Evaluating the impact of streaming systems design on application performance

Alessio Pagliari

▶ To cite this version:

Alessio Pagliari. Evaluating the impact of streaming systems design on application performance. Data Structures and Algorithms [cs.DS]. Université Côte d'Azur, 2021. English. NNT : 2021COAZ4011 . tel-03273377

HAL Id: tel-03273377
https://theses.hal.science/tel-03273377

Submitted on 29 Jun 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Évaluer l'impact de la conception des systèmes de streaming sur la performance des applications

Alessio PAGLIARI

Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis (I3S)

Présentée en vue de l'obtention du grade de docteur en Informatique
d'Université Côte d'Azur

Dirigée par : Fabrice Huet / Guillaume Urvoy-Keller

Soutenue le : 17/02/2021

Devant le jury, composé de :
Jean-Marc Pierson, Professeur, Université Paul Sabatier Toulouse 3
Guillaume Pierre, Professeur, Université de Rennes 1
Pietro Michiardi, Professeur, Eurecom
Fabrice Huet, Professeur, Université Côte d'Azur
Guillaume Urvoy-Keller, Professeur, Université Côte d'Azur

Evaluating the Impact of Streaming Systems Design on Application Performance

PhD Thesis

Alessio Pagliari
Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis (I3S)
Université Côte d'Azur

Directed by: Fabrice Huet and Guillaume Urvoy Keller

Jury Members:

Jean-Marc Pierson	Professeur, Université Paul Sabatier Toulouse 3	Reviewer
Guillaume Pierre	Professeur, Université de Rennes 1	Reviewer
Pietro Michiardi	Professeur, Eurecom	Examinator
Fabrice Huet	Professeur, Université Côte d'Azur	Advisor
Guillaume Urvoy-Keller	Professeur, Université Côte d'Azur	Co-Advisor

Defended on 17/02/2021

Thesis presented for the degree of
Doctor of Philosophy
in Computer Science

Résumé

Le traitement des flux de données (DSP) est un paradigme établi de Big Data qui permet de traiter et d'analyser les données en temps réel. Les applications de streaming sont composées d'une série de tâches, répliquées et réparties sur un cluster, qui effectuent des opérations sur les données entrantes, fournissant des mises à jour continues des résultats. Un large éventail de travaux a abordé plusieurs aspects du DSP : le placement des tâches, la tolérance aux pannes et la gestion des états ne sont que quelques exemples parmi tant d'autres. Dans cette thèse, on étudie les limites des plates-formes DSP actuelles, en se concentrant sur les performances du point de vue de l'application.

Dans la première partie, nous analysons les mécanismes de fiabilité des messages dans les plateformes de streaming. On découvre l'étroite interdépendance entre les mécanismes des plates-formes et les algorithmes d'ordonnancement des tâches. En particulier lorsque ces mécanismes sont mis en œuvre en tant que tâches non fonctionnelles. Ainsi, on présente deux algorithmes de planification pour optimiser les performances des applications, en tenant compte de l'impact du placement des tâches non fonctionnelles.

Dans la deuxième partie, on présente NAMB, un générateur de prototypes d'applications pour pallier les insuffisances des tests actuels d'applications de streaming. Tout d'abord, on introduit les principes fondamentaux sur lesquels repose NAMB, en présentant les modèles de description de haut niveau utilisés pour définir les applications de streaming. Ensuite, on illustre notre générateur de prototypes d'applications, en détaillant les défis de sa mise en œuvre. Enfin, on effectue une large évaluation, où l'on illustre de nombreux cas d'utilisation possibles de notre outil, en démontrant ses caractéristiques en tant que solution générique et flexible.

Mots-Clé: Traitement de flux de données, fiabilité des messages, mécanisme d'acquittement, ordonnancement, prototype d'application, génération d'applications, description de haut niveau

Abstract

Data Stream Processing (DSP) is an established Big Data paradigm that allows to process and analyze data in real-time. Streaming applications are composed of a series of tasks, replicated and distributed over a cluster, that performs operations on the incoming data, providing continuous results updates. A wide range of works tackled several aspects of DSP: task placement, fault tolerance and state management are just some of many examples. In this thesis, we study the limitations of current DSP platforms, focusing on performance from the application point-of-view.

In the first part, we analyse message reliability mechanisms in streaming platforms. We uncover the tight interdependency between platform mechanisms and tasks scheduling algorithms. Especially when those mechanisms are implemented as non-functional tasks. Thus, we present two scheduling algorithms to optimize application performance, taking into account the impact of non-functional tasks placement.

In the second part, we present NAMB, an application prototype generator to tackle the shortcomings of current streaming application testing. First, we introduce the fundamentals over which we base NAMB, presenting the high-level description models used to define streaming applications. Then, we illustrate our application prototype generator, detailing the challenges of its implementation. Finally, we perform a wide evaluation, where we illustrate numerous possible use cases for our tool, demonstrating its characteristics as a generic and flexible solution.

Keywords: Data Stream Processing, Message Reliability, Acking Framework, Scheduling, Application Prototype, Application Generation, High-Level Description

Acknowledgments

I premise I’m not good with acknowledgments, so don’t expect much from this page. The list of people to be thanked should be vast, surely I will forget somebody. Please excuse me if that will be the case. Also, the order below is not of importance, but rather of "how they popped-up in my mind".

First of all, as they are the advisors of my thesis I have to thank Prof. Fabrice Huet and Guillaume Urvoy-Keller. For the opportunity the gave me more than three years ago, the guidance and the help they gave me in a new topic that I never studied before and in this difficult challenge that is a PhD.

A special thank is for my friend and colleague, Andrea, with whom I shared all the PhD experience and this adventure in a foreign land. Thanks for all the discussions, complaining and beers.

I deeply thank the people I met during these years and who I consider great friends: Moudy, Sara and Antonia. With them I shared great times, workouts and paintings. Thank you for having eased my time in a foreign land.

I can’t forget of the people who stayed for a bit, but long enough to build a strong bond with, who also I will always consider great friends: Adam and Piergiorgio. Thanks for the beers, the jokes and all the crazy things.

I also thank all the people from the magic I3S’s third floor, for the coffee time, discussions and shared knowledge. I Thank the coffee machine, of which I took great care (right?). How could I ever forget the 3D printer and the "extremely useful" stuff printed by Fabrice. As well, I should thank all the other people I met during this adventure, I’m sorry if your name is not here.

Last but not least, one of the last people I met, but who quickly became the most important. Who stayed with me during the struggling times of a pandemic and, worst, the thesis writing. A special thank to a fellow researcher and my girlfriend, Ilaria.

Contents

Résumé i

Abstract iii

Acknowledgments v

1 Introduction 1
- 1.1 Big Data 2
- 1.2 Batch vs. Streaming 4
  - 1.2.1 Batch Processing 4
  - 1.2.2 Stream Processing 5
- 1.3 Research Methodology and Contributions 6
- 1.4 Thesis Outline 8

2 Background on Data Streaming 11
- 2.1 Data Stream Processing 11
  - 2.1.1 Applications 12
  - 2.1.2 Challenges 12
  - 2.1.3 Platforms 16
- 2.2 Apache Storm 17
  - 2.2.1 Architecture Overview 17
  - 2.2.2 Application 18
  - 2.2.3 Scheduling and Task Placement 19
- 2.3 Apache Flink 22
  - 2.3.1 Architecture Overview 22
  - 2.3.2 Applications 22
  - 2.3.3 Scheduling and Task Placement 23

3 Non-Functional Tasks Placement 27
- 3.1 Related Work 29
  - 3.1.1 Reliability and Fault Tolerance 29

3.1.2 Application Scheduling and Task placement 30
3.2 Storm's Acking Framework 32
3.2.1 Message Reliability 32
3.2.2 Backpressure 33
3.3 The Impact of ACKing 34
3.3.1 ACKers in task placement 34
3.3.2 Performance degradation 34
3.4 ACKers-aware Scheduling 40
3.4.1 One-per-Worker Strategy 40
3.4.2 Isolated Queue Strategy 41
3.5 Evaluation 42
3.5.1 Experimental Setup 42
3.5.2 Application 43
3.5.3 Methodology 44
3.5.4 Results 47
3.6 Conclusion 52

4 Application Prototype Generation: NAMB 55
4.1 Related Work 57
4.1.1 Benchmark and Evaluation 57
4.1.2 High-Level Languages and Application Generation 61
4.2 Motivation 62
4.3 Fundamental Characteristics 64
4.3.1 Data Stream Characteristics 64
4.3.2 Workflow Characteristics 65
4.4 High-Level Description 66
4.4.1 Workflow Schema 67
4.4.2 Pipeline Schema 70
4.5 Not Only a Micro-Benchmark 72
4.5.1 NAMB Design 72
4.5.2 Application Builder 73
4.5.3 Topology Generator 73
4.5.4 Data Generation 73
4.5.5 Multi-Platform Design 74
4.6 Implementation Challenges 75
4.6.1 Task Workload Simulation 75
4.6.2 Topology Design Decisions 77
4.6.3 Managing Platform Specifics 81
4.6.4 Data Generation 81
4.7 Conclusion 83

5 Evaluation and Practice of NAMB 85
5.1 Experimental Setup & Methodology 86
5.2 Application Design 87
5.2.1 Connection Routing 88
5.2.2 Parallelism Scale-up 89
5.2.3 Processing Load Balancing 90
5.2.4 Data Size 90
5.3 Application Prototyping 92
5.3.1 The Yahoo! Streaming Benchmark 92
5.3.2 Insights on Processing Load Tuning 94
5.4 Bottleneck Discovery 99
5.5 Platform Features Testing 100
5.6 Data Generator 102
5.6.1 Internal Generator 102
5.6.2 External Kafka Producer 104
5.7 Conclusion 104

6 Conclusion 107
6.1 Non-Functional Tasks Scheduling 108
6.2 Streaming Application Prototyping 109
6.3 List of Available Software 111
6.4 List of Publications 111

List of Figures

1.1 Batch Processing Workflow . . . 5
1.2 Stream Processing Workflow . . . 6

2.1 Example of Storm topology: one spout; two bolts (split); one final bolt (join). The last component has a parallelism level of 1, the others of 2 . . . 18
2.2 Example of Storm different ways of scheduling . . . 19
2.3 Example of Flink scheduling of a pipeline of 4 tasks . . . 24
2.4 Flink task sub-chaining with different connections . . . 25

3.1 Acking framework message exchange example . . . 33
3.2 Storm tuples throughput . . . 35
3.3 *RAS* ackers placement example with a configuration that allows 3 executors per worker; 3 workers used, 3 total ackers co-placed with an operational task . . . 36
3.4 CPU utilization of a Storm cluster: two different schedulers . . . 37
3.5 CPU load of a Storm cluster: two different schedulers . . . 38
3.6 Example of workers executors disruptor queues: sojourn time . . 38
3.7 Storm application network traffic using 7 char long java string tuples . . . 39
3.8 *OPW* placement example with a configuration that allows 3 executors per worker; 3 workers used, 3 total ackers placed one per worker; N.B.: ackers don't consume RAS resources . . . 41
3.9 *IQ* placement example with a configuration that allows 3 executors per worker; 4 workers used, 3 total ackers placed on a dedicated worker . . . 42
3.10 Simple streaming Word Count topology . . . 43
3.11 Results of the large-scale topology in the single cluster environment 48
3.12 Results of the small-scale topology in the single cluster environment . . . 49
3.13 Results of large-scale topology in the multi-cluster environment . 50

3.14 Results of the small-scale topology in the multi-cluster environment 51

4.1 Micro-Benchmarks Topology Layouts 58
4.2 Yahoo Streaming Benchmark Design [CDE+16a] 61
4.3 Datastream section of Workflow schema (yamb.yml) 68
4.4 Flow configured as: sinusoidal stream, with fixed rate and phase 68
4.5 Workflow section of Workflow schema (yamb.yml) 70
4.6 Per-task configuration with the Pipeline schema (yamb.yml) . . . 71
4.7 NAMB Architecture 72
4.8 Data Stream section of Workflow schema for the kafka source . . 74
4.9 Per tuple CPU measurements for real/simulated tasks 76
4.10 Streaming topology layouts as implemented by NAMB 77
4.11 Parallelism balancing examples; depth 4 diamond topology with a global parallelism of 18; balanced, increasing and decreasing configurations 78
4.12 Workload balancing examples; with a base processing value of 10; balanced, increasing and decreasing configurations 80
4.13 Configuration of the external data generator 83

5.1 Base configuration file for Workflow schema experiments. Highlighted values are changed during tests 87
5.2 Flink performance with different connection routing between source and task. 4 tasks with 24 instances each. 88
5.3 Flink performance when varying components parallelism using a rebalanced connection. The different series represented are defined by the number of source instances. 89
5.4 Flink performance with different workload distribution. 4 tasks of 24 instances each, connected via direct routing 90
5.5 Flink performances with different data sizes, 4 tasks of 24 instances each connected through rebalance routing 91
5.6 Throughput and latency percentage difference ratio between NAMB and Yahoo Bench results 92
5.7 Pipeline schema configuration file used for Storm. Highlighted values differ in the Flink configuration. 93
5.8 Application performance metrics in relation for busy wait cycles . 96
5.9 Latency Storm Runs Time Series of Latency 97
5.10 Backpressure and metrics standard deviation for busy wait cycles 98
5.11 Bottleneck discovery experiment in the three different configurations 100
5.12 Configuration file for the back-pressure experiment. 101

5.13 Back-Pressure test in Storm, with enabled and disabled reliability mechanism. . . . 102
5.14 Flow description. Highlighted distribution type changed in the experimentation. . . . 103
5.15 Internal data generator throughput with different stream distributions 103
5.16 External data generator (Kafka producer) throughput with different stream distributions 105

List of Tables

3.1 Summary of tests run for the large topology. Total executors: 376 + ackers 45
3.2 Summary of tests run for the small topology. Total executors: 26 + ackers 46

5.1 Real application simulation: results on Storm and Flink 92

Chapter 1

Introduction

Contents

1.1 **Big Data** 2
1.2 **Batch vs. Streaming** 4
1.2.1 Batch Processing 4
1.2.2 Stream Processing 5
1.3 **Research Methodology and Contributions** 6
1.4 **Thesis Outline** 8

A continuously increasing number of IT and non IT companies are heading towards a data centric business model. In the entirety of the industrial landmark the importance of data, most specifically information, is well established. In 2020, it is estimated that 4.5 billions users access the Internet via almost 30 billions devices [Cis20]. The amount of data generated has significantly increased compared to the past years, reaching a volume of 44 ZettaBytes of data generated in the Internet [UI14]. Given the variety of data exchanged by users, companies strive to exploit this data at best to obtain valuable strategic business information. One of the examples that can be given is targeted advertising. Several web-based companies exploit their users activity to analyze their behavior and offer interest-based advertising [Goo20]. Data can be also used to understand and predict the trends of the company, and get insights on its decisional process.

One of the main paradigms in data management is the Extract-Transform-Load (ETL), which describes the process of obtaining data, processing it and

storing the transformed value. Historically, ETL models relied on RDBMS (Relational Database Management Systems). However, given the important quantity of data generated nowadays, old ETL models to elaborate and analyze data are not enough. The paradigm born to analyze and process this huge amount and heterogeneity of data is called Big Data. This paradigm is nowadays widely spread and an established core technology in the majority of data-driven companies.

With new widely used Internet services (e.g. e-commerce, web search engines and social networks), legacy relational databases are not able to keep up with the amount of data to be stored and read. Consequently, Big Data applications required new data management methodologies to deal with these new challenges. From a *structured* paradigm for data management, new models moved to *unstructured data*. The former relies on pre-defined data, tables and fields, typically relational databases. The latter removes all the boundaries of structured data, and support more generic data formats, like texts, images or videos. To manage these formats non-relational databases (i.e. NoSQL) are used, proposing different paradigms to store data: document-based (e.g. MongoDB [MDB]), key-value (e.g. Redis [RED]), graphs (e.g. Neo4j [N4J]).

1.1 Big Data

Big Data has been precisely defined by IBM [Hub20] though the description of the *4 V's*: Volume, Variety, Velocity and Veracity. It became quickly a standard definition of what Big Data is and of the general challenges in the field. Through the years, the evolution of data and applications that would rely on Big Data, increased the number of challenges. It is common to extend the definition of Big Data with new *Vs*, e.g. Elder Research listed up to 42 V's [Sha17] in 2017. Some of the most recurring are: Variability, Visualization and Value; this last one is evident from the trend taken by the companies. Although the large set of Vs can be useful to understand all the nuances of Big Data, an already extensive and correct definition of the field is given by the original 4 Vs.

Volume The main reason why Big Data was needed can be attributed to the rapidly increasing *amount of data* that needs to be processed. As we previously stated, not only processing but storage as well are challenged by the volume of data. Given the increasing value of stored data (the more the better), companies cannot allow themselves to discard it. The collected data needs to be stored for future historical analysis, e.g. estimate company trends of the last months.

When dealing with Big Data, it is common to distribute the storage over a larger cluster rather than a single node. However, updating on-premises clus-

ters comes at high-financial costs. Hence, data moved from a local storage to a remote one with Cloud Computing. Cloud Providers (Amazon AWS, Microsoft Azure and Google Cloud Platform are the main players [Gar]) allow companies to outsource data storage and processing capabilities (useful as well for the challenges raised up by velocity) on remote clusters.

Variety Following the growth of relatively new Internet services, like Social Networks, E-Commerce and Video Streaming Services, the *diversity of data* generated and that needs to be dealt with increased significantly. Based on the service or the specific field, data can be structured, unstructured or even of multimedia type. Examples are: streaming video chunks (e.g. video streaming both on-demand and live), block of texts (e.g. posts or tweets), products metadata (e.g. online shopping), images and so on.

Big Data systems need to be able to deal with this variety and to merge data coming from different sources to obtain useful and valid information from it. For instance, the so called Data Lakes systems, used to administer data without following a common format, are implemented in Big Data solutions to store diverse data formats coming from different sources, enabling a centralized management of data.

Velocity With the advent of IoT (Internet of Things), the network is accessed by numerous devices of any kind (i.e. things). From smartphones to sensors, every single device that can connect to the Internet is a potential data generator. With IoT sensors, the amount of data generated per second increased exponentially. IBM estimated that 50000 GB were generated per second in 2018 [Hub17].

Thus, velocity describes the speed of data, both in terms of production and consumption of this data. New IoT applications are often reactive systems, that continuously monitor a given environment and trigger operations based on status changes. Sensitive applications, such as healthcare systems or security monitoring solutions require a fast response to changes in the monitored data. Big Data systems thus need to efficiently and quickly process the huge amount of quickly generated data, ideally in a real-time fashion.

Veracity We mentioned before the value of data. It is however important to first assess its correctness. In Big Data systems veracity is the aspect that defines the uncertainty of data. Data flows through networks, clouds and private systems that are not foolproof. Devices may stop functioning or lose connectivity, being unable to send data. Thus, data is subject to loss and incorrect retrieval. An incorrect dataset, that may lack important sections, can cause great cost to

a company. Such a dataset used for data analysis may generate wrong or incomplete information. If this is the case, business decisions based over that information can easily results in financial loss for the company.

It is thus important for a Big Data system and infrastructure to reduce any possible flaw. At the same time, data analysis techniques should always consider the possibilities for a non-consistent dataset. Hence, Big Data architectures and processing paradigms aims to ensure consistent datasets (i.e. with neither missing or incorrect data) and data processing correctness (i.e. no loss or wrong computations in the processing task).

1.2 Batch vs. Streaming

To get useful information, raw data has to be processed and transformed. As we saw, the main challenge when dealing with Big Data is to process high quantity of data that may be quickly generated. Two principal processing methodologies have been created to deal with such challenges. The first one is *batch processing*, used to manage significant amounts of data stored in distributed environments. The second one is *stream processing*, used to process quickly in almost real-time the stream of generated data.

1.2.1 Batch Processing

In Big Data analytics, batch processing helps to analyze massive quantities of stored, and commonly distributed, datasets. Batch processing spreads the workload over different distributed tasks that will run in parallel. The data is analyzed in batches by the different tasks. The distributed storage and the parallel execution gives batch processing an advantage over old data management techniques. Indeed, it allows more complex operations and an easier management of higher volume of data to process. For such reason batch processing is used for analytics of historical data, to get a global view of a fixed dataset.

MapReduce The paradigm over which batch processing is based on is MapReduce. It follows three main steps of processing: map, shuffle and reduce. In the *map* phase, each task analyzes the local data and applies the map function (e.g. counting word occurrences). Results are temporary stored as a set of key-value couples (e.g. word and count). During the *shuffle* phase, data is redistributed over the tasks, grouping the results by key, so to have all the occurrences of the same key processed by the same task. During the *reduce* phase, each task merges in parallel the results received (e.g. sum the count value of every word key).

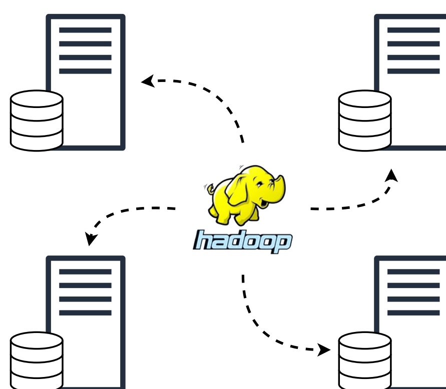


Figure 1.1: Batch Processing Workflow

The first public and most used platform to implement MapReduce is Hadoop [HAD]. With the help of a distributed file system, HDFS [HDF], Hadoop spreads the storage of data over the cluster, managing it as a unique source. To process the data, Hadoop places several tasks over the cluster nodes, in order to have them closer to the data sources.

Even though MapReduce is the most common paradigm in batch processing, other platforms, like Spark [SPA], enhance the batch processing capabilities. Spark includes the possibility to add intermediate steps in the processing pipeline. In this way, batch processing is not limited anymore by two fundamental operations but it allows to process data through successive operations.

1.2.2 Stream Processing

Stream processing doesn't target a fixed and stored dataset. Instead it processes a continuous stream of unbounded data generated live by external sources. It defines a sequence of tasks to be executed over data as it comes. The tasks are paralleled and distributed over the cluster to optimize the computation. The distributed and live processing paradigm allows for an almost real-time data processing. A common example is the monitoring of the Twitter feed for a particular hashtag, producing real-time statistics such as number of tweets with that specific hashtag or places of the world from where people are tweeting.

Data streaming commonly follows two stream processing paradigms: *once-at-a-time* and *micro-batch*. The first one, used by systems like Apache Storm [STOb], process data as it comes by quickly sending every received message into the processing pipeline. The second one - an emblematic is Spark Streaming [SST] - groups the incoming data in micro-batches, i.e. small buffers of

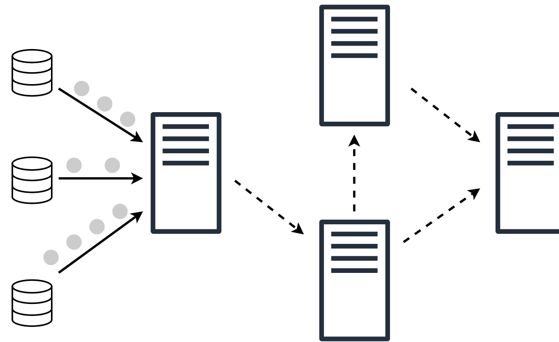


Figure 1.2: Stream Processing Workflow

data, that are then continuously processed in a batch processing fashion. Both paradigms have their pros and cons: once-at-a-time can achieve very low latencies, i.e. time between data reception and results output; meanwhile micro-batch reaches higher throughputs trading-off average latency, i.e. processes high-quantity of data per second.

1.3 Research Methodology and Contributions

In this thesis we study the impact of design choices on streaming applications performance. Different features of a DSP platform can influence the final application efficiency. Hence, when developing a platform or a streaming application, design choices are critical. At the *platform level*, they affect the implementation and optimization of the platform internal mechanisms. At the *user level*, they influence the data processing pipeline implementation. In both cases, improper design choices have a direct impact on the processing performance, that is the amount of data and the speed at which it can be processed.

Our contributions tackle the problem at the two levels. In this thesis, we study the *platform* design choices, analyzing internal platform mechanisms and their impact on performance, such as the reliability mechanism. In particular, its implementation as non-functional tasks and their placement problem. Likewise, we study how to help the *user* to better understand the application design choices impact on performance. Specially, we target the lack of a flexible application testing methodology.

Non-functional tasks (mis)placement The impact of the acking messages on the application performance is studied in the first part of the thesis. We analyze the relation between these non-functional tasks and the final application performance, considering how those tasks are managed by the application scheduler.

We uncover a recurrent misplacement of acking tasks in Storm, resulting in a significant degradation of throughput and latency. In this thesis we made the following contributions to this problem:

- First, we demonstrate the impact of non-functional tasks in a streaming application. Taking Storm acking framework case as an example, we show how an incorrect placement of the acker tasks may cause a significant degradation on the application final performances;
- Second, we introduce a placement algorithm that takes into account the non-functional tasks, in particular ackers, showing how an acking-aware placement of the application can improve application performances in terms both of processing latency and throughput.

Lack of flexible application testing We deal with the necessity of a tool that could ease experimental phases. We envision a generic multi-platform prototype generator. This has to be easy to use for streaming platforms, and it should provide a certain degree of customization, so to allow the study of different application design choices in a quick manner. Our contribution to this problem are:

- We devise a general description schema for streaming applications, that is easy to use without requiring specific knowledge of the platform we want to test the application on. To do so, we made a taxonomy of the key characteristics of a streaming application, i.e. the ones that impact the most performances; from those characteristics we built two high-level description schemas:
  - A generic description, that enables to specify the application through general characteristics;
  - A task-specific description, that allows an higher level of granularity to describe the application.
- We also develop an application prototype generator, which translates an high-level description into application code automatically; the generator can be adapted to different platforms so to test the same application behaviors and design choices over different environments, without having to write platform-specific code.

1.4 Thesis Outline

In this chapter we provided a general overview of Big Data characteristics and paradigms. We pictured the context and motivations of our work, listing the main contributions presented in the following chapters. Hence, below we provide a brief outline of the content of the rest of the thesis:

Chapter 2 introduces the main concepts of data stream processing, defining the main challenges currently addressed in literature. We give an overview of the wide spectrum of available platforms in the market, with a specific description of the platforms used in this thesis: Storm and Flink. Finally, we illustrate and compare their architecture and scheduling algorithms.

Our first contribution is presented in Chapter 3, where we further study the scheduling algorithms. We explore message reliability in DSP platforms, with particular focus on the Storm's acking framework. We expose the impact of acking on the application performance with regard to placement algorithms. We then propose two novel scheduling algorithms that optimize the placement of Storm tasks. Finally, we evaluate our solution by demonstrating the improvement in terms of application throughput and processing latency.

The next two chapters concentrate on our prototyping solution: NAMB. Chapter 4 depicts the lack of a general solution when it comes to evaluate or test streaming platforms from an application point-of-view. We firstly list all the fundamental characteristics of streaming applications that would impact their performances. Then, we introduce two high-level description models used to describe a streaming application. We present the deriving framework NAMB, Not only a micro-benchmark, a tool that overcomes the lack of a general solution for platform testing. The chapter illustrates in detail the architecture and the main components of the application prototype generator. It accurately describes the main implementation challenges. Moreover, it shows the modular nature of NAMB and its ability to be extended to support of more platforms.

In Chapter 5 we perform an extended evaluation of the system. We present several scenarios where one can exploit NAMB capabilities. We demonstrate that our proposed framework covers the initial requirements defined for a general testing solution. With the set of presented experiments we show that NAMB is: (i) flexible and quickly adaptable to several contexts, (ii) able to reproduce and simulate realistic applications, (iii) not bound to a single platform or environment and (iv) openly distributed.

The manuscript is finally concluded in Chapter 6, where we summarize all the contributions presented in the document and introduce possible future work.

Chapter 2

Background on Data Streaming

Contents

2.1 Data Stream Processing 11

2.1.1 Applications 12

2.1.2 Challenges 12

2.1.3 Platforms 16

2.2 Apache Storm 17

2.2.1 Architecture Overview 17

2.2.2 Application 18

2.2.3 Scheduling and Task Placement 19

2.3 Apache Flink 22

2.3.1 Architecture Overview 22

2.3.2 Applications 22

2.3.3 Scheduling and Task Placement 23

2.1 Data Stream Processing

Data Stream Processing (DSP) draws its fundamentals from database management systems and distributed systems. Initial data stream concepts date back to early 90s, when the database community started discussing about *continuous query* [TGNO92]. Successive research lead to the first real streaming foundations and paradigms in the early 2000s [FCKK20].

Initially exploited for Complex Event Processing (CEP), stream processing had to process data from multiple sources and perform combined queries. In

the context of CEP, the first systems were already able to rapidly process data coming from multiple sources. They merge the received data to compute data relations. The output of stream processors were the actions (i.e. triggers) to be taken in response to the monitored event.

Recently, with the onset of cloud computing, streaming architectures evolved. Modern DSP systems evolved to more distributed architectures that seek to scale systems by leveraging the capabilities of parallel data computation. Novel features such as state management and fault tolerance for higher processing guarantee became desirable properties. A wide spectrum of solutions (Section 2.1.3) from industrial to open source became available to perform data analytics. The variety of platforms was the result to an always wider application range that tackle newly rising challenges.

2.1.1 Applications

A streaming application is commonly composed of three phases. The first comprises the *data sources*, with connectors to the physical data generators, e.g. monitoring sensors that send updates or collectors of user activities on web services. Data sources continuously receive messages, namely tuples, containing the generated raw data. The source tasks will then inject data in the real application workflow. This workflow is commonly a pipeline of one or more *processing tasks*, that define the computed query. The query may consist of several branches. Data can be divided or merged to/from different branches, that finally merge towards the *sink tasks*. The sinks are used to manage the final result, i.e. elaborated data. The sink can be represented by simple logs, permanent storages (e.g. databases), as well as connections to other application source tasks.

To ease its management, a streaming application is commonly represented as a Directed Acyclic Graph (DAG). This allows to see each task as an independent entity (i.e. graph vertex), with incoming and outgoing edges. The application is managed as a job to be placed in the cluster. Tasks are individual processes that run on the cluster machines. The edges characterize the data communication between tasks. In this manner the tasks are distributed over the cluster, and network connections enable the data to flow from the source to the sink of the deployed application.

