TU WIEN Informatics

Latency-Aware Stream Operator Placement Optimisation in Fog Computing Environments

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Raphael Ecker, BSc
Matrikelnummer 01426241

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Dr.-Ing. Stefan Schulte
Mitwirkung: Vasileios Karagiannis, PhD

Wien, 28. November 2022

Raphael Ecker | Stefan Schulte

TU WIEN Informatics

Latency-Aware Stream Operator Placement Optimisation in Fog Computing Environments

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Raphael Ecker, BSc
Registration Number 01426241

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr.-Ing. Stefan Schulte
Assistance: Vasileios Karagiannis, PhD

Vienna, 28th November, 2022

Raphael Ecker	Stefan Schulte

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Raphael Ecker, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 28. November 2022

Raphael Ecker

Danksagung

Zunächst möchte ich meinen Mitstudenten für unsere Jahre der gemeinsamen Herausforderungen danken.

Ich möchte auch meinen Betreuern Professor Stefan Schulte und Vasileios Karagiannis für ihre kontinuierliche Unterstützung bei der Erstellung dieser Arbeit danken. Insbesondere ihre freundliche und entgegenkommende Bereitschaft zum Wissensaustausch und zur Durchführung von Reviews mit konstruktiver Kritik waren immens hilfreich.

Natürlich muss ich auch meiner Familie danken, die nicht nur diese Arbeit, sondern auch meine früheren Studien unterstützt und ertragen hat. Letztlich ist es ihre Leistung, die es mir ermöglicht hat, diesen Lebensweg überhaupt einzuschlagen.

Acknowledgements

To begin with, I want to thank my peers for our years of shared challenges.

I would also like to thank my advisors, professor Stefan Schulte and Vasileios Karagiannis, for their continued support during the writing of this thesis. In particular, their welcoming and forthcoming willingness to share knowledge and perform reviews with constructive criticisms have been immensely helpful.

Of course, I also have to thank my family, which has supported and endured not only this thesis but also my previous studies. Ultimately, it is their accomplishments that have enabled me to even take this path in life.

Kurzfassung

Das Internet der Dinge (Internet of Things, IoT) wird immer beliebter und damit steigt auch die Anzahl der beteiligten Geräte und der zu verarbeitenden Daten schnell an. Die verteilte Datenstromverarbeitung ist ein Software-Engineering-Muster zum Bau von Anwendungen als Topologien von Operatoren und wird häufig zur Verarbeitung von IoT-Daten verwendet. Fog Computing ist ein neues Paradigma zur Bewältigung dieser Datenmengen und soll niedrigere Latenzzeiten bieten, indem die Berechnungen auf Fog-Ressourcen, die näher an den IoT-Geräten oder Nutzern sind, durchgeführt werden. Das erfordert Verbesserungen bei der Verteilung von Operatoren auf Rechenressourcen. Von Fog-Ressourcen wird erwartet, dass sie im Vergleich zum Cloud Computing über vielfältigere Eigenschaften und Fähigkeiten verfügen und geographisch verteilt sein.

Diese Arbeit zielt darauf ab, die Platzierung von Operatoren in Fog- und Cloud-Computing-Umgebungen zu verbessern. Ein eingeschränktes Optimierungsproblem wird definiert und heuristisch mit drei Ansätzen gelöst: Dem Bergsteigeralgorithmus, einem Ameisensystem und einer Hybridlösung. Die Platzierungen werden dabei auch regelmässig zur Laufzeit erneut optimiert. Die Heuristiken sind in Apache Storm integriert und werden in einem emulierten Netzwerk mit zufällig generierten Topologien getestet. Sie werden mit zwei statischen Lösungen verglichen: Apache Storms Standard-Planer und R-Storm.

Die beste Heuristik übertraf den Standard-Planer wesentlich im Durchsatz und den Latenzzeiten. Im Vergleich mit R-Storm konnte durch die zuverlässigere Platzierung der durchschnittliche Durchsatz um 11% erhöht werden und die minimale Latenz um 36% reduziert werden. Außerdem hat die Lösung den Vorteil, dass sie dynamisch Anpassungen der Platzierung an unterschiedliche Lasten und Veränderungen in der Umgebung durchführen kann.

Abstract

The Internet of Things (IoT) is growing in popularity, and as such, the number of devices involved and the amount of data to be processed are rising quickly. Distributed stream processing is a software engineering pattern to build applications as topologies of operators and is often used to process IoT data. Fog computing is a new paradigm intended to handle the increasing data loads and offer lower latencies by moving computations onto fog resources closer to IoT devices or users. This necessitates improvements to the distribution of operators to computational resources. Fog resources are expected to have more varied capabilities when compared to cloud computing and to be geographically distributed.

This thesis aims to improve the placement of operators in fog and cloud computing environments. A constrained optimisation problem is defined and heuristically solved with three approaches: Hill-climbing, an ant system and a hybrid solution. The placements are also periodically re-optimised at runtime. The heuristics are integrated into Apache Storm and benchmarked in an emulated network with randomly generated topologies. They are compared against two static solutions: Apache Storm's default scheduler and the Resource Aware Scheduler.

The best heuristic outperformed the default scheduler significantly in throughput and latency. In comparison to the Resource Aware Scheduler, average throughput was increased by 11% because of the more reliable placement performance and the minimum latency was reduced by 36%. Additionally, the presented solution has the advantage of dynamically adjusting placements to different loads and changes in the environment.

Contents

CHAPTER 1

Introduction

1.1 Motivation

In computing, there has always been a drive to perform calculations quicker or more efficiently. As such, many applications across industries rely on near real-time computations. These can be split into two types: Applications, which accomplish their aims better by shortening the delay incurred by computations, resulting in benefits such as financial gains or better usability. In many scenarios, such as monitoring or trading on the stock market, any data and the potential to act on it is worth the most when it is first collected and loses in value as it ages [CRC16]. The second type of applications requires near real-time results to function correctly. This could be factory automations or autonomous driving with safety concerns or fraud detection systems for financial transfers and payments [CSI+20, VGT14, ATB+19]. Fog computing and stream processing are technologies that have been established with the aim of supporting near real-time computations [IEE18, dAVB18].

Fog computing is a computing paradigm of an infrastructure of computational resources available for rent, which, among other benefits, explicitly aims to support real-time computations. It is the logical continuation of the widely successful cloud computing with the aim of solving some shortcomings that have been found. Fog computing aims to provide geographically distributed computational resources in comparison to the centralised data centres of cloud computing. This geographic distribution allows it to provide computational resources, which are located closer to data sources or users. The reduced distances thereby also reduce the network latency involved when transferring data, providing new potential for real-time computations [IEE18].

Stream processing is a software engineering pattern for performing highly scalable computations with low latency. In stream processing, calculations are made by performing transformations on input data. A stream processing application is built out of a network

of such transforming operations, which can all be executed concurrently. Any data to be processed is then simply streamed through this network of operations, with a stream of computational results exiting it. As such, this pattern allows for software that is highly scalable due to the inherent concurrency and provides near real-time computations, because any input has only to be routed through and transformed by the network. Furthermore, the operations can be distributed by executing them on separate computational resources [dAVB18].

This process creates the obvious question of how the operators should best be distributed on available computational resources: the stream operator placement problem [VS20]. To utilise the full potential fog computing and stream processing provide in supporting near real-time computations, this point where both technologies intersect must similarly be considered. As such, it is necessary to optimise the placement of stream operators with the same goal in mind. With the geographical distribution of fog computing, the distance and, therefore also, network latency between computational resources gets larger. For this reason, the impact of network latency is more significant when an application is distributed among resources, in comparison to centralised cloud data-centres. Therefore, this thesis aims to develop a latency-aware stream operator placement optimisation for fog computing environments, to better support the applications desired by the industry with near real-time computational needs.

1.2 Aims of the Thesis

The consideration of the network and latency is not a new concept for the placement of stream processing operators, but one which is gaining interest because of fog computing. The primary aim is to create a new heuristic to solve the stream operator placement problem in fog computing environments. The following problems and research questions are answered or solved and represent the results of the thesis:

Requirements Analysis

To select a stream processing framework for the implementation of a new placement heuristic as well as the design of it, is necessary to understand the requirements of this task and the fog computing environment. This can be formulated as the following research question: “What technical requirements are necessary to be fulfilled for the placement of operators in a fog computing environment?”

Design

Before implementing a placement heuristic, the design has to be defined first. The design has to consider both the previously specified requirements and how to achieve the best results in a fog computing environment. The research questions to be answered with this result are the following: “What design choices can be made or are suited for fog computing environments? How should operators be placed on computational resources?”

Implementation

The primary aim and artefact created during the work on this thesis is the implementation of the placement heuristic. The implementation follows the design and has to fulfil the specified requirements. This step answers these research questions: "What stream processing framework is best suited for the implementation? How can the designed heuristic be implemented and what is its architecture?"

Evaluation and Comparison of Implementation

To ascertain the quality of the proposed placement heuristic, it has to be evaluated in detail. For this, a quantitative evaluation is performed by deploying the placement heuristic and benchmarking it with representative workloads. The research questions to be answered by this approach are the following: "What is the quality of the found operator placements? How scalable is the heuristic with an increase in the problem size? How efficient is the heuristic in finding placements? How does this approach compare to previous works?"

1.3 Methodology

The methodology to answer the research questions and achieve the aims of this thesis, while ensuring the validity of results consists of the following steps:

Literature Survey

The literature survey aims first to establish a baseline understanding and collection of definitions for various concepts which form the necessary background for this thesis. As such, stream processing, the Internet of Things (IoT), as well as cloud and fog computing are researched and explained. In particular, the relations and interactions of these paradigms as well as differences in definitions or common understanding are of interest.

Following this, the optimisation of stream processing applications and the stream operator placement problem are discussed. This includes the varying definitions used for the stream operator placement problem. Similarly, differences in placement algorithms and heuristics are summarised and broadly categorised into characteristics before concrete implementations are introduced. For these implementations, there is both the aim to showcase a wide spectrum of different ideas as well as focusing on the ones most relevant to fog computing and network awareness. To finish the survey, related placement problems in other areas of research are presented and compared to the stream operator placement problem.

Design

For the design of the placement heuristic, a requirements analysis is performed. Similarly, the intended use case and specific assumptions about the fog environment and

resulting limitations are clarified. The design includes the solution architecture, such as a component to measure network metrics and the placement heuristic, as well as their general interaction. Furthermore, the constraints and optimisation function used by the placement heuristic are formalised and details of the strategy are defined. The design, in particular, aims to improve on the state of the art of network-aware placement techniques.

Implementation

Based on the design, a Java implementation of the placement heuristic is integrated into the popular stream processing framework Apache Storm [TTS+14]. The selection of this framework and a broad overview of alternatives and their strengths, weaknesses and practical relevancy to this thesis are presented. The placement heuristic, in combination with components to measure network and operator metrics, allows the optimisation of the placement of stream processing operators without requiring changes to internal interfaces to ensure the general compatibility of the approach.

Evaluation

The proposed implementation is benchmarked in an emulated fog computing network with various workloads and compared against existing solutions. The resulting placements of operators are analysed using metrics such as throughput, resource consumption and latency. Individual experiments are repeated to allow for the aggregation of the results and to understand the consistency with which they can be achieved.

1.4 Thesis Structure

This thesis is structured as follows. Chapter 2 introduces the concepts of cloud computing, fog computing and stream processing with the Internet of Things (IoT) as a use case driving these developments. Additionally, the background on optimising stream processing applications is included in the chapter. The definition of the stream operator placement problem and a survey of algorithms and heuristics to solve it are provided in Chapter 3. Chapter 4 contains the requirements analysis, fog environment definition, framework selection and design of the heuristic used to create the novel implementation of this thesis. Chapter 5 presents the details of the implementation itself. In Chapter 6, the test procedure and environment are specified and the measured performance in addition to further analysis are imparted. Chapter 7 contributes a summary of the achieved results and a discussion on alternative approaches or open and follow-up research questions.

CHAPTER 2

Background

To discuss the stream operator placement problem, it is necessary to establish various key concepts and terminology first. Additionally, this chapter provides insights into the context stream processing applications operate in. As such, it also presents a first look at the conditions that form the requirements and constraints which become more relevant and formalised in later chapters. To accomplish this, Section 2.1 provides an introduction to cloud computing as a scalable infrastructure for the execution of applications. Section 2.2 presents an overview of stream processing as a paradigm for structuring computations and applications out of independently executed operations, thereby allowing for high levels of concurrency. The IoT is introduced in Section 2.3 as a relevant use case and industrial development at which stream processing applications excel at, and as a major motivation for the shift from cloud to fog computing. Section 2.4 explores fog computing as a new paradigm based on the principles of cloud computing but aimed at solving the limitations the IoT has exposed. A summary of techniques to optimise the logical and physical plan of a stream processing application is presented in Section 2.5. Some of these optimisations are dependent on the placement of operators and as such directly provide an introduction to the importance of the stream operator placement problem, which is the focus of the following chapter.

2.1 Cloud Computing

2.1.1 Overview

Cloud computing is a popular networking paradigm that is often utilised as the infrastructure to execute stream processing applications [dAVB18]. A cloud provides access to ubiquitous on-demand computing resources to users. These resources allow reducing initial investments in infrastructure and offer seemingly limitless scalability for applications [MG11]. The success of cloud computing has affected engineering practices with its focus on scalability and allowed other paradigms, such as big data, to prosper [HYA+15].

2.1.2 Definition

Cloud computing, at its core, is a powerful rentable infrastructure which relies on centralised data centres [RZH+20]. Computational resources are virtualised by using virtual machines or containers to deploy applications within them. They provide security by isolating applications and allow the efficient usage of resources by supporting co-location of applications [HMA19]. Resource utilisation is therefore improved, while also providing customers with the ability to adapt their application's resources to their demand quickly [HYA+15].

Cloud computing resources are often billed with pay-as-you-go or longer-term subscription pricing schemes. Storage and bandwidth are commonly charged with unit prices depending on usage. This model of only paying as much as necessary without large setup costs is one of the major business advantages cloud computing offers. In addition, there are also dynamic pricing models based on the demand and capacity of the cloud. These allow cloud computing to increase its efficiency by offering to sell unused capacity at reduced prices [AUK+15]. As such, this model is most useful for applications or tasks which do not have to run continuously and are therefore fitting for workloads of computationally intensive processes on existing datasets. The following content provides more insight into the types of service cloud computing can provide to users and bill them for.

2.1.3 Service and Deployment Models

Cloud computing provides three commonly recognised service models using its virtualised computational resources. These are Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS), which differ in the level of abstraction and therefore, also, control offered. As such, everything under one's control must also be maintained and affects the costs and necessary skill set to use the service [MG11]. Another less commonly recognised service model is Containers as a Service (CaaS), which has gained popularity in recent years [HMA19]. The following text provides an introduction to these models and afterwards also discusses their deployment. In Figure 2.1, a comparison of the service models is also visualised.

Software as a Service

A complete application is hosted in the cloud and offered to customers. It can be directly accessed with a user interface or by programs using predefined Application Programming Interfaces (APIs). Updates and other maintenance tasks of the application are not performed by the customer. Similarly, the service provider handles the management of data, the operating system, hardware or other underlying concerns [MG11].

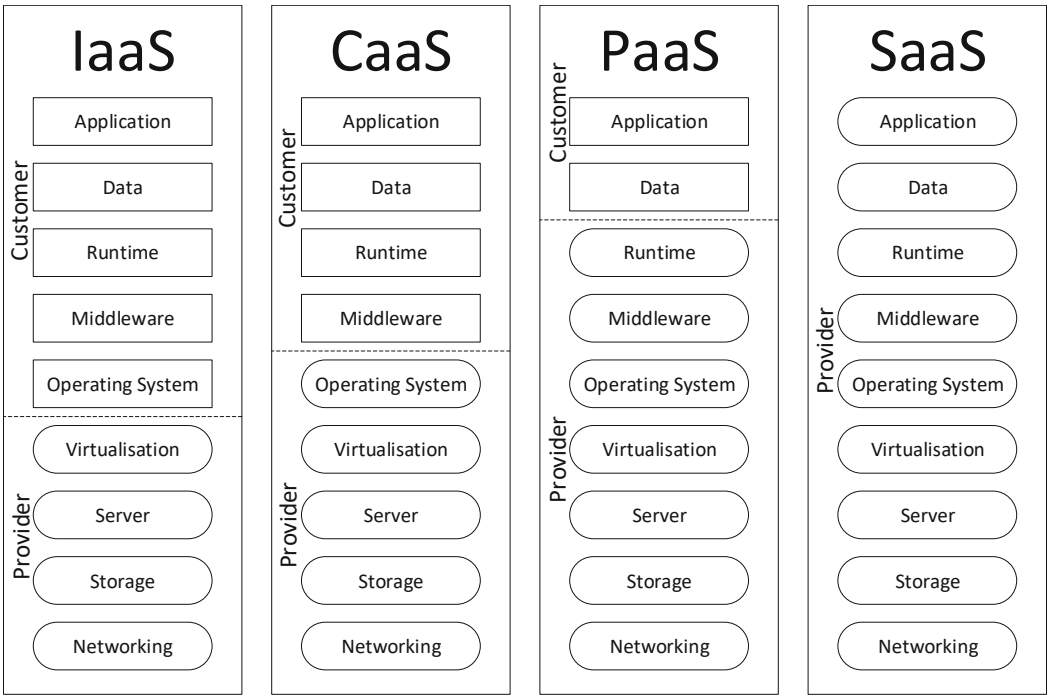


Figure 2.1: Comparison of common service models showcasing the different responsibility and ownership between the service provider and customer over components [YFN+19].

Platform as a Service

In this model, a platform in the cloud to execute customers' applications is offered. Customers are concerned with the development of the application, its data and configuration. Tools and services provided by the platform for the execution of the application, as well as, other lower-level concerns or infrastructure are managed by the service provider [MG11].

Containers as a Service

Containers are a more lightweight virtualisation technique in which applications can be packaged with configuration details and their software dependencies, previously referred to as a platform. They provide an isolated virtual operating system. The operating system kernel is not a part of a container and can therefore be shared between containers. This allows containers to be more efficient than traditional virtual machines as only a single operating system has to be launched instead of a new one in each virtual machine [HMA19].

As such, this technique gives customers more control than PaaS by allowing them to provide their own software stack for an application. The management of the operating system, hardware and networking are the responsibility of the service provider.

Infrastructure as a Service

In IaaS virtualised resources themselves are offered. These typically range from computing and storage to networks. Examples are virtual machines or load balancers. Customers can change configurations and manage their own operating systems or software stacks for computing resources. Only the provision of these resources and the underlying hardware such as the network are managed by the service provider [MG11].

Deployment Models

In addition to the many types of services a cloud may provide, there are also variations in its management. A cloud can be deployed to be accessible by different user groups. These groups are then the customers or consumers of a cloud by typically renting virtualised computational resources. This categorisation is referred to as deployment models and consists of private, community, public or hybrid clouds [MG11].

Private clouds only provide their services to a single organisation, while community clouds are shared with and potentially operated by multiple organisations. Public clouds are accessible by the general public and do not have further limitations such as membership in an organisation to utilise the service. Hybrid clouds are a configuration of multiple of the previously defined types. An example could be a private cloud that is generally used, but in need of additional capacity. The cloud and its applications could burst into a public cloud and rent additional computational resources, thereby creating a hybrid deployment. The ubiquitous amount of computational resources provided by the cloud have made it into a great infrastructure for big data analysis, such as for the data collected by devices of the IoT which will be introduced in Section 2.3 [MG11]. While the cloud provides ubiquitous computational resources for such purposes, these resources still have to be utilised by applications, which can scale efficiently. The following section introduces stream processing as a means to design such applications.

2.2 Stream Processing

The following section presents an overview of stream processing and introduces an example of the application of this software engineering pattern for a taxi service. The definition of the general properties and the behaviour of stream processing applications is provided in Section 2.2.2 to form a deeper understanding before the design of stream processing applications is discussed. It introduces the logical plan of stream processing applications, which contains their general structure, such as their operations and connecting data streams as it is displayed for the example in Figure 2.2. Section 2.2.3 then discusses the realisation of the logical plan when executing an application and the necessary components, such as a stream processing engine. Section 2.2.4 provides an overview of general properties a stream processing engine or its applications can have, such as determinism or fault tolerance.

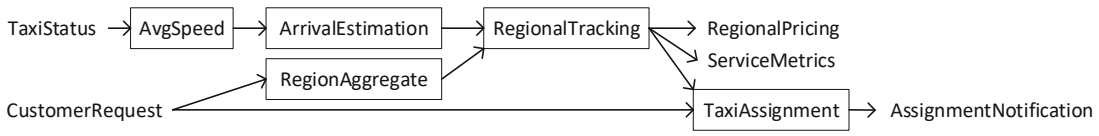


Figure 2.2: Example of a theoretical stream processing application to support a taxi service. Arrows represent the flow of events and boxes are the operations of the application. Inputs and outputs are represented by labelled arrows.

2.2.1 Overview

Stream processing is a highly scalable software engineering pattern to process large amounts of data in close to real-time [dAVB18]. This is accomplished by splitting the computation to be performed into independent operations, which can be executed concurrently. Stream processing is used to quickly process events or messages, such as sensor readings or user interactions. Data to be processed is routed through a directed acyclic graph (DAG) consisting of operations and message queues connecting them. These message queues or data streams at the core of this paradigm have coined its name. The operations transform the data while it passes through them and the complete stream processing application can thereby execute complex functionality [dAVB18].

