case concerns topics that their first appearance is taking place after the first six hours of the program’s execution. As the time series length is static at 180 time buckets, the completion of the first six times of execution signifies the change of the time series form, from a static time window to a sliding time window. As a result, the first occurrence of a topic in this case signifies the construction of its time series with the first 179 time buckets’ values to be 0 and the value of the last time bucket to be 1.

In the case of the reappearance of a topic, which had appeared previously, thus the corresponding time series has been created, again two cases are distinguished. As it happens with the first appearance of the topic, in this situation the distinction of the cases is associated with whether the topic had reappeared in the first six hours of program execution or later.

  - In the first case, where the topic had reappeared in the first six hours of the program execution, in line with the previous case the following actions are executed so as to the topic time series to be updated. Firstly, the specific 2 minutes time range is calculated from the difference between the tweet’s date and the start time of the program. Next, the corresponding time bucket’s value

¹http://twitter4j.org/
is increased by 1, meaning that there is a tweet posted in the specific time range that contains the topic. The rest 179 time buckets remain intact.

– After the first six hours of program execution, the form of the time series changes from static to sliding window, following a procedure that will be analysed in the following Bolt. Thus, when the form of time series is changed to sliding window, it is now considered that the first 179 time buckets are referred to past tense intervals, while only the last time bucket refers to the present time. Taking all the above into consideration, when a tweet referred to an existing topic will come, the update of the topic’s time series will be done by increasing its 180th time bucket by one.

At this point it is worth mentioning that both the creation of new time series and the update of the already existing ones, takes place directly in the first Redis database of the serving layer. Similar to those described in step 4 for the storing of training set’s time series in Redis, in this case time series are stored as key-value pairs too. The key of the pair is the topic of the corresponding time series while the value of the pair is a list, the built in Redis data type, that represents the time series.

To facilitate understanding, the procedures described above are illustrated through concise examples in Figure 4.4.

• **SlidingWindowBolt** The choice of using sliding windows form for the time series representation, has as its main objective the availability of each topic’s appearance history, in the tweets provided by Twitter API, of the past six hours at any moment. The availability of a part of topic’s appearance history is necessary for the frequently repeated calculation of the distances between the time series of the topics, which popularity is being checked and the time series of the topics that compose the training set. The part of the time series that corresponds to the last 2 hours of the topic’s appearance history participates in each process of the difference calculation. Despite that fact, the availability of the larger time window history, of 6 hours, is necessary for another frequently repeated procedure that is executed in the implemented framework. This is the update of the training set which takes place in the batch layer according to the results that arise from the current execution on the speed layer. This procedure will be explained in later paragraphs.

The renewal process of the time series is done in every two minutes, after the completion of the first six hours of program execution. The renewal process is taken place in the SlidingWindowBolt, where for each topic’s time series the first time bucket is deleted while a new one, with value equal to 0, is added in the end.

• **TimerBolt2min** For the correct timing of the renewal process of the time series, the usage of a timer is necessary. This Bolt implements the functionality of this timer exploiting a build-in feature of Storm, named Tick-Tuple. Tick-Tuple is used whenever a spout or bolt is needed to execute a task at periodic intervals. In the specific implementation, a Tick-Tuple is generated every two minutes from the beginning of the program’s execution. During the first six hours of program’s execution, the emerged Tick-Tuples are ignored. After the lapse of first six hours each Tick-Tuple signifies the communication of Bolt with Redis database and the retrieval of all topics which had been stored till then. Afterwords, the retrieved topics are provided to the SlidingWindowBolt, which is previously described, so that it can update their time series.

**Step 7a** This step is related to the calculation of the distances between the time series of the topics which are under examination and those of the training set. This calculation is a repeated procedure that takes place every 15 minutes. The calculation of the aforementioned distances is of utmost importance. As it mentioned before, the summarized distances between a topic’s time series and the trending topic’s time series of the training set
Figure 4.4: Topic’s time series creation and update example.

and the respective summarized distances between a topic’s time series and the not trending topic’s time series of the training set, are divided and the outcome ratio claims the potentiality of the topic to become trend. The whole procedure is performed by the implementation of the next three Bolts.

- **TimerBolt15min** This Bolt is responsible for two operations. In particular, its first responsibility is to provide to the Bolt that produces the final form of time series that will participate in the calculation procedures, with all the topics that are stored in the first Redis database, at the specific moment. The calculation of the time series’ distances takes place every 15 minutes. This is succeeded with the usage of Tick-Tuples feature.