2.1.2 Challenges

Modern DSPs are distributed and scalable. This nature poses a wide set of challenges to be faced [FCKK20], e.g. processing guarantees, fault reliability and state management. Below, we list the challenges that most relate to the content

of this thesis. Although some of them are not directly tackled, they constitute the context around which our work operates.

Windowing, Out-of-Order Processing, State

Data is processed as it arrives, one tuple at a time. This paradigm complicates keeping track of past events to compute historical relations. One of the fundamental principles in stream processing is windowing. Windowing (Section 4.3.1) allows to specify a period of time over which the system would keep memory of the occurred events. This enables over time statistics and not just live status. Moreover, it allows to perform computations based also on past events. The implementation and management of windows rise several challenges [FCKK20], especially:

- The implementation of the temporary storage in memory, which could be limited.

- The computational complexity to continuously re-elaborate the current status at every update.

- The semantic of how the window is defined, giving temporal or tuple-based boundaries.

Another major challenge is to process out-of-order tuples [ABC+15]. Given the distributed nature of stream processing, messages may arrive in a wrong order or may be lost. It can occur that at high rates and flowing through the network, messages be lost and never processed. Alternatively, taking different paths, or given the complex management of buffers, data may reach the application in a different order than it was generated. An important challenge of modern DSP is to take into account this behavior. Indeed, the main challenge and objective of data streaming is to achieve processing correctness, that guarantees to process all the generated data without loss. DSP systems try to use windows and watermarks [ABC+15] to wait for all the tuples to arrive. In recent applications, e.g. microservices and large-scale ETL, the state of the stream processor increased in importance. It needs to be kept and to be visible by external components [FCKK20], as the relation between new and past events may affect the result of the computation. A consistent state can be kept only having a consistent dataset, that is with no missing incorrect, or incomplete data. Thus, it needs support of fault tolerance and processing guarantee mechanisms. As a consequence, another major challenge is the architectural implementation of the state manager.

Processing Guarantees, Fault Tolerance, High-Availability

A stream processor shall provide guarantees on data processing. The three semantics of processing guarantee are: *at-most-once*, where there is no checking over the message processing and a tuple may be lost and never considered in the dataset; *at-least-once*, where each tuple is surely processed but a replay mechanism may duplicate them; *exactly-once* where every tuple is assured to be processed only one time. Modern DSPs implement the last two, so as to minimize the loss of data. In addition, to ensure processing correctness, a DSP engine needs other internal mechanisms. Fault tolerance and high-availability mechanisms are used to deal with random failures in the cluster. It can happen that a node in the cluster stops working, or a crash of the application may block a task from processing data.

Stream Processing Engines (SPEs) has to be ready to immediately deal with that occurrence. SPEs thus need to ensure the high-availability of the application, with minimum down times. Given the high rate and high volume of data, a second of downtime may results in millions of lost tuples. For such reason fault-tolerance mechanisms – i.e. automatic task re-placement or task replication [CGLPN17a,CGLPN17b,SZ16] – are commonly implemented. Moreover, a reliability mechanism has to be implemented to ensure message processing, e.g. check-pointing [CFE+15], ack mechanism [TTS+14].

Semantics and Query Language

The wide platforms marketplace generated a diverse semantic definition of processing methods and non-standard query languages. This resulted in a lack of a common parallel processing model in data stream platforms [FCKK20]. As an example, where a platform may have tuple-based windows, others can decide to have them time-based, or if some process messages as they arrive (i.e. real-time), others may use micro-batches. These differences complicate system configuration and application development. For every different platform it is necessary to understand how it behaves under certain conditions. In addition, some platforms may feature mechanisms that others lack, so that a same application over different systems cannot implement the same feature.

As a consequence, each platform comes with its API. Some platforms may require the definition of each task, specifying also the internal operations, others may provide a preset of operations that can be built together.

Thus, a major lack in the DSPs landscape is common processing semantics and query languages. Recent research efforts try to address this requirement for a common semantic behind data streaming. This would allow for future DSPs to advance in the same direction. A major work in this subject is from

Google [ABC+15], where they propose a common semantic for windowing and a common API set for all streaming platforms. Concerning the query language, modern platforms are exposing specific APIs which follows a more familiar SQL-style query language, as early introduced in [ORS+08].

Scalability and Scheduling

One of the V's of Big Data is Volume, and stream processing is facing this issue as well. New applications such as sensor networks, where thousands of sensors are deployed around a city, or Internet services like social networks, where millions of users post and upload media, generate high flows of live data that need to be quickly analyzed. Thus, an SPE has to be able to process an high volume of data.

A first solution to this problem is task scheduling. Leveraging on the distributed system characteristics, the SPE can spread the tasks of an application over the nodes of the cluster. This enables parallel computation of a higher quantity of data, than just processing it on a single machine. However, task scheduling itself is not enough. The application may need to be scaled to efficiently consume the incoming stream. A first and established solution is task parallelism. Replicating the same task, and load balancing data between the different instances, allows to extend the computational capabilities of an application, e.g. three task counting words can (optimally) process three times the data of a single task.

Both these solutions need to be optimized together. An application is composed by several tasks with several instances each. For an optimal placement of the application, it is needed the use a proper scheduling algorithm. This algorithm should take into account several characteristics of the underlying system, e.g. available and required resources, cluster composition, application characteristics, etc. Furthermore, recent works [dAdSVB18,CGLPN17b] that automate the scale up and re-allocation of the application at runtime adds more challenges, calling for a need of live task schedulers.

Evaluation and Benchmarking

As a result of all the challenges discussed up to now, the choice of the best fitting platform becomes a challenge itself. Applications and platforms require a prior evaluation. To have a basic understanding of a platform behavior and performances, one needs to know its internals, the specific semantics and the specific API set to write a basic application.

Benchmarking applications are used for such a task. Benchmarks are commonly ad-hoc workflows, used to evaluate the performances of an application

on a specific platform. A major challenge is the design of a comprehensive and efficient benchmarking application. A benchmark application should enclose all the necessary characteristics of the system that need to be evaluated. Commonly, the evaluated characteristics depends on the application context (e.g. field of study, platform evaluated, etc.) and user needs (e.g. processing reliability, fast results, etc.).

2.1.3 Platforms

The lack of a common semantic lead to the birth of several stream processing platforms, each of them implementing their own architecture and set of APIs. There is a wide choice of both open source and industry solutions. The majority of these platforms have been developed specifically for stream processing. However, other platforms already present in other fields (i.e. batch processing or messaging), adapted their programming models and executions semantics to include streaming capabilities. In a similar manner, there are streaming platforms born for streaming that are as well able to perform also batch computations, following the streaming paradigm by means of windows [ABC+15].

The first production-ready platform for stream processing, Storm [STOb], has been developed by Twitter. It is now part of the Apache project. Storm is the first real-time processor with high scalability. As Storm, the majority of open source platforms are today developed as an Apache project. Spark, originally an alternative to Hadoop for batch processing, implemented an extension called Spark Streaming [SST], that through micro-batches is able to run streaming applications. Flink [FLI] is based on the dataflow model semantics [ABC+15]. It focuses its architecture mainly on the design of windowing mechanisms and out-of-order processing. Kafka [KAF], originally a publish/subscribe message broker, evolved to enclose streaming capabilities (becoming an actual DSP), used mainly for micro-services. Other notable Apache platforms are Samza [SAM] and Ignite [IGN]. The first is a stateful streaming processor. The second is an in-memory framework that, in addition to stream computations, also includes in-memory storage and machine learning capabilities. Outside of the Apache foundation, but currently incubated, there is Twitter Heron [HER]. Heron has been developed as an alternative to Storm. With in mind the limitations of the latter, Twitter developed an ameliorated version of the popular stream processor, based on a lower level language (C instead of Java) and new technologies such as containerization.

Present-day streaming processing is commonly scaled using the cloud. This allows a better management and higher availability of resources. For such reason every major cloud providers now offer a stream processing service in their

suite. Google introduced Dataflow, known in the past as Millwheel [ABB+13]. Like Flink, it is directly built over the dataflow model. Amazon AWS offers Kinesis [KIN]. On Microsoft Azure, we can find Stream Analytics [AzS]. Beside these streaming-as-a-service platforms, we can find standalone industry solutions, like StreamSets [STS].

As can be seen, the streaming platform marketplace is rich. Several platforms can be used for different scenarios, some of them focusing on specific features that can be useful in very specific domains. The work presented in this thesis is principally done using Apache Storm and Apache Flink. The choice has not been done on the specific features offered by the platforms, but rather based on the popularity and novelty of the platforms. Thus, in the remaining of this chapter, we will give an overview of Storm and Flink, with details on their internal architectures, and a generic view of their most significant semantic properties.

2.2 Apache Storm

2.2.1 Architecture Overview

Storm is a distributed data stream processing system that relies on ZooKeeper [HKJR10] to manage the coordination between all its components and the cluster resources. Storm implements a master-slave design to manage the cluster and application execution.

The controller node, called *Nimbus*, manages the status of the cluster. It is in charge of managing new topologies to be scheduled, fault-tolerance and directly communicates with ZooKeeper. When a new application is submitted to the cluster, *Nimbus* will run the scheduling algorithm (Section 2.2.3) and place the application on the slave nodes.

The slave nodes are called *Supervisors*. Each *Supervisor* provides to *Nimbus* a set of Java Virtual Machines (JVM), called *workers*. Each *worker* is assigned a communication port and can contain several threads, known as *executors*, belonging to the same application. The communication port is both used as an identifier for the worker, as well as TCP port to create the connection used to transmit data between tasks.

Runtime examples of the Stom architecture will be shown in Section 2.2.3 in Figs. 2.2a and 2.2b.

2.2.2 Application

As with the majority of DSP platforms, the topology of the application is represented by a Directed Acyclic Graph (DAG), where the components are the vertices and the edges are the connections between components.The application running on the cluster is made of two types of components, Spouts and Bolts. Spouts represent data sources and inject streams of tuples into the workflow. Bolts can act in two ways: they can encapsulate simple operations to be performed on input tuples; or alternatively, they can act as sink tasks. Each component has its own level of parallelism (i.e. multiple instances of the same task) and can span multiple workers and executors to scale and distribute the application.

Spouts and Bolts are connected via several communication methods, called stream groupings [Stoc]. *Shuffle grouping* balances the load in a round-robin fashion over the destination executors. *Field grouping* decides the destination based on a key hashing function, ensuring that the same key will always be processed by the same executor, thus allowing the implementation of stateful components. *Partial-key grouping* [NMGS+15] improves on the previous one by trying to also enforce load balancing based on the incoming tuples frequency. Finally, *all grouping* performs a multi-cast, sending the same to all the following task instances, and global grouping redirect of all the tuples to a single instance of a component.

A sample topology, composed of one spout and three bolts, is shown in Fig. 2.1. All components except bolt B3 have a parallelism level of 2 which means they will take 2 executors at runtime.

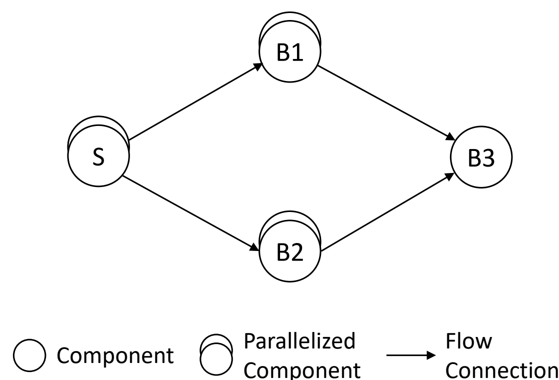
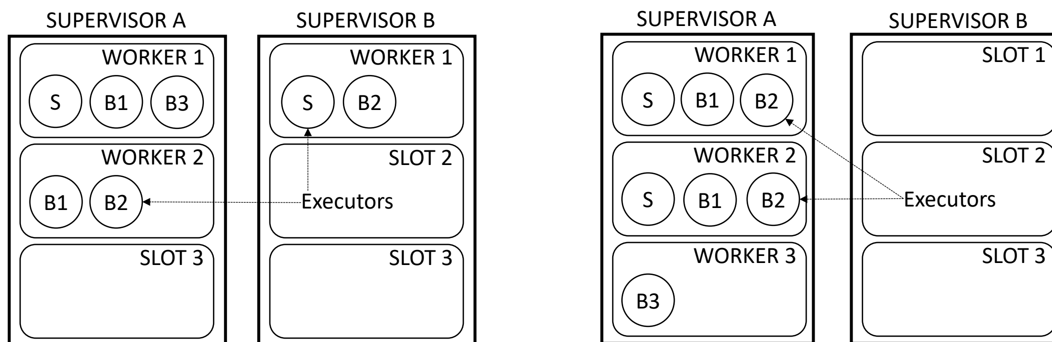


Figure 2.1: Example of Storm topology: one spout; two bolts (split); one final bolt (join). The last component has a parallelism level of 1, the others of 2

2.2.3 Scheduling and Task Placement

After submitting a topology, *Nimbus* will take care of its placement in the cluster applying a scheduling algorithm. The initial versions of Storm were released with an Even scheduler, which is still today the default option. Another scheduler, called the Resource-Aware Scheduler (RAS) and based on [PHH+15] is also available.



(a) *Even Scheduling*: placement example with 3 total workers; 2 executors per worker

(b) *RAS*: placement example with a configuration that allows 3 executors per woker; 3 workers used, 3 executors per worker

Figure 2.2: Example of Storm different ways of scheduling

Even scheduler

The Even Scheduler distributes the tasks following a round robin manner on a set of workers. The number of workers to use is specified by the user (default is 1), limited to the slots currently available. With this algorithm the executors are balanced over the supervisors, and so is the computation, if tasks have homogeneous CPU requirements.

The algorithm is based on a simple strategy as shown in Algorithm 1. **Step 1:** The available slots are sorted taking the supervisors in round-robin manner, e.g. if we have 20 slots in 5 supervisors, the first five slots will be each from a different supervisor. Then, the workers are assigned to the slots following the sorted slot list. **Step 2:** The topology executors are sorted by ID which by default entails to spouts first, then first bolt, etc., traversing the DAG in a breadth-first fashion. **Step 3:** The executors are finally assigned one by one to the workers, sorted as in Step 1.

For example, if a user requests 3 workers for the sample topology (Fig. 2.1) on a cluster composed of 2 supervisors with 3 slots each, the demand will be scheduled as follows. First, the slots will be sorted by worker and supervisor giving the following list: (Worker 1/Sup A, Worker 1/Sup B,Worker 2/Sup A...). The first 3 will be selected and the algorithm will then iterate over the components. The first spout component will be placed on 1/A, the second on 1/B... leading to the scheduling shown in Fig. 2.2a.

Algorithm 1: Even Scheduler

```
1 slots ←sortFreeSlotsBy(slot#, supervisor);
2 n ←min(requestedWorkers, availableSlots);
3 workers ←getFirstN(slots, n);
4 for e in topologyExecutors do
5     worker ← getNext(workers);
6     assign e to worker;
```

Resource Aware Scheduler (RAS)

The RAS scheduler aims at optimizing the resource utilization on the nodes while minimizing the network distance between tasks. In [PHH+15], the authors have defined the concept of network distance: inter-rack and inter-node connection are the slowest due to the network link, intra-node (i.e. inter-worker, through TCP) connection is faster, intra-worker (using serialization) is the fastest. Placing tasks with a higher connection level closer to each other should reduce the communication load and thus the latency. Moreover, this scheduler introduces the notion of memory and cpu resources provided by the cluster. The first one is specified as the amount of memory in megabytes. The second one, specified as cpu units, represents the computational power of a node and is normalized to $1\ core = 100\ cpu\ units$. The available resources per node has to be specified in the Storm configurations. When submitting a topology, a user can thus specify its resource requirements. If the scheduler is enabled but no requirement is specified, default values are used.

The RAS scheduler allows the definition of external scheduling strategies. The one in use by default is the `DefaultResourceAwareStrategy`, shown in Algorithm 2. **Step 1:** The strategy begins by first sorting components by the sum of their input and outputs connections. **Step 2:** Then, it selects the first executor of the first component in the list and assigns it to a worker. **Step 3:** it performs an ordering of its neighbors – always based on connections level – and places one executor for each of them. The loop is repeated until executors exhaustion.

Algorithm 2: Default RAS Strategy

```
1  components ←sortByInOutConn(topo)
2  for c in components do
3      repeat
4          e ←getNextExecutor (c);
5          worker ←getBestWorker(c);
6          assign e to worker;
7          nghbrs ←sortByInOutConn(C nghbrs);
8          for n in nghbrs do
9              e ←getNextExecutor (n);
10             if e is not null then
11                 worker ←getBestWorker(n);
12                 assign e to worker;
13     until e in c;
```

To select the best fitting worker for an executor, the algorithm sorts them by their network distance (racks or nodes) and the available resources, giving priority to the location where some other components of the same topology are already placed. At each executor placement, the scheduler decrements the occupied resources from executors.

In conclusion, the basic principle is to place as much executors as possible in the same worker; when this worker is full, it continues by filling other workers on the same node. If all the executors don't fit the same node, a new node in the same rack will be chosen, and so on. In this manner, the highest connected tasks will be placed the closest to each other as possible, reducing communication latency.

Applying the RAS scheduler to the sample topology of Fig. 2.1 will lead to the deployment in Fig. 2.2b. First it will compute the in-out degree of each components, taking into account the parallelism level. Spout S has 0 in links and 2 out links on the schema. Each out-link of S connects to components with a parallelism level of 2, the same as S. Hence, the total number of out-links is $2 \times 2 \times 2 = 8$. Bolt $B1$ has 2 in links from S, combined with its parallelism level this gives $2 \times 2 = 4$ in links, and $2 \times 1 = 2$ out links. Once sorted, S will be the first to be scheduled, and then it will be its neighbors $B1$ and $B2$. Since the algorithm gives priority to the smallest network distance, the components will all be placed in Supervisor A, i.e. the same node.

2.3 Apache Flink

2.3.1 Architecture Overview

From an architectural point-of-view, we can find several similarities between Flink and Storm. The distributed architecture of Flink follows as well a master-slave design. The master node is the *Job Manager*. It keeps the logic view of the applications in the cluster and runs the scheduling algorithm when a new application is deployed. Moreover, it manages the global status of the cluster, as well as other fundamental functions such as check-pointing and restore.

The slave nodes are called *Task Managers*. These nodes are where the application tasks will be placed and run. Each *Task Manager* contains a number of *Task Slots*, normally one per CPU core, to optimize the scheduling of tasks (Section 2.3.3). As Storm, also Flink is based over a Java runtime environment, thus the *Task Slots* are Java Virtual Machines.

2.3.2 Applications

The resulting logical view of a Flink application is always a DAG. As we said, it eases the scheduling and placement process. The two main components of Flink applications are: Connectors and Operators. Connectors are the source tasks that receive the flow of data from the internet and send it to the Operators. Operators are queries or custom operations performed over the data. They can also act as sinks. Similarly to Storm these operators can be parallelised to scale the application processing capabilities.

A major difference between Storm and Flink applications are on the Bolt and Operators implementation. In Storm each Bolt contains custom operations and are not constrained by default queries. Flink, on the other hand, in addition to custom operators, exposes a set of standard queries that can be performed over the data. Two main advantages of this solution are easier application development and optimization at system level for the query.

As any distributed system, data has to be transferred between tasks. The application can specify to which task data has to be assigned through connection strategies, same as Storm. Flink can load balance the tuples between tasks using the *rebalance* method, which will distribute data equally between processes. If we need to have all tuples of a certain kind processed by the same task, two tasks can be connected with the *keyBy* method. A key for each tuples is specified, so to send the same key always to the same task instance. In addition to methods similar to the ones we have in Storm, Flink supports also a *direct connection* method. If two tasks are directly connected (i.e. no rebalancing or grouping method is used), it creates a task-to-task connection. This connection

methodology will come in hand at the moment of task scheduling, as we will see in the next section.

2.3.3 Scheduling and Task Placement

Flink has only one default scheduler. Unlike Storm, it doesn't allow for an easy plug-in for external schedulers, except for customized versions like Alibaba's Blink [Tec18]. However, Flink's default scheduler, as opposed to the default one in Storm, tries to take advantage of application characteristics and cluster resources.

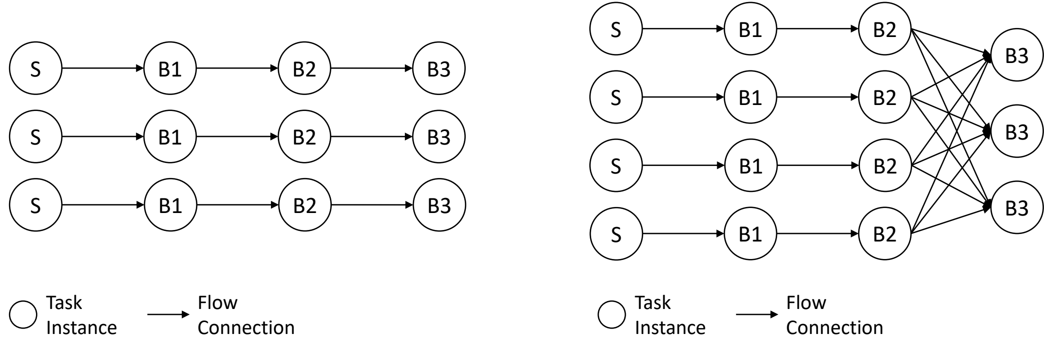
In Flink available resources are defined through Task Slots. A Task Manager is a JVM, thus it has a certain amount of memory assigned. Task Slots equally split the memory resources available in the task manager; e.g. if a Task Manager is configured with 1GB of RAM, and it has 4 slots (generally 1 slot per core), each slot will have access to 250MB of RAM. In this manner it ensures a certain level of resource isolation.

Tasks Placement

In each Task Slot, Flink places a pipeline of successive task instances. If an application has a defined parallelism level, each parallel pipeline is placed in a different task slot. The higher the parallelism level of the application, the higher the number of used slots. Consequently, the maximum parallelism level of a Flink application is limited by the number of available slots.

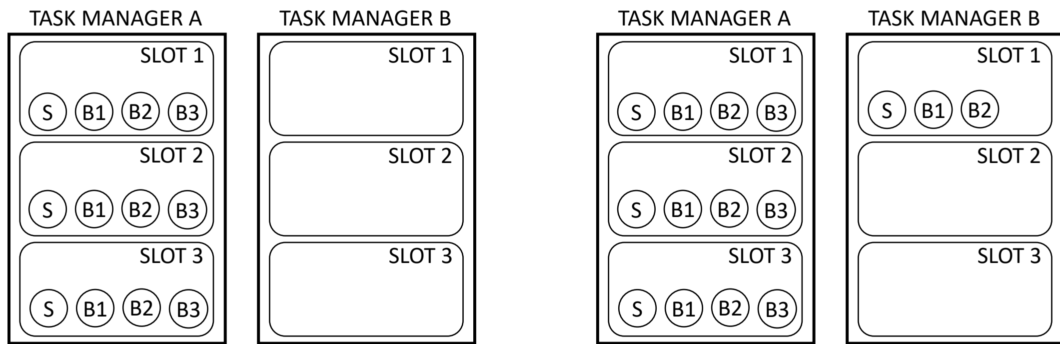
Flink places tasks from the same application close to each other. If multiple pipelines from the same application have to be placed, they will be placed in slots from the same Task Manager, to optimize communication latency, as with the RAS scheduler of Storm. Nevertheless, if we have to place more pipelines than the slots available in a Task Manager, the application will be spread over different Task Managers, spanning over the cluster nodes.

We take as example an application composed of 4 successive tasks connected in a direct manner, with no parallelism set. The 4 tasks will be placed all together inside the same slot. If the parallelism level for each task is 3 (Figs. 2.3a and 2.3c), we will have 3 different pipelines of 4 tasks each, placed in 3 different slots. However, if we have tasks of different parallelism levels, the pipelines to be placed will be of different length. If, among the 4 tasks: the first 3 has a parallelism of 4 and the last one a parallelism of 3 (Figs. 2.3b and 2.3d), Flink will have to place 3 pipelines of 4 tasks and one of 3 tasks (i.e. the first three task excluding the sink, as the three instances are already part of the other pipelines). The four pipelines will be regularly placed on four slots.



(a) Topology example with a parallelism of 3 per task

(b) Topology example with a parallelism of 4 per task, except the last one with 3



(c) Placement example with a parallelism of 3 per task

(d) Placement example with a parallelism of 4 per task, except the last one with 3

Figure 2.3: Example of Flink scheduling of a pipeline of 4 tasks

Operators Chaining

In Flink, each operator is a thread, so that they can be run in parallel. To optimize data processing and additionally reduce communication latency, Flink has an operator chaining mechanism. Operators, directly connected to each other (i.e. neither rebalance nor key grouping), form a chain of sub-tasks that can communicate directly, bypassing the network layer. A chain is treated as a single task (i.e. a single thread) and scheduled as such.

As shown in Fig. 2.4, if we have a 4 operator application, instead of having 4 different tasks, the ones directly connected will be merged together in a single task. If all 4 of them are directly connected (Fig. 2.4a), Flink will schedule it as a single task. If we have a more complex connection, e.g. rebalance or key grouping, the chain will be split where this connection happens. In Fig. 2.4b, we have

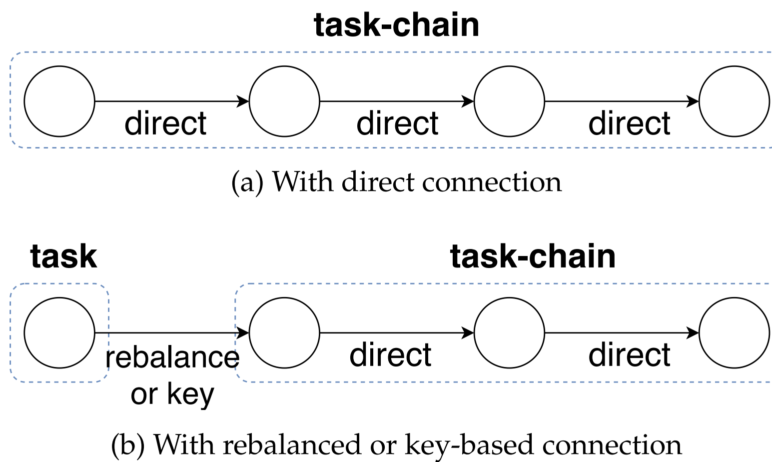


Figure 2.4: Flink task sub-chaining with different connections

a two-task application composed by the source task, and a single task merging the successive 3 operators. Once the logical view of the tasks is computed, the scheduling is performed as explained above.

Chapter 3

Non-Functional Tasks Placement

Contents

3.1 Related Work 29

3.1.1 Reliability and Fault Tolerance 29

3.1.2 Application Scheduling and Task placement 30

3.2 Storm's Acking Framework 32

3.2.1 Message Reliability 32

3.2.2 Backpressure 33

3.3 The Impact of ACKing 34

3.3.1 ACKers in task placement 34

3.3.2 Performance degradation 34

3.4 ACKers-aware Scheduling 40

3.4.1 One-per-Worker Strategy 40

3.4.2 Isolated Queue Strategy 41

3.5 Evaluation 42

3.5.1 Experimental Setup 42

3.5.2 Application 43

3.5.3 Methodology 44

3.5.4 Results 47

3.6 Conclusion 52

Modern DSP engines are designed to apply complex processing on data as a sequence of tasks to be executed [dAdSVB18]. Each task can be replicated

and distributed over the cluster to scale the application, increasing throughput capacity. The task scheduling algorithm plays an important role to optimize the task placement, trying to balance the computation load and minimize the end-to-end latency. Following the distributed paradigm, every DSP engine implements, in its own manner, non-functional components which implements mechanisms such as monitoring, logging or message processing guarantee. Regarding the latter, most middlewares propose a way to perform exactly-once processing, while all of them offer at-least-once. Message reliability can sometimes hurt the application performance [CDE+16a] as enforcing delivery guarantees on messages can come at the price of a reduced throughput or an increased processing latency.

Several works concentrate on the fault tolerant and reliability aspect of stream processing applications (Section 3.1). Every DSP engine proposes its own approach to the problem, optimizing the offered delivery paradigm. A large set of efforts focuses on improving fault tolerant checkpointing systems. However, to the best of our knowledge, no work has ever precisely analyzed the impact of message guaranteeing on the performance of DSP applications, especially, acknowledgment based systems.

In this chapter we show how an ack-based framework offering message delivery guarantee can have an unexepected impact to the performance of a DSP application. The two main systems implementing such mechanism are Google MillWheel [ABB+13] and Apache Storm [STOb]. We take advantage of Storm's open-source nature that allows us to fully analyze the framework. Storm deploys the tasks of an application in Java Virtual Machines (JVM) on the nodes of the cluster. We study the strategies used to manage acking by the two popular schedulers of Storm, namely the Even Scheduler and the Resource-Aware Scheduler (RAS) [PHH+15]. In both of them, the acking mechanism is materialized as tasks to be deployed on the cluster.

We demonstrate that the Resource-Aware Scheduler considers only marginally the acker tasks (a.k.a. ackers) during placement, resulting in degraded performance as compared to the Even scheduler. We next design and implement two task placement strategies for the RAS scheduler that take in account ackers and optimize their placement. The first strategy balances the acking load, collocating the ackers with other application tasks. The second strategy places the ackers in dedicated JVMs, separating them from the application tasks, alleviating the load of the incoming message queues.

We evaluate these two solutions in single cluster and multi-cluster environments, showing how in both cases we can improve the performance of the RAS scheduler. In particular we show how in a single cluster scenario, our strategies enable the RAS scheduler to catch up with the Even Scheduler in terms

of throughput while beating it in terms of latency. In the multi-cluster environment where the RAS scheduler approach enables it to outperform the Even scheduler [PHH+15], our acker placement strategies enable to further improve the RAS throughput.

The main contributions of this chapter can be summarized as follows:

- We demonstrate the impact of the middleware message delivery system on the application performance.
- Focusing on the Storm case, we show that the acking system generates a large share of the network traffic. Moreover, the placement of ackers can significantly impact the overall performance of the application.
- We present two ackers placement strategies that improve the performances of the two standard schedulers.

The rest of the chapter is organized as follows: related work is presented in Section 3.1. A detailed explanation of the acking framework is given in Section 3.2. Then, in Section 3.3 we precisely define the problem we study. In Section 3.4 we explain the two ackers placement strategies we propose, that we evaluate against the legacy Storm schedulers in Section 3.5. Section 3.6 concludes the chapter.