To provide an overview of the concept, Figure 2.2 shows an introductory example of the general structure of a stream processing application, which could be used by a taxi service. The example is primarily inspired by DSPBench's traffic monitoring application [BGM+20] and Kaggle's New York City Taxi Trip Duration challenge [Kag17], but also from the availability of even more available taxi-related datasets, such as the T-Drive trajectory data [YZXS11, YZZ+10]. The figure displays the flow of events through the application, signified by the arrows, and its operations, which transform the events to calculate the outputs. In this example, the application receives periodic updates from any taxi with information, such as their current position. This information can be used to calculate a taxi's speed and expected arrival times. Additionally, the application also receives driving requests from customers and assigns them to a taxi. These requests can be aggregated to estimate the demand of regions and combined with the knowledge of available taxis and the expected arrival times of assigned ones the supply of taxis can be calculated. The application can therefore update the price in regions based on the demand and available supply, which drivers can use to relocate into more demanded and thereby profitable regions. The use of stream processing allows these calculations to be performed in close to real-time while providing the scalability to handle the number of taxis or customers.

2.2.2 Definition

Many stream processing engines are based on the data flow paradigm, which inherently provides scalability [dAVB18]. The data flow paradigm proposes building applications as a set of smaller operations, which are all assumed to be executed concurrently. As such, the application can receive input data from other applications and transfers them to an

initial set of operations to perform the calculation. The operations receive the data and transform it before sending it to other operations. These operations are connected in a graph-based topology to describe the flow of data and therefore the processing of the application. Complex outputs can be calculated by applying successive transformations on input data with each of the operations. Inputs to the application are assumed to continuously receive new data events to process and as such the application also continuously calculates new outputs [DM74]. The outputs can then be displayed or transferred to other applications [VdAL18].

Thus, stream processing applications are commonly represented by DAGs. An example of such a graph representing a stream processing application can be seen in Figure 2.2. Within this graph, events which can also be referred to as data, messages, tuples or packets, are transferred along the directed edges between the nodes. The edges are therefore called data streams. Data streams are considered to be unbounded sequences of events, which are continuously received to be processed. In addition to the data streams appearing as edges, there are three major types of components in stream processing applications represented by the nodes in the graph [dAVB18].

Nodes in the graph with only outgoing edges are called data sources and form the inputs for stream processing applications. They receive emitted events as inputs for the application, for example, sensor readings or integrations of streams from other applications. These events are then processed on the fly by streaming them through the graph [VdAL18, MCA+17]. In Figure 2.2, both the *TaxiStatus* and the *CustomerRequest* are data sources.

The majority of the nodes in the graph are typically operations, which both receive events and output new events. They receive events via edges leading into the node, transform them or perform other calculations with them and then output results via the outgoing edges to the following nodes. It is important to note that an operation can have multiple input and output streams. In this case, each of these streams can contain different data types and information. For example, an operation might sort numbers into a stream of even and odd numbers, but more complex functionalities such as detecting spam emails or monitoring for different alerts can also be easily envisioned. Similarly, additional data streams can be used to output warnings or statistic information about the streamed data, which can then be processed in other parts of the applications. Stream processing engines provide various predefined operations such as counting, filtering, projections or aggregation of data, but modern ones also allow implementing user-defined operations. Additionally, stream processing engines handle the execution and various other properties of stream processing applications, which is discussed in the following sections [dAVB18].

Nodes, which have only incoming edges are called data sinks. These represent the outputs of the complete stream processing applications. Examples could be data sinks which store events in databases, transfer them to other applications for further processing or using and displaying the outputs [VdAL18]. In the previous example of Figure 2.2, the *RegionalPricing* information would be used by a customer-facing application, while the *ServiceMetrics* would be used for internal monitoring, such as a dashboard, or simply

stored for record keeping purposes. The *AssignmentNotification* would be routed to both the customers and the drivers via the responsible services or applications.

2.2.3 Execution of Stream Processing Applications

The DAG merely represents the logical plan of a stream processing application and describes their semantics. It defines what is being calculated and how these calculations are performed. The physical plan defines where and how the logical plan is executed to realise the application. To run the application, it is necessary to map the data sources, data sinks and operations to computational resources, which execute them. The data sources, data sinks and operations are then referred to as operators and can have a state associated with them storing necessary information for their execution. Operations can thus also be classified into stateful or stateless [dAVB18]. Details on how such a mapping can be created or in other words how operators are placed and what effect different possibilities have on the application are discussed in Chapter 3. The previous example of an operation sorting numbers into even and odd number streams is trivial to implement as a stateless operation. More complex functionalities, such as calculating the mean of a number stream, require memory, in this case, the count of the numbers received and their current sum, and is therefore a stateful operation. When the execution on a computational resource fails, such as due to a software crash or failing hardware, the operator can be migrated to other resources, to be executed there, by transferring the associated state [dAVB18].

Additionally, multiple operators can be deployed for a single operation in the logical plan. The replication of operators allows the application to scale to the current processing load. This requires splitting the associated state into independent sets to allow for concurrent computations. Accomplishing this can be difficult and largely depends on the data and use case of the application, because it must be ensured that the semantics or constraints of the operation or larger application are not violated. This concern is known as data parallelism and is a regular problem in the domain of parallel computing. Furthermore, because stream processing applications can scale with their load by spawning new operators or stopping existing ones, splitting, merging or synchronising the state of operators must also be possible [HVSD16].

While operators are concerned with the execution of an operation and can be considered as a unit of abstraction containing their state, the architecture of a stream processing application also contains the stream processing engine. This is because operators are not directly placed on computational resources, but are instead managed by the stream processing engine itself [MCA+17]. In this thesis, the processes or threads of the stream processing engine executing operators are referred to as workers. This term is similarly used for established patterns in parallel computing or distributed systems and the terminology is also common in existing stream processing engines such as Apache Flink [dAVB18]. Workers can execute multiple operators and represent the processing engine itself. Workers manage the transfer of data, the data streams, to other workers or potentially between local operators. Additionally, they handle functionalities such as

fault tolerance or the state transfer in the case of relocating operators or spawning new ones [MCA+17].

2.2.4 Properties of Stream Processing Applications and Engines

This subsection introduces further properties which stream processing applications or engines can exhibit. They can therefore vary between engines or can sometimes even be configured by the application. As such, this section aims to illustrate more of the complexities, research aims and differences between stream processing engines to provide a more complete overview of the technology.

While a stream processing application can be executed on a single computational resource, this thesis only considers stream processing that can be distributed. Distributed stream processing refers to the ability to distribute the work over multiple computational resources connected by a network. These can be deployed in proximity or distributed geographically over significant distances. Elasticity is another beneficial property distributed stream processing frameworks can exhibit and defines that the amount of used computational resources can increase or decrease. Elasticity, therefore, allows applications to adapt to changing computational demands of an application or a change in available resources [HVSD16].

In a distributed system defining the consistency in calculating outputs is important, especially considering the need for fault recovery and tolerance strategies to achieve high availability. These strategies are difficult problems, because the recovery includes received events, the states of operators and sent output events. Consistency is commonly defined by assurances on the delivery and therefore the processing of events in a data stream. Options in modern frameworks include at-least-once, at-most-once and the more expensive exactly-once [TKMN19].

Determinism or the lack of it is another property to differentiate stream processing systems in this domain. Determinism defines the ability to reproduce the same output if the same input is provided again. This property aids recovery strategies and is desirable for developers, because it assures the predictable behaviour of a system, but is also difficult to achieve. Operators process data independently and concurrently, and as such timing differences can cause different orderings of events in different runs causing different computational outputs [TKMN19].

To achieve properties such as fault tolerance, consistency and determinism, stream processing systems are typically built using the following three approaches. Events are collected into micro-batches to reuse strategies from traditional batch processing, distributed transaction protocols are used or snapshots of the state are created with each operation. In batch processing, large sets of data, called batches, are processed at once efficiently when they are available. With batch processing, fault tolerance, consistency and determinism are easier to achieve than in stream processing where inputs and outputs are continuously received and sent. Micro-batches, therefore, aim to combine

the advantages of both techniques by offering a solution in between both extremes of performing computations on individual events or only large sets of data [TKMN19].

Outputs in stream processing are calculated in one-pass and are available with a short response time [dAVB18]. The processing of an event only once immediately, instead of storing it and reusing it in future calculations such as datasets in batch processing, is also known as temporal awareness of stream processing applications. As such, stream processing applications can only consider recent events limiting potential use cases, but proving to be very effective in this speciality [PB19].

While stream processing is a highly scalable software development paradigm, the workload can still exceed the capabilities of a system. In such a case, the utilisation of computational resources can still be optimised to ensure the stability and throughput of an application. Back-pressure is a mechanism that can be implemented to reduce the rate of events to process, when the system or its individual components can not keep up with the current rate of received events. Back-pressure can be implemented in multiple ways, such as disk buffers to store events and later once the overload has diminished to replay the events. Another method is to reduce the number of events transmitted to the bottlenecked system or component. This can be done until the buffers of the previous component are full and the rate of events it receives has to similarly be constrained. As such, back-pressure can propagate within the application from the bottleneck back towards the data sources [MCA+17].

To increase the concurrency in stream processing applications further than the number of operations, it is necessary to parallelise their implementation or allow for the partitioning of streams into states that can independently be calculated. For example, in a company some computations might be calculated independently per department and thereby allow the creation of an independent operator for each department [MCA+17]. Windowing is another common technique in partitioning streams, but based on time. A window is a bounded set of events in a data stream containing the most recent or still relevant tuples. Depending on the type of window, an event can also be included in multiple windows. Windowing is used for aggregate computations such as calculating the average value of recent events [LMT+05].

The high level of concurrency inherent in stream processing applications, especially when combined with techniques such as windowing, allows for their excellent scalability. One area benefiting from the high scalability and capability to adapt to the current resource demand is the IoT. The large amount of data it produces and the need to quickly process it are strengths of stream processing [PB19].

2.3 Internet of Things

2.3.1 Overview

The IoT aims to empower devices by integrating the ability to communicate and collaborate. In fact, traditional non-communicating devices are often referred to using terms

such as dull devices in opposition to smart or IoT devices [JRNR20]. Systems formed by IoT devices allow for new functionalities and are leading to innovation in various domains, such as smart manufacturing, smart cities or healthcare [IEE20]. This section provides a general introduction into the IoT and its characteristics as well as some domains it can beneficently be used in.

The IoT consists of devices which can sense or interact with the world. These devices can then be connected to form IoT systems which allow for new functionalities or raising efficiencies. These systems use their collective powers to achieve common goals. Furthermore, the IoT systems can interact with each other or an IoT device can be a part of multiple systems [IEE20]. One of the large research aims is improving interoperability. It should be possible to integrate new devices to create added value in IoT systems rather than deploying new interconnected systems. IoT systems are considered to require some form of intelligence, meaning analysis or processing of the data, instead of just connecting them [HPS+15]. The cloud is commonly used to provide these services, because of its cost efficiency and scalability [WKB+19]. In many deployment models, this access is even necessary [IEE18].

2.3.2 Definition

The modern understanding of the IoT was created out of the convergence of three different visions: the things, the Internet and the semantic vision [SZ20]. These visions highlight different perspectives and therefore research directions of the IoT:

- The things vision consists of researching smart, identifiable objects and enabling integration between them with physical communication technologies [SZ20].
- The Internet vision focuses on the use and development of Internet protocols to integrate devices [SZ20].
- The semantic vision aims to improve the understanding of the meaning, representation and integration of data from different sources [SZ20].

As a result, IoT devices have the following three key properties [JRNR20]:

1. Ability to connect with other IoT devices
2. Identifiability
3. Capability to sense or interact with the environment

Additionally, IoT applications are often assumed to be context-aware to selectively provide fitting information or services [SZ20]. This context can depend on meta-information or physical properties such as the time or geographical location and may even be required

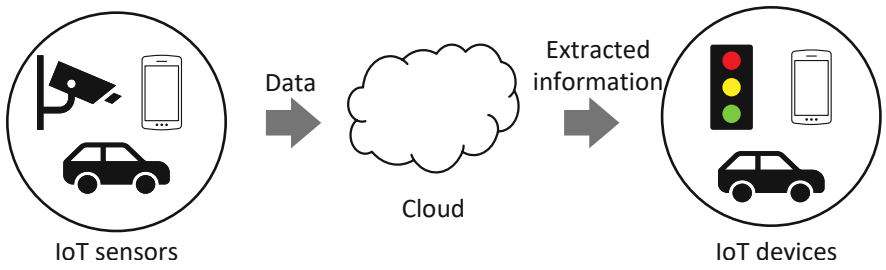


Figure 2.3: Example of different types of IoT devices cooperating to form an intelligent traffic management system to improve collective traffic flows.

to successfully integrate multiple information sources [IEE20]. IoT devices exist in large amounts and many utilise wireless communications [IGF+18].

IoT systems consist of sensors or IoT devices, which collect information, and IoT devices or actuators to interact with the environment [SZ20]. The collected information is processed or analysed on computational resources, such as the cloud, to intelligently control the environment [HPS+15]. Figure 2.3 shows a hypothetical traffic management system formed by various IoT devices. Devices such as cars or smartphones can transmit their current positions or surveillance cameras can recognise them. These can then be analysed in the cloud to identify aggregate movement patterns or problem areas such as traffic jams. Based on this information, the signalling of traffic lights or the route finding of cars and smartphones can be adapted. As a result, the collective information can be used to improve traffic flows to reduce waiting and travelling times for the population. This use case also highlights the scale of the data, as it is easy to imagine such a system spanning across a city and the need to quickly analyse the data from streets and intersections to reduce traffic in congested areas as early as possible [DSX+19].

2.3.3 Use Case Examples

The IoT has a versatile set of potential use cases. The following paragraphs provide an overview of applications and research domains. These are applications intended to benefit from this thesis or related works. Therefore, they can be used as a basis to consider the design decisions in this thesis, such as the establishment of requirements or constraints.

In Industry 4.0, also known as smart manufacturing and various other terms, the major aims are the automatic planning and optimisation of production and logistic processes [IEE20]. The context awareness of IoT systems allows for better monitoring and planning as well as to improve the flexibility of the system, such as allowing for customised products. Other aims are reducing defects and proactive maintenance [PZ17]. Smart farming has similar goals with tasks such as monitoring plants and animals or their environments, such as the soil, climate or greenhouses. Smart farming provides utilities such as detecting crop infections or diseases, assisting in raising productivity and improving efficiency, such as by reducing water consumption [FRA+19].

Smart cities are the idea of applying IoT to urban areas to improve large scale infrastructure. The monitoring can consist of dedicated sensors, cameras and vehicles or personal smartphones and other wearables providing information. This allows for the widespread collection of data points on interests such as pollution levels or traffic conditions. The data enables techniques such as smart healthcare or the previously introduced traffic management [DSX+19, AA20]. Smart cars and infrastructure for them or vehicle-to-vehicle connections to exchange data are also related [JRNR20]. Smart grids consist of an information, communication and energy infrastructure. They include ideas such as forming micro-grids, in which other local resources, such as generators, storage and users, can cooperate to form an electric grid even if the overall network fails. As such, they promote more distributed power generation, can reduce necessary peak or backup generation needs by smartly utilising batteries, and can generally optimise the network for efficiency and reliability among other advantages [FMXY12].

Even today there are already various possibilities with IoT systems in existing infrastructure. Airports can have several thousands of cameras which continuously create large amounts of data that need to be stored, but also could be further analysed. Several use cases are theorised or already exist to improve security, such as verifying licence plates of cars, utilising facial recognition, tracking baggage or as general alert systems by attempting to detect anomalies using machine learning systems. In addition, resources could be used to compress video streams allowing for cheaper cameras by reducing their required capabilities. Less capable devices also ensure that verifying their security is simpler. Compression is not only relevant to reduce storage costs, but also to reduce the transport cost in current cloud computing-based models. The cost of transferring all this data does not allow to support cameras with high resolutions, such as 1080p or 4k. Cloud-based installations are therefore considered to not scale sufficiently to support more modern standards [IEE18].

With all these devices collecting data, managing the scale becomes difficult. Even the widely used model of storing the data in a relational database to process later is sometimes considered obsolete in IoT systems [AA12]. Therefore, the following discussion considers some techniques for managing data at this scale.

2.3.4 Data Management

Sensors used for these applications continuously produce data and have to be processed. Video streams are a common example with their problematic high bandwidth requirements [IEE18]. These sensors typically do not stop collecting data. Hence, timely processing is necessary as otherwise an overwhelmed system can not keep up and the delay increases, until results from a data point are attained [PB19]. The cloud is commonly used to provide the necessary computational resources cost-efficient at scale [WKB+19]. Managing all this data is a research problem related to the IoT in itself. Various strategies are being researched to limit the scale to be able to handle the data. This ranges from efficient sampling techniques to early filtering or aggregation of values [AA12, HPS+15].

The persistent storage of all data is often considered not useful or even impossible. Thus, one strategy is to store data only for a limited time window while it is recent and relevant [PB19]. These constraints highlight the relevancy of this thesis and stream processing with its fitting features for the IoT by immediately processing data in a scalable way, so that even during peak loads the storage of events can be avoided. For traditional batch processing, dependent on storing and querying data, an ageing mechanism can be used to remove less useful data points. Such ageing mechanisms attempt to intelligently select the least valuable or least used data and delete it [AA12].

While all these strategies allow reducing transmitted or stored data, its scale is still difficult to manage. The widespread usage of the cloud for analysing IoT data is becoming a bottleneck and constrains potential use cases. In response, the idea of a new cloud-like infrastructure designed to solve related drawbacks, known as fog computing, is taking shape [IGF+18].

2.4 Fog Computing

2.4.1 Overview

Fog computing is a new paradigm adapting cloud computing to extend to the edge of the network, bringing computations closer to data sources and end devices [AVA19]. This section introduces fog computing based on the conceptual model defined by the National Institute of Standards and Technology (NIST) [IGF+18]. It is important to note that the discussion of the fog architecture has not yet reached a conclusion on a standardised definition [AVA19]. While multiple standards have been created, such as the NIST model and the recent IEEE adoption of the reference architecture of the OpenFog Consortium, they are not yet commonly agreed upon [IGF+18, IEE18]. This section provides an overview of the differences in fog computing definitions and their usage in research. The main definition is based on the NIST model, which has been selected because of its broader definition capturing the core principles of fog computing. The model also presents a more recent view on fog computing closely matching the original definitions [IGF+18, BMZA12].

Fog computing allows to process data closer to its sources, the smart or IoT device it originates from. As such, it is a more decentralised approach than the usage of centralised cloud-based data centres [IGF+18]. This is intended to be achieved by using comparably lower performance devices, such as Raspberry Pis, or smaller clusters of servers as a basis for the fog [YFN+19, VS20]. Performing computations at the edge instead of the cloud reduces the limitations of network bandwidth and latency for applications on end devices [Sim19].

2.4.2 Definition

Fog computing is seen as a three-layered model displayed in Figure 2.4 and consists of the cloud, the fog and IoT layers. The fog is therefore located as a layer between

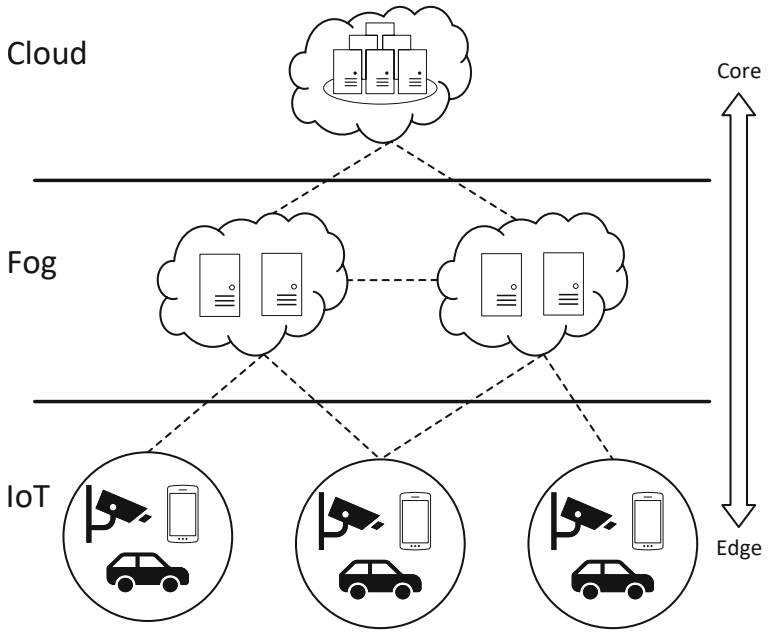


Figure 2.4: Common fog computing model consisting of layers for the cloud, the fog and IoT devices [DTD19].

the cloud and the IoT end devices. Applications can be executed in this three-layered model with components or services spread across the cloud and fog. It is important to note that applications can exist solely in the fog and independent of the cloud, which means without dependencies on the cloud or applications hosted there [IGF+18]. The amount of devices expected per layer differs between estimations. The NIST model predicts thousands of clusters in the fog layer with millions of end devices [IGF+18]. The newer IEEE standard assumes a much larger scale of tens of millions of fog resources with billions of end devices [IEE18]. In contrast to the different expectations of the device counts, the ratios between layers and therefore devices per fog resources are of similar scale in both estimations. This three-layered model is not universally accepted as there are multiple alternative definitions, which allow modelling additional concepts such as data consumption, management or specialisation of components [AVA19]. Similarly, there are models which further split the fog layer vertically to form hierarchies or other groupings [Sim19]. While the model and architecture of the fog are still in discussion, there is a unified vision of the advantages it provides and its properties.

2.4.3 Benefits and Characteristics

The distributed service fog computing provides has various benefits and characteristics. They are summarised in the following discussion. Furthermore, the differences to centralised cloud computing and the impact this has on applications is a major focus. The comparison to cloud computing is also summarised in Table 2.1.