The second responsibility of this Bolt is to ensure that in every calculation step, the number of trending topic’s time series of the training set, that participate in this procedure, and the respective not trending topic’s ones are equal. This is necessary, because as it mentioned before, while in the beginning of the execution, training set is consisted by 100 trending topic’s and 100 not trending topic’s time series, it is enriched through the steps of training set updates. At any step of the training set update procedure at most 10 time series of successfully characterized trending topics and at most 10 time series of topics that did not become trends are added to the training set, although the number of the time series of each
category is not always equal.

- **TimeBucketsBolt** As it was mentioned before, only 60 of the total 180 time buckets of each topic’s time series participate in the distance calculation procedure. With regard, which 60 time buckets of total 180 will participate in the calculation of the time series’ distance three possible cases are distinguished.

  - The first case regards calculations which are performed before the first 2 hours of program execution has elapsed. In this situation the first 60 time buckets of each time series participate in the calculation.
  
  - The second case regards calculations which are performed in the time period between the 2nd and the 6th first hours of program execution. In this situation, the index of the time bucket that corresponds to the current 2 minutes window is calculated. Therefore, in the time series’ distance calculation the part of time series that participates in, is the aforementioned time bucket along with the presiding 59 time buckets.
  
  - The third case regards calculations which are performed in any time interval after the 6th first hours of program execution. In this situation, the last 60 time buckets of each time series participate in the distance’s calculation procedure.

After the extraction of the appropriate time series part, a density check is performed to it. This happens so as to avoid the meaningless distance calculation in too sparse time series, as they correspond to topics with very low appearance rate. Thus, they are very unlikely to become trending topics in the near future. The criterion for a time series to pass the density check is to contain at least 16% non zero valued time buckets. The four normalization filters that have been presented at Step 3 are performed on the time series that succeed the density check. The outcome of this procedure along with the corresponding topic compose the output tuple of this Bolt.

- **CalculateSimilaritiesBolt** In this Bold all the necessary calculations take place in order to categorize a topic as potential trending topic or not. As it mentioned before, this Bolt takes as input tuples consisted of a topic and its corresponding normalized time series for time series that passed the density check in the TimeBucketsBolt. Each topic is characterized as trending or not according to the result of the following equation, that have been proposed in [1]:

\[
R(s) = \frac{\sum_{r \in R_+} \exp\left( -\gamma \min_{k=1,\ldots,N_{ref}-N_{obs}+1} \sum_{i=1}^{N_{obs}} d(s_i, r_{i+k-1}) \right) + \sum_{r \in R_-} \exp\left( -\gamma \min_{k=1,\ldots,N_{ref}-N_{obs}+1} \sum_{i=1}^{N_{obs}} d(s_i, r_{i+k-1}) \right)}{\sum_{r \in R_+} \exp\left( -\gamma \min_{k=1,\ldots,N_{ref}-N_{obs}+1} \sum_{i=1}^{N_{obs}} d(s_i, r_{i+k-1}) \right) + \sum_{r \in R_-} \exp\left( -\gamma \min_{k=1,\ldots,N_{ref}-N_{obs}+1} \sum_{i=1}^{N_{obs}} d(s_i, r_{i+k-1}) \right)}
\]  

(4.7)

The symbols used in the above equation are explained in Table 4.1. More precisely, the length of the training set’s time series is \(N_{ref} = 120\), corresponding to the 4 hours history of a topic before the moment that it had been announced as trending topic by Twitter if the topic belongs in \(R_+\) or to a random 4 hours window of topic’s history if it belongs in \(R_-\). Respectively, the length of an observed topic’s time series \(N_{obs}\) equal to 60 corresponding to the previous 2 hours history of the topic, from the time the calculation is taking place. From equation 4.7 it can be noticed that the difference in the lengths of the participating time series is covered by calculating the difference between the observed topic’s time series and all the possible sub-parts of the training set’s topic’s time series, which have length equal to \(N_{obs}\). This is been achieved by shifting the former time series to the latter, by a step that is equal to 1, until all possible differences have been calculated. The minimum distance of the ones that had been calculated with the aforementioned way is being kept, as it proposed in [1]. The minimum distances between the observed
A time series of the training set

$\mathcal{R}_+$: The set of time series of the training set that correspond to trending topics.