3.1 Related Work

3.1.1 Reliability and Fault Tolerance

Each DSP engine implements message processing guarantee in its own way. Storm implements an upstream backup, which keeps track of the messages along the processing path through acknowledgments [TTS+14]; similarly to Storm, Heron [KBF+15] and Millwheel [ABB+13] implement an acknowledgement-based upstream backup. In addition, both systems use an auxiliary checkpointing system to further improve message reliability. Apache Flink provides a checkpointing mechanism that continuously stores the state of the system [CKE+15]; Spark Streaming relies on Spark Resilient Distributed Dataset (RDD) support and the different guarantees provided by the external data sources [HL15]. Meanwhile, Samza adopts a changelog approach [NPP+17].

Fault tolerance of DSP is also a hot research topic [ABC+15]. Most works concern checkpointing and snapshot techniques. Zhuang et al. [ZWL+18] propose a novel Optimal Checkpointing Model for stream processing. This model proposes a dynamic calculation of an optimal checkpointing interval, aiming to

obtain an optimal processing efficiency. A more workflow-generic approach is taken by [HFC+18], where the authors tackle the problem of corrupted data failures, so called silent errors. They optimize the checkpointing overhead, for fail-stop errors, by proposing different combinations of scheduling algorithms and checkpointing techniques. Carbone et al. [CFE+15] propose an asynchronous snapshotting algorithm for stream processing dataflows, where they minimize the space requirement of snapshots.

Another popular solution is to perform replicas of the application. A dynamic replication scheme is presented in [HZK+15] that continuously monitors the system deciding the optimal technique to apply for the workflow. Cardellini et al. [CGLPN17a, CGLPN17b] formulate an optimal DSP replication and placement model, where they compute a number of replicas for each task to optimally scale the application. In [SZ16] is presented a DSP engine that implements a checkpointing system combined with a partial replication of tasks, in order to reduce the cost of the system recovery and the necessity of backup nodes.

Recent works are concentrating on upstream backup systems. The proposal from Li et al. [LWJ+17, LWJ+18] suggests a solution considering tasks-failures. In their work they describe a task allocation strategy that takes into account the impact of tasks recovery over the cluster resources.

All these works propose novel solutions to fault tolerant systems but none actually investigate the cost of their implementation on real applications. In our work we focus on message reliability and on its impact over the application performances at runtime.

3.1.2 Application Scheduling and Task placement

When talking about resource limitations, several works propose novel scheduling solutions directed to optimize resource utilization and minimize communication latency, with particular attention over network communication load in relation to task placement [AFM17]. A large set of works has been done in relation to Storm.

Similarly to R-Storm [PHH+15], extensively discussed in Section 2.2.3, two past works [ABQ13] and [XCTS14] propose different schedulers pivoting around the concept of network distance. Both works focus their solution on minimizing the network communication, implementing at the same time a monitoring system to optimize at runtime the application's resource utilization.

T-Storm [XCTS14] implements an online version of R-Storm including a new load monitoring component that collects different system cpu workloads and inter-executor traffic load. The algorithm optimizes cpu and communication

traffic in a more dynamic way, using these statistics, performing a seamless topology re-balancing during runtime.

Similarly, [ABQ13] propose a scheduler pivoting around the concept of network distance. It focuses on minimizing the network communication, implementing at the same time a monitoring system to optimize at runtime the application's resource utilization.

D-Storm [LB17] implements a dynamic version of the resource aware scheduler. In their implementation, they exploit a MAPE (Monitoring, Analysis, Planning, Execution) architecture to improve the placement algorithm.

Also [ZLZL16] focuses on assigning tasks to the same slot with other highly connected tasks. They present an online scheduler that thanks to a traffic relationship model that combined with cpu utilization is used to optimize the application load balancing.

All these efforts on minimizing inter-node communication has been summarized in [AFM17] where is concisely demonstrated the impact of the tasks placement in relation to the network traffic and how the proximity of high communicating tasks could improve the application throughput.

Cardellini et al. [CGLPN15a, CGLPN15b], try to extend the scheduling to a wider Quality of Service (QoS) point of view, focusing on distributed scenarios like Fog Computing [BMNZ14]. They implement an online scheduler that considers latency, CPU utilization and data traffic, trying to minimize the traffic between components and optimize the system availability.

Eskandari et al. [EHE16, EMHE18] implement two online schedulers based on graph theory. They consider network traffic and resource allocation, with the objective of minimizing data communication between tasks and optimize resource allocation. Both their solutions exploit graph partitionning methods to improve the tasks ordering and placement of R-Storm.

Another work [LXTW18], states that it is too hard to optimize the workload balancing through linear programming models. Thus, to tackle the problem they define a Deep Reinforcement Learning framework, implementing a model-free Deep Neural Network able to learn and predict optimal placements for data stream processing systems, without any model-specific boundaries. They are able to improve Storm performance and to find an optimal task placement during execution.

All the presented works, in diverse fashions, optimize the application's resource utilization; however, none of them directly consider the acknowledgment framework of DSPs. This work will show how a task-oriented acking system, such as the one of Storm, can impact the system performances. Specifically, we demonstrate how not only the application tasks have to be optimally

placed, but so should the tasks dedicated to acking. Even though Storm problems with large topologies and high traffic has been analyzed in different contexts [CDE+16b], nothing in the literature ever analyzed, from a runtime point of view, the impact of the reliability system over the application performances. These important academic efforts on improving resource allocation and computing load, generates global performance-oriented solutions. However, rarely the evaluation takes in account large topologies, or even the size is not specified at all. In our work we address the resource allocation problem, considering task allocation in relation to the Storm acknowledgment mechanism, aiming to improve the application performance.

3.2 Storm's Acking Framework

The acking framework in Storm serves two functions (as of Storm version 1.2.x): message reliability and backpressure. In the first case [Stod], to ensure the at-least-once processing property, every tuple in the workflow will be acked by each task. In case of failure, they will be re-transmitted by the previous executor. In the second case, each spout maintains the status of in-flight tuples and if the number of tuples waiting to be acked exceeds a threshold, the backpressure mechanism will be activated, slowing down the spouts.

3.2.1 Message Reliability

Storm uses special system tasks, called ackers, to manage the status of the tuples, which are designed to be lightweight [Stod]. The path of tuples in the DAG is depicted as a tree and the ackers are able to update step-by-step the completion status of each tuple running in the system, keeping track of eventual tuple duplication and joins. These tasks are implemented as any other executor [Stoa] and process incoming tuples in a similar manner. They can receive ack tuples from both spouts and bolts, through two different streams. The Spouts use the `INIT` stream to indicate the creation of a new tuple to the application workflow and the Bolts use the `ACK` stream to ack the tuple along the tuple tree.

For every new tuple, a spout generates a random ID and sends its XORed value to an acker (Fig. 3.1). Having different acker executors, the destination acker is decided through a mod hashing to always send the same ID to the same executor. The acker that receives the `INIT` message, keeps a two-entry table where it stores the source spout ID and the XORed value received. This table is stored in a specific data structure called `RotatingMap` [Stoa]. In addition to storing the table, this structure acts as a sliding window, to trace the time needed by a tuple to be processed. In this way is implemented a timeout for

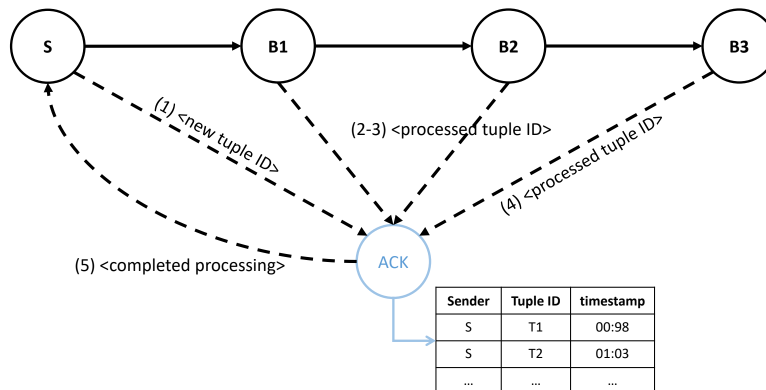


Figure 3.1: Acking framework message exchange example

tuple completion. By default if after 30 seconds a tuple has not yet been acked, it is considered as lost (i.e. failed). It will then be retransmitted by the last bolt which has acked it or by the spout if it was lost at the beginning.

Upon processing a tuple, a bolt can anchor resulting tuples to the original one, creating a tree-like relation. Every bolt in the topology will send, as ack, the XORed value of the ID of the tuple or the partial value. The partial value is used in case the tuple has been sent (i.e. duplicated) on different branches during its processing. This allows to keep track of tuples that are sent to multiple tasks (i.e. creating duplicates in different directions). To consider a tuple completed the acker will wait for the acks from all ramifications.

The root tuple will be fully acked only when all its children will be. This happens when the tuple reaches the end of the tree. The entry value in the acker table will be 0, thus the acker will send a message to the originating spout notifying it that the tuple completed its life cycle.

3.2.2 Backpressure

The acking framework is also used to enforce a backpressure mechanism. The mechanism is based on an integer value that can be set through the `max.spout.pending` configuration option. This value represents the maximal number of non-acked tuples allowed before triggering backpressure. If not set (by default), there is no ack-based backpressure.

Each spout keeps a list of non-acked tuples, i.e. how many tuples the spout

sends which are not yet completed processing. If, at some point during execution, the number of waited tuples is larger than the maximum defined value, the spout will slow down its emission rate, reducing the global throughput of the application. This can give time to the tuples to be processed and acked, avoiding overloading bolts memory by storing too many tuples in the incoming queue. The spout will slowly adapt the emission rate to always keep the number of waiting tuples smaller than the upper limit.

3.3 The Impact of ACKing

3.3.1 ACKers in task placement

Even Scheduler

Implemented as tasks, ackers need to be scheduled similarly to the other operational components of the topology. As we said in Section 2.2.3, the default scheduler places in round-robin the tasks of the application. The ackers are placed in the same way at the end of the operational task placement. By default the even scheduler places one acker per worker. Thus, the final placement will be of one acker per worker.

Resource-Aware Scheduler

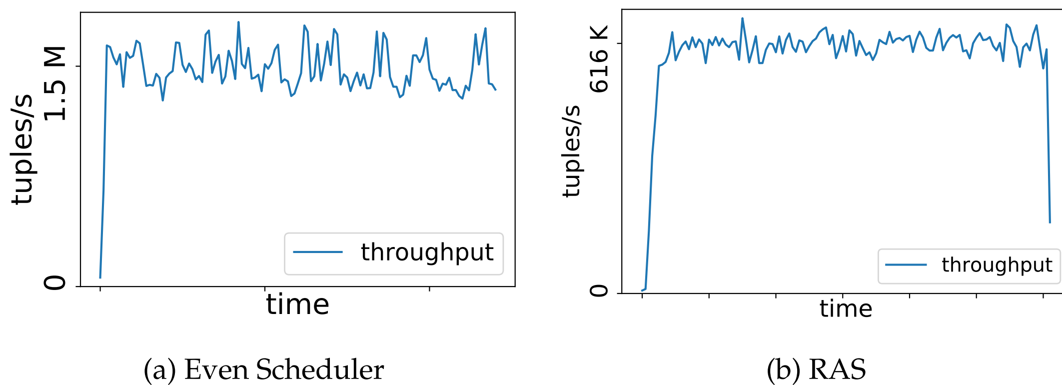
Similarly to operational tasks, in RAS, ackers consume resources. They are placed at the end of the process similarly to the even scheduler. For such reason, they will be placed in the already used workers if resources are still available, otherwise a new worker will be created. By default the RAS places only one acker task, differently from the one per worker of the Even scheduler. However, an higher number of ackers can be used if explicitly specified by the application developer. We discuss the consequences of this strategy in the remaining of the chapter.

3.3.2 Performance degradation

When dealing with resource optimization in DSP systems, the recurrent resources taken in account are: CPU, memory and network. Most commonly, literature solutions focus on the bottleneck generated by task communication [ABQ13, PHH+15, XCTS14, AFM17, ZLZL16]. Among these resources it results [ABQ13, XCTS14, CGLPN17b, CGLPN15b] that the most constraining one is the CPU. In fact, if the application tasks are not fast enough to process the incoming data

and excessive buffering may result in memory problems, the back-pressure system is able to slow down the application, giving time to the executors to process the queues. The communication load optimization has been undertaken through simple co-placement heuristics [ABQ13, PHH+15, EHE16, EMHE18], that are shown to improve the processing latency. However, CPU balancing is a more complex matter. Since it is hard to predict the impact of each task and the load generated a priori, the majority of works implements some monitoring at runtime [PHH+15, LB17]. When overload is detected, some re-balancing can be performed but it requires adapting the scheduler to these runtime statistics. In most cases the schedulers implemented by the DSP middleware expose some configuration parameters to adapt the application to the underlying system (e.g. RAS scheduler in Storm [PHH+15]). Anyhow, these systems require a pre-brenchmarking phase to understand the application performance and take appropriate measures.

Hence, the initial work of this thesis was focusing on analyzing the behavior and functioning of those schedulers. In this section we demonstrate the widely varying performance observed in our experimentation[1] (Fig. 3.2), for the two different Storm's scheduling algorithms.



(a) Even Scheduler (b) RAS

Figure 3.2: Storm tuples throughput

Both schedulers expose to the user some configuration parameters which can be used to direct the deployment of an application. The Even scheduler allows to set the number of workers to use. Specifying large values will lead to spread the tasks over multiple nodes of a cluster. With the Resource-Aware scheduler, it is possible to define CPU and memory requirements for every component [PHH+15]. These values will directly impact the number of nodes used by a topology. Predicting the optimal values for these parameters is a non-trivial

[1] the experimental setup is the same as described in Section 3.5

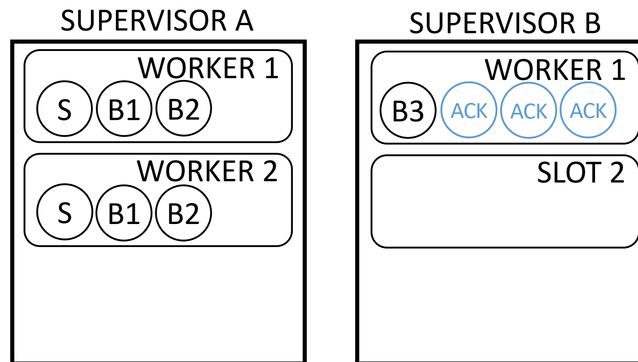


Figure 3.3: *RAS* ackers placement example with a configuration that allows 3 executors per worker; 3 workers used, 3 total ackers co-placed with an operational task

challenge. Hence, both schedulers usually require a tuning phase to find the most fitting values in order to optimize the throughput and latency of an application.

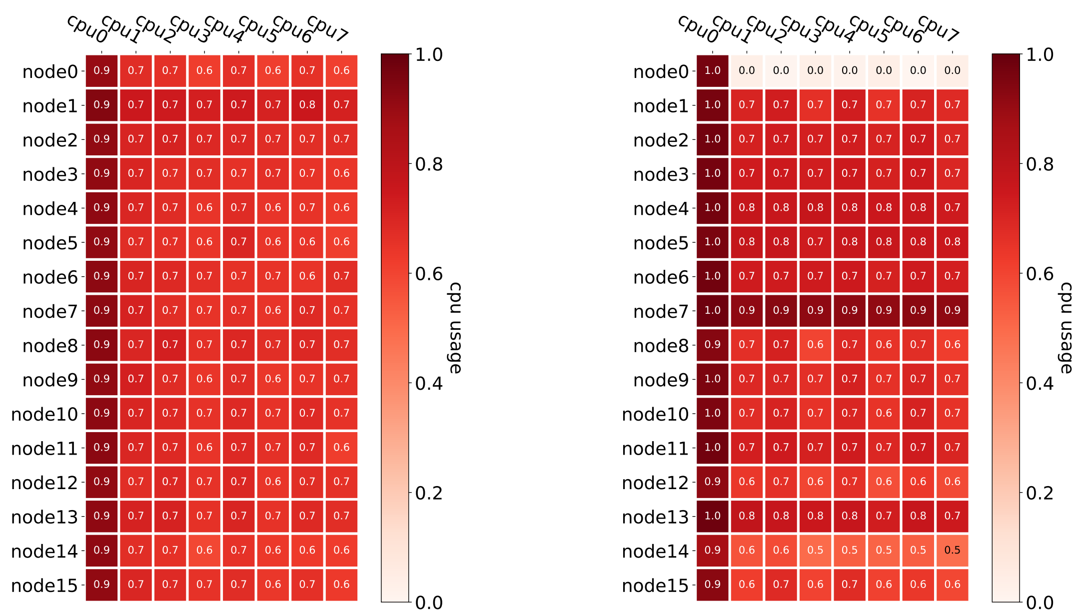
Nevertheless, even with a meticulous tuning of these values it is possible to encounter further placement problems caused by components out of user's control. As mentioned before (Section 3.2), Storm implements message guaranteeing through special system executors, which have to be placed by the scheduler. As shown in Section 2.2.3, the resource-aware scheduler tries to perform resource optimization with an offline algorithm. It is based on user defined parameters for both the cluster nodes and the topology components. Ackers' requirements have a default value of 10 CPU units and 128MB of memory. Default resource requirements can be changed through a configuration parameter, not directly from the application code as for the operational task customized requirements.

With some particular combination of user provided parameters, the placement of the topology components may precisely fit the used nodes, not leaving enough space for the acker executors. This produces a final placement where the ackers ends up in the last used node with still some free space, or even in a new separated node. In this situation, all ackers will be co-placed inside the same node, or even the same worker, together with operational tasks.

An example of such placement result can be seen in Fig. 3.3. In a configuration where a Worker has enough resources for 3 tasks, the last one (B3) has to

be placed on a third worker. The ackers are then co-placed with B3 as it is the only one with available space for the three ackers. The total of four tasks can be co-placed in the same worker as ackers consume, by default, less resources than an operational task.

This co-placement of ackers can cause a degradation of performances because of the increased load they put on the CPU. This can be observed by directly measuring the load on each node as shown in figures Figs. 3.4 and 3.5. When several ackers are placed inside node 7 (Figs. 3.4b and 3.5b), it increases the number of processes competing for the CPU, increasing the load, and thus slowing down the entire topology. Meanwhile, a more balanced ackers distribution (Figs. 3.4a and 3.5a) will even the load and ultimately offer better performance.

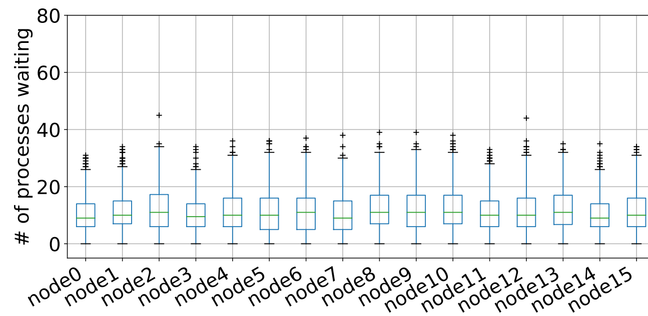


(a) Even Scheduler: ackers balanced over all nodes

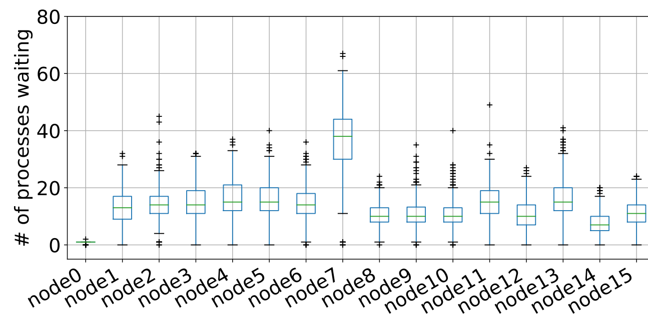
(b) RAS: Majority of ackers co-placed in node7

Figure 3.4: CPU utilization of a Storm cluster: two different schedulers

This can also be observed at the middleware level. Each executor has an incoming message queue called the *distruptor queue*, as well as an output transfer queue common to the worker where they are placed. When executors, i.e. ackers, are co-placed inside the same worker, their queues share the same memory space. When the load is well balanced, the tasks have a good processing rate and they are quickly consuming from the queue (Fig. 3.6a). However, if multiple ackers are co-placed inside the same worker node, with other operational

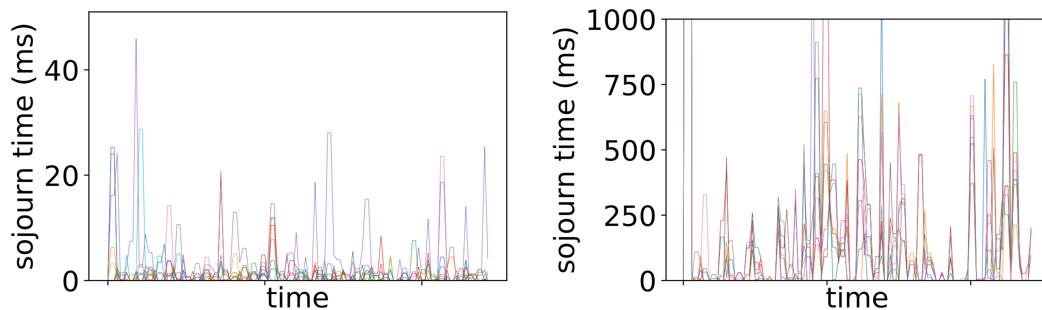


(a) Even Scheduler: ackers balanced over all nodes



(b) RAS: Majority of ackers co-placed in node7

Figure 3.5: CPU load of a Storm cluster: two different schedulers



(a) Worker in a balanced scheduling (b) Worker in a overloaded node

Figure 3.6: Example of workers executors disruptor queues: sojourn time

tasks, the tasks won't be able to keep up with the rate of incoming tuples. The competition for memory (i.e. buffers) between the acking process and the data processing will result in more congested queues with highly variable sojourn times (Fig. 3.6b).

Network Traffic To enable message processing guarantee, Storm needs to generate an acking message for each tuple processed by each task. As a consequence, this mechanism can generate a significant amount of traffic. Measuring network traffic — i.e. inter-worker and inter-node TCP traffic — during the experiments, we observe (Fig. 3.7) that in certain cases, the amount of traffic generated by the sole acking mechanism can be more than half of the total traffic produced by the running topology.

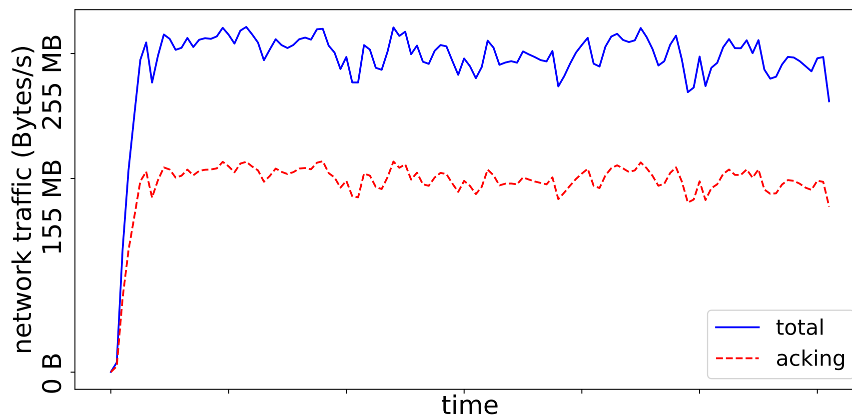


Figure 3.7: Storm application network traffic using 7 char long java string tuples

With large applications and a huge input rate, the number of tuples to be acked will be critical, exacerbating the problems described above. Even if the acker executors are designed to be lightweight, the high load of acking messages to be processed can exceed their capabilities, slowing down the processing rate. As a consequence, the number of pending tuples would increase and the back-pressure mechanism will choke the spouts. Simultaneously, the number of tuples waiting in the queue will build up, reaching the queue maximum and increasing the risk for new incoming tuples to fail.

Even at a small scale, the placement of ackers could impact the communication latency. If they are co-placed in a single node, it increases the probabilities that the acking messages travel through the network, instead of reaching directly a acker in the same node (or, better, in the same worker). In such a situation, the optimization performed by the RAS scheduler to minimize the communication is at risk of being nullified by the increased distance to ackers.

3.4 ACKers-aware Scheduling

In this section, we propose two extensions to the RAS scheduler that take into account ackers during tasks placement. The default number of ackers in the default RAS is equal to 1. This choice obviously leads to performance degradations with increasing application load. A more efficient acking strategy requires to: 1) set an higher number of ackers and 2) devise a strategy to map ackers to workers.

For the first problem, we rely on the heuristic used in the Even scheduler, which is to set a number of ackers equal to the number of workers. The heuristic is necessary because Storm's scheduler implementation requires to know a priori the number of ackers that are going to be used. Likewise, the number of used workers is not known in advance in the RAS scheduler, as opposed to the Even scheduler where it is set by the user. In the RAS scheduler, the number of workers is the result of the placement process based on the requirements of the component tasks in terms of CPU and RAM. To work around this issue, we use the components requirements with the CPU capacity of the nodes (resp. the JVM memory) to estimate the maximum number of executors a node can contain (resp. maximum number of executors per worker). We obtain the number of workers per node and the total number of nodes required. This gives us an estimation of the total number of workers.

The second problem, mapping ackers to workers, can be addressed using two different strategies that will be detailed below. To remove every resource constraint and allow us to chose where to place the acking tasks, we set to 0 their CPU and memory requirements. This gives us all the needed flexibility to implement the proposed scheduling algorithms.

3.4.1 One-per-Worker Strategy

The One-per-Worker Strategy (OPW) focuses on balancing the acking process computation load over the workers, instead of co-placing them in one single node with available resources. On large topology or large clusters, this translates into a more uniform load over all nodes. The idea is to replicate the scheduling strategy of the Even Scheduler by placing one acker per worker while preserving the RAS algorithm for the other components (Fig. 3.8).

The algorithm starts with an unmodified RAS placement strategy for the components (Algorithm 2 in Section 2.2.3). During this first phase, OPW gets the list of workers and during a second phase, acker placement is performed. Ackers are assigned in a round robin manner in each worker (Algorithm 3).

The benefit of this approach is that it combines the default scheduling of RAS (minimized communication latency) and a better load distribution for ackers

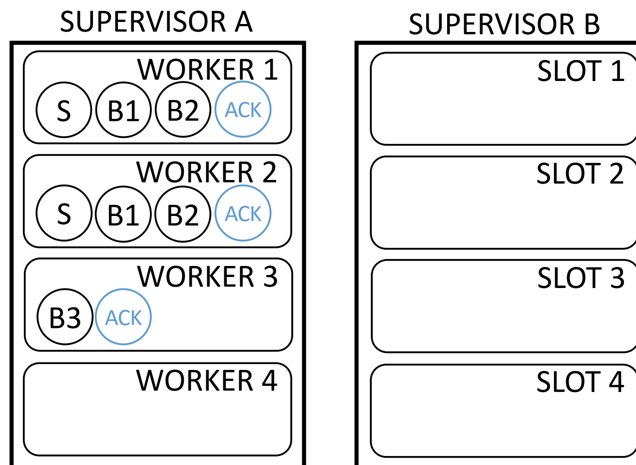


Figure 3.8: *OPW* placement example with a configuration that allows 3 executors per worker; 3 workers used, 3 total ackers placed one per worker; N.B.: ackers don't consume RAS resources

Algorithm 3: OPW: One-Per-Worker Strategy

```
1 RAS assignment of topology executors to workers;
2 for a in AckersToBePlaced do
3     w ← getNextUsedWorker();
4     assign a to w;
```

processing.

3.4.2 Isolated Queue Strategy

This second strategy, called Isolated Queue (IQ), isolates the messaging queues used by the ackers. The rational is to avoid competition with other executors over the incoming queue. This mechanism has the added benefit of reducing crashes due to incoming queues using all the available memory.

The IQ strategy deploys the same number of ackers as the OPW strategy, i.e. one per worker. But instead of placing one acker per worker, it groups them in a single worker, as it is shown in Fig. 3.9. Algorithm 4 shows the algorithm executed after the RAS algorithm has finished placing the other components. The IQ strategy first obtains the nodes where the current topology assignment has reserved some slots, and then creates one worker for each node. The algorithm then cycles the nodes and the ackers to be placed. If in the selected node, a ded-

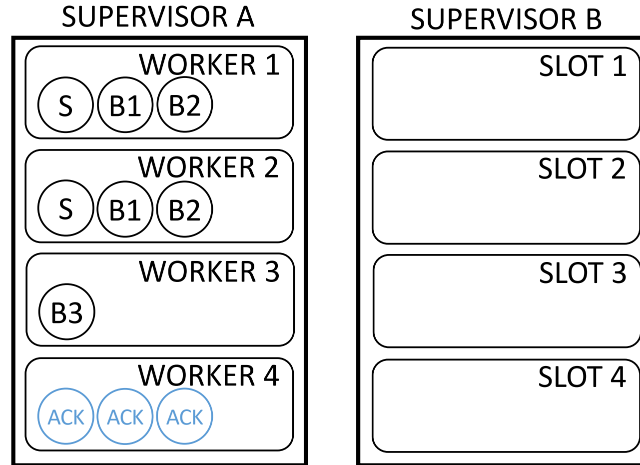


Figure 3.9: *IQ* placement example with a configuration that allows 3 executors per worker; 4 workers used, 3 total ackers placed on a dedicated worker

Algorithm 4: IQ: Isolated Queue Strategy

```
1 RAS assignment of topology executors to workers;
2 for a in AckersToBePlaced do
3     n ← getNextUsedNode();
4     w ← getAckerWorker(n);
5     if w is null then
6         w ← createAckerWorker(n);
7     assign a to w;
```

icated worker is not present it will create it and deploy a first acker in it. If it already exists, it will co-place another acker in the same worker.

3.5 Evaluation

3.5.1 Experimental Setup

To evaluate the impact of our strategies over the application throughput and average processing latency, we performed several benchmarks comparing the Even, RAS, OPW and IQ schedulers. The tests have been performed on the Grid5000 testbed[2], that allows us to reserve computing nodes in different clus-

[2]https://www.grid5000.fr/

ters. For our tests, we mainly used two clusters: the first one, `suno`, is located in the `Sophia` region and its nodes have two 4-cores Intel Xeon E5520 @2.27GHz with 32GB of memory. The second one, `parapide`, is in the `Rennes` region and offers nodes with Intel Xeon X5570 @2.93GHz and 24GB of memory. Inside a cluster all nodes are interconnected with 1Gbps links. The two sites are 850km apart and connected with a 10 Gbps dark fiber with a measured latency of 21ms.