Table 2.1: Summary of the presented characteristics of fog and cloud computing.

Characteristic	**Cloud Computing**	**Fog Computing**
Geographic distribution	Centralised [IGF+18]	Decentralised [IGF+18]
Computational resources	Clusters in data centres [RZH+20]	Lower performance (Clusters, Raspberry Pis, ...) [YFN+19, VS20]
Resource homogeneity	Homogeneous [BMZA12]	Heterogeneous [IGF+18]
Location in network	Core [DTD19]	Edge [Sim19]
Main advantage	High performance [IGF+18]	End device proximity [IGF+18]

The fog is distributed geographically. Computational resources can identify their location within this distribution and provide the information to applications. This location awareness can be utilised to ensure data is processed or stored where it is likely needed [IGF+18]. Possible examples are applications in the domain of augmented reality storing information about nearby points of interest. At the same time, this distribution allows for reduced latency and fewer limitations on the network bandwidth by being located at the edge of the network [Sim19]. In general, fog resources are assumed to have a reliable high bandwidth Internet connection, although this also depends on the computational resources used for the fog, which is still an open discussion and is presented in the following section [VS20]. As such, applications requiring close to real-time interactions are intended use cases for the fog. In comparison, batch-like processes generally do not benefit from the distribution of the fog and as such fit better to the more powerful cloud computing resources [IGF+18].

Applications, especially in the case of real-time interactions, may need to be aware of the mobility of their users. This allows for better placement of computational tasks or storage. In comparison to the cloud, scaling resources to the demand may also require identifying the location of the source. Otherwise, the utilisation of distant resources can not ensure the latency requirements an application may have. This mobility is possible because both users and IoT devices often depend on wireless connections [IGF+18].

Fog computing embraces heterogeneity, both in the applications and their data as well as the capabilities of its computational resources. As such, the interoperability of applications within a fog environment or installations of different fog service providers is important [IGF+18]. In contrast, the resources provided by cloud computing are more homogeneous [BMZA12].

Scalability plays a central role in supporting large amounts of devices expected to partake in a fog environment. As such, the efficient management of fog resources is necessary and largely automated. Individual fog resources or federations may therefore operate autonomously or form hierarchical structures. The automated management requires the programmability of the fog resources by the service provider and user across multiple domains [IGF+18].

The distributed nature of fog computing, the constrained resources of fog devices and the interactions with IoT devices have introduced new security challenges [LYCW18]. By overcoming these, the usage of fog computing also provides opportunities to improve the privacy aspects in comparison to the already established cloud computing. The geological distribution of fog computing resources allows control of where to process and store data. This can limit the handling of critical information to selected providers or even on-premise fog installations. Furthermore, the processing of data closer to its source can be used to minimise privacy concerning data in transit. That is because data can be filtered or aggregated early [PRB20].

Stream processing applications produce results in multiple steps, but these could each be executed by different fog providers. The exposure of information to a single provider can therefore be limited. For example, two different providers may execute tasks to aggregate data from different data sources while a third one joins their results and provides further analysis. None of these providers would have access to the complete data set or the capability to fully trace it. In contrast to clouds, this decentralised processing and potential storage can therefore be used to protect against service providers and limit exposure of data if a leak occurs [PRB20].

2.4.4 Service and Deployment Models

With fog computing intended as an adaption or extension of cloud computing and therefore sharing large similarities, it also supports the same service models. These are SaaS, PaaS, CaaS and IaaS [KDKZ17, IGF+18].

This similarity continues with the deployment models also matching the cloud computing ones. As such, private, community and public as well as hybrid deployment types, which are combinations of the previously listed ones, are again possible [IGF+18]. The rest of this section explores which hardware and further structuring of fog computing resources is often assumed or intended.

Fog computing uses multiple resources, these can be physical machines or virtual ones. They can function autonomously or form federations such as clusters or logical hierarchies. Such hierarchies can have merely organisational use or can for example be based on the proximity or latency between resources [IGF+18].

Following the three-layered model of fog computing introduced previously, the first layer consists of cloud resources. These could be complete data centres or specific physical or virtual computational resources of the cloud [IGF+18].

The second layer is the fog and its resources are referred to as fog nodes. Cloudlets are clusters of fog nodes and are a commonly used concept in fog computing. They functionally represent and are intended as mini data centres. They provide powerful computational nodes with high bandwidth and low latency connections [SBCD09].

The hardware or capabilities of fog nodes still have multiple interpretations. All resources from low-performance devices like single-board computers, such as the often-mentioned

Raspberry Pi, to small data centres or clusters are generally assumed [YFN+19, VS20]. There are also many places where a fog resource can be deployed. Locations such as cell towers, roadside units or even mobile targets like cars or trains are being envisioned and researched in related technologies [Sim19, VS20].

Another group of potential fog computing hardware is the network infrastructure itself. Low-performance devices such as routers, switches or access points could be used instead of having to create new installations [IGF+18, YFN+19]. They benefit from their ubiquitous access and existence everywhere between edge devices and the cloud. In the case of stream processing applications, data could potentially be processed directly in transit on these networking devices, instead of being routed to different computational resources close to the path taken. It will be interesting to observe if networking devices will be meaningfully utilised for fog computing in the long term. This is because the usage of networking devices is only possible as long as they are underutilised and have resources to spare.

Software-Defined Networking (SDN) is a paradigm gaining popularity in research and industry, aiming to maximise efficiency and minimise networking infrastructure costs. This is in part accomplished by managing the devices efficiently to increase maximum resource utilisation and to improve efficiency when underutilised [SAG+19]. These unused capabilities of devices are seen as a cost or inefficiency and are also the resources fog computing could consume.

SDN adds APIs to network devices to manage them via software. This splits the network into the data and control planes. The data plane contains devices, which handle the forwarding of the bulk data. The control plane merely oversees the network and reconfigures the data plane to adapt to changing demands and to raise the efficiency of the infrastructure. Dedicated control software allows for scalable and efficient management of the network by centralising its administration, while reducing complexity and costs [SAG+19].

The separation of the data handling and network management in SDN has an analogue in stream processing. For stream processing applications, many resources are available, where an application's operators can be executed. To handle the large number of possibilities, the placement of stream operators onto resources is automated and can target different optimisation goals [VS20]. This process solves the so-called stream operator placement problem [LLS08]. It controls the execution of stream processing applications and is the focus of the following chapter.

The third layer is the IoT layer at the edge of the network [IGF+18]. There is no consensus yet where exactly the actual separation between the fog and IoT layer is. In some definitions, the fog resources only exist within wired networks while others also include powerful IoT devices as possible resources. As such, not only end or IoT devices, which contribute to or use data processed by the fog, can be mobile, but also fog nodes themselves depending on the fog definition used by researchers [IGF+18, DTD19, MMA+17]. In these cases, topics such as energy efficiency or the availability of fog nodes gain in

importance [VS20]. Examples of such fog nodes could therefore be anything from IoT devices such as phones or laptops to installations in cars or trains [DTD19, VS20].

It was noted above, that under some fog definitions devices at the edge of the network are utilised as fog computing resources, but edge computing is another paradigm specialised for this environment. Edge computing shares many of the aims and methodology of fog computing, but only operates at the edge. Cloudlets are often associated with edge computing as well as Mobile Edge Computing (MEC), a concept in which similar computational resources are provided within the radio network. Mobile Cloud Computing (MCC) aims to directly utilise IoT devices as a computational or storage resource to provide cloud-like services. As such, the primary research challenges include the utilisation of such low-performance devices, which commonly are only temporarily available, are also mobile and limited by their battery capabilities [DTD19].

These closely related computing paradigms also highlight the question of if the separation of them still provides utility and if fog computing is developing to become the new umbrella term covering all these principles. In fact, the difficulty in defining and differentiating between edge and fog computing is often a major focus point of researchers. While their arguments are sensible, they usually share little similarity. Reasons such as a lack of hierarchies in edge computing, differing levels of heterogeneity in resources and different requirements or devices are examples of this debate [YFN+19, DTD19, IGF+18, AVA19]. Even the viewpoints that fog computing is an implementation or successor of edge computing exist [DTD19]. The lack of consensus and clarity also results in many researchers not differentiating between the terms [YFN+19, HV19]. For this reason, Chapter 4 provides the fog definition used for this thesis, discussing the applicability of different assumptions and interpretations in the context of stream processing.

While the discussion has been so far on the performance demands for stream processing applications and how their environment can be improved by utilising fog computing, the following section looks at the optimisation of stream processing applications themselves.

2.5 Optimisation of Stream Processing Applications

Stream processing applications are often seen as queries, which are continuously evaluated on newly received data [dAVB18]. As such, many optimisations in stream processing are also query optimisation strategies. The following discussion, therefore, starts by briefly introducing traditional query optimisation techniques. Their historic development has been naturally evolving them from more static techniques into one's closer matching the adaptability of modern stream processing systems. In fact, the first generation of stream processing engines was based on extending database models with the capability to process dynamic data [dAVB18]. A survey on current stream processing optimisations then follows this overview.

2.5.1 Cost-based Query Optimisation

The actual execution of a query consists of four phases in the case of traditional database querying and modern stream processing: parsing, optimisation, code generation and finally the actual execution. While parsing, a query tree is generated, which describes the flow of data and operation of data, closely similar to the DAG of stream processing applications for their topology. While a query defines the initial dataset to use and the final result, it still allows for variation in the intermediary steps. For these, the cost can be optimised by minimising the size of intermediary results and determining the best implementation to use for an operation. One method to minimise intermediary results is to change the order of operations. For database queries, this results in optimising the order joins are executed in, filtering data as early as possible and accessing the data efficiently. This methodology thereby selects efficient query plans by estimating their costs. In the last two phases, the code is then generated based on existing templates and executed [SAC+79].

Optimising the access of data relies on the usage of indexes, if available. Indexes are ordered lookup references for keys or other values of entries that can optionally be created and point to the segment containing the data point. Typically, to access the data all segments have to be read, but an index allows to skip any segment not containing useful data points for the query. The order of an index may not match the order of entries in the data segments and therefore individual segments might be accessed multiple times. This happens when entries are distant from each other in the order of the index, but are stored within the same segment. Additionally, accessing the index itself also has an associated cost. As such, using an index is not always an optimal choice and multiple indexes can exist for one set of data. Selecting the best access path therefore relies on estimations of their cost. This consist of CPU cycles and reading accesses. To perform these calculations, statistics are maintained, such as the count of distinct entries in indexes, the amount of values in data sets and their storage sizes [SAC+79].

These statistics are also used to estimate the cost to execute the query. The optimiser can adapt the evaluation order of operations and primarily select the best order of joining all relevant data sets. While the input and output sizes are fixed, it aims to minimise the intermediary result sizes as handling less data is always more efficient. The optimiser accomplishes this by estimating the result sizes of operations, which uses an estimation of the selectivity of operations. The selectivity of an operation is defined as the percentage of data points which will be included in the results [SAC+79].

2.5.2 Cardinality & Selectivity Estimation

The estimation of cardinalities and therefore selectivity is seen as the most important component of query optimisation [LGM+15]. To allow for these computations, many simplifying assumptions are made. These are the independence of attribute values and resulting join predicates as well as a uniform distribution of values within the domains of attributes [TDJ11].

These assumptions are known sources for estimation errors and as such techniques are developed to remove or improve upon them. By creating histograms of attribute values, the assumption of a uniform distribution can be removed. Similarly, there is a range of techniques aiming to handle the independence assumptions of attributes and associated join predicates [TDJ11].

In addition to the estimation accuracy, another problem is the propagation of errors in their estimates. Estimates are used as inputs for the estimation of the following operators and as such errors are propagated and can accumulate [YHM15]. Exponential error growth has to be assumed with an increase in the number of joins and can therefore overwhelm estimates in large queries [IC91]. Robustness against errors during the planning and execution of queries as well as the runtime management of workloads are therefore desired properties [YHM15].

As such, there is a variety of techniques aiming to improve robustness. Cardinality injection is a methodology using alternative information apart from just the statistics, such as data from previously executed queries, and can improve estimations and avoid errors. Similarly, plans can be selected with a performance that is less sensitive to estimation errors. The execution of multiple plans for a single query allows for more specific optimisations and reduces the chance of worst-case performance. One method to accomplish this is to partition the data and create an execution plan for each partition. This allows exploiting their specific correlations or value distributions. An alternative is to allow for more adaptability in the query plan or its execution as the following section describes [YHM15].

2.5.3 Adaptive Query Processing

Adaptive query processing allows for a query plan to be changed or its planning to be completed during the actual execution of the query. This provides benefits such as additional metrics or even index structures being available, less reliance on estimates and the possibility to adapt to runtime changes, such as performance fluctuations. Aims can be to improve the performance of individual queries or maximising the throughput of the complete workload. Additionally, in interactive environments returning partial early results can be beneficial to its users and therefore be of interest [GPFS02].

Estimation errors can be avoided by deferring some choices in the query plan until its execution, when more information is available. This is easiest to implement at materialisation points, blocking operations, which require all data to start their execution. These points allow a restructuring or reordering of any following operators, without having to worry about routing changes and state issues of currently executing operators [IDR07]. Another possibility is to only adapt the query plan if the collected statistics diverge too much from the estimated performance. Similarly, operators allow for various levels of adaptivity, such as their memory consumption or utilisation of new indexes if they are added to the database [GPFS02].

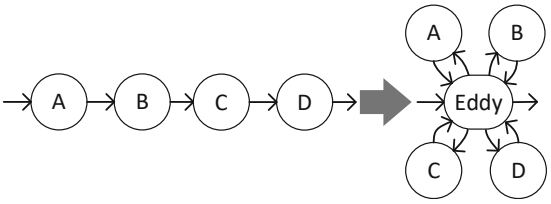


Figure 2.5: Eddies act as the central routing element deciding the order of the execution of operations. Data is only forwarded from the eddy once it has been processed by all operators which are a part of the eddy [HSS+13].

Query scrambling aims to reduce the impact of delays during the loading of data or slower than expected processing. In distributed systems, such delays and slower rates of received data can lead to an underutilisation of available resources. Query scrambling can adjust the query plan to execute some useful work in the meanwhile. This is done by inserting operations or changing their order of execution. As such, the efficiency of the query plan may be reduced to ensure there is always useful work to complete. This can accelerate the total query completion time by improving resource utilisation, but happens at the cost of the potentially worse overall efficiency of a new query plan [GPFS02].

Eddies route data between operators and as such can execute individual query plans for sets of data or even single data points. As such, eddies can be considered as a single n-ary operator consisting of multiple operators executing a more complex function [AH00]. Figure 2.5 shows an example of how the usage of eddies modifies the routing between operators. The routing of eddies is used to adapt the execution order of operators to changing performance of the execution environment or data characteristics and therefore operator efficiencies. Eddies can only be utilised when the order of operators can be changed without having to affect the state of the operators [GPFS02].

Overall, the historic development of query processing shows tendencies to increasingly become adaptive. Similarly, the large similarity of general stream processing and query processing highlights the value and interest into the technology. Many of these optimisations are also applicable to general stream processing. As such, the following section provides an overview of stream processing optimisations before then discussing the role of the stream operator placement problem.

2.5.4 Stream Processing Optimisations

This section provides a list of generalised techniques used for optimising stream processing applications. The section starts by reintroducing optimisation concepts from the previous discussion on query processing and their differences if applicable.

In the optimisation of stream processing applications, selectivity is also considered like in query processing. Selectivity in stream processing uses an alternative definition, which is the ratio of operator output events to input events. This allows selectivity to reach values larger than one to also model operators which send multiple events or events to

multiple outputs [HSS+13]. Another difference is the ready availability of selectivities. Databases can create statistics about the data they contain, but in stream processing, as previously mentioned in Section 2.2, the general assumption is only-once processing of events [PB19]. As such, the data can change over time and affect selectivities of operators, whose selectivity is not constant [HSS+13].

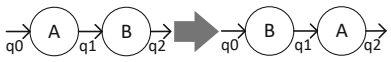


Figure 2.6: Reordering of operators to reduce the amount of intermediary results [HSS+13].

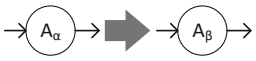


Figure 2.7: Selection of an alternative algorithm, which is better optimised in the given context [HSS+13].

Figure 2.6 shows the usage of selectivities to reorder operators to ensure ones with smaller selectivity and therefore smaller intermediary results are executed earlier. Techniques such as eddies, previously shown in Figure 2.5, can also be used in stream processing. The code generation executed during the translation from the logical to the physical plan can also select the best algorithm to use for the implementation. Deciding this depends on the semantics and constraints of all operations in the stream processing application. These may be readily available or have to be inferred depending on whether the programming provides well-defined high-level concepts or is based on low-level instructions. Figure 2.7 symbolises such an optimising change in the operator implementation selection [HSS+13].

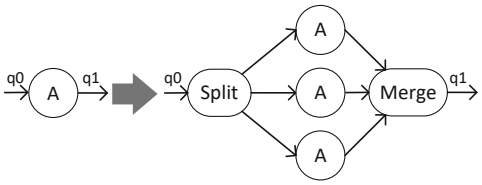


Figure 2.8: Application of operator fission to improve concurrency by creating independent parallel streams [HSS+13].

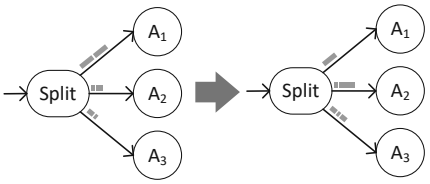


Figure 2.9: Changing the assignment of events to process can be used to balance the load for each operator [HSS+13].

Operator fission replicates operators to partition a stream into multiple parallel computations. Figure 2.8 shows the creation of such parallel regions. It requires partitioning the state of the operator or the operation being a stateless computation. Within fission areas and general computations, the load of operators should be balanced for optimal performance. Figure 2.9 shows that the distribution of data to operators can be improved to ensure more similar workloads. An imbalance in the workload could be created by reasons such as a slower than expected runtime environment or variations in the processing time of events [HSS+13].

Figure 2.10: Example of discarding events when the processing capacity is exceeded to ensure maximum throughput and stability for the application [HSS+13].

If a system is not capable of handling the data, then some load can be strategically shed to improve the throughput while aiming to minimise the negative impact. Figure 2.10 shows an example of such a technique, which discards events thereby reducing the accuracy of the computational results. Alternatively, cheaper operator implementations could be used, which may only approximate a result instead of an accurate computation [HSS+13].

Figure 2.11: Separation of the A operator into two separate operators to increase concurrency or to potentially enable other optimisations [HSS+13].

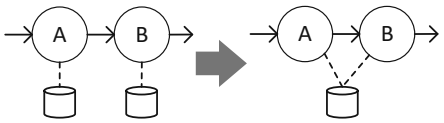


Figure 2.12: Elimination of redundant state between operators A and B by creating a shared state [HSS+13].

Separating operators into their smallest steps, as shown in Figure 2.11, allows for more reordering potential or extending a parallel region from operator fission. If many operators exist within an application requiring similar components, then sharing these between operators can save memory or storage resources. Examples of such components are large similar states, event queues or windows. Figure 2.12 shows the elimination of a copy of data, at the expense of having to safely manage the memory between the operators. Generally, this technique is limited to operators residing on the same computational resource [HSS+13].

Operators can also be fused into a single one, thereby avoiding the cost to transfer data, but reducing potential concurrency, because they can not run independently any more. As such, for this optimisation the cost of communication has to outweigh the loss in

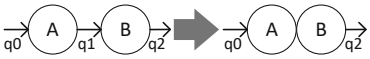


Figure 2.13: Operator fusion allows to remove the intermediary data stream q_1 [HSS+13].

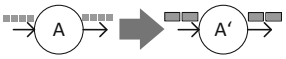


Figure 2.14: Communication and processing of data can be adapted to use larger batches of events instead of handling them independently [HSS+13].

concurrency. Figure 2.13 shows its application to eliminate the intermediary data stream q_1. Another method of optimising communication among other factors is to transfer and process data in larger batches as displayed in Figure 2.14. Batching can improve the throughput at the cost of latency [HSS+13].

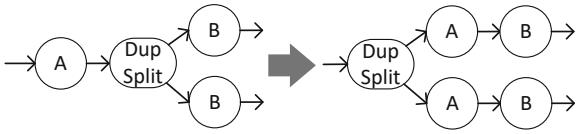


Figure 2.15: Elimination of one redundant instance of operator A in parallel streams by changing the graph to create a shared instance [HSS+13].

While all previously discussed optimisations aim to reduce the overhead, process data more intelligently or increase their concurrency, eliminating redundant elements can reduce the work itself. Operators or even sub-graphs of an application may be redundant, if they perform the same operations on similar data. In this case, additional instances of the sub-graphs or operators could be removed by sharing them with the associated streams. Figure 2.15 shows an example of this by eliminating the redundant A operator. This optimisation can even be applied across multiple applications, as long as the affected operators are stateless or their states can be combined [HSS+13].

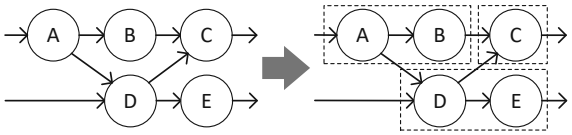


Figure 2.16: Operator placement assigns operators to be executed by specific computational resources. The assignments are symbolised by a box drawn around co-located operators [HSS+13].

The final optimisation is the placement where operators are executed. Figure 2.16 shows operators being assigned to different computational resources. This provides the basis to enable operator fusion and optimises for the usage of available resources and metrics affecting a stream processing application, such as available memory or latency [HSS+13]. The following section summarises the main takeaways of this chapter, before the following

chapter then provides a detailed introduction to the stream operator placement problem, it's variations and approaches to solve it.