$\mathcal{R}_-$: The set of time series of the training set that correspond to not trending topics.

$N_{\text{ref}}$: The length of each time series of the topics of the training set.

$N_{\text{obs}}$: The length of each time series of the topics their trending probability is being observed.

$s_i$: The $i$th time bucket of a topic’s time series that is being observed.

$r_i$: The $i$th time bucket of a topic’s time series that belong in the training set.

$\gamma$: A scaling parameter for fine-tuning.

$d$: The distance metric been used, among Euclidean, Squared Euclidean and Cosine distances.

Table 4.1: Explanation of symbols used in equation 4.7

topic’s time series and all the training set’s trending topic time series are summarized. Similarly, the minimum distances between the observed topic’s time series and all the training set’s not trending topic time series are summarized, too. Those two sums are being multiplied by a scaling parameter, $\gamma$, which value was set to 0.1 for the experimentation process. The ratio of the exponential of the above values is being compared against a threshold, which value has been set to 1.

Topics of which, the corresponding ratio is above the threshold, are being passed as input to the final Bolt of the topology.

Step 8a In this step topics that are reported as trending topics after the completion of an execution cycle, as it described in Steps 6 and 7a, are stored to the serving layer. The OutputResultsBolt is responsible to export information about the topic’s that are found to be potential trending topics, to the second Redis database of the serving layer. The information contains the topic itself, the current date which represent the specific time that the topic is being reported as trending topic by the framework, the ratio that had been calculated in the previous Bolt and the part of the topic’s time series that corresponds to the previous 2 hours history of the topic, from the moment that reported as trending topic. The final part of information, i.e. the part of the time series, is useful for the training set update procedure that will be presented in the next steps. The extraction of the appropriate part of the time series for this task is achieved in a similar way to the one that has been presented in TimeBucketsBolt, and it is related to the amount of time that has been passed since the program started its execution. This is the reason that, while only topic’s 2 hours history is participating in the procedure of topic characterization, 6 hours history of the topic is always been stored in the first database.

4.3.3 The training set periodic update

The steps 5-8a compose an endless repeated procedure that is being executed in the Storm cluster and contributes in the task of the early Twitter’s trending topics prediction by the implemented framework. A cycle of this procedure is performed every 15 minutes. Alongside, another repeated procedure, that is taking place in the Hadoop cluster is performed, complementing the the contribution of the total implementing framework for the aforementioned task. The second repeated procedure is related to the enrichment and the update of the training set, which aims in more qualitative results. The need of the execution of this sub-task arises from the low quality of the collected dataset that used for the creation of the initial training set. Recall that the initial training set was consisted of only 100 time series of trending topics, instead of the 500 that are proposed in [1].

Step 7b As it is depicted in Figure 4.1 the extraction of the new trending and not trending topics among with the corresponding time series, that will be added in the training set, contributing in the next execution cycle of
The speed layer, is performed in this step. The input data that are used for this task are coming from 3 different sources and are consisted of:

- all topics that are stored in the first Redis database of the serving layer at the specific time that the execution cycle is performed.
- all topics that have been characterized as potential trending topics from the procedure that has been executed on the speed layer until the aforementioned time.
- all topics that have been reported as trending topics form Twitter and have been collected through the Twitter API, until the aforementioned time.

A series of full outer joins is performed on the above datasets. This joins aim in the extraction of the true positives, false positives and true negatives, from the current results. As true positives are considered the topics that have both been reported from the implemented framework as well as the Twitter as trending topics, with the time of the framework’s report to precede the respective one of Twitter. At each execution cycle of this step at most 10 new trending topic’s with their time series that are available from Step 8a are added to the training set. Respectively, at most 10 not trending topics and their time series are added to the training set at each execution cycle. At most 3 of the aforementioned trending topics emanate from the false positives set, while at most 7 emanate from the true negatives set. False positives set is consisted of topics which while have been reported as potential trending topics by the implemented framework, they have not been reported by the Twitter until the last 2 hours. True negatives set is consisted of all the topics that have been observed but never been reported as trending topics neither by the framework nor by Twitter, until the specific time that the code of this step is being executed.

The repeated execution of this step is configured through a BASH¹ script that is running on the master node of the Hadoop cluster. The code of this as well as all the other steps that are performed in the Hadoop cluster had been written and is being executed with the usage of the JCascalog framework. The reason of the preference of JCascalog instead of the traditional MapReduce API is the more abstract way of coding of the first one. As a proof of this concept the code snippet that performs the full outer join for the false positives set extraction is presented in Appendix A, section 3. What JCascalog achieves in 5 lines of code would require 2 different functions, map() and reduce(), each one consisted of dozens lines of code in the traditional MapReduce API.