Based on the evaluation cluster adopted by [TTS+14], we deployed Storm 1.2.1 over 17 nodes: the Nimbus and the Zookeeper server are co-placed in one node, the remaining 16 nodes host the Supervisors. Each Supervisor consists of 20 available slots with a maximum memory heap of 1024MB each. For the RAS, OPW and IQ schedulers, the resources configurations in each node are of `800` cpu units (two 4-cores cpus) and `32768MB` (resp. `24576MB`) of memory for `suno` (resp. `parapide`).

3.5.2 Application

Based on the scenario previously described in Section 3.3.2, where we find a cpu overload problem in Storm applications, we focus our benchmarking on a CPU intensive application. Word Count (Fig. 3.10) is a canonical representative of this family of applications, used likewise in the BigDataBench suite [WZL+14] as a representative of social network analytics.

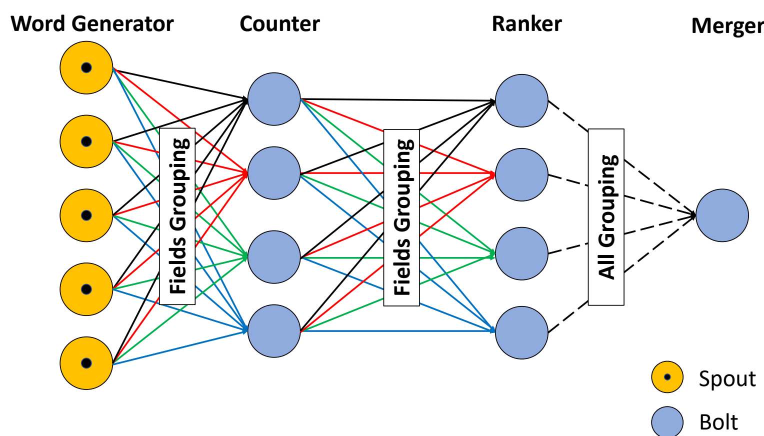


Figure 3.10: Simple streaming Word Count topology

The topology consists of four components. The Word Generators (i.e. the spouts), continuously inject in the topology random tuples obtained from a set of 1000 predefined words, under the form of `WordXYZ`, where `XYZ` is a number

that goes from `000` to `999`. The Count bolts receive the generated words as input, through a hash-based `fieldsGrouping` connection (the word itself is the key used for the grouping), and produce pairs `word, counter` as output, where `counter` is an incremental counter of the word's occurrences.The counter is kept in a rolling window updated and managed by a parallel thread. The Rankers receive these pairs as input, also through a `fieldGrouping` connection, and provide the ranking of the three most frequent words every two seconds. They use a sorted list of key-value entries (i.e. word, counter) that updates every time it receives a new count; if the counter of the received word enters the top three, it is added to the list and the fourth in terms of rank, is discarded, otherwise the received input is discarded. The final step is accomplished by the Merger. It is a single executor to which the Rankers connect through a `globalGrouping`. It receives the rankings from all the previous bolts and outputs the overall top three most frequent words, by simply merging all the three-words lists and log the top three words.

We set the resources requirements for the ackers as explained in Section 3.3.1, and adjust the pending tuples value, `max.spout.pending` (see Section 3.2). We set the pending value to 5000, which is high enough to allow the application a wide margin but that enables backpressure before worker memory saturation. Acking is enabled for every bolt. In this way we allow Storm to automatitcally enable the backpressure mechanism and we can study the impact of the acking framework over the application performances. All components have the same resource requirements in each test.

3.5.3 Methodology

Application Configuration To perform a comprehensive evaluation, we test the topology with two different levels of parallelism and with diverse resource requirements. Based over past works, [TTS+14, LXTW18] and only considering the topology components, i.e. excluding the system executors, we define a **large topology** with a total of 376 executors, and a **small topology** with a total of 26 executors. The large topology features 75 word generators, 150 counters, 150 rankers and 1 merger. The small topology is configured with 5 word generators, 10 counters, 10 rankers and 1 merger. We tune the different resources requirements to generate various scenarios and to better compare the four scheduling strategies:

- **Large topology:** the large topology has been investigated with several configurations (Table 3.1) corresponding to different network distances between tasks. The Even scheduler has been tested with 13, 16 and 60 workers. Increasing the number of workers increases the number of used

Scheduler		CPU Units	Mem (MB)	Workers	Nodes	Ackers
		n.d.	n.d.	60	16 | 32	60
Even		n.d.	n.d.	16	16	16
		n.d.	n.d.	13	13	13
	^	33	128	n.d.		
		30	128	58	15	60
RAS-Def		30	64	29	15	30
	*	30	32	22	15	15
	*	27	32	25	13	13
		33	128	47	16	48
	*	30	128	58	15	60
RAS-OPW		30	64	29	15	30
		30	32	15	15	15
		27	32	13	13	13
		33	128	63	16	48
		30	128	73	15	60
RAS-IQ		30	64	44	15	30
		30	32	30	15	15
		27	32	26	13	13

(^) not enough resources to schedule (*) out of memory crash

Table 3.1: Summary of tests run for the large topology. Total executors: 376 + ackers

nodes, changing the communication distance between the executors. The RAS, OPW and IQ schedulers have been tested with 5 different variants. Starting from a requirement per component of 33 CPU units and 128MB, to spread the topology as much as possible in the single cluster, down to a requirement of 27 CPU units and 32MB of memory, to consider the effects of aggregating the tasks on a smaller number of nodes and workers.

- **Small topology:** the small topology has been tested with a smaller set of configurations.The Even scheduler has been tested with 16 and 13 workers. With 16 workers we are able to spread the tasks as much as possible over the available nodes. While, with 13 workers, more executors will be co-placed. Thus, we can observe the effect of node sharing. With the RAS scheduler we have tested two different configurations. The first one

Scheduler	CPU Units	Mem(MB)	Workers	Nodes	Ackers
Even	n.d.	n.d.	16	16	16
	n.d.	n.d.	13	13	13
RAS-Def	400	128	15	14	13
	265	128	10	9	9
RAS-OPW	400	128	13	13	13
	265	128	9	9	9
RAS-IQ	400	128	26	13	13
	265	128	18	9	9

Table 3.2: Summary of tests run for the small topology. Total executors: 26 + ackers

maximizes the spreading of the topology. Since there is a total of 26 executors and 16 nodes with a capacity of 800 CPU units each, this can be achieved by requiring 400 CPU units for each component. This will result in 2 executors per node, which is the best achievable configuration given the number of available nodes. The second configuration places three executors per node by setting the CPU requirements to 265 units.

Cluster Setup We first benchmarked the application in a single cluster scenario, where the CPU limitation is more visible. Then, we moved to a multi-cluster scenario, so as to add a slower link that can impact the application latency. For the multi-cluster scenario, we added 15 nodes on the second cluster. The tests have been repeated with the same configurations used for the single-cluster case.

Evaluation Metrics We focus our evaluation on two key data stream metrics: application throughput and processing latency. Throughput is obtained by summing the number of emitted unique tuples by every spout executor over the duration of the experiments. Latency is the average latency computed by Storm ackers. For a given tuple, it is the time between its registration at an acker and the completion of the acknowledgment tree. Hence it is a direct estimator of the end-to-end processing time. Every test has been run for 20 minutes and the first and last 5 minutes were excluded from measurement to account for the warm-up and shutdown phases. The results presented correspond to a steady state of the topology.

3.5.4 Results

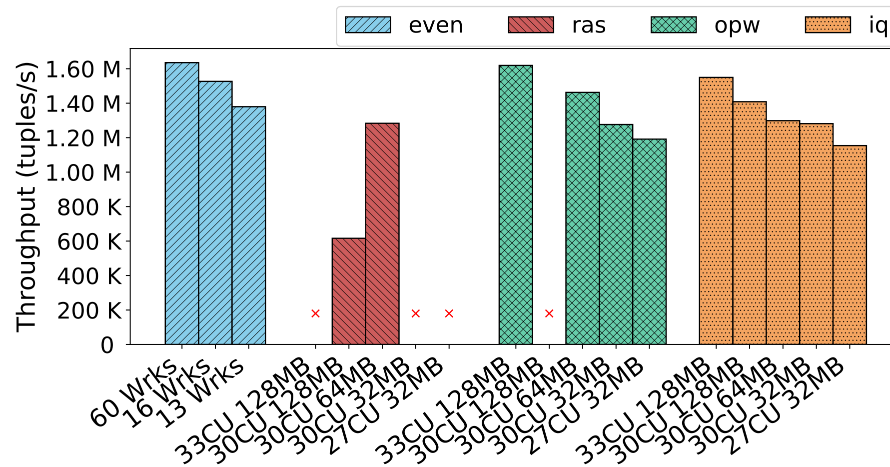
Single Cluster

Large Topology In this scenario, the Even scheduler (blue bars in Fig. 3.11) behaves as expected: increasing the number of workers increases throughput (Fig. 3.11a) but also processing latency (Fig. 3.11b). Indeed, increasing the number of workers while keeping the number of executors per node constant slightly increases the available memory for each executor. This results in more space available for the buffering queues. However, at the same time, a larger number of workers increases communication latency between executors in the same node, adding inter-worker communication.

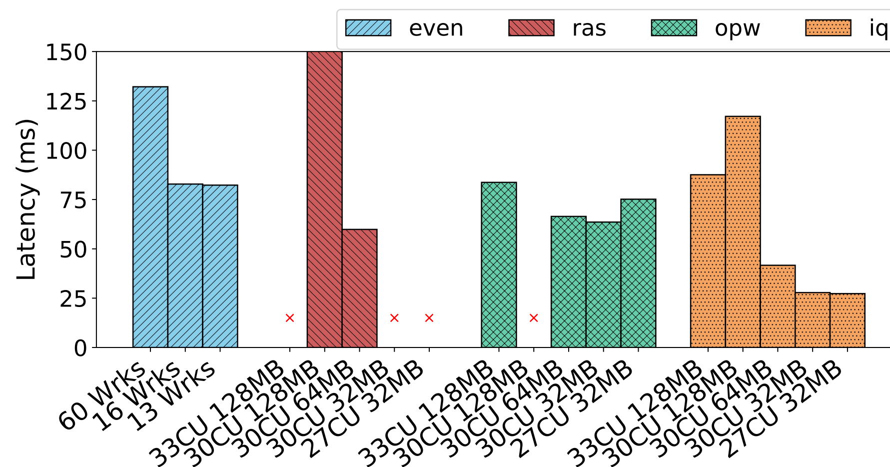
In our experiments with the RAS scheduler, some configurations would run out of memory and crash (missing values in Fig. 3.11a and Fig. 3.11b indicated by a cross). The only two working configurations were the ones with a requirement of 30 CPU units (CU) and with 128 and 64MB of required memory respectively. With a configuration of 33 CPU units per component, we don't have enough space in the cluster to place all the components. Meanwhile, with the last two RAS configurations (*30CU 32MB* and *27CU 32MB*), as well as the OPW strategy (*30CU 128MB*), we come across continuous memory dump crashes which prevents us from considering the results as valid. In all the failing scenarios, the cause is related to the acker executors. The recurring crashing JVMs are the ones containing only ackers, where we noticed that some ackers fill very quickly (in the first 10 seconds of run) the receiving queue and cause the worker to reach its memory heap limit.

Considering the only two valuable results (red bars in Fig. 3.11), we can observe the importance of a correct resource requirements configuration. With 128MB of required memory, we have an average of 8 executors per node. This number doubles when we decrease the required memory down to 64MB. The placement in the first case results in 4 workers per node with 2 executors each, In the second case we have 2 workers per node with 4 executors each. Compared to the former case, we have a better distribution of ackers. This increases intra-worker communication, hence reducing communication latency by 82% and also improving the throughput by more than 115%. As we can see, the impact of placement in the first test (*30CU 128MB* in Fig. 3.11b) drastically increases the completion latency to over 300ms in average.

Both OPW (in green on Fig. 3.11) and IQ (in yellow) improve on the original RAS scheduler, with both strategies significantly improving the throughput, catching up with the Even scheduler performance. Regarding latency, the OPW scheduler (green) cannot reach a faster average processing time than the default RAS result in its best configuration (30CU 64MB). However, in the best



(a) Throughput (higher is better)



(b) Latency (lower is better)

Figure 3.11: Results of the large-scale topology in the single cluster environment

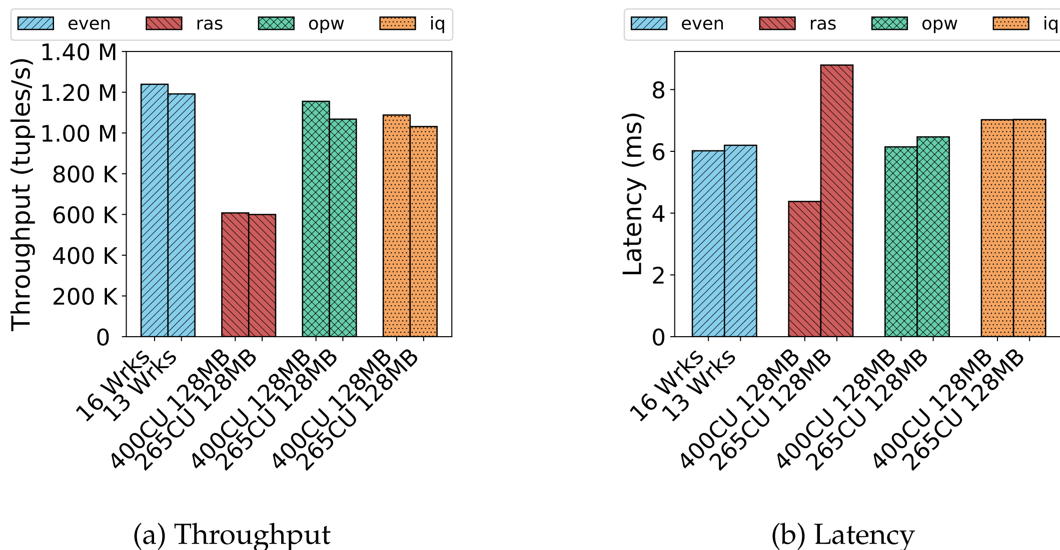
case (30CU 32MB), OPW results to be slower by just 4ms (i.e. 6% slower). When compared to the Even scheduler, swe are able to improve the process latency up to a 27% on equal throughput configurations.

Overall, while the OPW scheduler offers a good trade-off in terms of throughput and latency as compared to the RAS scheduler, the most meaningful improvements are observed with the IQ strategy. This is especially true in the last

three configurations (*30CU 64MB*, *30CU 32MB*, *27CU 32MB*), which considerably improve the latency of the default RAS scheduler. At a similar throughput level, the IQ strategy reduces the latency by more than 53% as compared to the RAS scheduler, and by 33% w.r.t. the Even scheduler. In specific scenarios where latency is more relevant than throughput, we can, for the best configuration, reduce latency by 66%, at the cost of a 7% decrease of throughput.

Small Topology Results with the small topology are presented in Fig. 3.12a and Fig. 3.12b for the throughput and latency respectively. We can observe that the Even schedulers performs very well in this scenario for both metrics. The RAS scheduler can achieve very good latency, but at the expense of a lower throughput in some cases. In addition, its performance varies in an unpredictable manner depending on the CPU and RAM requirements used for the tasks.

Differently from the previous scenario, with a small parallelism level, it results to be harder to improve the already good performances of the Even scheduler. However, we must point out that the throughput of the Even scheduler is 3 times higher than the resource-aware when using the default strategy (Fig. 3.12a). However, when a lucky placement occurs, the RAS can improve the Even lateny by a 27% (Fig. 3.12b), i.e. around 1.6ms (*400CU 128MB*). With



(a) Throughput

(b) Latency

Figure 3.12: Results of the small-scale topology in the single cluster environment

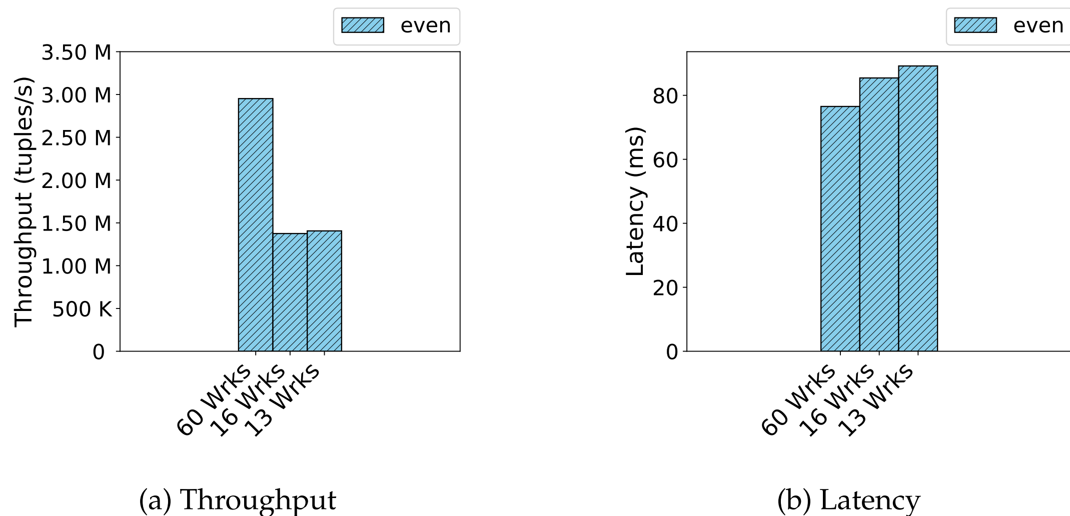
a requirement of 265 cpu units, precisely to fit 3 executors per node, the average latency escalates up to 8.8ms. Even though both tests with the default RAS strategy has the same throughput, in the second scenario we observe an higher latency caused by an higher sojourn time in the nodes containing the ackers.

In contrast, the IQ and OPW schedulers achieve stable performance irrespectively of the CPU and RAM requirements. They achieve performance on par with the Even scheduler, with slightly better results for OPW. A fair comparison between the OPW and Even schedulers can be done when both use the same number of physical nodes. As can be seen from Table 3.2, this is the case when comparing the 13 workers case of the Even schedulers and the (*400CU 128MB*) case of the OPW scheduler. The performances are almost similar in terms of throughput and latency for the two schedulers in these two experiments.

Findings: overall, on a single cluster, our strategies improve the performances of the default RAS in all scenarios. Moreover, in the worst case it is on par with the Even scheduler.

Multi Clusters

Large Topology A key advantage of the RAS scheduler, and also IQ and OPW, in a multi-cluster scenario is that they will pack, as far as possible, the topology in a single cluster. This is the case for the large topology scenario and, as such,



(a) Throughput (b) Latency

Figure 3.13: Results of large-scale topology in the multi-cluster environment

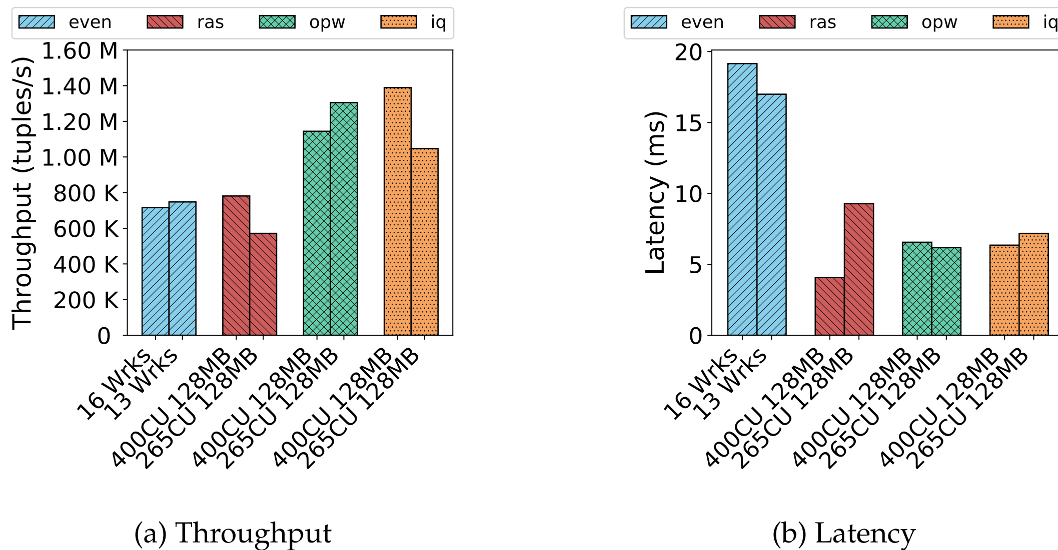
the results for the three resource aware schedulers are the same as in the single scenario (Fig. 3.11).

The Even scheduler, in contrast, is greedy and will deploy the topology over the two clusters, which causes the latency to ramp up from 6 to about 80 ms. But this can also lead to an increase in terms of throughput if it uses more physical nodes as is the case in the 60 workers scenario of Fig. 3.13a, where the throughput is almost double. This is given by the increased number of used nodes, 32 up from 16. We see a significant improvement (+ 82%) of the tuple processing (Fig. 3.13a) and lower (-42%) latency (Fig. 3.13b).

The slightly improved performances of the 13 and 16 workers tests (compared to the ones in the previous section) can be attributed to the more performing CPUs in the second cluster, where half of the topology is placed.

Small Topology In this scenario (Fig. 3.14), the Even scheduler clearly underperforms as compared to the resource aware schedulers, as it again deploys the topology over the two clusters. Also, the RAS scheduler again achieves less stable results than the IQ and OPW schedulers due to the balancing of the acking load they perform. This further enables them to improve throughput as compared to the original RAS scheduler while featuring similar latencies in this scenario.

Comparing the Even scheduler with the RAS, we see the true impact of the



(a) Throughput

(b) Latency

Figure 3.14: Results of the small-scale topology in the multi-cluster environment

resource-aware algorithm. In the best case the RAS is able to increase by the 10.8% the throughput and to reduce the latency by the 78.8%. The RAS achieves the same performances as in the single cluster. However, the Even scheduler, with so few executors feels the impact of having a slow link that separates them (blue bars in Fig. 3.14b).

As for the default strategy, the RAS with our implemented strategies, performs like in the single cluster environment. However, considering the trimmed performances of the Even scheduler we're able to improve its throughput (Fig. 3.14a) by up to 76%, with the OPW strategy, and by 86.6%, when applying the IW strategy. Even though the best latency is obtained by the default strategy, we see how our strategies maintain more stable results when changing the configurations. The RAS features (Fig. 3.14b) a latency increase (+ 127.8%) when passing from a requirement of 400 cpu units to 265. Meanwhile, our solutions increase, in average, by 61% the best latency obtain by the default RAS, but reduce the worst case by 29%.

Findings: our proposal improves on both default RAS and Even scheduler, when running on multiple clusters. The Even scheduler can sometimes achieves better throughput at the cost of using twice as many nodes.

3.6 Conclusion

With this work, we have demonstrated how implementing reliability in data stream systems can affect application performance if not done with care. This is especially true in acknowledgment-based systems, where the acking mechanism induces a significant processing and network load.

We exemplify the case with the Storm middleware, considering its two standard schedulers, namely the Even and the Resource Aware Scheduler (RAS). The RAS scheduler was devised for multi-cluster scenario where it is known to outperform the Even scheduler. We have demonstrated that its relatively worse performance in single-cluster scenario is due to its handling of the ackers components. We have improved the RAS scheduler, with new ackers placement strategies – OPW and IQ – that enable it to perform at least as well as the Even scheduler in both scenarios with small or large topologies. This means that it is possible to design a single scheduler that offers consistent performance irrespectively of the exact scenario.

Our work focused on the acknowledgment mechanism, demonstrating its possible impact on application performance. However, DSP frameworks usually provide other mechanisms (logging, monitoring, etc) which also rely on system tasks. In the same manner, they can have an impact on the processing

load. Thus, it is fair to conclude that all the non-functional tasks should be considered as part of the application during the scheduling process.

Chapter 4

Application Prototype Generation: NAMB

Contents

4.1 Related Work . . . 57
- 4.1.1 Benchmark and Evaluation . . . 57
- 4.1.2 High-Level Languages and Application Generation . . . 61

4.2 Motivation . . . 62

4.3 Fundamental Characteristics . . . 64
- 4.3.1 Data Stream Characteristics . . . 64
- 4.3.2 Workflow Characteristics . . . 65

4.4 High-Level Description . . . 66
- 4.4.1 Workflow Schema . . . 67
- 4.4.2 Pipeline Schema . . . 70

4.5 Not Only a Micro-Benchmark . . . 72
- 4.5.1 NAMB Design . . . 72
- 4.5.2 Application Builder . . . 73
- 4.5.3 Topology Generator . . . 73
- 4.5.4 Data Generation . . . 73
- 4.5.5 Multi-Platform Design . . . 74

4.6 Implementation Challenges . . . 75
- 4.6.1 Task Workload Simulation . . . 75
- 4.6.2 Topology Design Decisions . . . 77
- 4.6.3 Managing Platform Specifics . . . 81

4.6.4 Data Generation 81
4.7 Conclusion 83

In Section 2.1.3 we showed the wide range of Stream Processing Engines. All of them propose different approaches and architectures focusing on different streaming challenges [FCKK20], with the final objective of optimizing the stream processing performances by assuring reliability, high throughput and low latency. Stream processing evolved at a point which makes it capable, not only to process data streams, but also to stored data as well. The windowing semantics presented in [ABC+15] introduce a manner to perform batch processing through streaming platforms.

Several works have been done to improve these systems, from scheduling algorithms (Chapter 3) to deep architectural renewal [FCKK20]. The inclusion of new features, a radically different platform architecture and processing semantics, even the application designs itself, need to be analyzed and evaluated. This process is necessary to understand the behavior of a platform, and how it reacts to different design implementations. However, writing several applications to encompass all the possible design implementations and test diverse features, may be costly and time-consuming.

Current works commonly use test applications as benchmarks or mocks of production applications. Most of the available benchmarks tend to be bounded to the platform or scenario they are evaluated on. As a consequence, it makes those applications hardly usable in other contexts. Moreover, this ad-hoc approach does not always allow an easy tuning of the application. Indeed, even some slight changes of the application characteristics require modification of the source code. This becomes a limitation when the source is not available, or when the application internals are not correctly explained.

Thus, a reference solution is missing. We feel the necessity for a flexible and generic approach, that is context- and platform-agnostic and that would allow to easily configure the fundamental DSP applications characteristics, together with a detailed workload description.

With this work we propose a generic solution to the problem: an application prototype generator based on an high-level description model. We firstly define the high-level model, by presenting two schemas with different granularity level: the *Workflow schema* and the *Pipeline schema*. Based over fundamental data stream characteristics, the schemas support easy and quickly configurable topology description. These schemas will be used as input for an automatic generation framework.

The prototype generator we present is called NAMB, Not only A Micro-Benchmark. It is a framework that, given a generic application model, will automatically generate the defined application. It builds over the two generic high-level description models here introduced. Thanks to the combination of the high-level description schema and the application generator, NAMB allows for an easy and quick generation of a large set of micro-benchmarks as well as prototypes of realistic applications.

In this chapter we present the following contributions:

- We provide a detailed description and analysis of the fundamental characteristics of typical DSP applications.
- We present the Workflow schema, an high-level model to describe the global workflow and characteristics of a DSP application, allowing a quick and easy customization.
- We introduce the Pipeline schema, a second model to precisely describe each element of a DSP application, allowing an higher degree of flexibility.
- We propose NAMB, a platform for fast and flexible generation of prototype applications based on their high-level description.
- We demonstrate how NAMB can be used to evaluate the impact of design choices and create complex prototypes to analyze DSP systems, only by using high-level models instead of editing the application code.
- We make available a public release of NAMB at `https://github.com/ale93p/namb`, ready for Storm, Flink and Heron.

The rest of the chapter is organized as follows: the State of the Art of benchmark applications, is presented Section 4.1.1, and related works on high-level languages and application generation in Section 4.1.2. The work motivations and challenges are explained in Section 4.2. Section 4.3 presents the fundamental characteristics of data stream applications. The derived high-level language is introduced in Section 4.4. The prototype generator NAMB is presented in Section 4.5. Then, Section 4.6 focuses on the implementation challenges and details. Finally, we conclude in Section 4.7.

4.1 Related Work

4.1.1 Benchmark and Evaluation

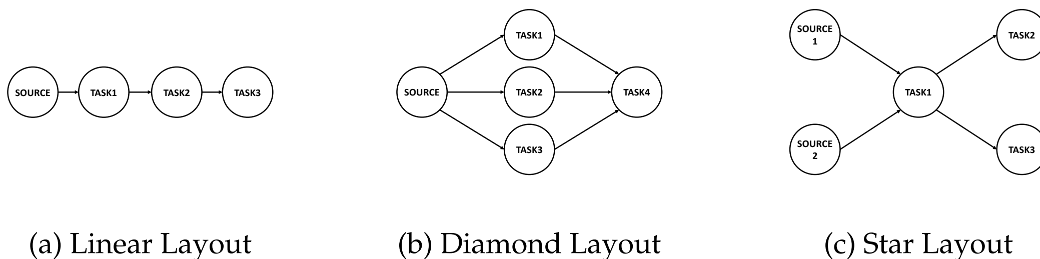
A taxonomy of the state-of-the-art DSP benchmark applications can divide them into three categories: micro-benchmarks, benchmarking suites and mock appli-

cations. While we seek to give a precise definition for each group, they are not mutually exclusive. An application can possibly be part of more than one group.

Micro-benchmarks

In this category are grouped simple applications that don't try to replicate realistic scenarios. The workload is composed of simple operations working on synthetic data.

Peng et al. [PHH+15] define three different base layouts for micro-benchmark topologies (Fig. 4.1): linear, diamond and star. The first one consists of a pipeline of tasks without any branch split or join; the diamond shape usually starts and ends with a single task and has multiple tasks in parallel in-between; the star layout has multiple source and sink tasks, linked by a single task. Those layouts are common and widely used in other works (as we will see in the following).