2.6 Chapter Summary

To summarise, this chapter has introduced stream processing as a means to architect highly scalable applications. A complex task is split into operations, each performing a partial step in transforming inputs to the intended outputs. The operations are executed by workers to form operators and are continuously applied to any new input the stream processing application receives. Operators can be placed on separate computational resources and can often also be replicated to further scale potential throughput. This depends on if an operator is stateless or can partition its state and associated streams without violating the correctness of the application.

The significance of the location in the stream operator placement problem is mirrored by the introduction of cloud and fog computing. Both provide virtualised computational resources to execute stream processing and also other kinds of applications on. The centralisation of cloud computing results in data having to be sent long distances before it reaches a data centre and can be processed. This introduces latency and requires more networking infrastructure. Fog computing alleviates this by adopting a decentralised distribution of computing resources. As such, individual resources provide fewer performance capabilities than a data centre for cloud computing, but provide the advantage of proximity instead. This development has further implications such as the potential of context awareness and better protection of privacy or other regulatory concerns, by processing data closer to its source and reducing potential exposure. Fog computing suffers from multiple in details conflicting definitions and difficulty to differentiate from related paradigms. As a result, this thesis further clarifies the definition of fog computing to be used in Chapter 4.

The IoT is a trend that has benefited from cloud computing, but is able to overcome newfound limitations with fog computing. IoT devices are connected to the Internet and sense or interact with the world. By allowing such devices to interact or share data, they can be used to provide additional use cases and value. This results in large amounts of data being collected and requiring processing. Stream processing applications are designed to handle such a continuous stream of new data and quickly calculate results. This is known as online processing.

Additionally, to allow the IoT to benefit from fog computing with even lower response times for results and thereby allowing for new latency-limited use cases, stream processing applications have to support fog computing and optimise for the response time. This requires a solution aware of the latency between computational nodes as they are more significant in the distributed fog computing setting than in cloud computing. It was also mentioned that the deployment of stream processing applications uses automation to optimise the applications' performance. This is known as the stream processing operator

placement problem and therefore must be designed with support for fog computing and this problem domain in mind.

The chapter has introduced all these paradigms and provided examples of use cases to understand the context of stream processing applications in the fog. The context is necessary to consider during the design of a solution as it provides the foundation for the requirements and constraints.

The optimisation of database queries has been used as an introduction to stream processing, because stream processing has been developed on the basis of databases and their optimisations. Therefore, many of the techniques and terminology that have been developed in this field form the foundation for the theoretical and practical optimisation of stream processing applications. At the same time, database queries provide a context which is easier to understand and analyse, with a known dataset limited in size, rather than the continuously processed infinite data streams of unseen data.

In particular, the general framework of estimating the cost of potential plans and selecting the best one, as well as the estimation of intermediary result sizes including the use of selectivity and considering the adaptivity of a query are highly relevant to stream processing. Selectivity considers in a traditional query the percentage of the inputs which are selected as the output for an operator, while in stream processing this has been extended to allow for operators, which create more outputs than they receive inputs. As the later sections have shown, the practical use of selectivity in stream processing is somewhat limited, given that the data to be processed is unknown and therefore the operators may behave differently. At the same time, the operators themselves may be user-defined functions. For these, the specific behaviour may have not been modelled and cost can therefore not be estimated, except for using historical data. Historical data is not always available and may not represent new data limiting the general applicability. The static sizes of intermediary results in queries can be similarly considered in stream processing by considering rates of events on the relevant data streams.

Following this, a large amount of optimisations for stream and query processing has been discussed. In addition to the operator placement problem itself, operator fusion is of significance for this thesis as the potential to apply it largely depends on the operator placement. Operator fusion aims to eliminate the overhead of an intermediary queue or data stream between operators by either merging them into a single operator by chaining the relevant functions or by already performing this step during the compilation of the stream processing application to benefit from additional optimisations a compiler may identify.

With the established context already in mind, the following chapter introduces the formal stream operator placement problem definition itself, the state of the art of relevant stream operator placement mechanisms and other closely related problems to prepare for the discussion of the design of a new solution.

CHAPTER 3

State of the Art

This chapter provides the necessary knowledge surrounding the stream operator placement problem, its definition and its solving techniques. The stream processing operator placement is defined at the beginning in Section 3.1. A survey of heuristics and solvers for the problem is presented in Section 3.2. This includes an overview of the key concepts and different techniques used and a set of selected previous approaches targeting the stream operator placement problem. Subsequently, an overview of related placement techniques or scheduling strategies is presented in Section 3.3 to fully summarise the state of the art closely related to the main topic of this thesis. Section 3.4 discusses how the proposed solutions differentiates itself and improves on the state of the art.

3.1 Stream Operator Placement Problem

3.1.1 Definition

The stream operator placement problem consists of operators of a stream processing application being placed on computational resources to be executed there. It is a combinatorial problem of M operators and N computational resources with $O(N^M)$ potential solutions. Every resource has a set of capabilities or available capacities, which are used to identify valid placements. Depending on the problem definition, these resources may have identical capacities and capabilities. They are then considered homogeneous or heterogeneous if they differ. The definition of the placement problem and implementation decide, what capacities and capabilities are used. For example, a capability of a resource could be the availability of a GPU and therefore potential to perform GPU-accelerated computations [VS20]. Samples of common capacity limitations used are available processing performance, RAM or bandwidth. Operators to be placed can similarly require a set of capacities and capabilities [LLS08].

Based on these, constraints that must be satisfied for a valid placement can be defined, such as guaranteeing a minimum amount of resource capacities an operator requires to perform its function. Similarly, it is possible to define an optimisation function, which assigns a score for each solution. A solver would then aim to optimise the score by, for example, maximising the stream processing applications throughput or minimising its resource usage [dAVB18].

Solving such placement problems optimally is relatively trivial for small instances and can be completed quickly [TLL14]. For large instances, it can become a significant issue, such as with many resources or operators, because it is currently unknown if they can be solved in polynomial time. With the optimal operator placement problem being an instance of a more general NP-complete problem, its computation, therefore, quickly becomes intractable with the size of the problem [LLS08]. Variants have therefore been established around different trade-offs in guarantees on the quality of the solution and its computational cost [VS20].

3.1.2 Variants of the Stream Operator Placement Problem

The following list presents three variants of the stream operator placement problem, based on the definition of the problem or the strategy to solve it.

Operator Placement Satisfiability Problem This group of solvers aims to find one solution that satisfies all constraints or the answer that no solution can be found. As such, optimisation functions are not utilised. Therefore, this problem is a constraint satisfaction problem (CSP) [TLL14].

Optimal Stream Operator Placement Problem Compared to the previous definition, this category uses the optimisation function to find the best possible solution satisfying all constraints. As such, these belong to the group of constrained optimisation problems (COPs) [VS20].

Heuristic Stream Operator Placement Problem By using heuristics or meta-heuristics instead of an exact solution strategy, a trade-off in solution quality and the time to compute it is created. Therefore, heuristics only aim to find a solution that is good enough for practical purposes and may stop early. For example, limits on the maximum allowed calculation time could be employed or only a subset of all computational resources could be considered [VS20]. While heuristics can be complete, a property guaranteeing that within the bounded time they are able to find the optimal solution, the computational cost can be prohibitive. Meta-heuristics are higher-level constructs which modify the search strategy, such as a heuristic, in an attempt to optimise it. For this thesis, the most important distinction between meta-heuristics and heuristics is that a meta-heuristic is not problem-specific. As such, well-known meta-heuristics are utilised for many optimisation problems. In comparison, a heuristic can be designed for a given problem and therefore employ a more unique strategy [BR03].

As a result, solution approaches for the satisfiability or optimal placement problem typically differentiate the most in their definition of the constraints or optimisation function. In comparison, the work on heuristics allows for more variety as the aim is to identify shortcuts or simplifications of the problem, which work well enough in practice and aim to approximate optimal solutions. The following section continues the overview of solving techniques by presenting common properties of solving strategies before selected solutions are presented.

3.2 Survey on Solving the Stream Operator Placement Problem

This section presents a survey of techniques to solve the operator placement problem. While the previous section has already introduced three types of solvers: exact solving techniques, heuristics and meta-heuristics, this survey first introduces general characteristics across all of these types. In practice, most placement techniques only support a limited subset of these matching closely to their intended use case or environment. In the following section, selected approaches are then introduced.

3.2.1 Survey on Characteristics

Structure of Scheduling Unit

The previous definition of the stream operator problem is a problem of assigning operators to resources, but the level of abstraction at which a heuristic or scheduler operates is not specified. As such, the unit which is used for the scheduling and its structure can vary. This is known as the structure of the scheduling unit and could consist of individual operators or one or even multiple DAGs at once. The unit of scheduling is also used for the definition of constraints and the scoring function. Scheduling a large structure at once, such as a DAG or even multiple ones, provides additional information over individual tasks but also increases the difficulty as more possibilities exist and more information has to be considered [VS20].

Mode of Submission

Additionally, the number of scheduling units submitted by a user can vary. Some solutions may only allow for a single scheduling unit, while others can accept a batch of multiple units. Solutions with multiple scheduling units may also place further constraints on them, such as if they respect the order of submission or if they only accept similarly structured scheduling units [VS20].

Granularity

The granularity of a scheduler describes the number of scheduling units processed at once. While every single placement request can be processed individually, allowing for

collections of requests to be placed at once provides additional information at the cost of increasing the difficulty [VS20].

Adaptivity

In Section 2.5.3, the importance of adapting the execution of a query to changes in the environment has been discussed. Similarly, the placement of operators could potentially be optimised with ongoing changes. Static placement techniques only consider an initially optimal placement, while online techniques aim to continue optimising placements if the environment changes [dAVB18]. To reduce the number of operator migrations and expensive optimisation computations, online techniques are typically only applied after significant changes have been detected. These are violations of constraints, drops of the performance below thresholds or simply waiting until a long enough period of time has passed [LLS08].

Quality of Service Constraints

In addition to optimising some metrics with the scoring function, Quality of Service (QoS) constraints can be defined to guarantee a minimum standard in all solutions. These are commonly focused on limiting the cost or ensuring most of the available computational resources are utilised instead of idling. In edge computing scenarios and certain fog resources, the energy consumption may also be considered, as the available energy may be limited, such as for battery-powered devices. In workloads on existing datasets, such as during batch processing, a deadline can be defined, when all work needs to be finished. This time from the start of an application to the completion of all work is also known as the makespan of an application [VS20].

Heterogeneity

Heterogeneity considers the support of computing resources with varying properties or capabilities, such as their computing performance or available storage [LLS08]. In Section 2.4, it has already been discussed that cloud computing consists of comparably more homogeneous environments than fog computing. As such, supporting heterogeneity is more important for fog computing and is less often considered for cloud computing scheduling approaches [VS20]. Network awareness, such as the consideration of bandwidth or network latency, causes a similar differentiation between computing resources but is typically separately highlighted as it is less common. Additionally, while the networking capabilities of a resource could be treated like any other heterogeneous properties, the fact that changes outside of the control of the current resource can affect the performance has to be considered [PLS+06].

Algorithm Coordination

The placement of scheduling units can be executed by a centralised algorithm, which has information on and controls all computing resources. Alternatively, the algorithm can

support decentralisation, thereby offering many schedulers, which perform placements. It is often accomplished by allocating a set of typically local computing resources to each scheduler instance, which independently manages them. This benefits the scalability of the approach as each unit only considers its assigned resources and placements can be executed concurrently by the various scheduler instances [LLS08].

Search Space Traversal

To allow a computationally expensive placement technique to scale with more available resources, many approaches artificially limit the search space to reduce the possibilities which need to be evaluated. A wide range of different strategies can be utilised to accomplish this goal. Examples include ending the search for improved solutions early [NCGP19, VS20], decomposing the placement of resources into multiple simpler problems [DLL+16], utilising artificially created hierarchies within the search space [NCGP19] or attempting to select initial resources by first solving a simplified problem [NCGP19] and using resources nearby [SNSD17] or on connections between data sources and sinks [CZM20]. In the following discussion, these approaches are introduced in more detail.

3.2.2 Solution Approaches

This section introduces selected approaches to solve the stream operator placement problem. While the previous sections have been grouped based on characteristics or problem classifications, this section presents a survey based on similar use cases, characteristics or problems a technique aims to solve. It aims to highlight unique approaches as well as work related to the field of fog computing.

Search Space Traversal

Many heuristics and meta-heuristics are based on strategies of attempting to improve already found solutions iteratively. For these exit conditions, such as time and iteration limits or changes in the rate of the solution improvement are often used. Some also use greedy strategies, which make locally optimal decisions and do not allow revisiting or changing them later in the search. Thereby greedily selecting options, which seem good at the moment, may turn out to be less optimal as the search progresses [NCGP19, VS20].

Another approach is to decompose the problem into multiple sub-problems, which can then be solved more efficiently. Their solutions are then merged into a final solution or are used as inputs for the following sub-problems. An example of such a problem is an initial assignment of work to cloud or fog resources to optimise the trade-off in computational efficiency of the cloud to the low latency of the fog. Following sub-problems can then handle each resource pool, the cloud and the fog, independently. The problem has therefore been simplified by discarding potential placements by first deciding between utilising the cloud or the fog, before specific resources are considered. The decomposed problem then approximates an optimal solution [DLL+16].

More commonly, potential resources are selected in an initial step before the placement of operators is considered. Geographically close resources such as cloudlets and colonies can be used to subdivide the search space into separate zones. Operators can then be independently placed within these zones or sent to other nearby zones if a local placement is not possible, because the zone is overused [SNSD17]. Governor similarly selects nearby resources initially, but bases this on the necessary communication path. If the data sources and sinks have known locations, then placing operators on paths connecting them can minimise latency and network costs. The selection of paths can depend on the metrics to optimise and may consider options such as low latency, high throughput or common paths between sources and sinks to reduce necessary resources [CZM20].

In contrast to a geographical separation and selection of resources, the use of a logical hierarchy is also possible. In such a case, computational resources are added into a logical tree structure as leaves, with inner nodes acting as aggregates of the resources of the nodes they contain. The placement of operators then starts at the root of the tree and attempts to place the operators optimally across the aggregated nodes for all sub-trees. Following this placement is attempted one level lower in all sub-trees that have been used for the previous placement. This navigation is then iteratively applied to the sub-trees until the minimal set of resources capable of handling the operators is found. Each iterative step, therefore, eliminates parts of the search tree, while performing this decision efficiently by using aggregated resources instead of considering each resource individually. The operators are then placed on selected resources represented by the leaves of the tree. Alternatively, this process can also be executed in a bottom-up search. To accomplish a bottom-up search, a small set of nodes in the hierarchy is selected for placement and if an assignment is not possible, the search space is expanded by searching on the larger parent level in the tree structure instead. The process iteratively expands the number of resources considered until either a placement is found or a placement using all resources has failed. Both strategies utilise a two-step process, with the first being the creation of the logical hierarchy, which is a clustering of resources based on their network capabilities, such as latency for example [NCGP19].

A similar idea in expanding the search space is to solve a placement problem with relaxed constraints first and use this solution to select the search space for the actual placement optimisation. The relaxed placement is faster to compute, because selected constraints are removed, thereby simplifying the problem. To ensure a placement with all constraints is possible using the resources previously found with the relaxed problem, additional neighbouring resources are added [NCGP19].

Geospatial and Mobility Awareness

Some approaches also aim to fully utilise the main principle of fog computing, moving computations closer to the end devices and network edge. They consider geographical distances in addition to cost or performance metrics. While there are only a few approaches aiming to minimise distances, these are quickly growing in complexity and are starting to utilise strategies such as predicting movement.

Under the assumption that a stream processing application is only deployed for a single edge device, it is possible to simply relocate the application if the device's distance to the computing resource becomes too large. To accomplish this, it is necessary to have access to and monitor the GPS coordinates of the device. Additionally, geographical regions are defined around computational resources. Whenever the device crosses a regional border, also often known as geofencing, the application is moved to the computational resource located in the newly entered region. In addition to merely considering the geographical context, this approach can also be expanded to define regions for other regions, such as privacy or regulatory contexts [WZR19].

Mobile fog is a general application model which allows for the implementation of stream processing applications. It is based on the idea that all computational resources exist in a hierarchy of geographical regions. As such, work can be completed directly in the region it originates from, or if the target is outside the current region is simply sent up the hierarchy until a resource is found which is responsible for both the source and target. In this model, the same processes are deployed on each computational resource and messages can only be transferred along with the parent or child relations within the hierarchy. Additionally, an API exists to manage the movement of devices between regions to allow their entry and exit of regions. This movement tracking is only supported, but not implemented by mobile fog, such as via the use of geofencing. The approach is also unique as it does not solve the stream operator placement problem in the usual sense, but instead provides an API for an application to route each individual message to a computational resource, which then processes it. Conceptually, it is, therefore, closely related to eddies, which were discussed in Section 2.5.3 [HLR+13].

MigCEP is an approach that not only places an operator once optimising for latency, but also aims to create a migration plan for each operator in advance. The migration plan consists of transfers of an operator to new computational resources, which satisfy latency constraints to the end device, while minimising the total cost of migrations. The individual transfers have defined initiation times and deadlines and are based on predicting the movement of data sources and sinks. The migration plans are also coordinated between operators and computational resources [OKRR13].

Network Awareness

Network-aware placement is closely related to heterogeneous scheduling. In this case, the optimisation also considers the network itself as a resource. This section discusses how some approaches utilise information on bandwidth or latency to improve their placements. The big difference to regular heterogeneous scheduling is that these network performance metrics are not of a single computation resource, but instead the connection between them. Therefore, the network as a shared resource creates new difficulties, such as external factors affecting the performance or a general lack of control over the resource. Because of this, basic information, such as the current utilisation, is easy to acquire for other resources, but has to be explicitly measured for the network. Furthermore, the performance can change because of external influences, and remeasurements become

necessary [PLS+06]. At the same time, this can quickly become a scalability issue, because the networking performance can differ between any two resources. To avoid the need to measure every possible link resulting in $O(n^2)$ performance, this area of placement optimisations typically uses incomplete information, as the following paragraphs will explain [SPPvS08]. Therefore, these approaches generally focus on performing a few measurements to predict the performance of all other links [ZLN+17]. As such, while heterogeneous scheduling typically has known capacities and usages with the control to dedicate a resource, network-aware approaches often use incomplete information and estimations, as this section showcases.

Before discussing placements considering latency, it is necessary to establish for this thesis the terminology or different viewpoints one can have of latency. For this, the definitions of Rosenberg et al. are used, but with necessary adaptions for the context of stream processing applications [RPD06]. At the lowest level, network latency occurs on any connection between two computational resources, which can include intermediary network devices on the communication path taken. The network latency furthermore includes the time until data can be transferred if a bottleneck exists and a waiting queue has to be created, the queueing delay. Similar to the network, the processing of data can also cause latency. To differentiate it from the network latency, the term of execution time is therefore used. It also includes potential queueing time if a bottleneck exists for the processing of data. Both the execution time and network latency can be considered on an individual operator or link level, or as accumulated time from the data source to the data sink of a stream processing application. When considering the DAG, this matches with the longest path in the graph when edges are labelled with the length of the respective type of latency. For an application, the execution time is the longest path if the network latency and execution times are used as distances. Furthermore, this matches with the total time needed for an event to be processed by the application. Latency can also be considered from an edge or cloud device, which sends to or receives data from the stream processing application. It is referred to as the response time or latency and is simply the application's execution time with the additional network link latencies to the edge or cloud devices and back. This is a metric that requires access to additional information or control over a client device, as it can differ between clients. The previous section on geospatial awareness, for example, indirectly optimises this metric by reducing the involved distances [RPD06].

To estimate latencies, position-based methods or techniques for completing matrices are commonly used [ZLN+17]. Approaches based on positions aim to estimate the current coordinates of any computational resource in a defined space and are, therefore, also known as embeddings [TR18]. Euclidean spaces are common examples, but alternatives such as hyperbolic spaces have also been used. Based on the positions of two computational resources, the distance in the defined space can be calculated, which is then the estimated latency. As such, the aim is to estimate positions, which minimise the error in known latencies and, therefore, deviations in the distances between resources. To accomplish this, spring-based forces are often used to push or pull resources apart or together based

on known latency measurements. These computations are repeated iteratively to update the latency estimate to current measurements and integrate new measurements [FW09]. Resources with stable and accurate positions can be used as landmarks, so that any new resource can compute its position only based on measurements to these landmarks. Google has used such a system to estimate the latencies to blocks of IP addresses to improve its content delivery networks' (CDNs) performance [SPPvS08].

Alternatively, matrix completion aims to estimate all missing values within a matrix M of latency measurements. This is accomplished with matrix factorisation, which aims to find two lower-rank matrices, U and V, such that $M = UV^T$. Matrix completion approaches provide two theoretical benefits over position-based approaches. In a position-based approach, their distance and, therefore, latency estimation is symmetric, while real-world comparisons show that the latency can be asymmetric. Furthermore, when comparing three resources, triangle inequality violations can exist in the real world but not in the models. As such, the sum of the measured latency for two sides may be smaller than the third side [ZLN+17].

Stream-based Overlay Network (SBON) uses an approach with spring-based forces to estimate latencies in a three-dimensional euclidean space called cost space. To reduce the variance in latency measurements, an additional filter is applied. Afterwards, an initial set of computational resources is selected for the placement for each task. This process also uses spring relaxation, but does not use the latency alone as the basis for the optimisation, but instead uses network usage. Network usage for any network link has been defined as $datarate(l) * latency(l)^2$. The *datarate* matches with the observed or estimated bandwidth usage between two operators and not the capacity of the actual network link. Each edge in the DAG of the stream processing application is modelled by a spring, which aims to minimise the data usage for this link with the increased importance for latency. As such, an approximate position for each operator is estimated, which minimises the total network usage of the DAG, similar to finding the coordinates to estimate latency. A small set of nearby resources is then selected for each operator as potential placements. The operator is placed on the resource closest to the estimated position, which fulfils all resource requirements. This process is periodically repeated to potentially migrate operators into more beneficial positions [PLS+06].