**Step 8b** The time series of the topic’s that have been chosen for the training set update are normalized with the normalization filters presented in Step 3. The prefix “+” or “-” is added on the corresponding topics, if they are trending topics or not, respectively. Finally, each pair of topics and time series is stored to the first Redis database so to participate in the next cycle of the procedure that takes place in the speed layer.

**Step 8c** The topics that have been chosen to update the training set in each cycle of this procedure are further stored in HDFS. The availability of the previous inserted topics in the training set is necessary so in every execution cycle of this procedure, to be ensured that the training will be consisted of distinct topics.

¹[https://www.gnu.org/software/bash/](https://www.gnu.org/software/bash/)
Chapter 5

Experimentation and Results

The experimentation for the performance evaluation of the implemented framework, that presented in the previous chapter, on the task of the early prediction of Twitter trending topics, based on the method proposed in [1], is implemented through 2 different cases. Each case is related to the distance metric used for the calculation of the distances between the observed topic's time series and the respective ones that correspond to the topics that compose the training set. In equation 4.7, distance metric is signified by the factor \( d() \). The two distance metrics that have been used in this experimentation procedure are the Cosine distance and the Squared Euclidean distance. Among the two aforementioned distance metrics only the last one, the Squared Euclidean distance metric, has been used in [1].

The experimentation procedure lasts 48 hours for each one of the two distance metrics. The trending topics provided by Twitter through its API are considered as the ground truth for the performance evaluation. In the following sections of the chapter, a series of charts will be presented. The charts present the performance in respect of:

- The fluctuation of the performance, related to the common trending topics reported by the implemented framework and Twitter, in the total 48 hours time window of execution, divided in 1 hour time ranges.
- The total percentage of the trending topics found in common both by the implemented framework and Twitter.
- The respective percentages of trending topics found before and after Twitter reported them, by the implemented framework.
- The mean times of the report of the trending topics, by the implemented framework, in relevance to their report time by Twitter.

5.1 Comparison of the distance metrics performance

Figure 5.1 illustrates the percentage of the true positives, i.e. the percentage of tweets reported both from the implemented framework and Twitter, for every hour the procedure is running. It is clear that when the method uses the Cosine distance metric it has significant better performance than when it uses the Squared Euclidean distance metric, almost during the whole time of execution.
The dominance of the Cosine distance metric is, also, illustrated in Figure 5.2, where the total percentages of true positives for both of the distance metrics, in the end of the 48-hour execution are presented. In addition, it is worth mentioning that the dominance of the Cosine distance metric is also proven from the comparison of the maximum percentages of true positive trending topics which, as it is depicted in Figure 5.1, they are 87.3% for the Cosine distance metric (18\text{th} hour) and 73.2% for the Squared Euclidean (7\text{th} hour).

5.2 Early trending topic prediction performance

In this section the performance of the method’s execution with the two metrics, is presented, regarding the percentages and the mean times of trending topics found before and after Twitter. Figure 5.3 presents the percentages of the true positive trending topics that reported from the framework before and after they reported by Twitter respectively, and the mean times of those reports, compared to the times of the Twitter reports, in four distinct times of the execution, the 12\text{th}, 24\text{th}, 36\text{th} and 48\text{th} hour. The total percentage of the true positive trending topics for each of one of the corresponding times is presented in Table 5.1.
Figure 5.3: The respective percentages and mean times of true positive trending topics reported by the framework before and after the Twitter.
The above results indicate that, while the usage of Cosine similarity in the implemented framework has in general good performance in the discovery of the patterns of the trending topics' time series, i.e. 78.39% of Twitter's trending topics has been reported as trending topics by the framework at the end of the execution, the performance on the matter of early report of those topics, i.e. before Twitter report them, is not so satisfactory. The percentage of the reports of trending topics by the implemented framework that have been done before the respective reports from Twitter is not only relatively small but it follows a downward path throughout the execution. This situation which is partially depicted in Figure 5.3 is further confirmed in Figure 5.4, which illustrates the whole performance of the execution.