(a) Linear Layout (b) Diamond Layout (c) Star Layout

Figure 4.1: Micro-Benchmarks Topology Layouts

In [XCTS14] three different micro-benchmark applications are proposed. The first one is composed only of a source and a sink. It is a canonical topology to evaluate the maximum throughput that can be achieved. The second one is another popular micro-benchmark, the streaming version of WordCount. It is usually available as example code in the main DSP systems such as Storm or Flink. It is implemented as a simple linear application that generates different sentences and counts the occurrences of single words in them. The last one is a sample log processing topology. While the last two topologies may be considered as real use cases, they are normally implemented as examples and not derived from production code.

The WordCount topology is also used by [CM18], to perform a performance comparison between several DSP platforms, and by [FAG+17] to evaluate the performance and capabilities of their novel self-adapting data stream processor.

Marangozova et al. [MMDPER19] propose an elastic data stream processor to dynamically adapt the parallelism of the topology. To evaluate their solution, they use a DDoS (Distributed Denial of Service) detection application that follows the diamond layout.

Eskandari et al. [EMHE18] evaluate their graph-based placement scheduler using micro-benchmark applications. They test different routing options in Storm and divide the micro-benchmarks between I/O intensive and CPU intensive. In the first case, they test the components to their maximum capacity without any additional processing, which are later added in the second category to stress out the CPU.

Similarly, [LB17] uses CPU and I/O bound micro-benchmarks to prove the efficiency of their task scheduler. Aljoby et al. [AFM17] use a simple diamond topology to prove the impact of placement and bandwidth over data stream applications. Kamburugamuve et al. [KRSF17], test the latency provided by InfiniBand [INB] and Omni-Path [BDH+15], using a simple diamond layout micro-benchmark.

Overall, micro-benchmarks often lack implementation details, missing the internal description. Thus, they don't give a clear idea of what is the actual workload of the application.

Benchmark Suites

A suite consists of a set of micro-benchmarks, or more realistic applications, developed with the specific objective of benchmarking a DSP system. Some of them may include a monitoring system to retrieve metrics and output the benchmark results.

In StreamBench [LWXH14], the authors define a set of workloads, characterizing them in terms of data type and computational complexity. They then propose 7 different benchmark applications representing those scenarios (Identity, Sample, Projection, Grep, WordCount, DistinctCount, Statistics), with varying processing complexity. The applications are tested with two different real-world datasets (textual and numeric).

BigDataBench [WZL+14] is a benchmark suite devised for both batch and streaming BigData platforms. They focus on Internet services. BigDataBench is composed of a large set of applications, each related to a specific Internet service. More than 30 micro-benchmark workloads are available and grouped under 5 macro-categories of services: Search Engine, Social Network, E-Commerce, Bioinformatics and Multimedia Processing. Among them Grep, Rolling Top Words (specific to Social Networks), Kmeans and Collaborative Filtering (spe-

cific to E-Commerce) are available for streaming data platforms. JStorm [JST] and Spark Streaming [SST] are currently supported.

The authors in [SCS17] propose RIoTBench, a suite designed specifically for IoT scenarios. They analyze the characteristics and the behavior of common applications in this context, describing 8 different common task patterns used in streaming applications for IoT. The suite regroups a large set of IoT micro-benchmarks to cover all these patterns. It also includes a set of representative IoT applications. They are inspired by realistic scenarios and implement four topologies representing the most common IoT workloads.

In Senska [HRM+17] instead of considering only a set of contexts to benchmark as what we have seen above, the authors propose a generalized approach to DSP evaluation being suitable for every environment. They define 9 basic tasks used in industrial applications (e.g. Transformation, Merging, Filtering) and use them to compose 5 different benchmark use-cases (e.g. Check Sensor Status, Check Machine Power).

Benchmark suites are designed for context-specific scenarios. They consist of applications built ad-hoc for a specific environment. They are limited by the static compositions of tasks, other than by the platform-specific implementation. This lack of flexibility does not allow customized workflow compositions or cross-platform benchmarks.

Mock Applications

These applications are valid representatives of what is usually deployed in a production environment. They implement common queries used in production applications, making them close replicas of what can be deployed in a real cluster.

The first benchmark application appositely developed to benchmark streaming data is Linear Road [ACG+04]. The authors propose an application that simulates a real scenario: an urban expressway system. The objective of the application is to monitor real-time traffic to apply an algorithm of variable tolling, to adapt the price to the road congestion.

The most popular benchmark application of this category is Yahoo! Streaming Benchmark [CDE+16a]. It has been widely used by companies to evaluate their solutions [Gri16, Yav17, KW17]. The application (Fig. 4.2) analyzes advertisement interactions on the Web. The data reaches the application as an events stream from Kafka. They are deserialized and parsed into different fields, keeping only the ad *view* events. They are then forwarded to the next task which looks into a Redis database to retrieve the associated campaign id. Finally,

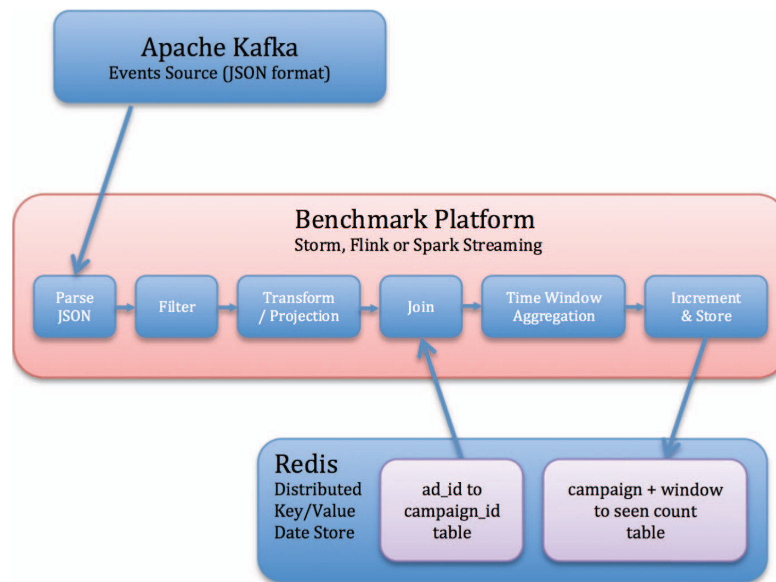


Figure 4.2: Yahoo Streaming Benchmark Design [CDE+16a]

events are grouped and counted by campaign id.

Yahoo! also presented two other typical industrial topologies in [PHH+15]: PageLoad and Processing. They are used to manage real-time advertisement events. Both are presented as a set of standard queries without any details regarding the actual workload.

Chatterjee et al. [CM18], in addition to the WordCount application described earlier, use two applications with real datasets: an air quality monitoring application and a flight delay analysis.

Mock applications, similarly to what we have seen in the previous section, offer static streaming workflows based for popular DSP systems. Although these applications are representative of real industrial workflows, they may not match the user-specific environmental needs.

4.1.2 High-Level Languages and Application Generation

Several higher-level languages for DSP [HBB+18] have been proposed. The main objective is to introduce mechanisms or concepts previously not supported, or optimize the application generation.

Apache Beam [ABC+15, BEA] tries to enclose all the key mechanisms that a DSP system should support, through a common set of Java APIs for the differ-

ent streaming platforms. A particular attention is paid to the windowing mechanism, and a new model for windowing is proposed. It can support out-of-order data arrivals along with session windowing. Then, *Beam Runners* translate the Beam APIs into platform-specific code, allowing cross-platform development and ensuring a common set of functions between all of them.

SECRET [BDD+10] defines a general semantic to describe the diverse Stream Processing Engines (SPEs) mechanisms. The framework aims to compare and ease the understanding of SPEs internal behaviors, which are normally differently implemented. In their paper, the authors mainly focus on the windowing mechanism implementing a framework aimed to continuously monitor the window status of the application.

The authors of SpinStreams [MDT18] present a framework to optimize the tasks implementation in DSP applications. Given a topology description in xml and the java functions to describe the tasks workload, SpinStreams applies operators fission (task replication) and fusion (merging two or more tasks into a single one) to optimize the query executions and improve application performances.

Another example of query language for DSP is Piglet [GHS16], an extension to Pig Latin [ORS+08] directed to Stream Processing functions. Given an high-level description of the stream processing queries, Piglet generates application code for the different SPEs. They define a SQL-based API to write the application that generalizes the supported platforms features. Similarly to SpinStreams, they rewrite the application code to optimize the query execution.

The above works propose high-level description models to define DSP topologies. They expose programming languages APIs to write the code, or require actual code for user-defined functions. The main objective is to generate executable application code for the different SPEs. At the same time, they focus on optimizing the query planning or introducing and generalize missing DSP features to these engines.

Our work goes on a different direction. We aim to generate prototype applications, with a simulated workload, i.e., we don't require the user to specify the internal code of each task. This approach would allow a quick and easy definition of the topology graph, without the need of writing the application code, through an high-level and simple description of the workflow .

4.2 Motivation

We just showed that the literature already provides us with a large variety of streaming applications and application generators. These works fulfil their ob-

jectives as they evaluate the specific case their designed for. However, none of them can be seen as a generic reference solution. We think that this past work presents various limitations in this regard:

- They are too **context-specific**. These benchmarking tools are commonly designed for specific scenarios such as IoT or Internet services, limiting their applicability to other fields of study. Moreover, they are implemented over a specific platform. It is thus necessary to rewrite the entire application to adapt it to other middlewares.
- Applications and generators require the **usage of actual code**. Either with specific APIs (e.g. Java or SQL), or with global definitions (e.g. XML) which anyway requires external code to define user-specific tasks.
- They are often **hard to replicate** due to the lack of a specific workload description. This is especially the case of micro-benchmarks, as they usually define a general objective for the evaluation – e.g. I/O intensive, maximal throughput – but fail to provide a detailed definition of the tasks internals.
- At the same time, they are **not flexible** enough. An imprecise description of the pipeline and hard-coded configurations do not always allow a quick and easy tuning of an application, preventing an easy study of different implementation choices.
- The software presented in the literature is often **not publicly available** or hard to retrieve. Private code to be required at the authors or expired websites, make complicated the usage and testing of these tools.

These drawbacks denote the lack of a generic solution, that could be used in every scenario and that could be easily and quickly adapted to the underlying environment. For such reasons, we think it is necessary to have a prototype application generator that could address those limitations, i.e. that is:

1. **generic**: able to run on or be adapted to any kind and future data stream frameworks and features.
2. **flexible**: with a well defined description model of the workflow, that can be quickly and easily customized.
3. **available**: as open-source, so as to be ready-to-use and continuously improved by the community.

To achieve these objectives, we need an initial description and model of a typical data stream application. Thus, we first define several fundamental characteristics common to DSP applications (Section 4.3), which have a significant impact on the characterization of an application workload. We then abstract these characteristics in a set of parameters configurable by the user through a high-level set of configurations (Section 4.4). Finally, we implement a framework that, given this model, will automatically generate the application to be deployed (Sections 4.5 and 4.6).

4.3 Fundamental Characteristics

Streaming applications are composed by a pipeline of tasks that continuously process data flowing into the application. We divide DSP applications fundamental characteristics into two categories: **data stream** and **workflow**. The former defines the input stream of the application. The latter describes how the data is transferred between tasks and how it is processed. The two categories are interdependent as the workflow is impacted by the characteristics of the data stream.

4.3.1 Data Stream Characteristics

Data Characteristics

In stream processing, data can assume various forms (from text to binary), depending on the application environment as well as the data sources. Hence, the dataset is defined by several properties. The cardinality of the dataset and the data size depends on the application and also the input format, e.g. JSON, XML, plain text. Depending on the application field, the number of different items as well as their popularity may also vary significantly.

Input Stream Characteristics

Data may arrive at different rates. Modern applications such as social networks not always feature a Constant Bit Rate (CBR) stream. Thus, we identified four key distributions that represent the majority of stream processes:

- CBR, e.g. a continuous feedback of a monitoring system;
- Bursts, e.g., extraordinary events in sensor-networks [ZKZ+15];
- Saw-tooth distributions, e.g. similar as before, extraordinary events that generate high number of tuples that slowly decrease.

- Sinusoidal human-related behavior, e.g. more activities during the day and less during the night [CERLDP16];

4.3.2 Workflow Characteristics

Connection

A DSP application is commonly represented as a Directed Acyclic Graph (DAG) of tasks, with the sources as roots and the sinks as leaves. The tasks can be arbitrarily connected to form different logical shapes. As an example, the three base layouts described in Fig. 4.1 are commonly basic building bricks for more complex topologies.

Scalability

Streaming systems are designed to manage high loads of data. Hence, applications need to be scalable. Most components of the topology can be parallelized to spread the incoming load over multiple instances. The global parallelism level of the application represents the total number of instances of all tasks of an application. However, this global value is not necessarily uniformly distributed, with some tasks requiring more computational capabilities.

Traffic Balancing

When data is transmitted to a parallelized task, the application has to decide how to distribute the tuples over the different instances. The various streaming platforms usually implement some standard grouping methods: (i) balanced routing, based on a simple round-robin algorithm that assigns each tuple to a different instance, enabling load balancing between the tasks; (ii) key-based routing, that sends each tuple to a specific instance using a hash function, allowing for stateful routing; or (iii) broadcast routing, where a tuple is replicated and sent to all the following level instances.

Message Reliability

Most DSP platforms offer a reliability mechanism to ensure message processing. As an example, Storm uses an acking framework with the aid of non-functional tasks while Flink relies on check-pointing to ensure exactly-once delivery [FCP]. These solutions can impact the application throughput, processing latency (as we saw in the previous chapter) or its message failure probability.

Windowing

As a consequence of the endlessness of stream applications, most of them try to give partial results over time periods, rather than waiting the end. This method of aggregation is done through windows [ABC+15, MTL+18, TGC+19]. A window is defined as a group of tuples commonly enclosed in a portion of time or in a specific number of tuples. Tumbling windows are non-overlapping, i.e the beginning of one window follows immediately the end of the previous one. Sliding windows, on the other hand, allows for overlapping. Hence, a tuple can be part of multiple windows. The implementation of a windowing system is complex. In particular, the need for storing data (even duplicated in case of sliding window) increases the resource requirements.

Data variability

As the data flow in the tasks pipeline, some tasks may alter the data (e.g. filtering, projections...). Hence the characteristics of the input dataset may change on the fly.

Workload

Each task composing the DAG performs various operation on the data and some might be more computationally intensive than others. Thus, the processing load is not always balanced over all tasks and bottlenecks might exist.

We listed and described the main properties that significantly impact the performance and the behavior of streaming applications. From those fundamental characteristics we derive the high-level model to describe streaming application prototypes, presented in the next section.

4.4 High-Level Description

The model defines a series of parameters, that covers at best the just described characteristics. We expose two different description models with different granularity of application description.

We define a generic schema for a coarse-grained description of the global behavior of the application, called the **Workflow schema**. To that first schema, we juxtapose a second one closer to a real application definition, named the **Pipeline schema**. This second schema allows the user to specify the exact DAG composition they want to evaluate, with parameters for each task. These two

approaches make it possible to easily and quickly define prototype applications, but also describe large and complex mock applications.

Both models follow the YAML standard [YAM] for their configuration files, which will serve as input to the generator (Section 4.5). It thus consists of a series of key-value pairs grouped into different blocks.

4.4.1 Workflow Schema

The Workflow schema allows the user to describe the global parameters of the application, without going into the individual task detail. From this high-level point-of-view, the schema gives the user a quick and easy way to generate simple micro-benchmarks. The simplicity of the schema allows to swiftly tune test-by-test some application features (e.g. data size, parallelism, computing load), and easily experiment with different design combinations. Following the categorization made in the previous section, we divided the model into two homonymous main categories.

Data Stream Section

The **datastream** section contains the input data and arrival flow characteristics. It supports the definition of a synthetic data stream or the connection to a Kafka cluster, to allow the connection to an external data source, thus the input of more complex and realistic data streams. In the following we focus on the synthetic generator, as it is the one that defines the stream as introduced in the previous section, whilst the kafka connection is addressed from an implementation point-of-view in Section 4.5.4.

The synthetic dataset is described by how many unique values it is composed, their appearance distribution, describing the probability of a value to be generated (e.g. *uniform* means equal probability for each value), and the size in bytes of a single tuple.

The input flow is defined through its arrival process, and its rate in terms of tuples per second. The arrival process distributions can be defined as CBR, burst, sinusoidal, sawtooth and reverse sawtooth. In addition to the rate value, common to every distribution, the description includes further parameters for non-CBR distributions, to customize their *variance* over time. For the bursts, it is possible to define the interval time between bursts and individual burst duration. For the sinusoidal and sawtooth, a *phase* parameter allows to define the cycle duration of the wave, e.g. the time for the sinusoid to go back to its starting point.

For example, Fig. 4.3 indicates that the benchmark will generate *synthetic* data. Each tuple will have a size of *8 bytes* and their values will be randomly

```
datastream:
  synthetic:
    data:
      size: 8
      values: 100
      distribution: uniform
    flow:
      distribution: uniform
    rate: 0
```

Figure 4.3: Datastream section of Workflow schema (yamb.yml)

chosen among *100* different ones with a *uniform* distribution. The arrival rate is not limited (indicated with value *0*), making the application produce as much data as it can, and *uniform*, i.e. not following any arrival process (with a limited rate, this would translate into a CBR arrival process). The unbounded rate will exploit the maximum capacities of the platform and testing environment, without setting any time interval between tuples.

```
flow:
  distribution: sinusoidal
  rate: 1000
  phase: 300
```

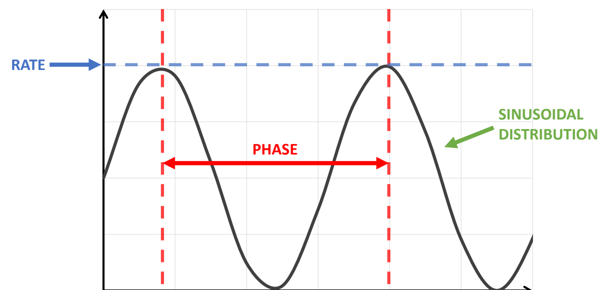


Figure 4.4: Flow configured as: sinusoidal stream, with fixed rate and phase

In a second example (Fig. 4.4), the arrival flow is defined as *sinusoidal*. For this distribution, an arrival rate of *1000 tuples/s* will correspond to the highest value reached by the wave, and the new *phase* value will define the time duration, in seconds, of a sinusoidal wave cycle.

Workflow Section

The high level description of the DAG is given in the **workflow** section. It allows to configure the logic composition of the graph and tasks connection as well as the internal task workload. The DAG depth defines how many levels of subsequent tasks are in the topology, including the sources. The global parallelism level is the total sum of all the tasks instances, and how it is distributed over the single tasks. One can further specify how components are connected to each other, as well as the grouping method (e.g. shuffle/round-robin, hash-based/by key) and the shape of the topology, defined through the basic micro-benchmark layouts (represented in Fig. 4.1).

The tasks workload is defined by their processing load. The value specified in the schema will be used to simulate the internal workload of the tasks. We rely on a busy wait function (Section 4.6.1) that will keep the CPU busy. In this manner we are able to produce a generic load not bounded to any specific data stream query. This allows for context-agnosticism. A DSP application may have a balanced load, where all the tasks have the same processing load, as well as unbalanced loads, especially applications for which the heaviest tasks are at the beginning of the DAG, or the reverse (we will go into more details in Section 5.2.3). The balancing method can be defined in the configuration to specify how to distribute the load value over the tasks.

The workflow description includes also a reliability flag, to enable reliability mechanisms where the platform supports it, e.g. Storm acking framework. Windowing and data filtering can be globally defined. The former specifies the type of window (i.e. thumbling or sliding), its duration in seconds and the interval between the beginning of two windows, in case of sliding windows. The filtering parameter specifies the percentage of data that will be discarded, e.g. a value of 0.33 means that only the 33% of incoming tuples will reach the sink.

In Fig. 4.5 we consider a *diamond* topology with round-robin (*balanced*) connections between tasks. The depth of *5* indicates there will be two more tasks after the diamond (Fig. 4.5 right side). With a *balanced* parallelism level of *24*, each of the 6 tasks, including the source, will have 4 instances. The processing time at each task will be simulated following the *decreasing* order (other modes such as constant or increasing are possible). This mode starts by assigning the defined *processing load* value to the first task and then decreases it by 20% for each following ones. Hence, the most computational intensive tasks will be at the beginning of the topology. Finally, the generated code will use the reliability mechanisms of the platform if available.

```yaml
workflow:

  depth: 5

  scalability:
    parallelism: 24
    balancing: balanced

  connection:
    shape: diamond
    routing: balanced

  workload:
    processing: 3.0
    balancing: decreasing

  reliability: true
```

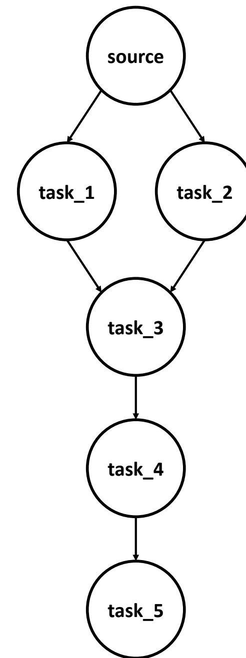


Figure 4.5: Workflow section of Workflow schema (yamb.yml)

4.4.2 Pipeline Schema

Differently from the Workflow schema, the Pipeline schema doesn't describe the topology from a generic point of view. It focuses on the description of each task and connection to accurately define more specific characteristics. Similarly to what a user would write to build a real application using platform-specific APIs. The main advantage of this schema is that it allows to tune the parameters at task level, giving more freedom to shape the application. This allows to test specific scenarios, such as finding the application bottleneck.

The configuration file consists of a main section to describe the pipeline of tasks. The key-values in this schema are overall the same as in the Workflow schema, but task-based. A task can either be a source or a processing unit, depending on the properties used to configure it. The *data stream* is described directly in the source task, defining how that specific task will generate data, or if it is connected to an external source (Section 4.5.4).

An advantage is that, differently from the Workflow schema, the user can also specify different sources with a specific behavior each. Meanwhile, the *workflow* is defined at a per-task level. Each task is defined by its own properties, as processing workload and parallelism. Each task will define the parent tasks and how they are connected to the previous level in the DAG. This allows to

create more complex topologies that do not strictly match one of the layouts provided in the workflow schema.

Fig. 4.6 shows an example of a configuration file with the associated topology for the Pipeline schema. A source (*word_generator*) will send data at a rate of 1000 tuples per second to 2 *counter* tasks using a hash-based routing. To simulate a real processing load (Section 4.6.1), we use here a busy wait loop of 4500 cycles (4.5 in the configuration file as the unit is a thousand of cycles). Finally, the data is sent to a *sink*, which will perform some light processing. The meaning of the *processing* parameter is explained and motivated in Section 4.6.1.

```
pipeline:
  tasks:
  - name: word_generator
    parallelism: 1
    data:
      size: 8
      values: 100
      distribution: uniform
    flow:
      distribution: uniform
      rate: 1000

  - name: counter
    parallelism: 2
    routing: hash
    processing: 4.5
    parents:
    - word_generator

  - name: sink
    parallelism: 1
    routing: balanced
    processing: 0.5
    parents:
    - counter
```

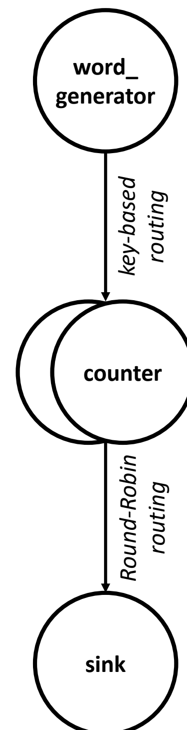


Figure 4.6: Per-task configuration with the Pipeline schema (yamb.yml)

4.5 Not Only a Micro-Benchmark

In this section we present the details of **NAMB** – **N**ot only **A** **M**icro-**B**enchmark – an application prototype generator for stream processing frameworks. Given a high-level definition of the workflow for the *Workflow schema* or the *Pipeline schema*, NAMB automatically generates the corresponding application for multiple platforms.

4.5.1 NAMB Design

NAMB prototyping process is based on two main phases. It processes the *high-level configuration* files provided by the user and passes the result to an *application generator*. Once the user has written their high-level YAML model, NAMB is executed through a main command line script, through which the user is able to specify for which platform the application will be generated. The script will create and run NAMB giving in input the configuration file, it will then directly deploy the generated application on the platform of choice.

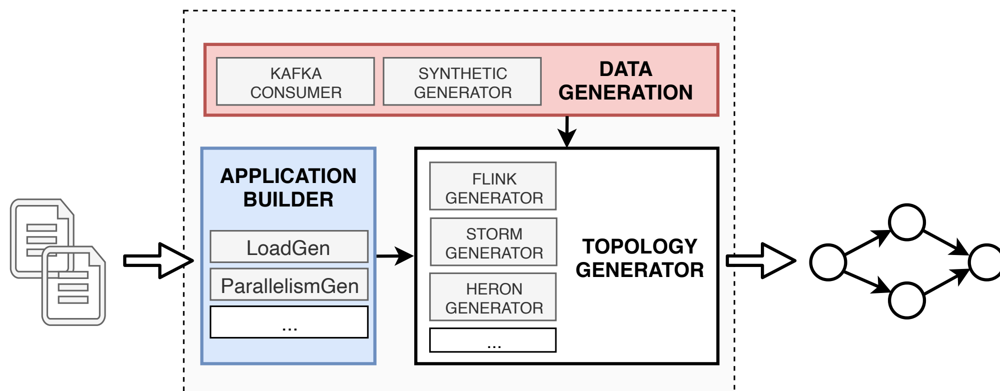


Figure 4.7: NAMB Architecture

NAMB core is composed by three main components (Fig. 4.7). The *Application Builder* that takes the input configuration and reads the defined parameters; the *Topology Generator* that, based on the specified platform, will translate those configurations into platform-specific code; and the *Data Generation* that will inject data into the pipeline, either synthetic data or data from an external source.

NAMB is designed to easily support add new platforms (Section 4.5.5). In its current version, NAMB supports Storm, Flink and Heron, while support of Spark Streaming is under development.

4.5.2 Application Builder

The Application Builder is the first component in NAMB's workflow. It is the interpreter of the input configuration. It parses the configuration file of the user and, based on the chosen schema, converts it in easy-to-use objects for the topology generator and the application generation. The Application Builder generates different objects based on the configuration schemas it is interpreting.

The Workflow schema allows the user to provide only a high-level description of the topology (no per task details). Hence, NAMB (other than storing plain values as the *topology depth* or the *topology shape*) has to adapt some of the global values, such as the *workload* or *parallelism*, to the tasks in the pipeline.

Meanwhile, for the Pipeline schema, it doesn't necessitate the adaptation done above. In this case, the Application Builder converts the tasks blocks described in the configuration file into a list of tasks objects, of which properties will then be used to define the actual tasks in the topology.

4.5.3 Topology Generator

The Topology Generator is the principal component of NAMB. It generates the application by translating the configurations given in input into platform-specific code. When running NAMB the user can specify between the supported Stream Processing Engines (currently Flink, Storm and Heron). Once the application is correctly generated it is directly deployed.

As we have seen, the Workflow schema gives a generic description of the topology, leaving incompletely defined some parameters such as the topology shape in relation with its length, or when to apply specific operations (e.g. data filtering). Hence, the Topology Generator needs to adopt some specific rules to standardize every topology generation (Section 4.6.2).

One of the main objective and characteristic of NAMB is to be cross-platform. In this way, given the same configuration file, NAMB can generate the topology for different SPEs. However, every SPE has a different architectural characteristics, implementing in their own way mechanisms such as schedulers or reliability. For that reason, the topology generation has to take in account these architectural differences (Section 4.6.3).

4.5.4 Data Generation

The Data Generation component is in charge of injecting data in the application. It can let the sources use an internal synthetic data generator or connect them to an external Kafka topic.

The *internal method* is used to generate synthetic data. The Data Generator is quickly configurable. The user can decide the characteristics of the dataset, as explained in Section 4.4.1, and the Data Generator will generate tuples for the topology.

The *external connection* allows to configure the data stream section in the Workflow schema (as shown in Fig. 4.8), or the source tasks in the Pipeline schema, to plug the source tasks to a Kafka cluster. In this manner, the data generation would not be bounded to the set of options offered by the Synthetic Generator. Moreover, in case of a system benchmarking, it will be possible to consider metrics, such as the event-time latency [KRK+18], that are not available using the Synthetic Generator.

```
datastream:
  external:
    kafka:
    server: localhost:9092
    group: test
    topic: topic
  zookeeper:
    server: localhost:2181
```

Figure 4.8: Data Stream section of Workflow schema for the kafka source

Fig. 4.8 shows the configuration section of the datastream, that can be used in place of the synthetic stream description seen in Fig. 4.3. In this case it doesn't *describe* the stream but it is used to define the connection to the external cluster. It requires to specify the connection properties of the Kafka cluster: the main server (i.e. the couple IPAddr:port), the topic and the group. For some platforms, e.g. Storm, it is necessary to specify as well the Zookeeper server.

4.5.5 Multi-Platform Design

System-Specific Configuration

In addition to the application configuration file, NAMB includes a configuration file for each supported platform. It is used to define system-specific properties that are external to the application, e.g. number of workers in Storm. It also includes a debug parameter that specifies an output log rate; this will print the sampled tuple information (e.g. timestamp, ID or value), that could be eventually used to extract statistics.

This file is then combined with the user-provided configuration to generate the application and deploy it on the specific platform.

Platforms Support

NAMB design is modular. This allows to add support for new platforms in an easy way. The main driver to implement is the *Topology Generator* for the platform. As all the configuration parsing and data generation is performed by parallel components, if anyone wants to add a new SPE to NAMB, they need to translate the schema model of the logical DAG using the platform-specific API. Even though the generator is the thicker component to implement, it will be also necessary to add support for minor companion components such as the system-specific configuration and the deploying option in the running script.

4.6 Implementation Challenges

4.6.1 Task Workload Simulation

To maintain the general nature of NAMB, the user does not have to specify the exact code of a task. This avoids specific query operations on data, which commonly results in context-specific processing, not general enough for our objectives. Instead, NAMB simulates the load through a busy wait loop function. In this manner it is able to simulate processing workload, easily configurable in the schema. A parameter is used to set the number of loop cycles, allowing NAMB to replicate the processing load of common tasks used in stream processing. The value is specified in *thousands* of cycles, e.g. a value of 1.5 correspond to a busy wait of 1500 cycles, a value of 0.2 to 200 cycles, and so on.