Multi-operator Placement Algorithm (MOPA), a closely related approach, uses $datarate(l) * latency(l)$ as a metric for the optimisation instead. This aims to minimise network utilisation rather than the skewed optimisation in favour of latency [RDR10]. Minimising the network utilisation has also previously been attempted, although based on the peer-to-peer path routing of distributed hash tables [Ac04].

The QoS-Aware Scheduler is also based on SBON and has been implemented as a distributed scheduler into Apache Storm. It uses $datarate(l) * latency(l)$ as a metric for the spring-based force minimisation as well. Instead of the three-dimensional cost space with three latency dimensions for computational resources, it has only two latency dimensions with one utilisation and availability dimension each for a total of four dimensions. After the virtual placement in the cost space has been found, utilisation

and the availability of a resource are used as a part of the distance computation to find the nearest and most similar resources to the virtual placement. This way, the scheduler attempts to find resources with low utilisation, the percentage of idle time, and high availability. As such, it can improve the utilisation of resources based on their observed behaviour but is not aware of the heterogeneous demands of the operators. Still, only the latency dimensions and the datarate are considered during the spring-based force minimisation, meaning that the optimisation of these network metrics is the focus [CGPN15b, CGPN15a].

Planner decides the sub-graphs of the DAG that should be executed on the cloud or the edge. Based on existing constraints, the maximal sub-graph that can be placed on the edge is computed. The sub-graphs for the cloud and edge are then defined by selecting links within the maximal sub-graph to be cut, which are the locations where the data has to be transferred between the edge and cloud. For this decision, the metric $inputrate(o) * selectivity(o) * outputsize(o)$ is used, which equals the $datarate(l)$ from the previous approaches. The selectivity is not necessarily known, because any operator could be a user-defined function. Therefore, for known operator types or user-defined functions, default values are used. At runtime, these could then be updated with their real metrics. Because of the default values, the optimiser primarily moves filter operations towards data sources at the edge. Furthermore, the main assumption is that data is created at the edge and results should be sent to the cloud, thereby defining where data sources and sinks are generally placed. Processing data from the edge for the edge is, therefore, outside the scope of the solution. The approach does not aim to reduce global network utilisation, but instead merely the transfer from the edge to the cloud while maximising the usage of edge resources [PCSA18].

Another heuristic uses a similar idea of splitting operators across the edge and cloud, but by identifying regions in the DAG first. The process of creating these regions and their placement is visualised in Figure 3.1. For each individual edge in the DAG, a region is created, except when a series of edges exists, where the intermediary operators only have one input and output. To find the regions, joins and forks, which are defined as operators with more than one incoming or outgoing edge in the DAG, are searched, because they are the start- or end-points of such regions. The series of edges between these points is then merged into a larger region, such as in the case of L, I and A in Figure 3.1. These regions form a hierarchy, which closely resembles the DAG, by adding input and output relations between regions for any operator in multiple regions. Each data source and sink is annotated with the information if it is placed in the cloud or edge. Using these annotations and the region hierarchy, operators can be placed using two strategies. The first strategy aims to optimise the response time of all paths by greedily placing operators of the hierarchy iteratively in a breadth-first search order on the computational resource with the lowest latency which satisfies the operator's constraints. The second strategy applies simple rules to decide for each operator if it should be executed in the edge or cloud. An operator which outputs only to a data sink or other operators placed in the cloud, is placed in the cloud, while any operator on a path to an edge-located data sink

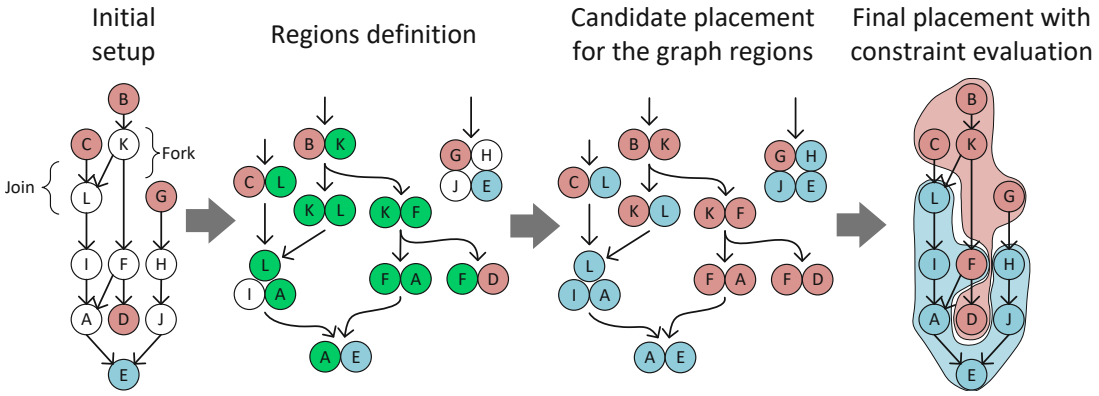


Figure 3.1: In these steps, operators intended to be placed at the edge are coloured red and ones in the cloud are blue. Identified joins and forks are coloured green. The first step is the definition of the DAG consisting of operators identified by their numbers, which is then converted to a hierarchy of regions. The third step shows the resulting operator classification, after applying simple rules. In the last step, the coloured regions signify their actual deployment location. The circle's colour is the initial candidate location [VdAL18].

is also placed on the edge. Operators intended for the edge are then similarly placed considering their latency and any operator which can not successfully be placed without violating constraints is reassigned to the cloud. In the figure, this case is displayed for the operator F. The second strategy, therefore, does not consider latency in the cloud placement [VdAL18].

Hiessl et al. also minimise the latency by considering the highest response time of all paths in the topology, based on network latency and processing delay. Additionally, it optimises the availability, bandwidth usage to enact a topology and the cost to continue operating it, which are all weighted elements of its scoring function. It supports heterogeneous CPU, memory and disk-space requirements. Interestingly, the solution limits the number of operators to one per computational resource at most and, as such, removes the potential of co-location requiring optimisations. The placement problem is modelled as a COP using Integer Linear Programming (ILP) and solved optimally with the pre-existing solver CPLEX or alternatives. The placement is then periodically recomputed and adjusted to provide online scheduling functionality. As such, the computational complexity of calculating an optimal solution is higher and potentially limits scalability [HKH+19].

Maximum sustainable throughput considers the idea that the input rate of events to process is not constant. As such, rather than focusing the optimisation on current bandwidth usage or latency, the scalability of the system should be considered. The approach assumes that networking is the limitation and therefore does not consider other resources which could become a bottleneck, such as the processing performance or memory capacities. It is necessary to know the selectivity of any operator and the

bandwidth capacity of any network link. With this information, the expected data rate can be calculated for any link between operators in the DAG if given an input rate for the application. This closely matches the intermediary result size estimation for database queries discussed in Section 2.5.1, but using data rates instead of constant sizes. The sum of data rates over a link can then be put in relation to the bandwidth capacity to calculate a metric similar to the utilisation: a maximum scalability factor for data rates on this link. The link with the smallest scalability factor is the first link to become a bottleneck and, therefore, what limits the input rate or maximum sustainable throughput. The optimisation, therefore, aims to maximise the smallest scalability factor by formulating a CSP. The problems of acquiring accurate operator selectivities and measuring or estimating the capacity of all network links are not a part of this solution. Additionally, the general insight into achieving maximum sustainable throughput is to spread the usage across more network links or computational resources to increase capacities [LGI20]. This is, of course, a goal contrary to minimising costs or latency and enabling stream processing optimisations such as operator fusion. Under the assumption that the problems of measuring operator selectivities and bandwidth capacities can be resolved, then taking this metric into consideration and balancing it with the typical objectives of minimising resource usages or costs might lead to more promising results.

Maximum Cumulative Excess (MaCE) is a metric to estimate the worst-case latency provably within small bounds for a stream processing application. It requires a lot of historical information to predict future performance, such as the rate of input events over a series of time intervals, the selectivity of operators, the cost of processing an event and the processing capacity of computational resources. MaCE uses queueing theory to model bottlenecks in the processing of events and thereby predict latency. The main idea is to calculate the excess of events an operator receives in a time interval, but is not able to process, because of a resource bottleneck. For the following time interval, a new batch of events would have to be processed and the excess of the previous interval as well. As such, the excess can accumulate, providing the name for this metric. The largest excess across the application then matches the expected latency, if operators always process the oldest events first. The timestamp of an event depends on when the input data has entered the system or for the output of an operator matches to the oldest time of any input used in the calculation. The modelling can be extended to include bandwidth limitations of network links, by modelling them identically to processing bottlenecks, and network latencies. Overall, MaCE is a metric which depends on specific event processing orders, but can be generalised for batches of events to reduce the cost of maintaining the order. Additionally, because the model requires to know the input rates, it is built for workloads with repeating patterns. This is not necessarily a problem, because the input load of workloads is often similar for longer periods of time or can have regular patterns, such as changes in activity over a day [CGB+11].

Apache Spark Streaming

Apache Spark is a batch processing framework, that also supports the stream processing of data. It accomplishes this by collecting events into mini-batches, which are then independently processed. The collection of such mini-batches, therefore, introduces a configurable level of latency, but provides a stream-like computational model with low enough latencies for many applications. Each mini-batch then consists of multiple tasks, which need to be placed similarly to operators [CDE+16]. The placement of tasks and the differences to stream processing are further discussed as a related problem in Section 3.3.2. In general, it is possible to adapt or customise the frameworks of one processing paradigm into another [PB19]. Apache Spark is a popular framework often known for its streaming capabilities and is therefore discussed in the following section, even though the placement logic is based on the batch processing paradigm.

To better support the heterogeneous requirements a task can have or a computational resource provides, some interesting placement heuristics exist. Symbiosis separates tasks to place into CPU-bound and network-bound tasks, depending on if the input of the task is already cached or co-located on the same machine. Afterwards, it aims to symbiotically execute tasks by co-locating tasks of the different categories, thereby preventing resource conflicts. Furthermore, this avoids the difficulty of defining or predicting resource usage or limitations and monitoring the current utilisation. This is especially difficult for tracking bandwidth usage, as the new information requires interface modifications, which can limit compatibility with existing software. Rather than finding the optimal placement, this heuristic can be largely considered as avoiding easier to detect bad placements [JMLL16]. RUPAM uses a similar concept with queues for each category of tasks, but has extended it with GPU, disk I/O and memory categories. These categorisations have been predefined for some tasks, but primarily depend on usage records of previous executions. Some tasks are also placed in all queues [XBLK18].

Sparrow aims to decentralise task scheduling by randomly probing potential placements and enqueuing a reservation for the task on probed machines. With random probes, the need for global knowledge is avoided. Once the first machine with a reservation is ready for the execution, the task is retrieved and other reservations are cancelled. This delayed placement or allocation is known as late binding. A task is, therefore, simply sent to any of the available placement heuristics, which can perform the placement without any need for coordination with others. This allows for scalable concurrent placements with low latency for a demand of thousands or even millions of tasks per second [OWZS13]. Due to its simplicity and success, it has inspired various schedulers and placement heuristics. The technique has also been improved by sharing reservations with other tasks of the same scheduler. It helps in counteracting the randomness inherent to the probing and queue-based placement strategy, which improves the order of task executions [LLZM17]. Hopper uses a similar concept, but speculatively spawns additional task copies to counteract slow executions of individual tasks, which may cause large delays for the complete workload. Once a task finishes, slower copies are terminated [RAWY15]. The following section discussing related optimisation problems, such as batch processing.

3.3 Related Optimisation Problems

The discussion of Apache Spark has shown that there are optimisation problems from other paradigms closely related to the optimisation of stream operator placements. In fact, they share such a similarity that implementations of one paradigm can be adapted to represent another paradigm, as in the case of the streaming support of Apache Spark. Similarly, the processing of database queries has been used as an introduction to establish some techniques and terminology for stream processing. At their core, all of these problems consist of assigning some operators or tasks to resources in an efficient or cost-effective manner. For this reason, this section not only introduces related optimisation problems, but also has a focus on the discussion of their differences, especially when compared to stream processing.

3.3.1 Database Query Processing

In Section 2.5.1 ff., traditional query processing for databases has been discussed as an introduction to stream processing. The focus has largely been on their similarities, so this section aims to highlight the differences to stream processing. As such, the placement of operators for databases and stream processing is closely related, but also has some distinctions.

With stream processing, the determinism and consistency of the application can have various implementations, as discussed in Section 2.2.4. Meanwhile, for databases, the query result must match the currently stored content. As a result, while in stream processing individual events are processed and streamed between operators, in databases this optimisation, also known as pipelining, can not be applied to blocking operators [HSS+13] To sort as a part of a database query, the input to sort has to be completely known first. Therefore, the operator has to block the execution until the input has been collected and can only then start producing outputs [WvG93]. In Section 2.5.3, adaptive query processing and the possibility to adapt or finish a database query plan during its execution has been discussed. This can be, for example, accomplished at blocking operators, as any following operator similarly has to wait for the blocking operator to collect its input first [IDR07]. In comparison, based on the data flow paradigm for stream processing applications, all operators can be executed concurrently, while database query execution can delay the execution and placement of operators dependent on blocking operators [dAVB18, DM74].

The difference in consistency definitions also leads to variations in the available information. For databases, statistics, such as histograms, can be collected, while stream processing is based on an unbounded sequence of events, which is continually received and, therefore, previously unknown. As discussed in Section 2.5.1, these statistics allow us to predict the behaviour and cost of the workload better. Furthermore, database management engines provide a limited set of functions with cost models, while stream processing engines often allow the implementation of user-defined functions. Streaming processing optimisations,

therefore, focus on elasticity to react to load differences rather than predicting the behaviour of user-defined functions [SAC+79, dAVB18].

3.3.2 Task Scheduling for Batch Processing

Closely related to query processing is batch processing. In batch processing, a workload, called a job, consists of tasks, which for their execution may depend on the results of other tasks, similar to operators in query processing. In contrast to scheduling or placing operators, tasks must be finished before any following task can start. This means the previously discussed pipelining optimisation can only be applied by placing the complete pipeline in a single task. Tasks share, therefore, some similarities with blocking operations in their behaviour. For this reason, tasks are often structured into stages, which can start once all tasks in the previous stage have been completed. Tasks within a stage do not have to start or end at the same time, but could be executed one by one. As a result, practical task scheduling approaches can limit themselves to the placement of individual tasks, rather than having to consider possibilities such as pipelining. At the same time, because the tasks do not necessarily have to be long-lived, use cases exist where thousands or even millions of tasks need to be scheduled every second [OWZS13].

3.3.3 Virtual Machines and Containers

In Section 2.1, cloud computing has been introduced as a rentable computing infrastructure. It provides services for the execution of containers or virtual machines. For these, a mapping of the virtual resources to the physical resources, which execute them, has to be created and can similarly be optimised to improve the QoS or minimise costs. The main difference to the placement of stream processing operators is that the application to be executed, installed on the virtual machine or container, and its behaviour is generally unknown. As such, the placement typically relies on defined resource or communication requirements in addition to factors such as the current resource utilisation or total resource capacity [FMIF18].

3.3.4 Virtualised Network Functions (VNFs)

VNFs aim to replace the deployment of traditional networking appliances, such as load balancers, CDNs or firewalls, with software executed in virtual environments. As such, they are a fundamental change to how networks are designed and maintained [AAB19]. VNFs provide benefits such as easier management, fast deployments, scalability and high availability for these functions [LT19]. At the same time, the sharing of resources can also reduce capital and operational expenses [AAB19].

The virtualisation also allows for the potential of optimising the placement of VNFs for their deployment or migration. Similar to the previously mentioned placement problems, there are varying objectives, such as cost and resource usage optimisation or improving QoS-metrics. Their optimisation can be considered independently or have chain or graph-based dependencies similar to the operators in stream processing [LT19].

The following section discusses how the proposed placement heuristic of this thesis innovates on the state of the art and separates itself from existing solutions.

3.4 Differentiation of the Proposed Solution to the State of the Art

As the title of this thesis implies, latency is an important part of this solution and, as such, the network-aware solutions that have been presented in the previous chapter share the most similarity and are now compared to highlight the uniqueness of this solution. Table 3.1 presents a brief comparative overview of their capabilities, which are discussed in more detail in the following.

First off, the table clearly shows that many solutions are already aware of latency or optimise it, but also that many of them limit themselves to only optimising network-related metrics and are not aware of resource limitations. SBON [PLS+06] only uses latency and bandwidth during the actual optimisation. Afterwards, the virtual placements are attempted to be realised by finding the nearest neighbours that fulfil a variety of constraints. This means that heterogeneous properties are only satisfied and are only indirectly optimised, which is also the case for the solution of Veith et al. [VdAL18]. The QoS-Aware scheduler [CGPN15b] recognises this potential and adapts SBON by considering the utilisation of computational resources and attempting to thereby maximise the available resources to operators, but this is still unaware of heterogeneous capabilities. In contrast, the proposed solution tries to maximise resource utilisation and make full use of heterogeneous capabilities by trying to consolidate placements on computational resources, which also maximises the possibility of co-location requiring optimisations. The researchers also report cases of instability in the placement that can negatively impact the availability, because of the decentralised process these approaches use.

Another rare feature of the solution of this thesis is being an online scheduler and, as such, the capability to adapt and optimise according to changing conditions. Additionally, this reduces the risk of misconfiguration because the schedulers can use information gathered during the execution and can reduce the overall configuration necessary. Of the presented solutions in the table, only MigCEP [OKRR13], Hiessl et al. [HKH+19], SBON [PLS+06] and its derivatives, MOPA [RDR10] and the QoS-Aware scheduler [CGPN15b], support this.

This solution also avoids the consideration of bandwidth capacity, which has not been clarified by many papers how the metric is actually acquired and if it was actually implemented or only theorised. Being a property of the network, it can differ between any two computational resources, change over time and is affected by the actions of other users, making it difficult to have accurate values by relying on simple configurations and expensive to repeatedly measure. In the case of Lambert et al. [LGI20] and Governor [CZM20], this is not a problem because they were evaluated using simulators where such values can be simply provided. For SBON [PLS+06] it is not clear if constraints

Table 3.1: Summary of main properties of previously discussed network-aware placement heuristics.

		Hetero-geniety			Network Awareness				
Name	**Online Scheduler**	**CPU**	**Memory**	**Other**	**latency**	**bandwidth usage**	**bandwidth capacity**	**Optimi-sation Targets**	**Main Mechanisms**
MigCEP [OKRR13]	✓	✗	✗	✗	✓	✓	✗	latency, bandwidth, migrat. costs	decentralised migration plans, pathfinding
Planner [PCSA18]	✗	✗	✗	custom (e.g. CPU)	✗	✓	✗	latency, bandwidth	graph partitioning
Veith et al. [VdAL18]	✗	✓	✓	✗	✓	✓	✗	latency	graph partitioning
Lambert et al. [LGI20]	✗	✗	✗	✗	✗	✓	✓	bandwidth	greedy/ graph partitioning/ COP solver
Governor [CZM20]	✗	✗	✗	✗	✓	✗	✓	varying	pathfinding, greedy assignment
SBON [PLS+06]	✓	✓	✓	disk-space	✓	✓	✓	latency, bandwidth	decentralised spring-force minimisation
MOPA [RDR10]	✓	✗	✗	✗	✓	✓	✗	latency, bandwidth	decentralised gradient descent, nearest neighbour
QoS-aware S. [CGPN15b]	✓	✓	✓	disk-space	✓	✓	✓	throughput, latency, bandwidth, availability	decentralised spring-force minimisation
Hiessl et al. [HKH+19]	✓	✓	✓	disk-space	✓	✗	✓	latency, enactment & operation cost, availability	COP solver
This work	✓	✓	✓	✗	✓	✓	✗	latency, bandwidth, resource consolidation	heuristic COP solver

on the bandwidth capacity were used for the evaluation or only a theoretical possibility. With the QoS-Aware Scheduler [CGPN15b] this is even less clear because it only states that it follows the approach of SBON and never mentions the usage of possible constraints or other information considered. Only with Hiessl et al. [HKH+19] is this clarified, as it only considers the bandwidth when an operator is downloaded to the worker during the enactment of a placement. As such, this is only a metric of the central store to each worker, allowing the achieved rate during the last download to be reused.

Finally, there are, of course, differences in how the placement problem is defined and solved. For example, while SBON [PLS+06] optimises latency on the local network link level, the approach of Veith et al. [VdAL18] and Hiessl et al. [HKH+19] consider the accumulated network and compute latency at the data sinks, while this solution only considers the accumulated network latency. Hiessl et al. optimise properties, such as the availability, bandwidth usage to enact a topology and the cost to continue operating it. It uses constraints to support heterogeneity but also disallows more than one operator on a single resource, thereby preventing co-location. In contrast, this thesis is purely focused on performance metrics, such as maximising the benefit of co-location optimisations or reducing bandwidth usage during execution. The formulation as a COP in this thesis allows to easily include other similar metrics if they are available and of concern. Additionally, like most of the presented solutions, this thesis uses heuristic solving strategies to reduce the risk of scalability concerns. The specific heuristics used are hill-climbing and an ant system, differing from all of the discussed solutions. Furthermore, the solution was also implemented in and evaluated with Apache Storm This ensures that the design can be realised and help identify systematic errors.