On the contrary, the mean times of the early reports of the true positive trending topics, in comparison to the times of the Twitter's reports, are quite satisfactory. This good performance is enhanced by the fact that the mean times of the later reports, in comparison to Twitter's, are significantly smaller. This situation is also depicted partially and in total in Figures 5.3 and 5.4, respectively.

As regards the usage of the Squared Euclidean distance metric, the above results indicate that there is a tend for better performance in the respective of the trending topics reported before Twitter, compared to the respective one that arises from the usage of Cosine distance metric. This better performance is true for each one of the four inspected times where the difference reaches the value of 13.4% at the end of the procedure. Of course this predominance of the Squared Euclidean metric over the Cosine distance metric as regards the percentages of trending topics reported before Twitter, is overshadowed by the unambiguous dominance of the latter in the performance of finding true positives trending topic regardless the moment they have been reported.

Similarly, to the execution with the Cosine distance metric, the execution with the Squared Euclidean has a downward trend over time, however as it depicted in Figure 5.5 this downward trend is not so abrupt as there are long time ranges with relative stability. However this stability is a result of the constant presence of specific trending topics in Twitter's list for many hours, during the current execution. As it have been observed during the various test of the execution procedure, the constant presence, that lasts many hours, of some topics in Twitter's trending topics list is a common phenomenon.

From the perspective of the mean times, Figure 5.3 indicates that with the usage of the Cosine distance metric the method achieves to report trending topics earlier than it does with the usage of Squared Euclidean metric. This observation is further enhanced from the comparison of the corresponding subfigures of Figures 5.4 and 5.5.
(a) Percentages of trending topics reported before and after the Twitter per hour of execution.

(b) Mean times of trending topics reported before and after the Twitter per hour of execution.

Figure 5.4: **Cosine:** Total performance of the method’s execution regarding the trending topics that reported by the framework before and after they reported by Twitter.
Figure 5.5: **Squared Euclidean**: Total performance of the method’s execution regarding the trending topics that reported by the framework before and after they reported by Twitter.

### 5.2.1 Discussion on the Results

Table 5.2 provides a comparison of the results of the current thesis (related to the execution with the Cosine distance metric) and of the [1] where the implemented method was initially proposed. While at first sight the results of the two implementations seem to have large deviation, this couldn’t be a safe conclusion due to the reasons that will be discussed immediately after.
Table 5.2: Comparison of results of the implemented method used in real time analysis (current thesis) and in retrospective analysis ([1])

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Total % of true positives</th>
<th>Total % found before Twitter</th>
<th>Sample of tweets used in method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current thesis</td>
<td>78.4%</td>
<td>34.6%</td>
<td>1%</td>
</tr>
<tr>
<td>Thesis [1]</td>
<td>95%</td>
<td>79%</td>
<td>10%</td>
</tr>
</tbody>
</table>

In our view, the small sample of provided tweets had the greatest negative impact on the results, as it prevented the generation of a good quality time series both in the procedure of the initial training set and in the procedure of real time execution. Another fact that is likely to adversely affect the results is the restricted time of execution, which is limited in 48 hours, while the retrospective analysis using the method in [1] is applied in a dataset that ranges in one month. Further details about the way the aforementioned situations as well as other problems that encountered during the implementation of the distributed framework and the data collection, affect the results, are provided in the next chapter.

In addition, some general observations, which may contribute to the diversification of the results are the following. Firstly, the fact that in the retrospective analysis the training set as well as the test are concern the same time period i.e the same one month time range, could influence for the better, their total results. Secondly, it could be safely assumed that the mechanism employed by Twitter to identify trending topics have been changed and further improved from 2012, where the execution of the method in [1] took place, to date. Those changes on the mechanism, for example the adoption of Storm, probably contributed in the earlier discovery and report of the trending topics by the Twitter itself.
Chapter 6
Discussion on Problems and Future work

The continuous increasing rate of data generation from a huge variety of sources, the problems that arise from this situation, and the distributed technologies that are developed to tackle those problems were the main subjects of this thesis. More specifically, the research focused on two parts. The first part regards the bibliographic review of the most widely used tools and frameworks that have been built aiming to contribute in the field of Big Data processing. The second part regards the development of a distributed architecture framework, which have been built using some of the aforementioned tools, and its usage for the solution of a real life problem, the real time trending topics prediction on Twitter.