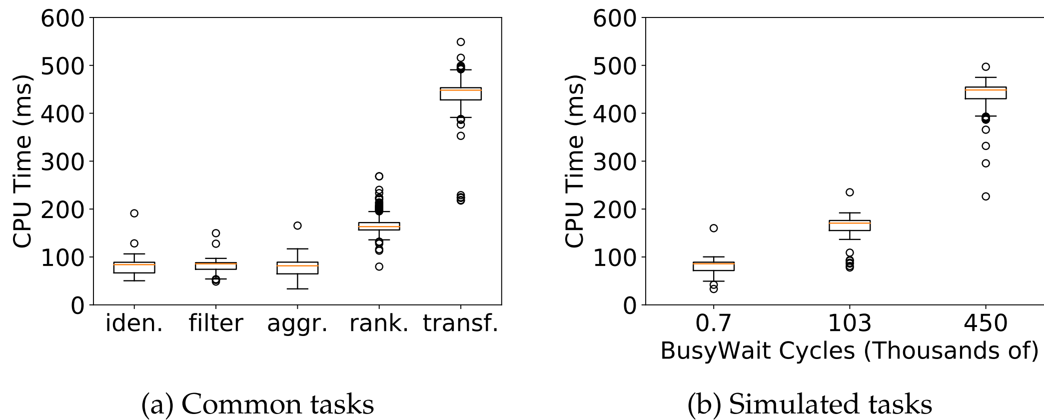
Proof of Concepts

To demonstrate the equivalence that can be obtained between busy wait loops and a real load, we have performed experiments on Apache Storm (version 1.2.1) on a 4-core node. Based on a set of representative works in the DSP domain [PHH+15, CM18, SCS17, HRM+17, ČTLČ16] we derived 5 key DSP tasks:

1. *Identity*: a task that just forwards the tuple as it is, without any processing;
2. *Transformation*: a task that transforms the input data, e.g. a parsing function that divides a JSON or XML text into an array of fields;
3. *Filter*: a task that filters data based on its value or a specific field, e.g. if/else rules;

4. *Aggregation*: a task that accumulates the input data over time, e.g. arithmetic operations;

5. *Sorting*: a task that sorts in a specific order the input data over time, e.g. ranking of word occurrences.



(a) Common tasks (b) Simulated tasks

Figure 4.9: Per tuple CPU measurements for real/simulated tasks

Each of these tasks was executed and its CPU time measured using Java's ThreadMXBean [TMX] API. Then busy wait (i.e. simulated load) tasks were created and evaluated in the same conditions, to find equivalent simulated tasks. With a carefully chosen number of cycles, we were able to reproduce a load similar to the original task.

In Fig. 4.9a we can see the results for the real tasks. *Identity*, *filter* and *aggregation* are under an average CPU time of 100 ms, *aggregation* is slightly more variable than the other two, however we can consider the processing load of the three tasks equivalent. The *ranking* task double the load of the previous three, being quite stable around the a CPU time of slightly less than 200 ms. *Transformation* is the heaviest tasks and it spikes up to almost 500 ms, around 2.5 times the *ranking* task, but with a larger variability.

Once we have the load characteristics, we try to find equivalent simulated tasks. As they have close processing load results, we can simulate the *identity*, *filter* and *aggregation* tasks using the same busy wait cycles. In Fig. 4.9b, we can see how a busy wait of 700 cycles can in average replicate the same load of the three tasks. Even though the results shows higher variability than the real task, we can in average simulate the *ranking* task with a synthetic task of 103000 busy wait cycles. More precise results are obtain for the *transformation* task, which can be correctly be simulated with 450000 busy wait cycles.

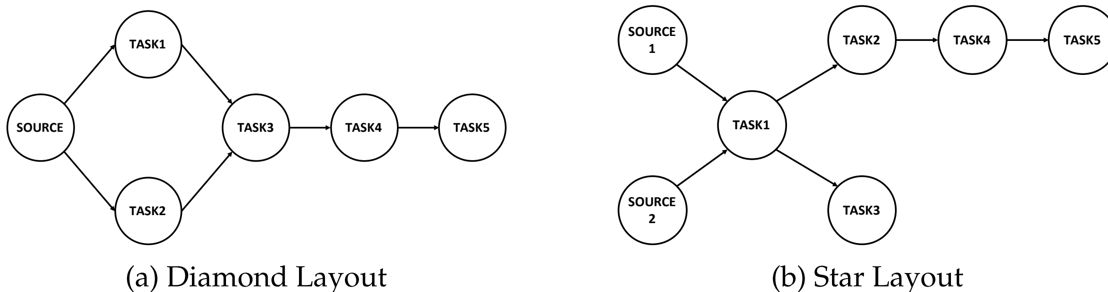
Finally, we consider NAMB synthetic tasks able to equivalently simulate real tasks by using busy wait functions.

4.6.2 Topology Design Decisions

As previously mentioned, when translating the Workflow schema to the actual platform-specific topology, we need to define specific implementation rules. Some may just require a value adaptation, others may actually shape the topology and its design. The Pipeline schema is exempt of these translations as it doesn't define global and generic values, but specific ones by task.

Topology Shape

The user configures the topology shape and the depth of the DAG. Given this combination, the Topology Generator has to define the logical composition of the DAG. If the shape is *linear*, the translation is straightforward, we will have 1 source and $n - 1$ tasks connected sequentially (for a DAG depth of n). If the shape is *diamond*, instead of replicating the diamond through all the topology, we apply it only at the beginning. This results in having 1 source connected to 2 tasks at the same level, that will then join to a single task. The topology then continues as a linear chain. A similar rule has been applied to the *star* topology. In this case, we will have 2 sources that will join to a single task. The latter then splits to 2 tasks. From this point, we decided to continue the topology with only one branch. This means that one of the two branches is a sink and the other continues as a linear chain.



(a) Diamond Layout (b) Star Layout

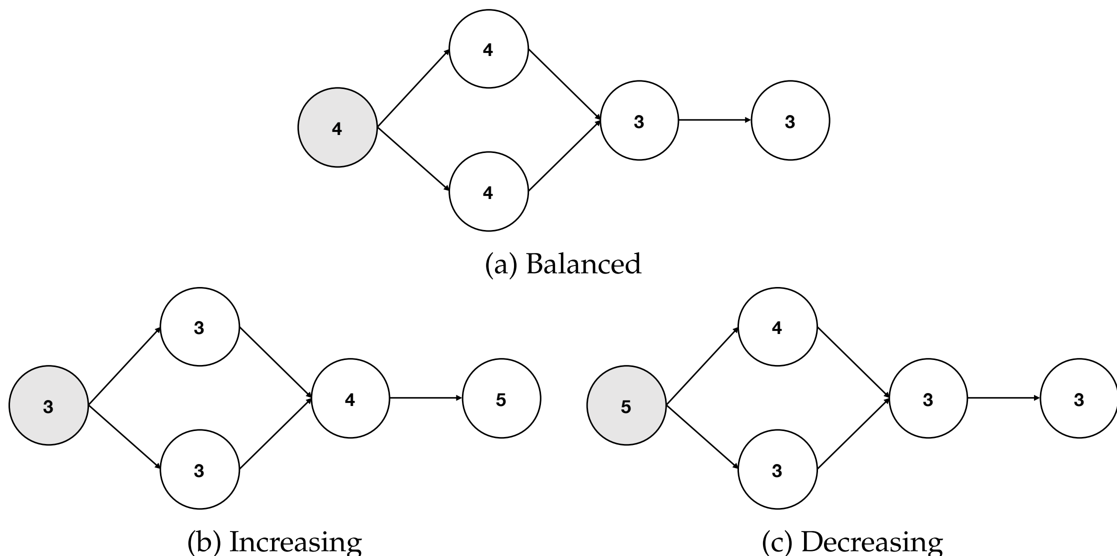
Figure 4.10: Streaming topology layouts as implemented by NAMB

For instance, considering a topology of depth 5, with a *linear* layout there will be 1 source and 4 working tasks, pipelined one after the other. With a *diamond* layout (Fig. 4.10a) the topology will still have 1 source (at the first level of depth). However, this time the tasks will be 5. The first two will be placed right after the source (at the second level of depth), splitting the stream, which then

merge in the third task (third level of depth); after the third task the topology continues as linear until the last one (at the fifth level of depth). Also the *star* layout generates 5 tasks (Fig. 4.10b). In this case the application has two sources, which merges in the first task. This first task immediately splits the stream in two branches. As explained above, the remaining tasks will be placed after the *upper branch* (from a logical point of view), in a linear manner.

Parallelism

The schema specifies a global parallelism value. The value represents the total amount of task instances of the application, i.e. the sum of each task parallelism level. For such reason, the specified value needs to be distributed over the tasks in the application. How it is distributed is based on the balancing technique specified in the configuration. If the user specifies *balanced* parallelism, NAMB will split equally the instances over the tasks (including the sources) while *decreasing* will create a decreasing series of instance numbers that will be assigned sequentially to the tasks, vice-versa the *increasing* configuration. The *pyramid* will assign increasing values until the middle task of the topology and then decreasing until the sink.



(a) Balanced

(b) Increasing

(c) Decreasing

Figure 4.11: Parallelism balancing examples; depth 4 diamond topology with a global parallelism of 18; balanced, increasing and decreasing configurations

An example of the parallelism distribution with different configurations is presented in Fig. 4.11. The example shows a diamond topology of depth 4 (i.e. 1 source, in gray, and 4 tasks, in white), with a global parallelism of 18 (i.e. a

total of 18 task instances). The *balanced* configuration (Fig. 4.11a) tries to equally assign a parallelism value to each task (including source). In this case it can't assign the same parallelism to each task, as the value is not equally divisible between all tasks. Hence, it will give more instances to the first tasks. The *increasing* (resp. *decreasing*) configuration (Fig. 4.11b, resp. Fig. 4.11c) will assign different parallelism values to each task, distributing them from a lower parallelism level (resp. higher) climbing up (resp. lowering down) to an higher (resp. lower) value.

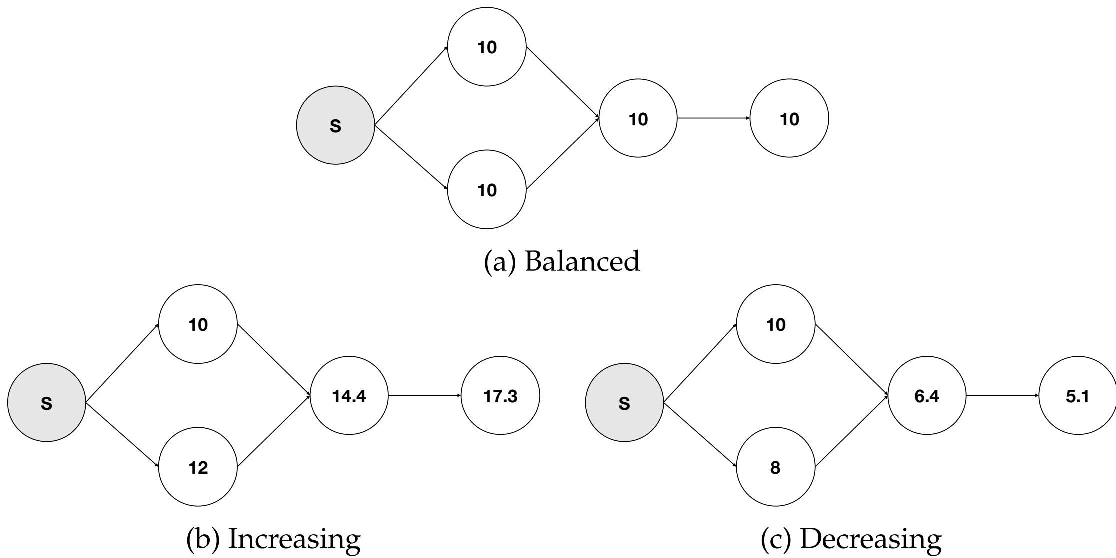
Processing Workload

A similar process is done also for the processing workload. The user specify a value for the processing load of the application (as seen in Section 4.6.1). As for the parallelism, the workload value has to be distributed over the tasks. Differently from the parallelism representation, the workload value doesn't represent the total processing amount of the application. In this case, the value represents the starting-point for the loads, i.e. the value assigned to the first task. As for the parallelism, several distribution methods are implemented: *balanced*, which assigns the same *workload processing* value to all the tasks; *increasing*, which assigns the *workload processing* value to the first task and increases it by 20% for each following ones; *decreasing* which works the same way but decreases the workload; and *pyramid*, which merge together the *increasing* and *decreasing* methods at the same manner as for the parallelism.

Fig. 4.12 shows an examples of the various balancing configurations. The examples are on a diamond topology of depth 4 (i.e. 1 source, in gray, and 4 tasks, in white). With a base value of 10 (i.e. 10000 busy wait cycles) the three different distributions set a different value in each task (excluding the source). A *balanced* configuration (Fig. 4.12a) will assign 10 at each task. *Increasing* (Fig. 4.12b) will start from a value of 10 for the first task, then 12 (i.e. +20%), follows a value of 14.4 (i.e. +20% of 12), and so on. Vice-versa, with a *decreasing* configuration (Fig. 4.12c), still starting from a first value of 10, to the second task will be assigned a processing load of 8 (i.e. -20%), then a value of 6.4 to the third (i.e. -20% of 8), and so on.

Traffic Routing

As introduced in Section 2.1.3, in streaming applications tasks can be connected to each other through different grouping methodologies. These techniques describe how the data traffic is routed between the tasks. The schema has a field to specify which kind of grouping we want in the application.



(a) Balanced

(b) Increasing

(c) Decreasing

Figure 4.12: Workload balancing examples; with a base processing value of 10; balanced, increasing and decreasing configurations

By default, the defined method is applied at every step of the topology. However, given the differences between platforms, this rule has to be bypassed by some platforms to accommodate specific architectural optimizations (Section 4.6.3).

Data Filtering

A filtering parameter allows to reduce the data volume at runtime. To keep the schema generic, it does not specify *when* or *how* to apply the filtering.

For the *when*, we set a fixed position. To be able to test the application with the two volume loads, filtering is applied at the middle of the topology. As an example, if the DAG depth is 8, the filtering will be applied at the third task (i.e. level 4 of the DAG). In the case of a non-linear topology, if the middle level corresponds to the double-task level (i.e., 2 for diamond or 3 for star), filtering will be applied to both of them.

For the *how*, the user can set a filtering value. This value represent the percentage (normalized to 1) of received messages that the task will forward to the next one.

Windowing

Following the same reasoning for the data filtering, the windowing has been associated to a single task (i.e. the one that would perform the aggregation). For

convenience we set the windowing task as the second task of the topology, or the first in case the application is composed just by the source and one task. The user can define the type of window (e.g. thumbling, sliding) and the window size.

4.6.3 Managing Platform Specifics

NAMB generates applications for multiple SPEs. When generating the application from the Workflow schema or the Pipeline schema, it is necessary to take into account the major design and implementation differences between platforms [GHS16]. Each platform implements in a different manner the way to define data routing among tasks, reliability mechanisms and scheduling strategies.

An example of such discrepancy can be seen between Flink and Storm. Those platforms implement different grouping methods to connect the tasks, e.g. Flink: direct connection, absent in Storm (Sections 2.2 and 2.3). Likewise, these differences may reflect in the scheduling algorithm and different optimization made by the platforms (Sections 2.2.3 and 2.3.3).

In the current version of NAMB, whereas in Storm and Heron the specified routing is applied between each task, in Flink it is only applied at the first level, between the source and the first-level tasks. This preserves the chains of subtasks (Section 2.3.3) and produces a more realistic and *optimized* prototype.

While the majority of applications generators perform also query optimization, it is not our main focus. Our objective is to generate prototypes. For this reason, we take into account platform specifics only to make design decisions for the generation. However, it would be easy for someone to implement an optimized generator for a specific platform, taking into account the previously described design rules and the internal mechanisms of the platform.

4.6.4 Data Generation

Synthetic Data Distribution

NAMB implements an internal synthetic Data Generator. The generator manages the form of the data as well as the composition of the dataset, its size (i.e. how many values the dataset is composed of) and the appearance probability of the single values.

The synthetic sources will produce strings of a specified size from a set of unique values, sequentially increasing the characters starting from the right-hand side one, composing a series of the form: $[aaa, aab, ..., aaz, aba, ..., abz, ...]$ (in case of a 3-byte long data). Even though we acknowledge that in DSPs we

may encounter different types of data, we currently only generate strings. As we saw in Section 4.6.1, no actual processing is performed directly over the tuples, so the actual data type does not impact the application behavior. However, the size of data impacts its transfer over the network between the tasks and the distribution of tuple values (uniform or biased) impacts the routing of tuples in the topology.

Concerning the appearance probability, the high-level description accepts two different distribution types. A uniform distribution will give an equal probability to each value to appear in the stream. To simulate the popularity of some data, it is also possible to specify a *non-uniform* distribution. In the first case, the size of the set matches the unique values as each of them will be present only once; whereas, to have an heterogeneous probability of values generation (when *non-uniform* is set), the generator will create different copies of the same value.

The Data Generator manages also the arrival flow properties. It defines the rate at which the data will be produced, as well as its arrival distribution. Data may arrive at a constant bit rate, or present some variability, as already discussed in Section 4.4.1.

External Generator

The synthetic generator is subject to several limitations as already discussed in Section 4.5.4. As an example, the Java implementation of the Synthetic Generator does not allow to set an inter-tuple interval under 1ms, except for the unlimited rate (bounded only by the processing capabilities of the node). This limitation could be easily overcome with a Kafka producer written in C.

For such reason, we use the external generator Section 4.5.4 capability to connect a Kafka broker to NAMB. As an external module of NAMB, we implemented a Kafka producer in C. The producer follows the same design as the internal one. A major difference is given by the configuration file. Even though it is tightly based on the properties of the yaml file, C doesn't support yaml parsing. For simplicity reasons, we implemented the configurations through the `libconfig` library.

As we can see in Fig. 4.13, the configuration file is divided in 4 sections. The sections *datastream* and *flow* are the same used in the Workflow schema and Pipeline schema. The only adaptation is for some property names, as libconfig requires unique names overall and not per section like yaml. E.g. *ddist* and *sdist* in place of the keyword *distribution*. The section *kafka* specifies the connection to the Kafka broker (i.e.where to send the messages). Finally, a *global* section is used for generic configurations. The figure shows the *debug* option, used in the same manner as for the NAMB configuration, to specify the frequency of tuple

```
datastream =
{
  data = {
    values = 100;
    size = 10;
    ddist = "uniform";
  };

  flow = {
    rate = 2000;
    phase = 300;
    sdist = "sawtooth";
  };
};

kafka =
{
  broker = "localhost:9092";
  topic = "namb-topic";
};

global =
{
  debug = 0.002;
};
```

Figure 4.13: Configuration of the external data generator

details logging.

4.7 Conclusion

In this chapter, we have presented NAMB, *Not only A Micro-Benchmark*, a generic application prototype generator. NAMB features two high-level description models for streaming topologies to produce a working streaming application prototype for a target platform.

Based on the analysis of the main characteristics of DSP applications, we have devised the Workflow schema and the Pipeline schema. They allow a precise, high-level, definition of a streaming application. The first one can be

used to quickly write prototypes for a set of canonical topologies. The second one can accurately reproduce complex applications.

The prototype generator, NAMB, uses the two schemas to generate prototype applications. To remain platform- and application-independent, we simulate the tasks processing workload, through an equivalent busy wait function, that replaces complex application-dependent code. A Data Generation component, with support for synthetic and external data, is used to inject tuples to the application. In addition, we give specific insights on the implementation challenges addressed to translate the generic description to actual code.

Using NAMB, a user can investigate the impact of design choices on the overall performance by simply modifying the configuration file instead of the real application. In the next chapter, we empirically show how our application generator exploits these high-level descriptions to automatically generate prototype applications.

The current version of NAMB, along with the configuration files used in this thesis, is available on GitHub[1]. It supports Storm, Flink and Heron. The external C generator is as well publicly available[2].

[1] https://github.com/ale93p/namb

[2] https://github.com/ale93p/namb-c-datagen

Chapter 5

Evaluation and Practice of NAMB

Contents

5.1 Experimental Setup & Methodology 86

5.2 Application Design 87

5.2.1 Connection Routing 88

5.2.2 Parallelism Scale-up 89

5.2.3 Processing Load Balancing 90

5.2.4 Data Size 90

5.3 Application Prototyping 92

5.3.1 The Yahoo! Streaming Benchmark 92

5.3.2 Insights on Processing Load Tuning 94

5.4 Bottleneck Discovery 99

5.5 Platform Features Testing 100

5.6 Data Generator 102

5.6.1 Internal Generator 102

5.6.2 External Kafka Producer 104

5.7 Conclusion 104

In this chapter, we present different scenarios where we use NAMB to perform fast and easy testing of SPEs using prototype applications. We showcase different scenarios for the two description schemas, Workflow schema and Pipeline schema, using NAMB on multiple platforms, namely Apache Storm and Apache Flink.

In the previous chapter we motivated the need of a general solution to perform streaming platform and application testing. In this respect (Section 4.2) we highlighted a list of limitations in the current state of the art. Furthermore, we listed a set of characteristics that a general solution should have.

We identified the need for a **generic** and **flexible** solution. Hence, we designed and implemented NAMB and the application high-level description to be: (i) able to *correctly define* the prototype application; (ii) able to test *different features* that could be adapted to *every context and scenario*; (iii) that can be *quickly and easily customized*; (iv) able to run on *multiple platforms*.

We devise a set of tests aimed to demonstrate that NAMB, coupled with the high-level schemas, covers all the features listed above. These tests are presented in the following sections: in Section 5.2, we show how by quickly changing the high-level description, one can easily explore different application design choices; in Section 5.3 we show that apart from micro-benchmarks, NAMB can as well replicate real applications performances; in Section 5.4 we exploit the Pipeline schema to analyze possible application optimizations; in Section 5.5 we use NAMB to evaluate the impact of platform internal mechanisms; finally, in Section 5.6 we describe the characteristics of the internal and external generators (Section 4.6.4), making a comparison between the two.

5.1 Experimental Setup & Methodology

All the experiments were done on a 4-node Linux cluster on the Grid'5000 testbed[1]. Each node has two 4-core Intel Xeon CPUs and 32GB of memory. The cluster is interconnected by a 1Gbps network. Out of the 4 nodes, one is used as a master node (Nimbus for Storm and Job Manager for Flink), and the other three as worker nodes (resp. Supervisors and Task Managers). Other tools, as Zookeper or Redis, are co-placed in the master node.

We use throughput and latency as the two evaluation metrics. The throughput is measured as the total number of tuples produced by the sources task per millisecond. The latency is the processing latency [ABC+15], the average time spent by tuples between the source and a sink. We exploit the `debug` option of NAMB (Section 4.5.5) to sample the processed tuples. We set it to sample 1 tuple each 2000. Such a value allows to retrieve enough samples to compute a statistical analysis, without affecting application performance with I/O operations. Hence, throughput and latency are computed over those samples.

For the presented experiments, we have used Storm and Flink. We use one or the other platform based on the test, to prove the capability of NAMB to run

[1] `https://www.grid5000.fr/w/Grid5000:Home`

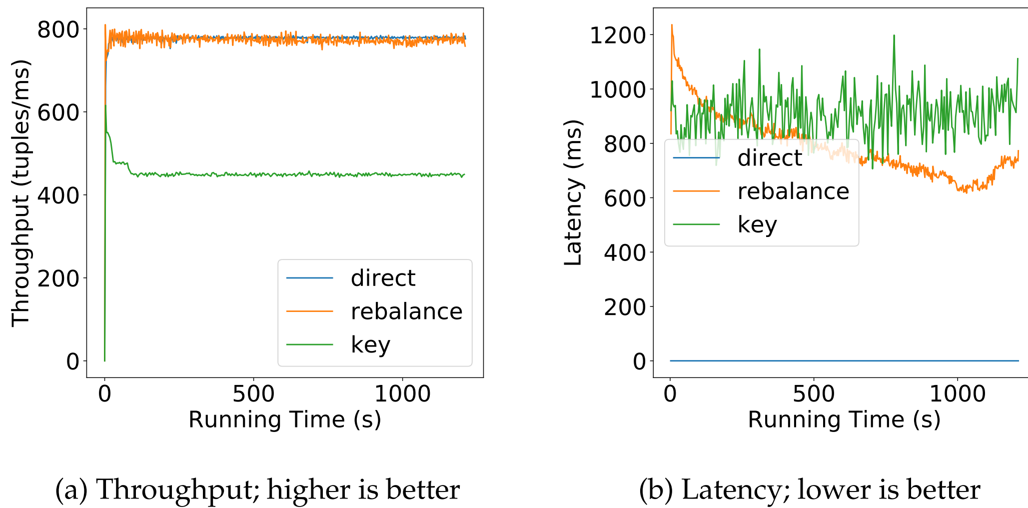
on different platforms. We highlight the fact that any supported platform can be used to reproduce the tests presented in the following.

5.2 Application Design

To demonstrate the benefits of the Workflow schema, we show how starting from a common linear topology, we can quickly evaluate the impact of small changes in the design choices. We evaluate it on **Flink** with 4 different micro-benchmarks, using the base configuration file shown in Fig. 5.1. For each experiment, we focus on a single parameter change (highlighted in gray in the figure). The considered topology is made of a single source and 3 other tasks organized in a *linear layout* with a *balanced* parallelism distribution. The topol-

```
datastream:
  synthetic:
    data:
      size: 10
      values: 100
      distribution: uniform
    flow:
      distribution: uniform
      rate: 0

workflow:
  depth: 4
  scalability:
    parallelism: 96
    balancing: balanced
  connection:
    shape: linear
    routing: none
  workload:
    processing: 10
    balancing: balanced
```

Figure 5.1: Base configuration file for Workflow schema experiments. Highlighted values are changed during tests



(a) Throughput; higher is better

(b) Latency; lower is better

Figure 5.2: Flink performance with different connection routing between source and task. 4 tasks with 24 instances each.

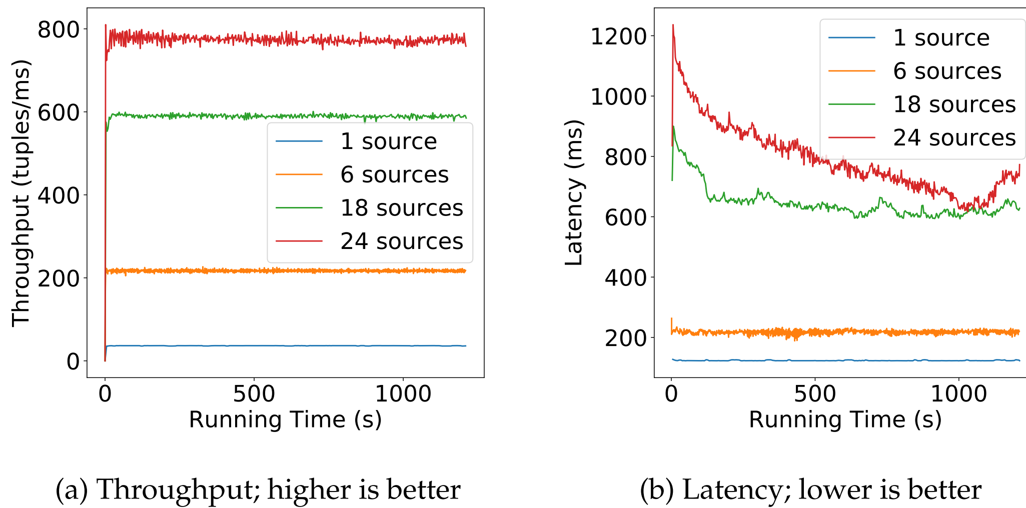
ogy is fed with a *uniform* synthetic data stream of 100 unique values of 10 bytes each, without rate limit.

5.2.1 Connection Routing

In this test, we analyze the impact on the performance of the different routing systems of Flink. We fix the parallelism level to 96, the upper-bound imposed by Flink in our hardware environment [FJS].

A parallelism of 96 creates 24 instances per task (including the source). The connection routing, or grouping, specifies how the tuples are distributed between task instances. We test three different grouping schemes by changing the *routing* type parameter. The direct connection (*none* value in the configuration) directly connects the source to the task. As a consequence, Flink groups all the tasks in the same chain, as explained in Section 2.3.3. The rebalance routing (*balanced* in NAMB) equally distributes the tuples between all task instances. And finally, grouping by key (*hash*), distributes the tuples based on the hash value of the tuple. For these last two methods, Flink creates two different tasks (Fig. 2.4b), therefore the tuples will need to travel the network to be routed to the assigned task (i.e, the network layer is not bypassed with the sub-chain approach).

Fig. 5.2a shows that balancing and direct routing achieve the same throughput, while the key-based (i.e. hashed) grouping offers significantly lower performance. On contrast, in Fig. 5.2b we can see a significant impact on latency



(a) Throughput; higher is better

(b) Latency; lower is better

Figure 5.3: Flink performance when varying components parallelism using a rebalanced connection. The different series represented are defined by the number of source instances.

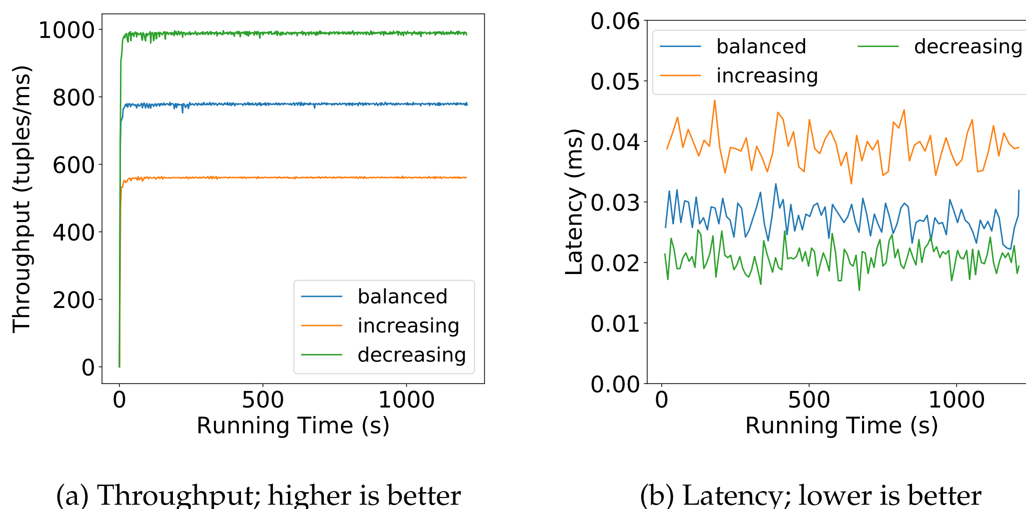
when the tasks are directly connected and co-placed: the latency is significantly lower than the other two methods, only 0.03 ms of average completion time compared to the 701 ms and the 907 ms for balanced and hash respectively.