The following section provides a brief summary of the most important concepts that are presented in this chapter.

3.5 Chapter Summary

This chapter has primarily discussed the stream operator placement problem and the state of the art in solving it. Operator placement, in general, is the problem of assigning a set of operators to a set of computational resources while minimising costs or maximising some benefits and considering constraints, such as resource usage and capacities. As the survey has shown, different definitions of specific operator placement problems exist in three different categories. As a satisfiability problem, the aim is to find any valid assignment, without considering costs or further benefits. The optimal placement problem considers finding the best solution with respect to the constraints, costs and benefits. For the heuristic problem, the focus is on the trade-off of the computational cost of finding a placement and the resulting quality of the solution.

This chapter has presented many of the characteristics which can be used to differentiate operator placement techniques. In practice, some of them primarily describe the interface or interaction with the placement logic and are less relevant to their behaviour. The survey has shown that the strategies used to traverse the search space or limit it can

be particularly unique. Additionally, both the properties which are considered and the constraints or optimisation function defined on top vary for different use cases leading to a lot of differentiation between placement techniques. Other characteristics include the optimisation of individual operators or complete applications at the same time, potential decentralisation of placement techniques or continuous optimisation by adapting placements rather than computing them once.

Many network-aware placement heuristics rely on latency estimations to improve their scalability. Two techniques can be used for this, matrix completion and spring forces-based estimations. While matrix completion provides better theoretical benefits, because they allow for triangle inequalities, existing placement techniques only utilise spring forces-based estimations. This is likely for their much simpler model, where every resource has a position in a cost space. The positions are updated by simply iteratively attracting towards or repelling from other resources so the distance in the cost space matches the measured real-world latency more closely. Additionally, the model naturally matches closer to a distributed computation, given that each node can individually estimate its position iteratively. At the same time, adaptions such as only using a few peers rather than all, known as landmarking, allow for linear scaling costs with a reduction in accuracy. Current approaches minimise the bandwidth usage or the product of the bandwidth usage and latency, known as network utilisation. Some solutions also consider the available bandwidth capacity, but this usually has to be manually specified and therefore does not consider changes in the environment, such as the usage of other users on the shared network. Overall, bandwidth usage can only be reduced by the placement of operators if operators can be co-located and therefore allow for the operator fusion optimisation.

The spring force-based estimations are also used by many approaches to identify initial neighbourhoods for the placement of an operator. Alternatively, the computational resources on network paths from data sinks to sources have also been considered. Hierarchical placement heuristics or geospatial-aware solutions also form neighbourhoods on logical or physical proximities. These approaches often define regions or are based on the idea of small computational clusters, such as cloudlets, as potential infrastructure for the fog environment.

Metrics such as MaCE and the maximum sustainable throughput explicitly model the latency or throughput of an application. These more detailed models depend on the availability of selectivities, which are available from estimation or historical data from previous executions of the application. This also assumes that general patterns and distributions in the future data to be processed match the already observed data.

Operator placement has also been compared to related problems, particularly the processing of database queries, which has also been used as an introduction in the previous chapter, and the scheduling in batch processing, which in cases like Apache Spark has also been used for stream processing with mini-batches. Similarly, virtual machines, containers and VNFs have been considered. Overall, the differences in these problems largely are not the resources or constraints considered during placements, but rather the semantics of the workloads or available information. For example, database queries know

the data to be processed in advance and stream processing receives previously unknown data. For VNFs, virtual machines or containers their behaviour might not be modelled and can only be treated as a black box. At the same time, there are differences with the expected runtime of a task and the necessary placement speed. In the case of batch processing and mini-batching, solutions such as Sparrow show that there can be a need to place thousands of short-lived tasks, which is an entirely different demand than that of stream processing. Another difference is the execution of tasks or operators. In database queries, operators can be blocking or pipelined. For stream processing, everything is pipelined and the execution in batch processing is based on stages. Virtual machines, containers and VNFs can be placed in isolation or based on communication requirements or dependencies. This creates differences in when the placement of operators is considered and the set of operators that can be considered at once.

With the insights gained from the state of the art in stream processing and related optimisation problems, the following chapter discusses the design of a heuristic for the operator placement problem.

CHAPTER 4

Heuristic Design

This chapter presents the major design considerations and decisions made for the heuristic operator placement. First, the requirements for an operator placement heuristic are analysed in Section 4.1. Section 4.2 further clarifies which assumptions are being made about fog computing, because there have been varying competing definitions in the Background chapter. Section 4.3 presents the reasoning behind using Apache Storm for the implementation and Section 4.4 then provides a more detailed introduction to its architecture and general knowledge related to the placement of operators in Apache Storm. Following the introduction of the necessary context of the operator placement heuristic, the actual components are considered. The design of the latency estimation is then specified in Section 4.5. Section 4.6 contains a simplification of the operator placement problem in Apache Storms API, as well as the formulation of the constrained optimisation problem consisting of the definition of the optimisation function and the necessary constraints.

4.1 Requirements Analysis

This section aims to specify the requirements and first major design decisions for the operator placement heuristic. Based on the previous survey of the state of the art, the design of a network-aware operator placement heuristic consists of solving three major problems.

1. A strategy to efficiently traverse or limit the size of the potential search space, while limiting the potential negative impact on the solution quality.

2. A system to efficiently and accurately estimate the latency between N computational fog resources to avoid repeatedly performing $O(N^2)$ expensive measurements.

As this is a distributed system, the need for coordination or synchronised processes between resources should be minimised to ensure scalability and to improve reliability.

3. The design of the actual placement heuristic by modelling an optimisation problem, which makes the best use of the available information and selecting or designing a solver.

Stream processing applications are ideally continuously but, in practice, only long-running tasks. As such, the actual runtime of the placement of operators is less critical than in some related problems which have been previously discussed, such as batch processing. Still, the computation of the placement should occur within a second or potentially multiple seconds for particularly complex topologies or environments. This is based on the idea that there are still some use cases for which faster placements are more relevant. For example, in interactive environments such as when developers are developing, testing and potentially submitting said topology multiple times, long wait times are unwanted. A delay of at most one second in such cases should therefore limit the potentially negative impact in these interactive environments.

The placement heuristic must support heterogeneous computational resources and be network-aware. The network is a shared resource and measuring the available capacity of network links is not only expensive but would have to be repeated periodically to account for the changing behaviour of other users. Therefore, the network usage is optimised indirectly by reducing the total amount of bandwidth required by co-locating operators, but this, of course, does not guarantee the absence of a bottleneck on individual links. As such, a valid placement has to fulfil multiple constraints and optimise the usage across various resources. To ensure long term performance, the placement heuristic furthermore needs to be adaptive and handle new placements for failed operators and computational resources.

The aim of the placement heuristic is to support a near real-time system. This is because stream processing is typically not considered as a real-time system as this requires immediate processing of the data [fTISA19]. With the involved latency to transfer data to the stream processing system and common mechanisms such as back-pressure or mini-batching, which further aggregate data and delay the processing of data, it is difficult to refer to this as immediate computations. Therefore, the definition of a near real-time system is used, which aims to provide computational results with low latency, but not immediately [fTISA19]. Furthermore, this thesis does not aim to guarantee computations within specified deadlines but instead provides a reasonable effort in providing computations close to real-time. This primarily comes down to selecting operator placements to reduce the overall network latency, as having some latency is inherent in a distributed system, and avoiding latency caused by the queueing of data when a computational resource is overloaded, such as in the case of back-pressure. As a result, this thesis intends to support stream processing applications that benefit from lower latency, but do not have strict latency requirements for their correctness.

Additionally, the latency to be directly optimised is only the network latency, as the execution times of operators are indirectly optimised by avoiding overloading a resource. The aim is, therefore, to minimise the total time needed to process an event within the application and not the response time, which would further include the network latencies of devices interacting with the stream processing application [RPD06].

The following section specifies the assumptions which are made about the fog and, therefore, provides the context and the requirements the operator placement has to be considered in.

4.2 Specification of Fog Environment

In Section 2.4, current definitions for the fog have been provided and various differences have been highlighted. These definitions primarily differ in their details to provide a full specification. The following discussion aims to clarify which definitions or assumptions about the fog are utilised or if they are even relevant for the context of this work. As such, rather than providing a full specification, the aim is to define a minimal set of necessary assumptions and argue why further considerations are not necessary.

One of the points with diverging opinions was what computational resources would be utilised and what layers they could belong to. During the discussion of existing placement techniques, approaches have been presented to separate a stream processing application into groups of operators executed in the cloud or at the edge. This minimises bandwidth usage or latency across the connections between these layers [PCSA18, VdAL18]. To optimise the overall application latency, all connections should be considered. But this, in turn, means that a strict theoretical understanding and separation of resources into the three-layered fog model is not necessary for the context of operator placements. Similarly, for the placement of an operator, only the capabilities of a computational resource matter. If the resource in question is actually a network device, a single-board computer or server matters little to the placement procedure itself, as long as their capabilities are known or measured.

One exception to this is IoT devices and other edge computing resources. Special considerations have to be made for them, given their varying reliability, mobility, battery limitations, often wireless communication and potential short-term availability. For this reason, approaches in this field typically differ largely from the regular cloud or fog placement methods discussed in the previous chapter as new solutions aim to improve their handling of these difficulties in edge or fog computing [VS20]. As a result of this greatly increased complexity, this thesis, as well as most research presented in the previous chapter, simply does not consider or utilise such resources. This supports the conceptual separation of fog computing and edge computing, although, with continued progress, researchers may also want to focus more on this challenge in the context of fog computing. Edge computing in the context of this thesis is, therefore, considered as the research into more localised networks, such as, for example, ad hoc networks and collaborative processing. Similarly, cloud computing is considered as a paradigm with an infrastructure

that solely relies on the cloud layer, while fog computing uses at least the fog layer and potentially the cloud.

Another reason for the exclusion of edge devices is based on economic considerations. A device that moves or has a battery does have additional costs over stationary ones, while these features also lead to complications for stream processing. These are the conflicts of having long-running operators with limited battery life or the interest in consistent low latency while a device is moving and potentially even losing connection. At the same time, to be able to contribute to the fog, these devices would have to have an excess of resources while in use that is still large enough to be worth the extra effort. Even if the investment into new devices or repurposing of available existing devices is deemed economically to integrate into the fog, then this still does not mean that stream processing applications are their best or easiest method of utilisation. In this sense, stream processing is a paradigm competing for the edge resources with other paradigms. In comparison, batch processing typically does not require continuous connectivity to transfer data, low latency or even long-running tasks on a single resource. This means that batch processing or an adaption such as micro-batching are examples of paradigms that are more closely aligned with the capabilities of such edge devices. As a result, using edge devices for these paradigms should be easier or more effective, ultimately resulting in the acceptance of higher costs than in the case of stream processing.

To summarise, devices that could be utilised in this theoretical fog environment and will be considered for placements are ones with higher expected reliabilities and not additional complexities, such as movements or battery limitations. How powerful, what device they are exactly or where they could exist in the three-layered model is less of a concern as these are considerations that likely will change over time or simply be established by the first large-scale and successful fog environment installations. Concrete examples of what these devices could be, therefore, computationally weaker devices, such as routers or Raspberry Pis, individual servers to clusters, entire data centres and clouds from cloud computing itself.

With the environment of fog computing and the requirements for the stream processing operator placement clarified, the next section discusses the selection of the stream processing framework for the implementation.

4.3 Stream Processing Framework Selection

The selection of the stream processing framework is discussed at this point because while these frameworks share many commonalities, they also have various uniquenesses in details that have to be considered during the heuristic design. This section aims to clarify why Apache Storm has been selected for the implementation. As such, the aim is not to provide detailed introductions to each framework, but mainly a comparison of their largest advantages and disadvantages for the context of this thesis.

4.3.1 Apache Storm

Apache Storm is a stream processing framework that has been very popular for many years. Its staying power has allowed it to form a large community, bringing a wealth of information and additional tooling to the project. Additionally, the framework is used in production in companies such as Twitter [TTS+14].

For this thesis, a framework is necessary which provides low latency, so that the effect of different placements on latency is clearly noticeable. Apache Storm is generally known as a framework that provides good latency metrics, but it is usually outperformed by Apache Flink [CDE+16, KRK+18]. Apache Storm has continued improving its latency and made it one of the large re-engineering topics for the release of version 2.0. With this release, it was claimed that Apache Storm is the first stream processing engine to achieve latencies below one microsecond, but the scientific community has not yet done any larger benchmarks to study the real-world performance [Apa19].

Additionally, Apache Storm not only provides interfaces to define new placement heuristics but also ones to extend the already existing strategies. This has made Apache Storm a popular choice for testing operator placements [ABQ13, XCTS14, PHH+15, MAI21, QR21]. In turn, the placement of operators has been improved over time, in part by including work suggested by researchers [Apa22g]. All of these beneficial properties have led to using Apache Storm for the implementation of this thesis, as many other researchers have similarly decided before.

4.3.2 Popular Frameworks

This section provides an overview of popular alternatives to Apache Storm, which have also been considered for this thesis.

Apache Spark

Apache Spark is designed as a batch processing framework that also offers stream processing. As such, what is known as stream processing in the framework are micro-batches, which are collected and then processed as independent batch processing tasks, before reassembling them into a stream [ZDL+12]. This difference has already been discussed in Sections 3.2.2 and 3.3.2. It is a closely related paradigm with similar capabilities, but ultimately different to stream processing. Apache Spark, therefore, has a different placement problem because short-lived independent tasks rather than continuous streaming operators are placed. It also provides worse performance for latency, especially in the case of the minimal observed latency, as this includes the time to aggregate the next micro-batch [KRK+18]. For this reason, Apache Storm has been selected over Apache Spark. Apache Spark offers interfaces for placement mechanisms, thereby supporting the ability to perform such research, but they are not as conveniently available as the ones of Apache Storm. In 2018, a new execution engine was added to allow sub-millisecond latency, referred to as continuous processing rather than micro-batching, but it is still considered an experimental feature with various limitations [Apa18, Apa21].

Apache Flink

As discussed previously, Apache Flink is a framework that has always performed very well in benchmarks for latency [CKE+15]. Unfortunately, modifying the placement logic is a lot less accessible than in other frameworks, which makes it less desirable given that the placement logic is the focus of this thesis.

Of course, there is also a variety of other popular frameworks or even cloud-specific services, such as for example Google Cloud Dataflow [Goo22], Amazon Kinesis Data Streams [Ama22], Apache Samza [Apa22b] and Apache Pulsar [Apa22a], but these do not provide any advantage for this thesis over the already mentioned frameworks. Similarly, these frameworks are usually not used in previous research on operator placements which, in turn, would further limit the comparability of any result this thesis achieves.

4.3.3 Research Frameworks

Another group of frameworks that have been considered as an alternative to Apache Storm are research frameworks. These are either developed with specific aims or to provide a more adjustable platform for experimentation. As a result of their smaller communities, they often have less well-established tooling and documentation, if the framework itself is available for access. ProgCEP, which has been explicitly designed to support the research of operator placement algorithms, is not publicly available [LK19]. As such, while research frameworks can provide certain advantages for the work on this thesis, they also present a trade-off as they typically have a more specialised focus and are more experimental in their setup. While frameworks such as VISP have been considered, they have been evaluated to not provide sufficient benefits for this task over ones already regularly used by the industry [HVWD16]. The availability of extensive documentation and the work of the supporting communities, such as tooling or available benchmarks, among other benefits with alternative frameworks, is simply more beneficial as the popular frameworks in use already fulfil the requirements for this thesis. Additionally, while research frameworks are often easier to extend and adjust with their smaller scope, documenting any difficulties when performing the same task in more established solutions can provide additional insights.

Following the selection of Apache Storm as the framework for the implementation, the next section introduces Apache Storm in detail with a focus on concepts related to the placement of operators. As such, this is also the final necessary element to fully specify the context for the design of the operator placement heuristic.

4.4 Overview of Apache Storm

This section provides an overview of its specific terminology and the architecture of Apache Storm's services. As such, this section defines the context of the placement problem and presents some related details, in particular, how concurrency is handled in Apache Storm and its significance to the placement problem.

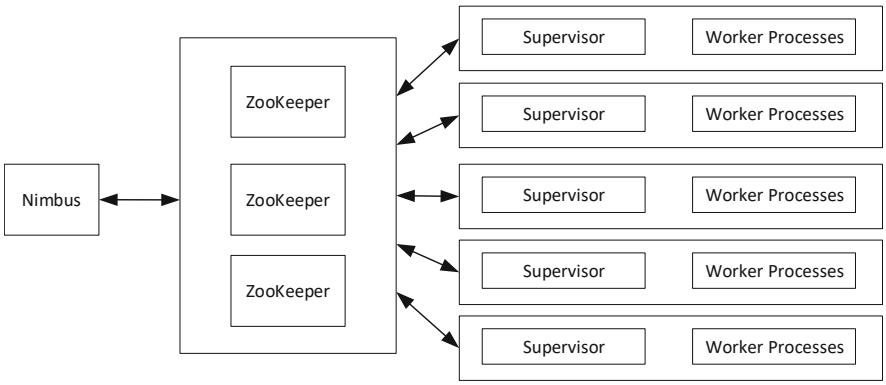


Figure 4.1: Storm Architecture consisting of Nimbus, ZooKeeper nodes and supervisors with worker processes [TTS+14].

4.4.1 Apache Storm Architecture

Apache Storm uses a naming scheme that diverges from the traditional operators, data sources and sinks naming for components in stream processing topologies. Data sources are called spouts, operators are bolts and data sinks are also referred to as bolts. Meanwhile, custom logic for a data sink is defined by implementing a new sink. When spouts and bolts are executed, they are referred to as tasks rather than operators [TTS+14]. To avoid confusion, this thesis will continue to primarily use the more widely accepted and previously defined terminology of operations, operators, data sources and data sinks.

The architecture of a deployed Apache Storm cluster is showcased in Figure 4.1. Nimbus is the master of the cluster. It handles many of the management tasks and provides an API to control the cluster. Especially relevant for this thesis, the placement of operators is also a responsibility of Nimbus. There can only be a single active Nimbus instance, but in the case of a failure, the rest of the cluster can continue operating normally. Tasks, such as the deployment of new stream processing applications, can then only be executed once Nimbus has been launched again [TTS+14].

Multiple ZooKeeper nodes in the cluster store state and configuration information reliably and thereby facilitate coordination between Nimbus and the supervisors. As such, Nimbus and the supervisors write modifications of the state or the configuration of the application to Zookeeper. From there, it can be read by any of the services during the initial startup, recovery from failure or simply to retrieve updated information. Any required information for the execution of stream processing applications, which is not already stored in the configuration or on local storage, is therefore stored in and read from Zookeeper to allow for failure tolerance. Both Nimbus and the supervisors are designed to fail fast and stop in case of an error. Because their state is stored in Zookeeper or the local storage, they are considered stateless and can be recovered by restarting them [TTS+14].

Each supervisor consists of a supervisor process and the worker processes. The supervisor then manages the communication and the state of its worker processes running on the

same computational resource. The worker processes handle the execution of operators of the stream processing applications. Each worker process uses a pool of threads, called executors, where within the operators are placed. Each executor is used as a potential slot for the placement of operators and one or multiple operators can be placed within a single one. Internally in Apache Storm, the worker processes are referred to as *WorkerSlots* and can be easily confused with the *ExecutorDetails*, which actually refer to the operators or tasks to execute. During a placement, one or multiple *ExecutorDetails* from the same topology can be assigned to a single *WorkerSlot*. The worker then uses threads for the execution of the *ExecutorDetails* as necessary [TTS+14].

A minimal deployment of an Apache Storm cluster, therefore, consists of one Nimbus, ZooKeeper and supervisor instance each. In practice, additional ZooKeeper and supervisor instances would be deployed to improve the reliability of the overall cluster, even if the performance from additional computational resources would not be necessary. This also applies to extra instances of Nimbus, which can be used as a backup service to fail-over to as there can only be a single leader. Storm UI is a Web service typically deployed along the cluster to provide a user interface to manage the cluster via the Nimbus management API [Apa22i].

This section has introduced supervisors and their model that allows for concurrency. Workers are individual processes that each manage a pool of threads, with each thread known as an executor. The following section discusses more closely how Apache Storm decides the level of concurrency during the execution of an application and related limitations.

4.4.2 Concurrency in Apache Storm

Apache Storm does not support dynamically adjusting the number of operators, or tasks, within a topology. Instead, this is a static number defined during the initial submission of the topology to the cluster. As a result, there is an inherent upper limit to the concurrency and scalability of a launched topology, which can only be increased by stopping and relaunching the topology. Furthermore, all operators or tasks within the topology must be executed to ensure all data is being processed [Apa22m].

In practice, this means that the actual level of concurrency is defined purely by the placement of operators. A placement mechanism can place all operators on a single worker process, thereby only utilising the concurrency available to that worker, or spread the operators across all workers to maximise the concurrency within the limitations of all available computational resources. Apache Storm allows users to define parallelism hints for the placement logic, which define how many executors should be utilised to execute all instances of an operator [Apa22m].

These hints can also be updated during the execution of a topology. This is known as rebalancing and can change the concurrency of the active topology by engaging the placement logic again. These hints are only a guideline. A placement mechanism may

create a different level of concurrency because the hints may be unachievable due to other constraints or because they are simply ignored by the implementation [Apa22m].

An alternative to manually triggering rebalancing is adaptive operator placements. The following section presents the state of support for adaptive placements in Apache Storm.