In the first section of this chapter there is a discussion about the problems encountered during the method’s execution on the implemented framework and the reasons that caused those problems. In the second section the discussion that takes place concerns the future work that is planned to be done from the perspectives of i) the solution of the problems encountered, ii) the extensions and changes that should be done so as to achieve better performance, iii) some potential uses of the implemented framework in other problems of the Social Networks Analytics and other fields.

6.1 Implementation problems

The first and most significant problem encountered concerns the quality of the initially generated training set. As it mentioned in chapter 4, the trending topics’ time series that became part of the initial training set were poor both in terms of quality and quantity.

In particular, it has been already mentioned that the majority of topics’ time series, which have been constructed from the one month collected datasets, were too dense. At this point, it is worth mentioning that there were several cases where the corresponding trending topics’ time series were consisted only of zero values. This means that there were no reference about the particular trending topics in the tweets that had been collected during the respective 1 month time period, while in many cases the corresponding references were few. This fact led to the rejection of the majority of the trending topics’ time series resulting in the final training set to consist of only 100 trending topics’ time series and consequently only 100 not trending topic’s time series.

From the perspective of the time series’ quality, the problem that encountered arises from the fact that even in the time series with good density, the values of their time buckets were, in many cases, relatively small. This means that although there were tweets that were related to the specific trending topics, their number was small for the needs of the implemented method.

Both of the aforementioned situations are due to the the small sample, 1%, of tweets provided by Twitter API. It seems that while the time range of collecting sampling data for the training set was the same in the current implementation and in [1], the difference in the size of those samples (1% instead of 10%) greatly
affects the outcome of the procedure.

The second big problem faced in this implementation arises from the restricted infrastructure resources. The restricted resources affect the procedure that has been executed in the implemented framework in three distinct points.

The first point where the restricted resources cause problem in the implementation is in the procedure of the dataset processing for the generation of the training set, in the Hadoop cluster. It has been mentioned that the volume of the collected data exceeds 300 GB. Considering that i) the total size of the HDFS in the cluster is a little less than 500 GB (section 4.1.1 provides details of the Hadoop cluster) ii) data stored in HDFS are replicated for data loss prevention and iii) there is a need for some remarkable free space to store the intermediate output of the Mappers since the Reduce step begins, follows that the whole data processing is not possible to finish in a single step.

By reducing the cluster’s data replication factor value to 2 instead of the default value 3, the aforementioned procedure has been finished in 2 steps, i.e. by splitting the dataset into two parts. The consistency of the output data files and their aggregation for the composition of the training set had been achieved through the dfs-datastores library's (presented in section 2.1.5.2) functionalities for absorbing and consolidating the generated pail files.

The other two points of the procedure that are being affected by the restricted infrastructure resources are in the real time execution that takes place in the Storm cluster. In contrast with the ability to tackle the problem in the previous step, regarding these two points the problem was impossible to overcome.

More specifically, the first issue encountered in the Storm cluster is related to the limited disk size of the nodes that compose it, i.e. 10 GB per node. The small disk size of the disks has as result their fullness after a few days of execution mainly due to the various log files (nimbus log files, zookeeper log files, workers log files, etc.) that being generated. This situation prevents further execution of the procedure and this is the reason that the execution lasts 48 hours in each case. By extension this situation affects the maximum improvement that could have been achieved by the procedure of the incremental training set update, on the results. The reason for this is the fact that the number of the trending topics provided by Twitter in 48 hours, and thus could be part of the true positive topics that are inserted in the training set, is relatively small and the improvement that could cause on the results is not so essential.

The second issue encountered in the execution part that takes place in the Storm cluster is related to the restricted number of cpu cores that compose it. Due to this restricted number of cpu cores the possible parallelism degree of the algorithm was limited. Thus there was a number of compromises in the design decisions of the corresponding algorithm, with the cost of the potentially negative affection on the final results.

The main compromise that was made is the decision to consider only the hashtags of a tweet as topics and therefore only hashtags are considered as objects for observation in the proposed implementation. This fact further contributes to the delay in the enrichment of the training set in the notion that from the trending topics’ list provided by Twitter only half of its contents can be utilized in the periodic check for true positives that will become part of the training set. As a consequence it may affects the improvement of the obtained results.

6.2 Future work

From the above, it appears that there is a number of crucial tasks to be done so the implemented method could be more effective. Furthermore, there are some interesting extensions that could be implemented to improve the method itself and the usability of the implemented distributed framework. In the remaining of this chapter the future work that is planned to be done, is presented, divided into the following four areas:

- Future work that aims to fix the major problems that have been presented in the previous section.
• Future work that aims at the performance testing of different variations of the proposed method for trending topics prediction.