5.2.2 Parallelism Scale-up

Here, we evaluate the impact of the parallelism level on the entire application. We have set routing to rebalance mode.

In this evaluation we try four different parallelism levels, starting from the lowest possible value (4, i.e. 1 instance per task) until the highest one (96, i.e. 24 instances per task). In the configuration file, we just need to change the *parallelism* value and re-deploy NAMB.

From the results in Fig. 5.3 we can see how, as expected, the throughput increases when increasing the application parallelism (Fig. 5.3a). More sources and more tasks to process data, increases the global emission rate. As we have enough tasks to process the data generated by the sources, the final throughput equals the sums of all the sources. Starting from a single source emission rate of 36 tuples/ms, the throughput linearly increases with the number of source instances: 218 tuples/ms with 6 sources, 646 tuples/ms with 18 sources, up to 791 tuples/ms with 24 sources. On the other hand, the latency increases as well (Fig. 5.3b) due to the placement of the tasks on different nodes, following the *network distance* principle [PHH+15].



(a) Throughput; higher is better

(b) Latency; lower is better

Figure 5.4: Flink performance with different workload distribution. 4 tasks of 24 instances each, connected via direct routing

5.2.3 Processing Load Balancing

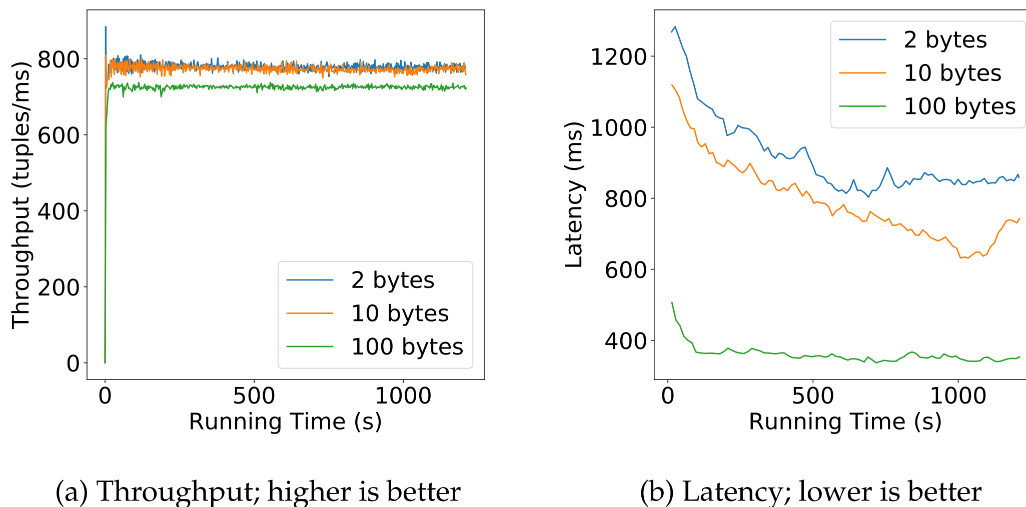
In this experiment, we try to change the workload balancing, to see the impact of a computing bottleneck on the application performance. In the NAMB configuration, we just need to change the *workload balancing* property. We keep the parallelism level to 96 and the routing to direct (*none*).

We investigate three different distributions: *balanced*, *increasing*, and *decreasing*. As seen in Section 4.6.2, the three distributions will respectively: assign the same load value at each task, assign the base value to the first task and then increasing the load at each successive task and vice-versa.

As expected, increasing the computing load greatly lowers the throughput compared to a balanced one (Fig. 5.4a). Also, if the load decreases, the application can process more tuples. The completion latency (Fig. 5.4b) follows the same trend. What is interesting in this experiment is that it shows that, in Flink, having a high load on the first task will not necessarily create a bottleneck. Indeed, since Flink's built-in scheduler will co-place (as a single thread) all directly connected tasks, the overall load is more important.

5.2.4 Data Size

Another important factor to take into account when developing an application is the size of data. In most cases, the data will have to be transferred between machines, impacting the overall performance of the application.



(a) Throughput; higher is better

(b) Latency; lower is better

Figure 5.5: Flink performances with different data sizes, 4 tasks of 24 instances each connected through rebalance routing

This test explores three different data sizes, 2, 10, and 100 Bytes, representing a number, a string and a JSON message respectively. We will use a rebalance routing strategy to ensure to have network communication but without the overhead of the hash functions.

Fig. 5.5a shows that the data size has a modest influence on the throughput. However, the impact on the latency is high. The larger the tuples, the lower the latency (Fig. 5.5b). Flink uses internal buffers [Kru] which are flushed either after some timeout expires or when they are full. Having large tuples triggers the second condition faster, decreasing the average completion latency of the tuples.

In this section, we have shown how NAMB can be used to quickly generate a set of prototypes. By slightly modifying the configuration file, a user can investigate the impact of multiple different features of a DSP. In our case, on one side, we were able to confirm the impact on throughput of the parallelism level and the load distribution of the tasks. On the other side we discovered the significant impact of the key grouping connection and the impact on latency of higher data sizes. This kind experiments, can ease the user design of their application.

5.3 Application Prototyping

5.3.1 The Yahoo! Streaming Benchmark

Using the per-task granularity of the Pipeline schema, we can reproduce existing applications and create prototypes with similar performance. For this evaluation, we use the Yahoo! Streaming Benchmark (see Fig. 4.2 and Section 4.1.1). To validate our approach, we compare the results on two different platforms: **Storm** and **Flink**. As in [Yav17], we have modified the Yahoo Streaming Benchmark to remove the Kafka producer and use a local ad-hoc data generator instead, so as to maximize the application throughput. Besides, it makes the benchmark generator more comparable to NAMB internal synthetic one.

For both platforms we have set a parallelism level of 1 for each task except for the last one, for which it was set to 2.

Fig. 5.6 shows the relative difference between NAMB and the Yahoo Stream-

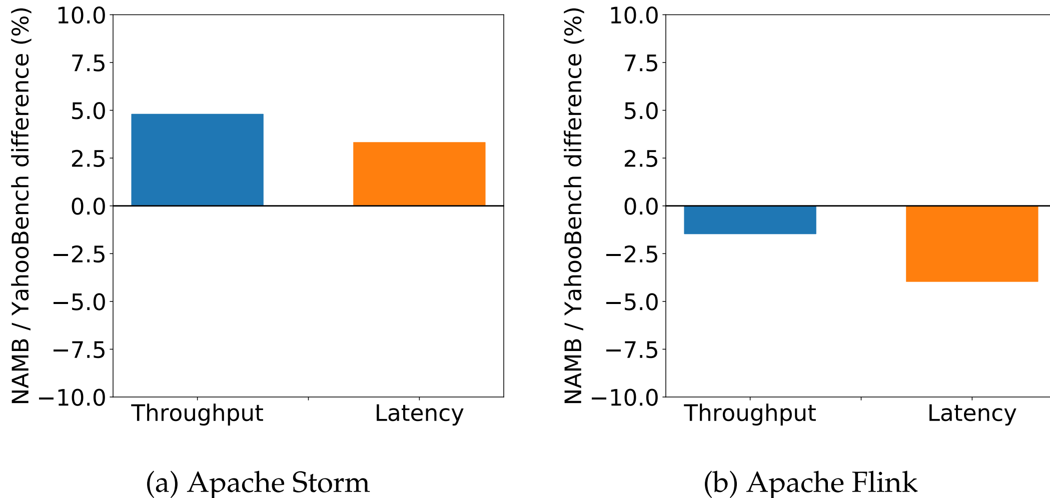
	Throughput (tuples/ms)	Latency (ms)
Yahoo Bench	101.66	48.39
NAMB	106.58	50.13

(a) Apache Storm

	Throughput (tuples/ms)	Latency (ms)
Yahoo Bench	128.83	15.03
NAMB	126.89	14.42

(b) Apache Flink

Table 5.1: Real application simulation: results on Storm and Flink



(a) Apache Storm (b) Apache Flink

Figure 5.6: Throughput and latency percentage difference ratio between NAMB and Yahoo Bench results

```yaml
pipeline:
  tasks:
  - name: ads
    parallelism: 1
    data:
      size: 180
      values: 1000
      distribution: uniform
    flow:
      distribution: uniform
      rate: 0

  - name: event_deserializer
    parallelism: 1
    routing: balanced
    processing: 6.9
    parents:
    - ads

  - name: event_filter
    parallelism: 1
    processing: 0.7
    filtering: 0.333
    parents:
    - event_deserializer

  - name: event_projection
    parallelism: 1
    processing: 2.2
    resizeddata: 52
    parents:
    - event_filter

  - name: redis_join
    parallelism: 1
    processing: 3.0
    parents:
    - event_projection

  - name: campaign_processor
    parallelism: 2
    routing: hash
    processing: 2.1
    parents:
    - redis_join
```

Figure 5.7: Pipeline schema configuration file used for Storm. Highlighted values differ in the Flink configuration.

ing application for both platforms. We can see that NAMB is able to obtain almost the same performance as the original application (Table 5.1a). The Yahoo Benchmark in Storm (Fig. 5.6a) reaches a throughput of 102 tuples/ms, whereas NAMB achieves 107 tuples/ms, less than 5% difference. Meanwhile, the average latency is around 48 ms with the YahooBench and 50 ms with NAMB (slightly more than a 3% difference). The difference with Flink (Table 5.1b) is similar in terms of throughput with only a 1.5% difference, 129 tuples/ms for the YahooBench vs 127 tuples/ms with NAMB). The difference in latency is 4%, 15 ms with the YahooBench and 14.5 ms with NAMB.

From these results, we can observe that Flink gives slightly better performance. This is because of its task grouping policy, which reduces network communications. Storm, on the other hand, uses 6 Java Virtual Machines. However, this result could change with different configurations files. Hence these results cannot be directly used to assert the performance of one platform over the other.

In Fig. 5.7 we see, as an example, the configuration used for Storm. For these two experiments, we did not use strictly the same configuration file. Indeed, the computational load of Storm and Flink is different for the same application (Section 4.6.3). We adapted the processing loads in Flink to be 60% of the ones used in Storm. This was, however, the only difference. The rest of the configuration parameters being strictly identical.

5.3.2 Insights on Processing Load Tuning

The choice of a correct processing value for a given task is non trivial. We just show how distinct platforms react differently to the same processing load. To find the matching performance that replicates the Yahoo! Benchmark we had to specify two different sets of processing values for Flink and Storm. In the following we further investigate the causes of this necessity, trying to get better insights on this application behavior.

First of all, we have to consider that those tests were run without a data generation rate limit. In this manner the generation reaches the platform stress point, enforcing backpressure at the source, limiting the application to its maximum achievable throughput. Hence, we are in an unstable condition. Moreover, given the lack of a common semantic in data streaming (Section 2.1.2), the internal implementation of the backpressure mechanism is different between Flink and Storm.

Thus, we analyze the behavior of the busy wait function with a limited rate, which doesn't cause backpressure. To test the two platforms reaction to backpressure, we gradually increase the processing load value, up to the point where the task takes more time to process the tuple than the source to produce it. At

this point, the incoming buffers start to fill and backpressure is enforced.

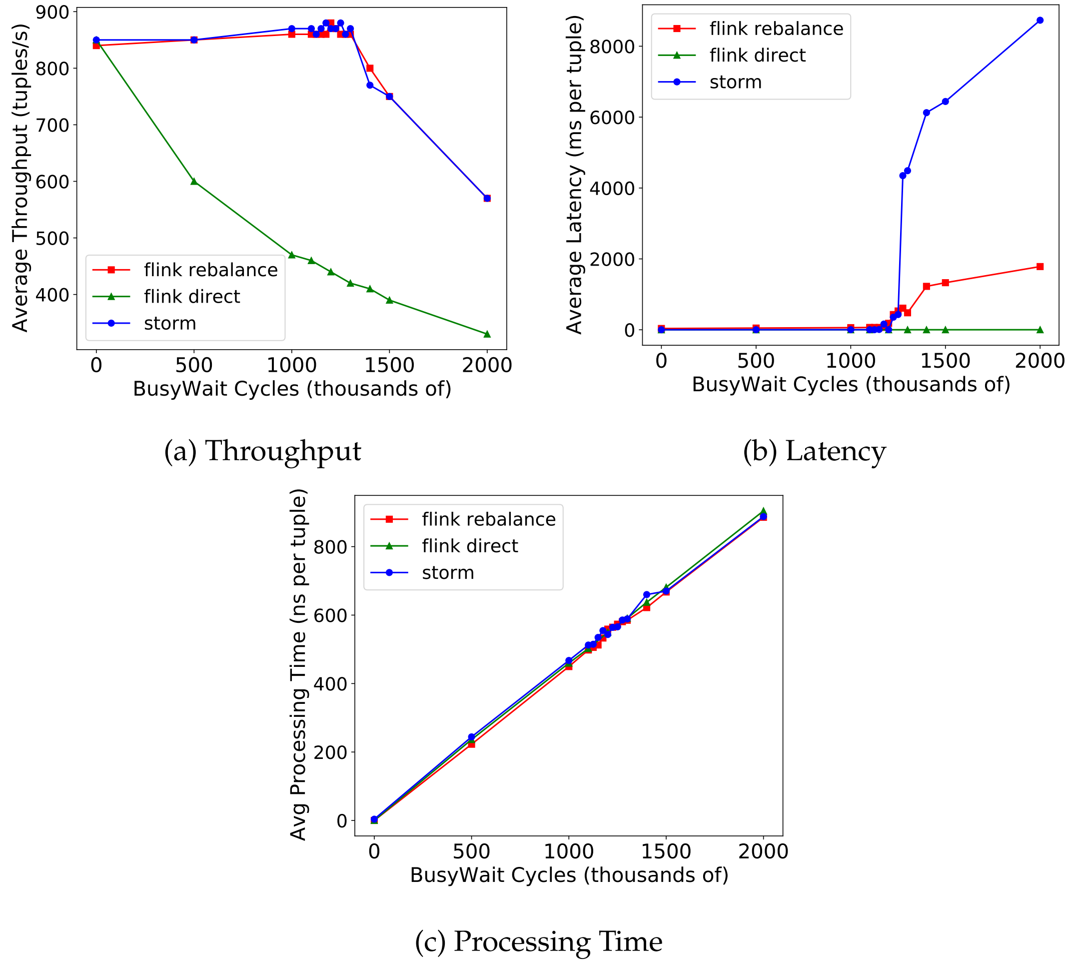
We run a simple source-task micro-benchmark. We limit the generation to 1000 tuples/s, so as not to immediately trigger backpressure with low processing loads. We start from a processing load value of 0 (i.e. 0 busy wait cycles) and we increase it up to 2000 (i.e. 2 millions busy wait cycles), largely above the processing capacities. We perform the tests in Storm and Flink, both using one single JVM, i.e. one worker for Storm and one task manager slot for Flink. Since Storm creates a thread for each task, while Flink has operator chaining (Section 2.3.3), we tested Flink with two different configurations: one using a *rebalance connection*, that creates one thread per task, to make it comparable to Storm (we will refer to it as *Flink rebalance*); the other using a *direct connection*, that applies operation chaining creating a single thread for both tasks, to see which is the difference (we will refer to it as *Flink direct*).

Processing Load Impact on Application Performance

As expected, looking at the throughput (Fig. 5.8a), *Flink direct* is the one behaving differently. The advantage of the single thread is the nonexistent communication cost, shown by a stable low latency (Fig. 5.8b). However, the trade-off is that the two tasks are not parallelised. Being co-located in the same thread, the source and task operations are done sequentially. In other words, the source operations to produce a new tuple won't run until the task finish to process the current tuple. Hence increasing the processing load of the task impacts as swell the source capacity, reducing significantly the generation throughput. A quick takeaway can be to never use a direct connection between the source and the task to achieve the best performance.

Meanwhile, *Storm* and *Flink rebalance* have an almost identical throughput evolution (Fig. 5.8a). In both cases it is constant until a processing load of 1300 and we see it that starts to decrease at 1400. This indicates that there is no difference between the two platforms in terms of task processing, meaning no specific optimization is applied by either platform. This is confirmed as well by the identical (average tuple) processing time measured for all three configurations (Fig. 5.8c). Instead, we see a significant difference in latency (Fig. 5.8b). Even though for both platforms, latency follows the same trend, we can see that after the processing value of 1250, the latency in Storm spikes significantly higher than Flink rebalance.

Fig. 5.8c shows that same processing load values have the same computation time in both platforms. Yet, in the previous section we had to configure two different values for Storm and Flink to achieve the same application performance. From our analysis, it appears that only latency is impacted differently in the two platforms. As we stated at the beginning of the section, the previous tests were



(a) Throughput

(b) Latency

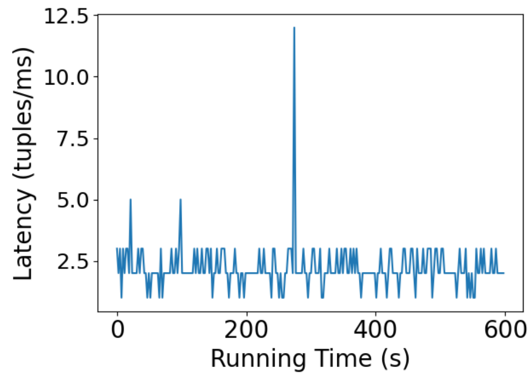
(c) Processing Time

Figure 5.8: Application performance metrics in relation for busy wait cycles

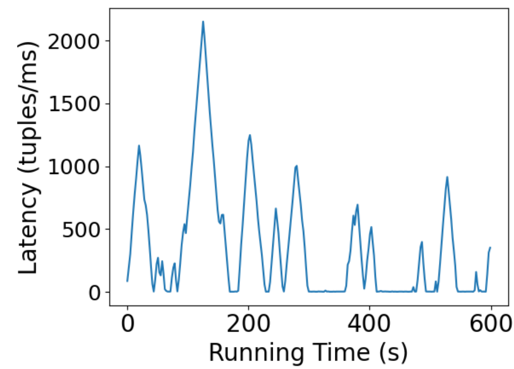
run under a backpressure condition. Hence, in the following, we further study the role of backpressure in relation to the processing load.

Backpressure Characterization Through Streaming Time Series

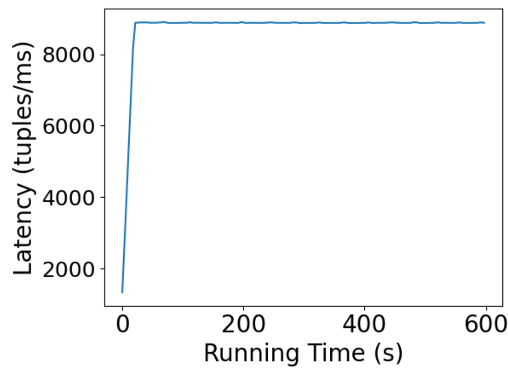
To investigate this parameter, we look at the specific time series of the test runs. We notice that both platforms follows the same pattern: we present Storm as an example in Fig. 5.9. In the presented figures we show the latency, as throughput behaves speculatively. Initially, with low processing load values, we find a stable condition (Fig. 5.9a), where the application runs in its **normal state**. Then, increasing the load, throughput and latency are highly variable (Fig. 5.9b), as the application **continuously alternates** normal and backpressure state. The mem-



(a) Normal State (500’000 BW Cycles)



(b) Alternating State (1’225’000 BW Cycles)

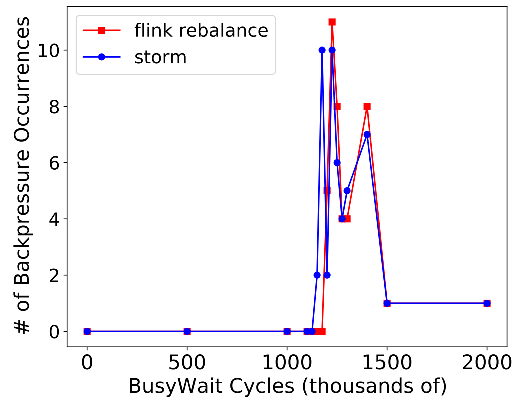


(c) Persistent Backpressure State (2’000’000 BW Cycles)

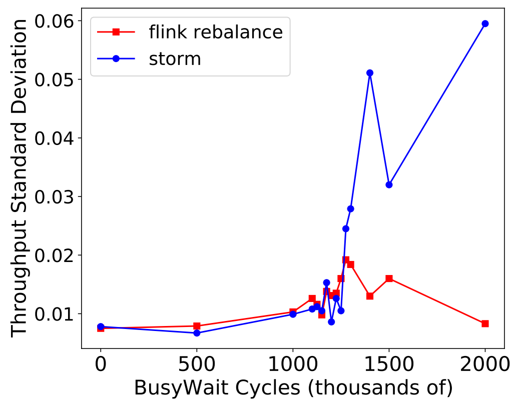
Figure 5.9: Latency Storm Runs Time Series of Latency

ory buffers continuously fill, generating high latency spikes and consequently significant throughput reductions. Finally, when the load is excessively high, the application enters in a **persistent backpressure state**. We find a single initial backpressure event (Fig. 5.9c) that limits the application rate at its maximum achievable throughput. The latter is the same condition under which the tests where run in the previous section.

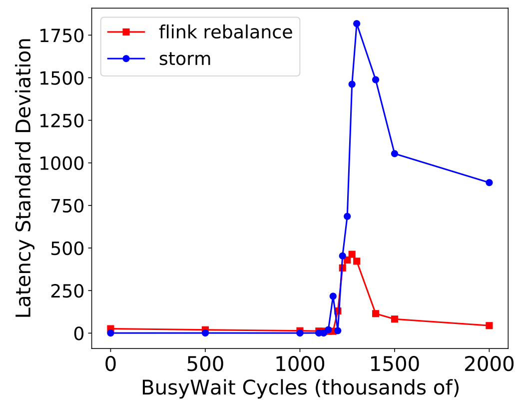
Hence, we made a preliminary study to better understand how to characterize the backpressure behavior through the time series, as it impacts a correct definition of a task processing load value in NAMB. A first discrete metric is the number of backpressure occurrences. The three conditions described above can be seen if we look at the number of backpressure events for each processing load (Fig. 5.10a). For this preliminary study we manually verified their pres-



(a) Backpressure Occurrences



(b) Throughput STD



(c) Latency STD

Figure 5.10: Backpressure and metrics standard deviation for busy wait cycles

ence in the time series – however, we remark that it would be optimal to have the platform to log every time the application enters a backpressure state. In the plot (Fig. 5.10a) we can see how increasing the processing load value increases the number of backpressure occurrences. Then, they start to decrease down to a single occurrence. This is given by the fact that every backpressure event is also characterized by its duration.

Thus, at heavier processing loads the application will present less backpressure occurrences but of a longer duration, significantly impacting the computation of average throughput and latency. We deduce that the average values for the two metrics are not accurate enough to find correct processing load values for real applications. A first attempt to find a better evaluation metric, that would better characterize the time series in relation to the backpressure

events, is to consider the standard deviation (STD) of the two evaluated metrics (i.e. throughput and latency). As we can see, the STD values for throughput (Fig. 5.10b) and latency (Fig. 5.10c) follow similar patterns to the one of the backpressure occurrences (Fig. 5.10a). Moreover, standard deviation can also give us a rough idea of the backpressure heaviness, e.g. longer backpressure events, higher standard deviation.

In summary, we showed that average throughput and latency are not enough to tune the processing load value. We demonstrated that Storm and Flink manage backpressure differently at high processing loads. This results in different application performance for the same load value. Hence, the necessity of different configurations for the two platforms. To this end, we used two new metrics to better characterize the time series resulting from the streaming application run: backpressure occurrences and standard deviation of throughput and latency. Through these analysis we showed that when it comes to the choice of the processing value, we do not need to just consider the application overall performance. This study underscores the need to consider the impact of backpressure. However, other features implementation differences, in other platforms and in different conditions, may require the same attention.

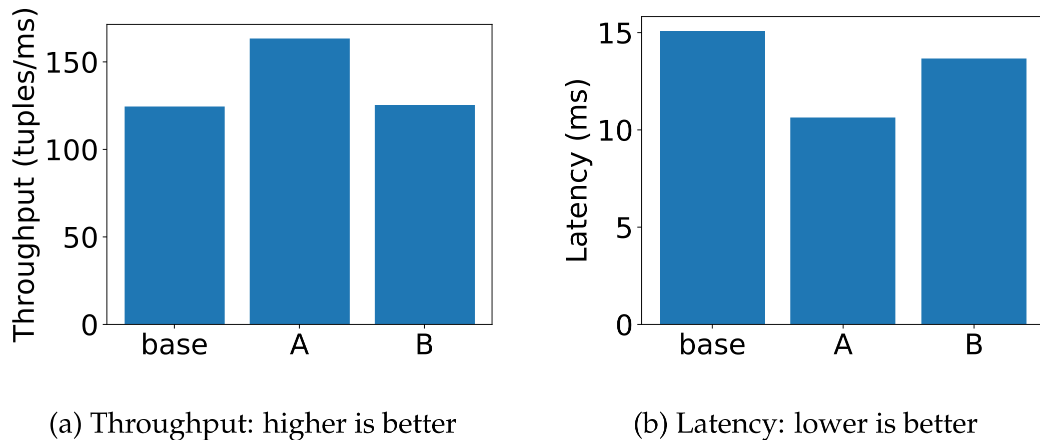
5.4 Bottleneck Discovery

Once found the correct configuration for the Yahoo! Benchmark (like in Section 5.3.1), we can exploit NAMB's flexibility to investigate the behavior of an application under various conditions. We consider a scenario where developers are interested in finding potential bottlenecks in their application. Without loss of generality, we chose **Flink** as SPE.

We used the base configuration shown in Fig. 5.7 (experiment base). We used two others configurations as possible *optimizations*: one where we halved the processing load of the *event deserializer* (experiment A); and another where we made the same *optimization* to the *campaign processor* (experiment B). The goal is to evaluate the impact of some potential code optimization on these two components.

The results of the evaluation in Fig. 5.11a shows that, in terms of throughput, the event deserializer is one of the possible bottlenecks in the application. Meanwhile, reducing the load of the campaign processor does not have an impact on throughput. On the other hand, considering latency (Fig. 5.11b), we see that in both cases we have an improvement of the tuples processing time. Yet, halving the event deserializer produces a more substantial improvement.

In this section, we showcased NAMB capability to perform quick analysis



(a) Throughput: higher is better

(b) Latency: lower is better

Figure 5.11: Bottleneck discovery experiment in the three different configurations

on the application design, that would help to better understand the impact of the application design over the performance. In this case, we take the study of the bottleneck as an example, however, NAMB can be used at the same manner to study different application design choices.

5.5 Platform Features Testing

We use here the Workflow schema to quickly test specific platform mechanisms. In this experiment, we analyze the back-pressure mechanism of **Storm**, enabled by the acking framework. We have studied the impact of the framework in the first part of this thesis (Chapter 3). Back-pressure ensures a limit to the application throughput, to avoid overloading the system in case of a too high rate of incoming tuples. The acking framework is also used to ensure message reliability. Its activation in Storm requires adding specific code in every task. In NAMB, this can be done by simply setting the reliability parameter to `true` in the *workload* section.

Moreover, as we mentioned before (Section 4.5.4), the *synthetic generator* cannot be set to go below 1ms of interval between tuples. However, we need to stress the system to trigger the back-pressure mechanism. We use multiple Kafka producers and a Kafka server to generate an high rate of tuples. The latter (Fig. 5.12) is co-located with the master server.

The generated prototype is a source with two tasks in a linear topology, with a processing of 0 for each task, so as to let the application have the minimum processing time, thus reaching the maximum throughput. We set Storm to use

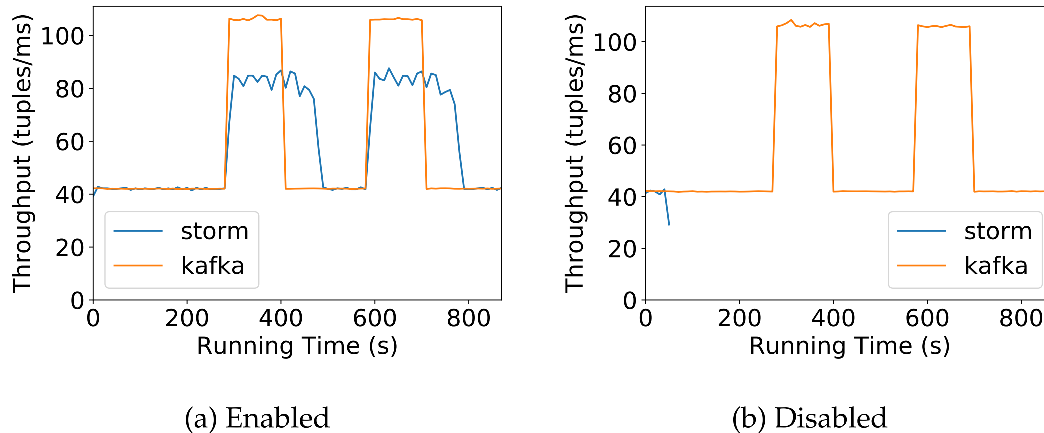
```yaml
datastream:
  external:
    kafka:
      server: <master_server>:9092
      group: test
      topic: test
    zookeeper:
      server: <master_server>:2181

workflow:
  depth: 3
  scalability:
    parallelism: 3
    balancing: balanced
  connection:
    shape: linear
    routing: balanced
  workload:
    processing: 0
    reliability: true
```

Figure 5.12: Configuration file for the back-pressure experiment.

3 workers, one for each application component.

The external Kafka producer is set to have two production phases. Most of the time it is in a steady phase, with a fixed data generation rate. However, at regular intervals, the rate is increased during a so-called burst phase. We tested Storm once with the reliability mechanism enabled and once with it disabled.