4.4.3 Adaptive Operator Placements in Apache Storm

Apache Storm does support adaptive placements. The engine periodically triggers the placement logic completely independent of any operators or topologies actually requiring placements. This allows developers to adapt the placement with each of these cycles or, for example, to implement QoS constraints that trigger a placement computation if they are close to being violated. Still, there are also two major limitations to the adaptivity of Apache Storm:

- As the previous section has discussed, Apache Storm does not natively support adjusting the number of tasks or operators in a topology. As such, performing scale-outs and -ins are not possible. Cardellini et al. have previously extended Apache Storm to add support for this and thereby extend the potential elasticity, but the modifications have not been integrated into the official Apache Storm version yet [CNL16].

- Zhang et al. found that Apache Storm manages its executors on the process rather than the thread level. When tasks are migrated, Apache Storm stops and restarts the worker process and the related threads. This is not only inefficient and slow, with interruptions caused in the range of seconds, but also means that operators can be affected by a migration process even though their placement does not change. Zhang et al., therefore, modified Apache Storm to manage operators on the thread level and avoid the need to stop and restart the worker process or operators in other threads that are not migrated [ZJW+19].

While these limitations do reduce the potential design space and lead to large short-term inefficiencies in the migration of operators, they ultimately do not hinder the development of an adaptive operator placement mechanism. Similarly, future updates of Apache Storm may rectify these issues, especially when considering that solutions have already been proposed. In the past, Apache Storm, for example, did not have support for the migration of operator states for which changes have been proposed that are now implemented in Apache Storm [CNL16, Apa22k]. The aim of creating an adaptive operator placement can, therefore, not only be realised but may even see further improvements in the future.

The following section discusses the first problem of the design of the placement heuristic to solve: estimating the network link latencies.

4.5 Network Link Latency Estimation

Apache Storm does not provide extensive information on the latency from transfers of data. Therefore, these measurements between supervisors need to be collected additionally. The latency estimation does not have to be directly integrated into Apache Storm and is therefore designed to be an independent component to be executed along with it.

For the estimation of latencies, an approach using spring-based forces is used. It is primarily inspired by Google's landmarking-based implementation [SPPvS08] and SBON [PLS+06], which are also discussed in Section 3.2.2. The use of a spring-based estimation rather than a matrix completion is preferred because a spring-based estimation naturally leads to a highly decentralised design without the need for coordination, which is a defined requirement. Each supervisor independently estimates its coordinates in a three-dimensional cost space, the latency space. In this space, the distance between two coordinates directly correlates to the estimated latency between the points. The coordinates are initially randomised and are then periodically adjusted based on latency measurements with other peers [PLS+06].

The pseudocode to execute one of these periodic estimations is presented in Algorithm 4.1. To decentralise the estimation, a supervisor adjust its position by assuming the positions of other peers as fixed (Lines 1-3). Spring-based forces are then applied between the supervisor and its peers to pull the supervisor into a position closer to the measured latencies (Lines 8-12). This is iteratively repeated until the total movement across all peers becomes insignificantly small or a maximum amount of iterations is reached (Line 6) [PLS+06]. The iteration limit is mostly relevant when a supervisor estimates its position for the first time or when large changes in the network occur. Otherwise, the previous position is likely already close enough to the new estimate, resulting in smaller adjustments and early cancellation of the estimation process. In the field of graph drawing, an iteration limit of 100 was deemed sufficient and has, therefore, also been used in this work [Ead84, Kob12]. At this point, a supervisor has locally estimated a position that minimises the error of the estimated latencies and of the real-world measurements [PLS+06]. The new position is then returned from the function (Line 16).

Outside of the presented pseudocode, the supervisor then shares its new position in a Key-value database and waits until it starts the latency estimation process again to further refine its coordinates with new measurements or updated positions from its peers. These wait times are slightly randomised to prevent peers from potentially being synchronised in their waiting and, therefore, both regularly updating in parallel based on each others' old data. 60 seconds were used as the intended wait time between estimations, except for the first five rounds, in which it was reduced to four seconds. Alternatively, this coordinate store could have been implemented with a more scaleable peer-to-peer-based approach, but it was not because of the increase in complexity. Additionally, the workload is already small for an entire cluster, so the increased centralisation is not a problem.

To calculate the spring-based force in Algorithm 4.2, a unit vector of the distance between the supervisors is calculated (Lines 1, 3). The force is defined as $c_1 * log(d/c_2)$,

Algorithm 4.1: Pseudocode for one estimation update of a supervisors position in the latency cost space.

```
   Input: estimatedPosition=current coordinate, minEpsilon=minimum required
          movement to continue iterating
   Output: updated estimated position
 1 peers=peerSelection()
 2 getUpdatedPeerPositions(peers)
 3 measurePeerLatencies(peers)
 4 maxMovement=MAXFLOAT
 5 iteration=0
 6 while maxMovement>minEpsilon and iteration<100 do
 7 |   movement=Vector(0,0,0)
 8 |   foreach peer in peers do
 9 |   |   d=calculateForce(peer,estimatedPosition)
10 |   |   movement=movement+d
11 |   end
12 |   estimatedPosition+=movement
13 |   maxMovement=movement.length
14 |   iteration++
15 end
16 return estimatedPosition;
```

with c_1 as a constant, c_2 as the ideal distance, the measured ping, and d as the distance in the latency space (Line 2). This force is then applied in the correct direction by multiplying it with the unit vector (Line 4). The logarithmic scale prevents the forces between very distant positions from becoming too large in comparison to the smaller forces [Ead84, Kob12]. Additionally, a variant using the linear force calculation `force=multiplierConstant*(d.length-peer.latency)` based on SBON and its use of the Vivaldi algorithm is also being evaluated [PLS+06, DCKM04].

Algorithm 4.2: Pseudocode to calculate the movement based on spring-based forces [Ead84, Kob12].

```
   Input: peer, estimated Position
   Output: d=movement update vector
1 d=peer.position-estimatedPosition
2 force=multiplierConstant*log(d.length/peer.latency)
3 d=d/d.length
4 d=d*force
5 return d;
```

Similar to Google's landmarking implementation, not all supervisors but only a small subset of them are used as peers [SPPvS08]. In contrast, rather than selecting specific peers, n random peers are used. After an estimation, only i peers with the lowest latency are kept and j new random peers are selected, such that the full selection of n peers exists again. This specific selection with a bias for more local resources is made because the accuracy of latency estimation for the closest resources matters the most for the operator placement. This is because these resources are far more likely to interact with each other once a placement occurs. After all, they offer low latencies. A network-aware operator placement heuristic that aims to reduce latencies, then, of course, aims to make use of these resources. In contrast, the accuracy of the global latency estimation is less important because having more distant resources interact hinders the aim of achieving lower latencies. Still, j random peers are used to ensure that the global estimation is still somewhat accurate, even if ideally less relevant.

The actual latency measurement is performed using the natively installed ping utilities on Windows or Linux because a Java-based implementation would require extended privileges during the execution to perform ICMP pings rather than relying on a TCP fallback [Ora22]. Pings are a measurement of the round-trip time and, as such, can not measure potential differences in the directions of a link. This is an additional reason for the selection of a spring-based estimation rather than matrix completions. As discussed in Section 3.2.2, a spring-based estimation can only model symmetric and not one-directional latencies, but a matrix completion can. If one-directional latencies are never measured, to begin with, then modelling them also does not provide a benefit. To approximate one-directional latencies, the round-trip time is halved so that if the latency of an Apache Storm topology is estimated, it resembles the actually observed latency more closely. Pings are handled by the operating system, CPU efficient and are therefore even relatively accurate under heavy loads [SPPvS08]. This could be the case when a node, for example, might be overloaded by a stream processing application. Of course, pings can still have high variance depending on the actual network conditions. Therefore, the measurement is repeated multiple times and the median of them is computed to return a more stable measurement. Using such aggregate filters is a common practice. Google's landmarking implementation also utilises the median and SBON uses a moving percentile filter. Google uses a history of measurements for the calculation and Pietzuch et al. found that the aggregation of ten measurements allows for stable latency estimation [SPPvS08, PLS+06]. For this reason, the computation of the median uses ten total measurements: the five current measurements and the last five previous measurements. Including the last measurements provides some additional short-term stability similar to Google's measurement history.

With the understanding of how the latency is measured and the previously discussed architecture and limitations of Apache Storm, the modelling of the optimisation problem is discussed next.

4.6 Modelling of the Constrained Optimisation Problem

The modelling of the constrained optimisation problem consists of three main components: a simplification, the optimisation function and the constraints. The following sections each describe one of these components.

4.6.1 Simplification

The first step in efficiently solving the placement problem is a simplification. The separation of workers on a supervisor in Apache Storms architecture is mostly a logical one. Each worker has access to all resources of a supervisor. Similarly, multiple workers are not necessary to achieve concurrency because workers are already pools of threads, as discussed previously. The documentation of Apache Storm even recommends only using one worker per topology on a supervisor, as multiple workers would introduce additional overhead for the communication between processes [Apa22c]. With this performance consideration in mind and without any requirement to distinguish between workers, there is no need to model each individual worker in the placement.

Instead, operators can be assigned to a supervisor if one of its workers is already occupied by the same topology or at least one worker is still free. This means the worker instances can instead be modelled as a number of available topologies that can be run on the supervisor. In a later step, the actual assignment from the supervisor level to the worker level is trivial to execute.

This greatly simplifies finding assignments and co-locating operators, which also raises the question of why Apache Storm models and exposes individual workers to the placement logic. At the moment, the *DefaultScheduler* and *ResourceAwareScheduler* in Apache Storm seems to attempt to achieve this result in some way, although each one has a unique implementation for it [Apa22n]. Additionally, while Apache Storm does allow scheduling multiple topologies at the same time, they are considered individually by this placement heuristic to simplify the problem. This is also the case with all existing schedulers in Apache Storm, specifically the *DefaultScheduler*, *IsolationScheduler*, *MultitenantScheduler* and *ResourceAwareScheduler* [Apa22n, Apa22h].

Following this simplification, the actual optimisation problem can be defined starting with the scoring function.

4.6.2 Scoring Function

With the previous design decisions in mind, latency is a core component to be considered. The placement problem definition has been set up as a score minimisation problem because latency can grow arbitrarily large and is considered a key metric to optimise. The following therefore presents the optimisation function that scores a potential operator placement of a single topology and shall be minimised to achieve better performing placements.

The optimisation function shown in Equation 4.1 consists of four components with additional weights w_1 to w_4 to modify the amount of influence each specific component has. Each of the scoring components has a range of $[0, 1]$, except for the latency function $s_{lat}(x)$ with a range of $[0, \infty]$. For this reason, w_1 was set to $\frac{1}{1000000}$ such that it becomes a relatively insignificant part of the placement that mostly acts as a deciding factor between similarly scored solutions. All other scoring functions were equally weighted by setting the constants $w_{2..4}$ to 1.

$$s(x) = w_1 * s_{lat}(x) + w_2 * s_{sup}(x) + w_3 * s_{co}(x) + w_4 * s_{event}(x) \tag{4.1}$$

$s_{lat}(x)$ is representative of the highest estimated network latency that would be accumulated during the processing of an event in the topology T. As such, for any operator $s \in T_{source}$ we can define that $s_{lat}(s) = 0$. For other operators in the topology, this can be extended to $\forall o \in T : s_{lat}(o) = max(s_{lat}(p) + l_{p,o} : p \in o_{predeccesors}))$ with $l_{a,b}$ being the estimated network link latency from the operator a to b. The latency score of the topology is then simply the maximum among all sinks: $s_{lat}(T) = max(s_{lat}(s) : s \in T_{sinks})$

$s_{sup}(x)$ is used to condense the placement of operators. It is defined as $s_{sup}(x) = \frac{|supervisors \in x|}{|supervisors|}$. The idea behind this score is to reduce the total amount of supervisors used. Unused supervisors could then be temporarily shut down to optimise the cluster by reducing unused resources.

$s_{co}(x)$ is a measure of the co-location of operators. If $p(o)$ is the placement of an operator o then $s_{co}(x) = \frac{|e_{o_1,o_2} \in T : p(o_1) = p(o_2)|}{|e_{o_1,o_2} \in T|}$

$s_{event}(x)$ was originally intended as a score to minimise the bandwidth usage directly by forming the fraction of the placement's bandwidth usage and the theoretical usage if no operators are co-located. During the implementation, there were various difficulties in accessing accurate bandwidth information that are described in the implementation section of this thesis. Therefore, rather than measuring the bandwidth used, it was adapted to events emitted over not co-located edges. This metric is, therefore, highly similar to the co-location and with $t(e)$ being the events emitted on an edge e it can be defined as $s_{event}(x) = \frac{\sum_{e_{o_1,o_2} \in T : p(o_1) \neq p(o_2)} t(e_{o_1,o_2})}{\sum_{e_{o_1,o_2} \in T} t(e_{o_1,o_2})}$. In practice, $t(e)$ can not be measured exactly because Apache Storm only outputs a metric about the emitted events for each operator. $t(e)$ can, therefore, only be approximated by dividing the operator-based statistic by the count of outgoing edges. This metric can, as a result, not account for any skew in the distribution of data to succeeding operators. This is furthermore not ideal because Apache Storm allows defining custom groupings that affect how data is distributed to the operators. Given the practical limitations, a more accurate score can not be calculated because the data is not available at this level of detail.

With that said, even if the size of tuples is unknown, there can still be a significant difference in the number of tuples an operator outputs, because of their potentially different selectivities, as previously discussed in Section 2.5. These differences in the selectivity also

accumulate across the different paths in a topology and therefore necessitate this metric in addition to $s_{co}(x)$. At the same time, it does not fully replace $s_{co}(x)$. This is because the co-location of operators and the resulting removal of overhead and synchronisations was generally one of the most important factors for the actual performance. As such, $s_{co}(x)$ optimises the general performance and $s_{event}(x)$ ensures that the most impactful operators are co-located.

The following section discusses the modelling of the constraints, of which the CPU utilisation is also important for the topologies' performance.

4.6.3 Constraints

Most constraints for the placement problem are relatively basic and do not need to be explicitly checked, because the placement heuristic can indirectly guarantee them. These are the constraints of having each operator assigned to exactly one supervisor and not assigning operators of more topologies to a supervisor than workers are available, because each worker can only execute operators of a single topology. For this reason, they are not explicitly modelled.

The only constraints that need to be checked are memory and CPU usage violations and that no more topologies are assigned to a supervisor than it has workers. If the memory of all operators that are placed on a supervisor exceeds its capacity, then that can result in crashes and the supervisor, therefore, causes a constraint violation of the excess assignment relative to its capacity.

The CPU usage constraint is more involved because it slightly differs for a currently running topology and when searching for potential new placements. Initially, it was attempted as a part of the scoring function to prioritise placements with high average utilisations while leaving some available processing capacity to handle spikes or changes in demand. This was difficult to tune and the utilisation is a highly sensitive metric to changes in the input rate or simply moment to moment, thereby resulting in fluctuating scoring that led to observations of unstable placements in some experiments. In these cases, the placement heuristic would repeatedly update the placement to adjust to the minor utilisation changes. It was therefore modelled as a constraint, which made it easier to tune, and differently defined for current and new placements to improve stability.

When a topology is already placed, then observing a high utilisation is good in the sense that resources are being effectively utilised. For this reason, the utilisation of a supervisor is allowed to reach up to 95% before it is considered a constraint violation. This ensures some headroom for minor usage spikes because utilisation is being considered over a time frame while avoiding overutilisation that leads to events needing to be queued and therefore delayed at the operator. When searching for new placements, it was instead found to be beneficial if a larger buffer is available in the CPU utilisation. This is first off because the real utilisation might be growing currently due to increasing input topology rates and would, therefore, otherwise necessitate a new placement in the near future. Secondly, requiring a lesser utilisation in new placements effectively creates a range

of acceptable utilisations and can therefore improve stability by avoiding downscaling. Requiring utilisation of at most 50% on any supervisor in new placements was found to provide more stable placements while ensuring high utilisation. This is again because there might be usage spikes. Similar to the memory constraint, utilisation by which the 95% or 50% utilisation is exceeded relative to the intended amount is the amount of the violation.

The scoring can therefore be considered purely as a force to find smaller, more co-located placements, while the constraints oppose this by ensuring a lower bound. This usually leads to supervisors being utilised slightly below 50% on new placements, as otherwise, a more co-located placement with higher utilisation would likely be possible.

With the optimisation problem being fully defined, the following section discusses the implementation to solve the placement problem.

CHAPTER 5

Heuristic Implementation

This chapter presents how the design has been realised into a functional implementation. Section 5.1 discusses how Apache Storm's architecture was extended to support the additional data collection and the implementation of these external components. Topological sorting and Apache Storm's capability to execute cyclic topologies are presented in Section 5.2. Section 5.3 summarises the actual implementation related to solving the previously defined constrained optimisation problem. Finally, Section 5.4 presents a more detailed insight into the state of Apache Storm's scheduling API based on the experiences and insights gained while working with it.

5.1 Solution Architecture

This section aims to provide an overview of the implementation by introducing the architecture of the solution. It is discussed how the placement heuristic can be integrated into Apache Storm and how Storm's architecture has been extended with the deployment of the solution.

5.1.1 Heuristic Placement Architecture

The heuristic placement implementation was developed for the recently released version 2.4.0 of Apache Storm [Bip22]. It was also partially developed during the versions 2.2.0 and 2.3.0, thereby providing some additional insights into the placement-related changes of Apache Storm over time.

Apache Storm provides two Java interfaces to implement a placement logic. The *IScheduler* interface allows the implementation of a completely custom scheduler. In contrast, the *IStrategy* interface allows creating only a new placement logic for the Resource Aware Scheduler, thereby reusing some of its logic [Apa22g]. At their core, both interfaces are very similar by providing slightly different definitions for a *schedule* function and,

otherwise, some configuration or clean-up functions. The Resource Aware Scheduler itself has originated from researchers, previously under the name of R-Storm, and the scheduling implementation in Apache Storm has also been improved by researchers over time [Apa22g, PHH+15].

The *IScheduler* interface has been chosen for the implementation of this thesis, because the *IStrategy* interface scheduling functionality is only executed during the initial submission of a topology or when other external changes occur. While the *IScheduler* interface does not provide complete control over when a placement is computed, it is at least called periodically and therefore allows implementing an online scheduler, which can reliably adapt the placements regularly. Furthermore, the *IStrategy* interface does not provide any extensive utility for the development of a prototype which the *IScheduler* does not also provide. For example, both interfaces support the definition of CPU and memory constraints and accessing the current resource usage during the scheduling. One advantage the *IStrategy* interface of the Resource Aware Scheduler does provide, is the ability to use a different strategy for each topology, but while this may be interesting for a production environment, it is of little use for the development of the prototype.

The steps to deploy a custom *IScheduler* or *IStrategy* implementation are relatively similar. These are further complicated by information on the process being spread across many parts of the documentation without references to each other. Additionally, any error reporting for these steps is minimal, if existent at all, in the logs. Therefore, the following steps provide a summarisation of the most important information for a successful deployment:

1. First, the implementation has to be compiled and combined with non-Storm dependencies to create a packed jar file, which is the same process as packaging a topology before submission to a cluster for its execution.

2. The jar file has then to be deployed on all Nimbus instances. To do this, the file has to be copied to the *extlib-daemon* folder in the Apache Storm installation folder on Nimbus instances, unless another path for it has been configured.

3. Either as a command-line argument for Nimbus instances or in the *conf/storm.yaml* configuration file, the scheduler to be used has to be specified with the fully qualified class name. In the case of an *IStrategy* implementation, this has to be set to the Resource Aware Scheduler with the class name as *org.apache.storm.scheduler.resource.ResourceAwareScheduler*.

4. For a custom *IStrategy* implementation, the usage of this strategy has then to be specified in the startup code of every topology using *setTopologyStrategy(className)*.

Additionally, to access metrics from the operators, these have to be collected and made available to the scheduler on Nimbus. To accomplish this, a very basic implementation of the *IMetricsConsumer* interface sends all metrics to ZooKeeper, where they are accessible

by the scheduler. The deployment of this component is very similar to a scheduler and consists of the following steps:

1. First, the implementation has to be compiled and combined with non-Storm dependencies to create a packed jar file.

2. The jar file has then to be deployed on all supervisor instances. To do this, the file has to be copied to the *extlib* folder in the Apache Storm installation folder on supervisor instances, unless another path for it has been configured.

3. The metrics consumer can either be configured for an individual topology in the startup code of the topology or by registering it in the *conf/storm.yaml* configuration file. The modified configuration file has to be deployed on all supervisor instances.

The last component is the latency estimation between hosts of supervisors. It is simply installed and executed along with any supervisor, but operates independently of them. To transfer this data to Nimbus as well as exchange data to compute the estimations, Redis, an in-memory key-value store, is used [Red21]. For reliability purposes or improved scalability, a Redis cluster could be deployed. For this research, a single instance has been chosen because a failure of an instance is unlikely within an experiment and it can simply be repeated in comparison to a live production environment. Additionally, the actual workload Redis has to handle is comparably very small, and as such, even a single instance should not become a performance bottleneck, even for very large clusters. Redis has been chosen over the reuse of ZooKeeper, because there is no need to persistently store the latency data. Furthermore, ZooKeeper is then solely used and configured by Apache Storm components. The architecture thereby resembles the previously introduced minimal Storm architecture more closely.