• Future work for testing the proposed method in different approaches, regarding the implemented framework’s workflow.

• Future work that concerns the usage of the implemented architecture in tasks of field of Social Network Analytics, other than the one that have been applied in the current thesis, as well its usage in problems that belong to other fields.

As regards the first bullet, the fix of the major problems presupposes the existence of a more powerful infrastructure. The acquisition of a more powerful infrastructure will allow the overcome of the problems encountered both in the batch and in the speed layer of the implemented framework.

More precisely, a Hadoop cluster with larger capacity and more nodes, would allow the collection of a bigger dataset for the generation of the training set. This can be achieved by the creation of many Twitter accounts that can communicate with the Twitter API for the collection of a larger sample of the posted tweets in the range of one month. Then, with a more powerful Hadoop cluster, the tasks of tweets duplicate removal (acquired from different accounts), and the generation of the training set from the larger dataset, would be trivial, and it would definitely improve the initial training set both in terms of quantity and quality.

Regarding the problems encountered in the Speed layer, the ability of establishing a more powerful Storm cluster will allow the increase of the, implemented algorithm’s, degree of parallelism. This means that more workers will be able to execute more tasks and by extension this will allow the implementation of new tasks, with the most significant to be the usage of techniques for topic extraction from the tweets’ text, other than the hashtag entities. Specifically, the most notable topic extraction methods, presented in [75] are going to be implemented. Furthermore, a Storm cluster with nodes of larger capacity will allow the longer execution of the method which, in conjunction with the bigger quantity of observed topics, will lead to more reliable conclusions about the method’s performance evaluation.

Having the aforementioned problems fixed, a number of variations of the proposed method is planned to be tested. One category of method’s variation concerns the usage of different distance metrics for the time series comparison. More specifically, the performance of time series oriented distance metrics like, ERP (Edit Distance with Real Penalty) [76], MSM (move-split-merge) [77] and SDTW (Sparse Dynamic Time Wrapping) [78] are going to be examined. The other category of method’s variations concerns the normalization filtering of the time series as it presented in Section 4.3.1. We plan to evaluate the method by using different combinations of the proposed normalization filters without using them all in once. Furthermore, we plan to evaluate the method with different values of the filters’ parameters, than those we used in this thesis.

From the perspective of the implemented algorithms execution workflow, the plan is to make it more “loyal” to the principles of the Lambda Architecture. This case, also, indicates the necessity of a more powerful infrastructure and the addition of a disk-based distributed NoSQL database (we plan to use ElephantDB) to serve the scopes of the Serving layer. Furthermore, this case indicates the transfer of the Redis database that stores the real time created time series from the Serving layer to the Speed layer. Those changes are essential because the great increment of the number of the observed trending topics (using the methods for topic extraction from tweets’ text) will probably cause overhead between the periodic distance calculation on the Speed layer as well as overflow of the in-memory Redis database. The plan is to transfer this task to the Batch layer. After this changes, the Storm cluster in the Speed layer will be responsible only for the topic’s extraction and the update of the changes of the corresponding time series in the Redis database. The Redis database will store only the indexes of the time buckets that their value have been changed and the respective values (so as to reduce the size of the data stored in it) while the whole time series will be stored in the Serving’s layer database and will be
periodically updated according to the changes stored in Redis. The Batch layer will periodically communicate with the Serving layer so as to acquire the time series, perform the calculation and output the results.

Another interesting plan is to use the implemented distributed architecture for other tasks that belong to the field of Social Network Analytics. The first priority is to try the performance of the architecture in the task of real time spam detection on Twitter. A variance of proposed methods are candidate for implementation and execution on the implemented architecture, both from the perspective of spam users detection, i.e. spam Twitter accounts (for example methods proposed in [79], [80]), as well as from the perspective of spam content detection, i.e. discovery of tweets with spam content in their text, by exploiting the common patterns on those texts and other content-based features (for example methods proposed in [81]).

Finally, while in the current thesis, the initial pretension was to use the implemented distributed framework to address the problem of popular topics prediction in Social Networks, however, taking a closer look, it can be seen that this framework could be used in many problems from areas different from Social Network Analytics. In other words, it could be safely assumed that the framework created can be used in any application that requires real time time series classification. At this point it is worth mentioning some indicative use cases in which the implemented framework may fits.