As we can see in Fig. 5.13, when the reliability mechanism is enabled, the back-pressure regulates the maximum throughput of the application. In Fig. 5.13a, the back-pressure limits the throughput to 80 tuples/ms during burst phases, which is lower than the Kafka production rate (over 100 tuples/ms). After completing the processing of all the Kafka tuples, Storm throughput goes back to the steady phase. On the contrary, we can see in Fig. 5.13b, how, without the reliability mechanism enabled, the application almost immediately fails. Even during the steady phase, it cannot process incoming tuples fast enough. The Java VM reaches the Garbage Collector threshold right after the application starts to receive the first tuples, leading to out-of-memory errors.



(a) Enabled (b) Disabled

Figure 5.13: Back-Pressure test in Storm, with enabled and disabled reliability mechanism.

Using NAMB, this kind of experiment can be quickly performed without having to modify platform-specific code, but by just changing one parameter in the high-level description.

5.6 Data Generator

To show the behavior of the extended data generator we perform different experiments with different stream distributions. We show the behavior and limitations of the internal data generator compared to the connection to the external Kafka producer.

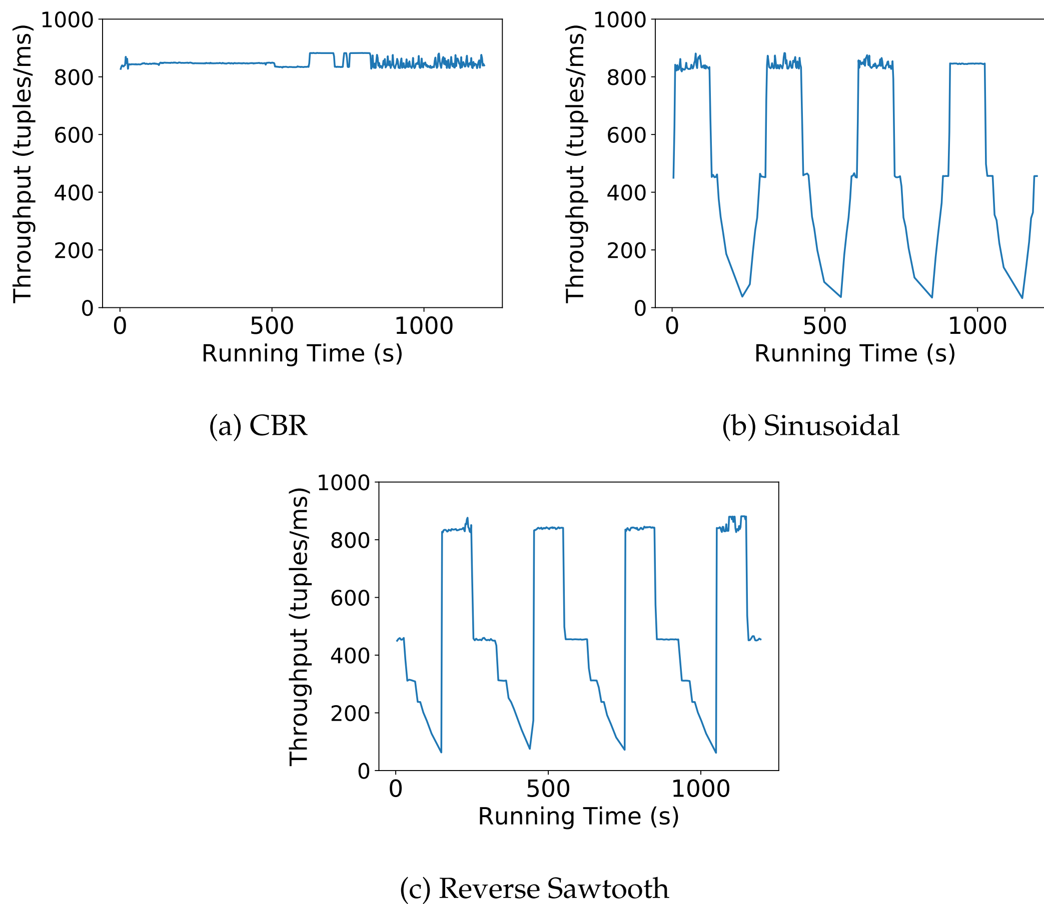
5.6.1 Internal Generator

The previous experiments (Sections 5.2 to 5.4) were performed using the internal data generator with a *CBR* distribution and an unlimited rate (Fig. 5.1). Using the same configurations, we show the behavior of the data generator by setting a predefined rate and over different arrival process. In this set of experiment we only changed the flow description as in Fig. 5.14.

We show the results of the internal generator for three different distributions (Fig. 5.15): CBR, sinusoidal and reverse sawtooth (so as to simulate an extraordinary event followed by a slowdown). For the last two distributions, we set the phase duration to 300 seconds. We set the generation rate to 1000 t/s, which is the maximum achievable by the Java generator. As previously mentioned (Section 4.5.4), Java's sleep precision cannot go under the millisecond.

```yaml
flow:
  distribution: sinusoidal
  rate: 1000
  phase: 300
```

Figure 5.14: Flow description. Highlighted distribution type changed in the experimentation.



(a) CBR

(b) Sinusoidal

(c) Reverse Sawtooth

Figure 5.15: Internal data generator throughput with different stream distributions

In Fig. 5.15a we immediately see the low precision of the internal generator. The noise is due to several factors, such as the sleep precision itself and other processing done in the task.

The sinusoidal (Fig. 5.15b) and the reverse sawtooth (Fig. 5.15c) show another limitation of the Java's sleep function. In our environment a nanosecond precision is not achievable. As a consequence, both distributions result as a sequence of steps rather than a continuous rate change. In particular, we see a different behavior between higher and lower throughput phases. This is visible mainly at two levels. First, at higher throughput we have more tuples to sample, i.e. more points on the plot and vice-versa for lower throughput where in fact we observe less accuracy. Second, the processing overhead of the task doesn't allow the throughput to reach the configured value of 1000 tuples/s. We can see a better curve at low throughput by it stays limited for a longer time during the high throughput phase.

Despite these limitations, the internal generator is able to reproduce the defined waves, which is enough for some micro-benchmarks.

5.6.2 External Kafka Producer

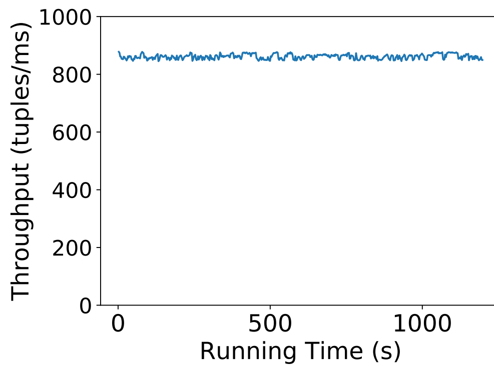
The external generator (Sections 4.5.4 and 4.6.4) aims to overcome the previous limitations. For sake of comparison, we show the same distributions as above (Fig. 5.16). However, in this case we set the rate to 2000 tuples/s, to show the capability of the external generator to reach higher throughput than the internal one.

Firstly, we see in Fig. 5.16a that the CBR scenario features a negligible noise, as compared to Fig. 5.15a. Moreover, Figs. 5.16b and 5.16c are more accurate and don't show step changes in the throughput, thanks to the microsecond accuracy achieved with the sleep function implemented in C.

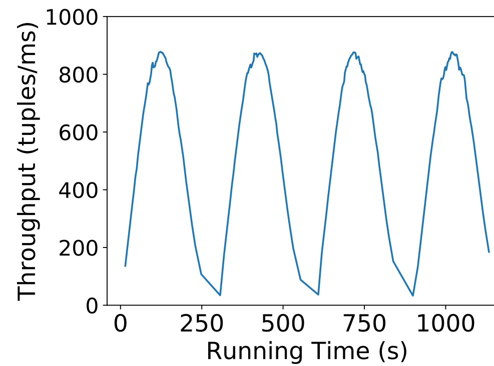
The better precision of the external generator shows that one may need develop a custom generator, to simulate more realistic data flows using real datasets.

5.7 Conclusion

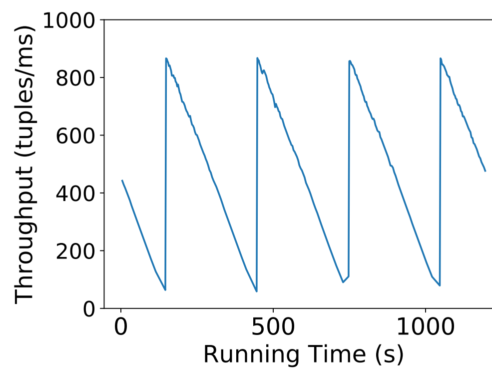
In this chapter, we evaluated through numerous experiments our streaming application prototype generator, NAMB. We have shown how NAMB can be used to quickly define and run prototyped applications over multiple platforms, with the advantage of modifying a single configuration file, instead of the platform specific API code.



(a) CBR



(b) Sinusoidal



(c) Reverse Sawtooth

Figure 5.16: External data generator (Kafka producer) throughput with different stream distributions

We used Flink to show the impact of small design choices, such as data routing and size. With the same platform we showed how can be discovered possible application optimizations, i.e. how to detect bottleneck. Using Storm, we showed how one can quickly study the behavior of internal platform mechanisms, along with tasks performance optimization evaluation. We used both systems to show how NAMB could replicate more complex and realistic applications, taking as example the Yahoo Streaming Benchmark, and achieving the same performance as the original one. We addressed the problem of the choice of processing value through a preliminary study of the matter. We finally showed the characteristics and limitations of the internal data generator component trough a comparison between the internal and the external ones.

With these various examples and test cases, we demonstrated that NAMB

offers a **generic** and **flexible** solution to the benchmarking problem pointed in the previous chapter.

Chapter 6

Conclusion

Contents

6.1 Non-Functional Tasks Scheduling 108

6.2 Streaming Application Prototyping 109

6.3 List of Available Software 111

6.4 List of Publications 111

The core of research work carried out in this thesis revolves around Big Data Analytics, and in particular Data Stream Processing. We study the limits of current DSP platforms, from an application design and platform features point-of-view.

We first investigate the impact of message reliability mechanisms in streaming applications in Chapter 3. We uncover a link between tasks scheduling algorithms and reliability mechanisms. Especially when non-functional tasks are used to enforce this kind of mechanisms, we show how a wrong placement of those tasks impacts application performance. To work around this issue, we present two scheduling algorithms that optimize the placement of the application, improving its performance.

Then, we tackle the shortcomings of current streaming application testing. Hence, we present NAMB, Not only A Micro-Benchmark (Chapter 4), a tool for application prototype generation. We use two high-level description models used to define streaming application. We describe the architecture of our tool, also analyze its implementation challenges. Finally, in Chapter 5 we use NAMB in different use-cases to demonstrate its generic and flexible nature.

6.1 Non-Functional Tasks Scheduling

Through an in-depth analysis of the Storm's acking framework, we demonstrate how a poor placement of the application tasks, ackers in particular, can degrade throughput and latency.

We firstly give a detailed explanation of the Storm acking framework, showing the internal implementation, and how it materializes in a set of application tasks that need to be placed and consume resources as every other task. We exhibit the significant amount of traffic generated by ack messages. We illustrate how an higher latency and lower throughput can result as a consequence of these two aspects.

We present an upgraded version of the Resource-Aware Scheduler (RAS), that we name Acking-Aware Scheduler. We propose two new strategies based on the default one implemented in RAS. The two strategies, OPW (One-per-Worker) and IQ (Isolated Queue), take actively in account the acking tasks and their impact, and place them consequently. In OPW, we alleviate the processing load by distributing these non-functional tasks over the workers. In IQ, we unload the buffering queues by placing the acking tasks in an isolated worker, avoiding memory competition between operational and non-functional tasks.

Through an extended evaluation phase, we show how our two proposed scheduling algorithms can improve the application performances. We compare our solution to the two internal schedulers in Storm: the Even and the RAS scheduler. In a single cluster environment our placement strategies improve both default schedulers, and in the worst case they perform as the Even Scheduler. In a larger multi-cluster environment, our solutions improve significantly over both the Even Scheduler and the Resource-Aware Scheduler, in terms of throughput and latency.

Future Directions In conclusion, we expose how a mismanagement of non-functional tasks, can substantially impact the final performances of the deployed application. In Streaming Platforms, where performances constitutes a high-priority requirement, this aspect should not be underestimated when designing and implementing the application.

Different reliability systems already exist and are implemented by existing streaming platforms. Thus, it has to be taken into consideration that alternative message reliability systems may be more performing than one implemented through non-functional tasks. Also, the design of novel ways to optimize the platform internals to make the framework lighter and the job schedulers aware of those mechanisms, could constitute a promising future work direction.

6.2 Streaming Application Prototyping

In the second part of the thesis, we study application and streaming platform testing. Among all the test applications available in literature, we uncover a lack of a generic and easy-to-use solution to quickly test diverse types of applications over different platforms. To this end, we present NAMB, a solution that is easily usable (independently from the knowledge a user may have) and that allows a quick definition of various application designs.

Following an initial taxonomy of the currently available benchmarking applications, we extrapolate the main features that significantly impact a streaming application performances. Based on these fundamental characteristics, we define an high-level and generic model to describe a streaming application. We introduce two different description models, that from a totally generic point-of-view describe the fundamental characteristics of the application we want to deploy on our platform. We propose a generic model, the Workflow schema, and a per-task defined model, the Pipeline schema. The models take shape as YAML configuration schemas, enabling the user to *write* a streaming application without knowledge of the specific platform APIs.

NAMB, Not only A Micro-Benchmark, is an application prototype generator. We design and implement the tool to generate a streaming application based on the high-level description provided by the user. We initially implement it with support for Storm, Flink, Heron and with an ongoing implementation of Spark Streaming. We design the tool to be modular and allow an easy support of new streaming platforms. We describe the challenges to convert the high-level description into actual platform code. It has to adapt also to the architectural differences of the specific streaming platform. We exemplify the problem by comparing Storm and Flink task management. We simulate the task processing load, to keep NAMB general and context-unaware, without the need of implementing actual queries. We demonstrate how a busy wait function can correctly simulate the load, facilitating the user description of the application.

Overall, we demonstrate the potential of the two description models. We show how the Workflow schema can easily and quickly allow for application customization, enabling an efficient manner to test application design choices that may impact the performances. We show that the Pipeline schema can be used to prototype real applications, achieving close performance results to the ones of the replicated application. Using this use case we focus on the challenges of the processing value decision. Analyzing the behavior of the application in relation to the load, we define more specific metrics for a correct load value choice. Given the capabilities of the two models, we show practical use cases where we can exploit NAMB. We use the quick tuning feature to discover where may reside the bottleneck operator, understanding which param-

eter should be optimized (if possible) to improve the application performance. We study and analyze the behavior of an internal platform mechanism, as the Storm reliability framework, to understand if fits with our requirements and what is its impact on performance. Finally, we show the capabilities of the internal synthetic data generator, illustrating its limitations, and demonstrating how an external data generator could deal with them.

NAMB, together with the high-level schema, allows to tackle all the missing features of current streaming benchmarking applications. It is context-unaware, so it can be used to create prototypes for every environment. It is flexible, allowing a wide range of customizations to describe the application we want to deploy. It is accessible, as it doesn't require to write application code. It is not restricted to a single platform, allowing an easy support for more technologies. Last but not least, it is available as an open source software.

Future Directions NAMB is already available and ready to use. Nevertheless, it can take advantage of further improvements. First of all, one can add the support of more streaming platforms, as Spark Streaming and Kafka Streaming. From a design and implementation point of view, we have margins of improvement both on the high-level description and the implementation of the tool itself.

The high-level description can be extended with more features of modern and upcoming streaming applications, as well as extending the definition of the currently supported ones. We already extended the initial NAMB data stream description, from a more basic definition to the one presented in our published works, with specific fields to customize the stream. The same work can be done for other fields, like the characteristics of the dataset (e.g. probabilities of occurrence, data format) or the details of internal mechanisms (e.g. windowing or reliability).

In parallel, the translation of the models in actual application can be improved. It should be possible to find better generalizations, or adapt them to modern trends, e.g. where to place the windowing task (one selected task or over all the application), how to shape the application, how to distribute the parallelism.

A major subject of interest is the study of an optimal methodology for the processing value choice. An in-depth study of the way to translate the load value from one platform to another, depending on the type of operation performed by the application, is an interesting future work. It will help to perform efficient and direct comparison of performance between platforms.

6.3 List of Available Software

Here we list, and we link to, the software produced during this thesis work, made available as Open Source:

- NAMB - Not only A Micro-Benchmark.
  `https://github.com/ale93p/namb`
- NAMB C DataGen - *C Data Generator For NAMB*.
  `https://github.com/ale93p/namb-c-datagen`

6.4 List of Publications

The contributions illustrated in this thesis have been material of the following publications:

International Conferences

- A. Pagliari, F. Huet, and G. Urvoy-Keller. "On the Cost of Acking in Data Stream Processing Systems" 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2019.
- A. Pagliari, F. Huet, and G. Urvoy-Keller. "NAMB: A Quick and Flexible Stream Processing Application Prototype Generator", 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2020

International Workshops

- A. Pagliari, F. Huet, and G. Urvoy-Keller. "Towards a High-Level Description for Generating Stream Processing Benchmark Applications", 2019 IEEE International Conference on Big Data (Big Data). IEEE, 2019

National Conferences

- A. Pagliari, F. Huet, and G. Urvoy-Keller. "Cost of Acknowledgment in Data Streams", Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS), 2019

Bibliography

[ABB+13] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[ABC+15] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment, vol. 8*, pages 1792–1803, 2015.

[ABQ13] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 207–218, 2013.

[ACG+04] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 480–491, 2004.

[AFM17] Walid AY Aljoby, Tom ZJ Fu, and Richard TB Ma. Impacts of task placement and bandwidth allocation on stream analytics. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pages 1–6. IEEE, 2017.

[AzS] Azure Stream Analytics. `https://azure.microsoft.com/en-us/services/stream-analytics/`.

[BDD+10] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J Miller, and Nesime Tatbul. Secret: a model for analysis of the execution semantics of stream processing systems. *Proceedings of the VLDB Endowment*, 3(1-2):232–243, 2010.

[BDH+15] Mark S Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D Underwood, and Robert C Zak. Intel® omni-path architecture: Enabling scalable, high performance fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 1–9. IEEE, 2015.

[BEA] Apache Beam. `https://beam.apache.org/`.

[BMNZ14] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. Fog computing: A platform for internet of things and analytics. In *Big data and internet of things: A roadmap for smart environments*, pages 169–186. Springer, 2014.

[CDE+16a] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1789–1792. IEEE, 2016.

[CDE+16b] Sanket Chintapalli, Derek Dagit, Robert Evans, Reza Farivar, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, and Boyang Peng. Pacemaker: When zookeeper arteries get clogged in storm clusters. In *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*, pages 448–455. IEEE, 2016.

[CERLDP16] Matthieu Caneill, Ahmed El Rheddane, Vincent Leroy, and Noël De Palma. Locality-aware routing in stateful streaming applications. In *Proceedings of the 17th International Middleware Conference*, pages 1–13, 2016.

[CFE+15] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603*, 2015.

[CGLPN15a] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Distributed qos-aware scheduling in storm. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 344–347. ACM, 2015.

[CGLPN15b] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. On qos-aware scheduling of data stream applications over fog computing infrastructures. In *Computers and Communication (ISCC), 2015 IEEE Symposium on*, pages 271–276. IEEE, 2015.

[CGLPN17a] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Joint operator replication and placement optimization for distributed streaming applications. In *proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools on 10th EAI International Conference on Performance Evaluation Methodologies and Tools*, pages 263–270. ICST (Institute for Computer Sciences, Social-Informatics and . . . , 2017.

[CGLPN17b] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator replication and placement for distributed stream processing systems. *ACM SIGMETRICS Performance Evaluation Review*, 44(4):11–22, 2017.

[Cis20] Cisco. Annual internet report (2018–2023) white paper, 2020. URL: `https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.htm`.

[CKE+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[CM18] Subarna Chatterjee and Christine Morin. Experimental study on the performance and resource utilization of data streaming frameworks. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 143–152. IEEE, 2018.

[ČTLČ16] Milan Čermák, Daniel Tovarňák, Martin Laštovička, and Pavel Čeleda. A performance benchmark for netflow data analysis on distributed stream processing systems. In *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*, pages 919–924. IEEE, 2016.

[dAdSVB18] Marcos Dias de Assuncao, Alexandre da Silva Veith, and Rajkumar Buyya. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *Journal of Network and Computer Applications*, 103:1–17, 2018.

[EHE16] Leila Eskandari, Zhiyi Huang, and David Eyers. P-scheduler: adaptive hierarchical scheduling in apache storm. In *Proceedings of the Australasian Computer Science Week Multiconference*, pages 1–10, 2016.

[EMHE18] Leila Eskandari, Jason Mair, Zhiyi Huang, and David Eyers. T3-scheduler: A topology and traffic aware two-level scheduler for stream processing systems in a heterogeneous cluster. *Future Generation Computer Systems*, 89:617–632, 2018.

[FAG+17] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment*, 10(12):1825–1836, 2017.

[FCKK20] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. A survey on the evolution of stream processing systems. *arXiv preprint arXiv:2008.00842*, 2020.

[FCP] Apache Flink Checkpointing. `https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/state/checkpointing.html`.

[FJS] Apache Flink Job Scheduling. `https://ci.apache.org/projects/flink/flink-docs-stable/internals/job_scheduling.html`.

[FLI] Apache Flink. `https://flink.apache.org/`.

[Gar] Gartner. Gartner says worldwide iaas public cloud services market grew 31.3in 2018. `https://www.gartner.com/en/newsroom/press-releases/2019-07-29-gartner-says-worldwide-iaas-public-cloud-services-market-grew-31point3-percent-in-2018`.

[GHS16] Philipp Götze, Wieland Hoffmann, and Kai-Uwe Sattler. Rewriting and code generation for dataflow programs. In *GvD*, pages 56–61, 2016.

[Goo20] Google. Personalized advertising policies, 2020. URL: `https://support.google.com/adspolicy/answer/143465?hl=en`.

[Gri16] Jamie Grier. Extending the yahoo! streaming benchmark. *URL http://data-artisans.com/extending-the-yahoo-streamingbenchmark*, 2016.

[HAD] Apache Hadoop. `https://hadoop.apache.org/`.

[HBB+18] Martin Hirzel, Guillaume Baudart, Angela Bonifati, Emanuele Della Valle, Sherif Sakr, and Akrivi Akrivi Vlachou. Stream processing languages in the big data era. *ACM SIGMOD Record*, 47(2):29–40, 2018.

[HDF] Apache Hadoop HDFS. `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`.

[HER] Apche Heron. `https://heron.incubator.apache.org/`.

[HFC+18] Li Han, Valentin Fèvre, Louis-Claude Canon, Yves Robert, and Frédéric Vivien. *A generic approach to scheduling and checkpointing workflows*. PhD thesis, Inria, 2018.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, 2010.

[HL15] Guenter Hesse and Martin Lorenz. Conceptual survey on data stream processing systems. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, pages 797–802. IEEE, 2015.

[HRM+17] Guenter Hesse, Benjamin Reissaus, Christoph Matthies, Martin Lorenz, Milena Kraus, and Matthias Uflacker. Senska–towards an enterprise streaming benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 25–40. Springer, 2017.

[Hub17] IBM Big Data Hub. Infographic. extracting business value from the 4 v's of big data, 2017. URL: `https://www.ibmbigdatahub.com/infographic/extracting-business-value-4-vs-big-data`.

[Hub20] IBM Big Data Hub. Infographic. the four v's of big data, 2020. URL: https://www.ibmbigdatahub.com/infographic/four-vs-big-data.

[HZK+15] Thomas Heinze, Mariam Zia, Robert Krahn, Zbigniew Jerzak, and Christof Fetzer. An adaptive replication scheme for data stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 150–161, 2015.

[IGN] Apche Ignite. https://ignite.apache.org/.

[INB] InfiniBand Trade Association. https://www.infinibandta.org/.

[JST] Alibaba Jstorm. https://github.com/alibaba/jstorm.

[KAF] Apache Kafka. https://kafka.apache.org/.

[KBF+15] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250, 2015.

[KIN] Amazon Kinesis. https://aws.amazon.com/kinesis/.

[KRK+18] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1507–1518. IEEE, 2018.

[KRSF17] Supun Kamburugamuve, Karthik Ramasamy, Martin Swany, and Geoffrey Fox. Low latency stream processing: Apache heron with infiniband & intel omni-path. In *Proceedings of the10th International Conference on Utility and Cloud Computing*, pages 101–110, 2017.

[Kru] Nico Kruber. A Deep-Dive into Flink's Network Stack. https://flink.apache.org/2019/06/05/flink-network-stack.html.

[KW17] Aljoscha Krettek and M Winters. the curious case of the broken benchmark: Revisiting apache flink® vs. databricks runtime. *URL: https://data-artisans. com/blog/curious-case-broken-benchmark-revisiting-apache-flinkvs-databricks-runtime*, 2017.

[LB17] Xunyun Liu and Rajkumar Buyya. D-storm: Dynamic resource-efficient scheduling of stream processing applications. In *Parallel and Distributed Systems (ICPADS), 2017 IEEE 23rd International Conference on*, pages 485–492. IEEE, 2017.

[LWJ+17] Hongliang Li, Jie Wu, Zhen Jiang, Xiang Li, Xiaohui Wei, and Yuan Zhuang. Integrated recovery and task allocation for stream processing. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2017.

[LWJ+18] Hong-Liang Li, Jie Wu, Zhen Jiang, Xiang Li, and Xiao-Hui Wei. A task allocation method for stream processing with recovery latency constraint. *Journal of Computer Science and Technology*, 33(6):1125–1139, 2018.

[LWXH14] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 69–78. IEEE, 2014.

[LXTW18] Teng Li, Zhiyuan Xu, Jian Tang, and Yanzhi Wang. Model-free control for distributed stream data processing using deep reinforcement learning. *Proceedings of the VLDB Endowment*, 11(6):705–718, 2018.

[MDB] MongoDB. `https://www.mongodb.com/`.

[MDT18] Gabriele Mencagli, Patrizio Dazzi, and Nicolò Tonci. Spin-streams: a static optimization tool for data stream processing applications. In *Proceedings of the 19th International Middleware Conference*, pages 66–79, 2018.

[MMDPER19] Vania Marangozova-Martin, Noël De Palma, and Ahmed El Rheddane. Multi-level elasticity for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2326–2337, 2019.

[MTL+18] Gabriele Mencagli, Massimo Torquati, Fabio Lucattini, Salvatore Cuomo, and Marco Aldinucci. Harnessing sliding-window execution semantics for parallel stream processing. *Journal of Parallel and Distributed Computing*, 116:74–88, 2018.

[N4J] Neo4J. `https://neo4j.com/`.

[NMGS+15] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David Garcia-Soriano, Nicolas Kourtellis, and Marco Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *2015 IEEE 31st International Conference on Data Engineering*, pages 137–148. IEEE, 2015.

[NPP+17] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. Samza: stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, 2008.

[PHH+15] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. R-storm: Resource-aware scheduling in storm. In *Proceedings of the 16th Annual Middleware Conference*, pages 149–161, 2015.

[RED] Redis. `https://redis.io/`.

[SAM] Apache Hadoop. `https://hadoop.apache.org/`.

[SCS17] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. Riotbench: An iot benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience*, 29(21):e4257, 2017.

[Sha17] Tom Shafer. The 42 v's of big data and data science, 2017. URL: `https://www.elderresearch.com/blog/42-v-of-big-data`.

[SPA] Apache Spark. `https://spark.apache.org/`.

[SST] Apache Spark Streaming. `https://spark.apache.org/streaming/`.

[Stoa] Storm. Acking framework implementation. URL: http://storm.apache.org/releases/current/Acking-framework-implementation.html.

[STOb] Apache Storm. https://storm.apache.org/.

[Stoc] Storm. Concepts. URL: http://storm.apache.org/releases/current/Concepts.html.

[Stod] Storm. Guaranteeing message processing. URL: http://storm.apache.org/releases/current/Guaranteeing-message-processing.html.

[STS] StreamSets. https://streamsets.com/.

[SZ16] Li Su and Yongluan Zhou. Tolerating correlated failures in massively parallel stream processing engines. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 517–528. IEEE, 2016.

[Tec18] Alibaba Tech. Alibaba blink: Real-time computing for big-time gains. https://medium.com/@alitech_2017/alibaba-blink-real-time-computing-for-big-time-gains-707fdd583c26, 2018.

[TGC+19] Jonas Traub, Philipp M Grulich, Alejandro Rodríguez Cuéllar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. Efficient window aggregation with general stream slicing. In *EDBT*, pages 97–108, 2019.

[TGNO92] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous queries over append-only databases. *Acm Sigmod Record*, 21(2):321–330, 1992.

[TMX] Java Thread MX Bean. https://docs.oracle.com/javase/8/docs/api/java/lang/management/ThreadMXBean.html.

[TTS+14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156, 2014.

[UI14] EMC Digital Universe and IDC. The digital universe of opportunities: Rich data and the increasing value of the internet of things, 2014. URL: `https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm`.

[WZL+14] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *2014 IEEE 20th international symposium on high performance computer architecture (HPCA)*, pages 488–499. IEEE, 2014.

[XCTS14] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. T-storm: Traffic-aware online scheduling in storm. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 535–544. IEEE, 2014.

[YAM] YAML. `https://yaml.org/`.

[Yav17] Burak Yavuz. Benchmarking structured streaming on databricks runtime against state-of-the-art streaming systems, 2017.

[ZKZ+15] Nikos Zacheilas, Vana Kalogeraki, Nikolas Zygouras, Nikolaos Panagiotou, and Dimitrios Gunopulos. Elastic complex event processing exploiting prediction. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 213–222. IEEE, 2015.

[ZLZL16] Jing Zhang, Chunlin Li, Liye Zhu, and Yanpei Liu. The real-time scheduling strategy based on traffic and load balancing in storm. In *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on*, pages 372–379. IEEE, 2016.

[ZWL+18] Yuan Zhuang, Xiaohui Wei, Hongliang Li, Yongfang Wang, and Xubin He. An optimal checkpointing model with online oci adjustment for stream processing applications. In *2018 27th International Conference on Computer Communication and Networks (ICCN)*, pages 1–9. IEEE, 2018.