- Finding specific pattern in data obtained by sensor’s networks. For instance the current framework could be used to predict weather phenomena from time series which correspond to changes in atmospheric pressure, temperature, etc.
- Finding specific pattern in electricity consumption studying data from sensors connected to to the Electrical Grid\(^1\).

All the above use cases could be served by the implemented framework provided that the following parameters will be changed according to the specific demands of each problem:

1. Suitable initial training set which describes instances of the target behaviors.
2. Determination of the appropriate similarity measure.
3. Determination of the appropriate sliding window time range.
4. Determination of the appropriate observation time range.

\(^1\)http://en.wikipedia.org/wiki/Electrical_grid
Appendices
Appendix A

Source code samples

A.1 Thrift file

```java
namespace java thrift.tweets

enum TweetType
{
    TWEET = 1,
    RETWEET = 2
}

struct Text
{
    1: required string originalText;
    2: optional string cleanText;
}

struct Location
{
    1: required double latitude;
    2: required double longitude;
}

struct User
{
    1: required i64 userId;
    2: optional string name;
    3: optional string screen_name;
    4: optional Location location;
    5: optional string createdAt;
    6: optional string description;
    7: optional bool isVerified;
}

struct URLEntity
{
    1: required string url;
    2: optional string displayUrl;
    3: optional string expandedUrl;
}
A.2 Pail structure
public interface PailStructure<T> extends Serializable {
    public boolean isValidTarget(String... dirs);
    public T deserialize(byte[] serialized);
    public byte[] serialize(T object);
    public List<String> getTarget(T object);
    public Class getType();
}

public abstract class ThriftPailStructure<T extends Comparable> implements PailStructure<T> {
    private transient TSerializer ser;
    private transient TDeserializer des;

    private TSerializer getSerializer() {
        if (ser == null) ser = new TSerializer();
        return ser;
    }

    private TDeserializer getDeserializer() {
        if (des == null) des = new TDeserializer();
        return des;
    }

    @Override
    public byte[] serialize(T obj) {
        try {
            return getSerializer().serialize((TBase)obj);
        } catch (TException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public T deserialize(byte[] record) {
        T ret = createThriftObject();

        try {
            getDeserializer().deserialize((TBase)ret, record);
        } catch (TException e) {
            throw new RuntimeException(e);
        }
        return ret;
    }

    protected abstract T createThriftObject();
}

public class TweetPailStructure extends ThriftPailStructure<Tweet> {
    @Override
    public Class getType() {
        return Tweet.class;
    }
}
public class SplitTweetPailStructure extends TweetPailStructure {

    @Override
    public List<String> getTarget(Tweet object) {
        try {
            ArrayList<String> directoryPath = new ArrayList<>();
            String dateStr = object.date.createdAt;
            DateFormat dateFormat = new SimpleDateFormat("EEE MMM dd HH:mm:ss ZZZZZ yyyy");
            Date date = dateFormat.parse(dateStr);
            Calendar cal = new GregorianCalendar();
            cal.setTime(date);
            directoryPath.add("\" + cal.get(Calendar.YEAR) + ");
            directoryPath.add("\" + (cal.get(Calendar.MONTH) + 1) + ");
            directoryPath.add("\" + cal.get(Calendar.WEEK_OF_MONTH) + ");
            directoryPath.add("\" + cal.get(Calendar.DAY_OF_MONTH) + ");
            return directoryPath;
        } catch (ParseException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public boolean isValidTarget(String... strings) {
        return true;
    }
}

85
if( strings.length != 4 ) return false;

try {
    return strings[0] != null;
} catch (Exception e) {
    return false;
}

A.3 Full outer join performed using JCasclog

Subquery falsePosTopicsQuery = new Subquery("?fp")
    .predicate( tTopicsQuery, "?topics", "!!tmpVal1" ) // Twitter's topics
    .predicate( pTopicsQuery, "?topics", "!!tmpVal2" ) // Framework's topics
    .predicate( new FalsePositives(), "?topics",
               "!!tmpVal1", "!!tmpVal2" ).out( "?fp" );

public class FalsePositives extends CascalogFunction
{
    @Override
    public void operate(FlowProcess process, FunctionCall call)
    {
        if( call.getArguments().getObject(1) == null )
            call.getOutputCollector().add( new Tuple( call.getArguments().
                                                      .getString(0) ) );
    }
}