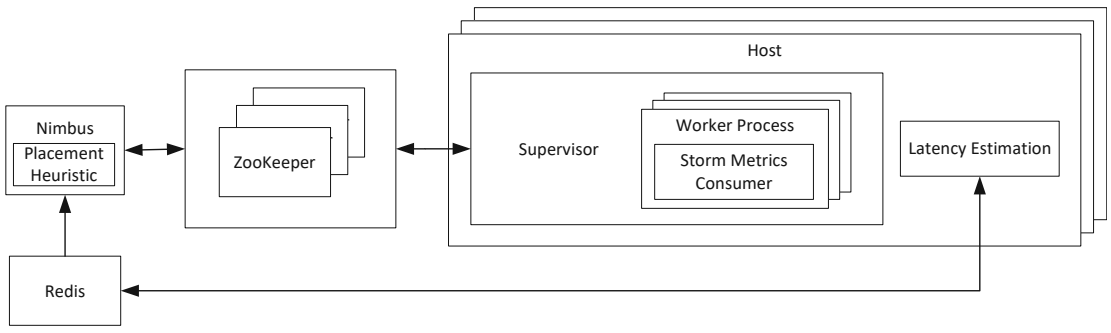


Figure 5.1: The heuristic placement architecture and communication model, including the regular Apache Storm architecture: The extended architecture primarily adds a Redis instance and the latency estimation component on the hosts of supervisors. It also shows that a metrics consumer has been configured for storm supervisors to gain access to various statistics during the execution of a topology.

The complete architecture and communication model is therefore summarised in Figure 5.1. In addition to Storms' regular communication between Nimbus, Zookeeper and the supervisors, there are three additional flows of data related to the operator placement heuristic that is being executed on Nimbus. When the Storm Metrics Consumer receives metric data from operators, it is written to Zookeeper. Additionally, periodically the latency estimators retrieve the list of potential peers and their positions from Redis, use this data to adjust their own position estimation and store it on Redis. Finally, the placement heuristic on Nimbus only retrieves the data from ZooKeeper and Redis when a placement occurs, resulting in one-directional data flows. These three data flows, therefore, do not require synchronisation and the components do not directly interact with each other because Redis and ZooKeeper store the data until it is being used.

Otherwise, it is also noteworthy that, in theory, multiple supervisors can be deployed on a single host, although there are few actual reasons to do so. This is because the number of worker processes of a supervisor, and therefore its scalability, can be changed by updating its configuration. Similarly, using additional supervisors would add the overhead of multiple supervisors and reduce the leftover resources for the workers and any actual stream processing. The metrics consumers in Apache Storm are executed in workers, but are not directly components of them. Where exactly metrics consumers are executed is not consistent, because they are internally treated as operators of a topology with other operators configured as inputs to them. The metrics consumers are therefore placed similarly to all other operators and can also be replicated. As such, a worker may host no metrics consumer or even multiple metrics consumers from one topology launched in the cluster [Apa22j].

One advantage of this architecture is, that it allows for the implementation of the heuristic without requiring any modification of Apache Storm itself. All the necessary functionality already exists in Apache Storm, can be extended with well-defined interfaces, can be configured or simply added externally as another component. While this is, in principle, simple, there are some differences across the interfaces of Apache Storm. For example, metrics are not directly accessible within Nimbus, but require creating an extra component to forward this data from the supervisors. Then the id of a task from the metrics API matches to any id between the *startTask* and *endTask* variables of the *ExecutorDetails* object. This handles a single executor executing multiple operators, which is not the default behaviour, but can be changed via the configuration.

The following section continues with the architecture by briefly discussing how the operator metrics from Apache Storm are accessed, related difficulties and how the *MetricsConsumer* has been implemented.

5.1.2 Metrics Consumer

Apache Storm does not directly provide access to the metrics of a topology, which is why the *MetricsConsumer* is used to forward that data. Alternatively, Storm UI also exposes the metrics with a Representational State Transfer (REST) API, but it would have to be

repeatedly queried for potential metric updates rather than them being pushed [Apa22l]. In the end, the *MetricsConsumer*-based implementation also relies on Nimbus polling the data, because the actual metrics are stored in ZooKeeper as described in the architecture. As a result, both implementation variants of accessing the metrics are highly similar, as they require the use of optional Storm components in their deployment, be it Storm UI or the *MetricsConsumer*.

The reason for using the old API is that the *___recv-iconnection* metric or rather its *messageBytes* field has not yet been transitioned to the v2 API. Similarly, these seem to not be reported in Storm UI. The *messageBytes* were intended to be used by the heuristic that has been implemented to estimate the bandwidth usage of a topology. Unfortunately, they are only reported if operators are not co-located and, therefore, unavailable in general. An alternative in which the initial placement avoids co-location to collect this metric was also attempted but ultimately was reconsidered due to the inconsistent collection of this metric. The main idea was to identify the overall bandwidth usage to estimate the average size of individual events for each operator. This could then be used to estimate the bandwidth, even if operators are co-located, by using the statistic of emitted events. The slow collection occurs particularly because the *___recv-iconnection* metric is only reported every 60 seconds, which is the standard in Apache Storm. For the purpose of online scheduling, the metrics collections interval has been reconfigured to ten seconds to match Storm's scheduling interval. The configurations *storm.cluster.metrics.consumer.publish.interval.secs*, *topology.builtin.metrics.bucket.size.secs*, *topology.v2.metrics.tick.interval.seconds* and *executor.metrics.frequency.secs* were adapted accordingly. Once the bandwidth usage of an operator has been measured, it could be used to calculate an average event size, such that the bandwidth usage can then be estimated even if the operator is co-located. Because of the difficulty of reliably acquiring up-to-date bandwidth information and the need for an initially bad placement, the decision was made to approximate bandwidth usage purely by considering the number of emitted events, as presented in the design.

One limitation of Apache Storm is that, by default, the CPU usage is not monitored. Instead, it provides the capacity metric, which is the percentage of time an operator is not idle. Considering that best practices avoid the implementation of operators with blocking functions, this directly means that capacity can directly signify some resource bottleneck. The cause of this bottleneck could be another resource, such as the storage or network, but in the most general case describes the CPU usage or its idle time. For this reason, the scoring function and constraints from the optimisation problem definition are implemented using the capacity metric rather than the actual CPU usage. Capacity is, of course, a metric that is only relevant to the current computational resource. It is therefore transformed into a general CPU cost to ensure the support of heterogeneous resources. The capacity of the operator is multiplied by the supervisor's CPU capacity to compute the CPU cost of the operator. This achieves point costs that can be considered in conjunction with point-based capacities of supervisors that are manually defined using the existing functionality of Storms Resource Aware Scheduler [Apa22g, PHH+15]. The manual supervisor CPU capacity configuration could be avoided by executing a

benchmark on the start-up of the service to measure it directly instead. Prior to version 2.3.0 of Apache Storm, the capacity metric also had to be calculated manually even if Storm already used it in its interface [Eth21]. The custom calculation for it in version 2.2.0 was also implemented in the *MetricsConsumer*.

This collection of metrics is continued in the following section with the measurement and estimation of network latencies to create a network-aware placement heuristic.

5.1.3 Network Latency Estimation

The network latency estimation implementation directly follows the design and architecture considerations already made. Redis acts as a store for each supervisor's estimated positions and as a lookup table to find peers. As discussed during the design, the component is implemented using Java and uses the natively installed ping utility, if available, for increased accuracy in the collection of latencies. To ensure the latencies of the estimation can be mapped to the supervisors, the same method to generate the id of a supervisor in Apache Storm is essentially reused. This is simply *InetAddress.getLocalHost().getCanonicalHostName()* of the Java Standard Library, which returns the local hostname. With both the latency estimation and Apache Storm identifying supervisors by their hostname matching the data becomes trivial.

The following section discusses topological sorting, which is a key mechanism for predicting the latency of a topology using the estimated network link latencies for the purpose of scoring an assignment.

5.2 Topological Sorting

A simple method to estimate the latency of a stream processing application would be to sum or average all the individual latencies. While this can help in identifying better placements, it completely ignores how the latency is distributed across the application. As such, there could be some paths with very short or excessively long latency. For this reason, the latency of an application has been previously defined in Section 3.2.2 as the highest latency of any path through the DAG. An efficient method to calculate this is to simply set the latency to reach an operator to the maximum of all its predecessors in addition to the network link latency to the current operator. By applying this iteratively to all operators once their predecessors have been calculated, the latency experienced at the sinks can be calculated and therefore be used to estimate a complete application's latency. The order in which the latency of the operators can be iteratively calculated stays identical unless the DAG is changed. As such, this order can be calculated once rather than recomputing it for every latency estimation. Furthermore, this order is known as a topological order for which well-known solutions to compute them exist [Kah62].

In a topological order of vertices contained in a graph, a vertex is sorted to be in a position succeeding all other vertices with a path to the current vertex [Kah62]. This provides an ordering of vertices, such that for any vertex all in the graph preceding

vertices are ordered before it. The operators of a DAG can be mapped directly to the vertices in this problem. This is a helpful order for calculating metrics such as the latency of a stream processing application because this order ensures that once the latency for any operator in a DAG is updated, the preceding operators' latencies are already known and can simply be referenced. Calculating such an order is relatively cheap with an $O(|E| + |V|)$ runtime for any graph with the set of edges E and set of vertices V. To accomplish this using Kahn's algorithm, sources in the graph are iteratively removed, including all of their outgoing edges. Following vertices may thereby also turn into new sources, until all vertices in the graph have been removed. The order of the removal of vertices is then also their topological order [Kah62].

Unfortunately, this only works for directed acyclic graphs, which have been discussed so far as the theoretical foundation for stream processing. In practice, Apache Storm does allow for the definition of cycles in a topology and functions mostly correctly with them. For example, the Resource Aware Scheduler's support for cyclic topologies is specifically mentioned in the original paper [PHH+15]. However, it is not clear if this feature is actively being supported in Apache Storm. In the original Storm paper, it was briefly mentioned, but given that none of the other official sources or documentation state support for cyclic graphs and only showcase DAGs as examples, it leads to the question of whether support of this feature is still intended [TTS+14]. Since the release of Apache Storm 2.3.0, cycles are detected during the submission of a topology and presented as the message of an exception, but the submission is not aborted and the user is not being warned about this potentially causing issues or limitations [Eth21]. As such, this may simply be a warning to help developers notice a misconfigured topology quicker. This ambiguity can lead to the somewhat incorrect and common assumption that a topology in Apache Storm is based on DAGs. The one feature of Apache Storm that strictly only supports DAGs is the guaranteed message processing [Apa22d]. With cycles, an event, or other ones created by operators processing it, potentially never fully completes its processing or becomes completely acknowledged because some descendant event may not exit the cycle. This situation or problems following from it can be manually circumvented, but the end result is that topologies with cycles are only partially supported by Apache Storm. Furthermore, given that the feature is not advertised or warned about, it seems to be a rare case for stream processing in Apache Storm in general.

As such, while cyclic graphs are potentially not officially supported by the Apache Foundation, they are currently still valid topologies that any placement mechanism should be able to place correctly and, therefore, an aim of this thesis. Of course, this creates a problem with estimating the latency of a topology as a cycle may be traversed once or even an infinite amount. Depending on which extreme the behaviour of the actual stream processing application is, the optimisation of the latency in the cycle may outgrow any other metric to optimise. Given that this is less than ideal and cycles are not fully supported, a seemingly rarely used feature and infinitely cycling events are even less likely, the decision was made to support cyclic graphs by treating them as acyclic graphs for the consideration of latency. Additionally, there is also the consideration

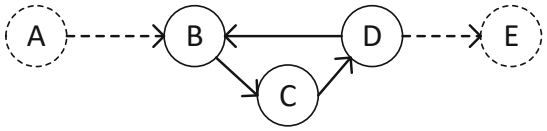


Figure 5.2: Example of a cyclic topology being processed by the Eades, Lyn and Smith heuristic. It shows a graph with sources and sinks already removed, signified by the dashed lines, and is the state at which a cycle would be detected. All nodes left in this graph have the same out- and indegree for edges. As such, which vertex is selected and which edge is therefore removed relies purely on the iteration order over the vertices. In the optimal case the edge from D to B should be removed to cause a minimal impact on the pathing or structure of the graph.

that if latency is a design target of a stream processing application, then having cycles increases the complexity of reasoning about latency, especially when considering the risk of errors during the application design or implementation. As such, designing cyclic applications would likely be avoided and restructured or unrolled, creating once again fully supported DAGs. With that said, the following discussion aims to remove cycles in a graph by selecting edges to ignore and thereby treating the graph as a DAG for the consideration of latency.

5.2.1 Eades Lyn Smith Feedback Arc Set Problem Heuristic

Removing the minimal number of edges from a cyclic directed graph to transform it into a DAG is known as the minimum feedback arc set problem. The heuristic by Eades, Lyn and Smith was implemented to accomplish this aim, because of its speed, simplicity and capability to directly output a resulting topological ordering [ELS93]. Algorithm 5.1 contains the pseudocode of the heuristic. In principle, this heuristic works very similarly to Kahn's algorithm that has been previously introduced. Rather than just removing sources (Lines 3-6), the heuristic also iteratively removes sinks (Lines 7-10) and inserts the two types of vertices into two respective stacks: the left stack and the right stack, rather than directly into a single queue (Lines 5, 9). If afterwards the graph still contains vertices, then a cycle must exist (Line 11). Therefore, the vertex v with the $max(\text{outdegree}(v) - \text{indegree}(v))$ is removed and pushed onto the left stack (Lines 12-14), thereby treating it like a source even though it has edges pointing to it. Once the graph is empty, the right stack is pushed onto the left stack to create the topological ordering (Line 17) [ELS93]. Figure 5.2 shows a simple example of breaking a cycle by removing, or actually ignoring, an incoming edge.

Alternatively, this could be implemented with a fixed-sized array, with the length of the array matching to the vertex count and the stacks could be replaced by pointers. The left and the right stack would then be replaced by two pointers, one for each end of the array, which would mark the next position to insert a vertex into the array. After an insertion, the pointer would then be moved closer to the other end of the array.

Algorithm 5.1: Pseudocode for the Eades, Lyn and Smith heuristic to solve the minimal feedback arc set problem [ELS93].

```
   Input: G=directed, potentially cyclic, graph
   Output: vertices in topological order after the minimal number of cyclic edges
           was removed
 1 left=new Stack(); right=new Stack();
 2 while G is not empty do
 3 |   while G has a source s do
 4 |   |   remove s and its edges from G;
 5 |   |   push s onto left;
 6 |   end
 7 |   while G has a sink s do
 8 |   |   remove s and its edges from G;
 9 |   |   push s onto right;
10 |   end
   |   // graph has a cycle preventing removal of more vertices
11 |   if Graph is not empty then
12 |   |   find vertice v with max(outdegree(v)-indegree(v));
13 |   |   remove v and its edges from G;
14 |   |   push v onto left;
15 |   end
16 end
17 pop entries from right and push onto left;
18 return left;
```

It is important to note, that not every operator has to be a vertex in this graph. This is because all instances or replications of an operator behave identically and can be considered as a single vertex. In Apache Storm, a component refers to the operation itself rather than all the operator instances. Therefore, the operation or component is used as a basis for the topological sorting. Once the components are sorted, they are simply replaced with their operator instances and thereby create a topological order of all operators. This process provides the benefit of calculating the order on a smaller graph and not having to, for example, resolve multiple replications of the same cycle again. In practice, it also means that calculating the topological order is mostly independent of the replication and, therefore, the deployed scale of the application, except for the linear cost at the end to replace each component with its operator instances.

The minimal feedback arc set problem only considers removing the minimum number of edges and not how this affects the pathing or structure of the graph. This does not match to the goal of calculating metrics such as latency, for which the impact on these calculations should be minimised. Therefore, the selection of the node and, in turn, edges must be adapted when cycles are being resolved. The initial example in Figure 5.2

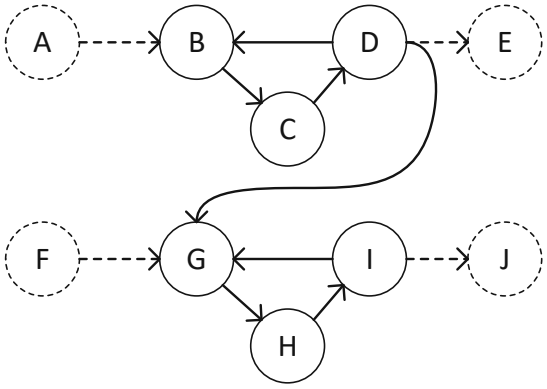


Figure 5.3: This example highlights another issue, mainly that just removing an edge in any cycle can lead to a suboptimal result. If vertex G would be selected, both the edges (D,G) and (I,G) would be ignored. A better solution would be to solve the (B,C,D) cycle first so the (D,G) edge could be naturally removed by the iterative source removal.

highlights such an example, where the best edge can be easily identified, but the heuristic may instead remove any of the edges in the cycle. This can be avoided by requiring that a vertex to be selected must have previously had an incoming edge from an already removed source. We can assume that for any cycle, at least one such node must exist because the sources and sinks of the topology are, by definition, not part of a cycle, as is any path that has been previously iteratively removed until the cycle was detected.

The graph in Figure 5.3 shows another problem once multiple cycles exist in the topology. Primarily, the heuristic should only remove necessary edges to eliminate cycles and the removal of some cycles may allow some additional vertices to be iteratively processed, rather than requiring their out-of-order removal too. This means an optimal order of which cycle to remove first needs to be found. So far, only cycles have been considered for their simplicity, but any more complex structure consisting of multiple cyclic elements must also be simplified. The important concept for such cyclic structures is that any vertex has a path to any other vertex in the structure, which is generally known as a strongly connected component [Tar72]. This is also the problem for calculating latency, because none of the vertices in such a structure can be considered as the first or last element. As such, the following section presents their detection and resolution.

5.2.2 Strongly Connected Components

Strongly connected components are well studied and, as such, algorithms to find them efficiently exist. Tarjan's algorithm is based on depth-first search and has a runtime of $O(|V| + |E|)$, to not only identify all strongly connected components in a directed graph, but also return them in inverse topological order [Tar72]. The pseudocode of Tarjan's algorithm can be seen in Algorithms 5.2 and 5.3.

The general idea of the section of Tarjan's algorithm displayed in Algorithm 5.2 is to

Algorithm 5.2: Strongconnect function for Tarjan's algorithm to identify strongly connected components [Tar72].

```
   Input: v=Vertex to start search from
   Data: Stack S, int index initially initialized to 0
   Output: outputs sets of vertices which form strongly connected components in
           inverse topological order
 1 v.index = index;
 2 v.lowlink = index;
 3 index = index + 1;
 4 push v onto S;
 5 v.onStack = true;
 6 foreach (v, w) in v.successors do
 7 |   if w.index is undefined then
 8 |   |   strongconnect(w);
 9 |   |   v.lowlink = min(v.lowlink, w.lowlink);
10 |   else if w.onStack then
11 |   |   v.lowlink = min(v.lowlink, w.index);
12 |   end
13 end
14 if v.lowlink == v.index then
15 |   start a new strongly connected component
16 |   do
17 |   |   w = S.pop();
18 |   |   w.onStack = false;
19 |   |   add w to current strongly connected component;
20 |   while w != v;
21 |   output the current strongly connected component;
22 end
```

perform a depth-first search (Lines 6-8) with the *strongconnect* function, while keeping track of the search tree by giving each vertex a unique index (Lines 1, 3). Additionally, the *lowlink* is stored for each vertex, which is the lowest index which can be reached from the vertex in the current search subtree (Lines 2, 9, 11). As such, the search tree can be fully expanded from any vertex, while updating the indexes and pushing the discovered vertices onto a stack (Lines 4, 5). On the tail-end of the recursive depth-first search (Line 14), the vertex where a cycle starts is identified by comparing the vertex's *index* to the *lowlink*. Only if these still match, a new component is formed which contains all following vertices on the stack, which are the vertices in the child search tree of this vertex (Lines 15-20). If the current vertex is a part of a cycle and not the first node in the search tree of this cycle, then the *lowlink* has been updated to the assigned index of the vertex in the search tree, where the cycle actually starts and as such a component is not created. If the vertex is not part of a cycle, then the *lowlink* is never updated and the vertex,

therefore, forms a component alone. Because the formation of components occurs on the tail-end of the recursion and, therefore, essentially in the opposite direction of the edges, the components are formed in the inverse topological order [Tar72].

Algorithm 5.3: Pseudocode for Tarjan's algorithm to identify strongly connected components [Tar72].

```
Input: G=directed graph
Output: sets of vertices which form strongly connected components in inverse
        topological order
1 index = 0;
2 S = new Stack();
3 foreach v in V do
4 |   if v.index is undefined then
5 |   |   strongconnect(v);
6 |   end
7 end
```

Algorithm 5.3 shows that this depth-first search is started from every vertex which has not been discovered previously to ensure that all vertices are discovered. The depth-first search can only discover following vertices and ones connected to the graph [Tar72]. If all the sources of the graph are already known, then it would be enough to start the searches by iterating over all sources rather than all vertices in Algorithm 5.3.

The reason why strongly connected components can be returned in a topological order even in a cyclic graph is, that the strongly connected components can be used to transform the graph into a DAG. Figure 5.4 visualises this process. If the graph is mapped to a graph only consisting of the strongly connected components represented as vertices and edges between vertices of different components are mapped to an identical edge between the components, then the resulting graph is always a DAG. This is a property trivial to prove by contradiction, because if we assume the resulting graph contains a cycle and, therefore, can not be DAG, then all the components in this cycle have paths to each other and would form a new and larger strongly connected component. This contradicts the assumption that the strongly connected components have already been found and, therefore, a cycle can not exist. The resulting graph of such a transformation, therefore, has to be a DAG.

Following the previous iterations of the heuristic's execution on a graph to create a topological ordering of vertices, all sources or sinks have already been removed. As such, any vertex that neighbours one of the previously removed sources or sinks is within a cyclic structure or, in other words, a strongly connected component, because otherwise, it would be a source or sink that has already been removed. Therefore, at least the first and last component in the order returned by Tarjan's algorithm must contain a cycle, because it returns strongly connected components in inverse topological order. The heuristic can

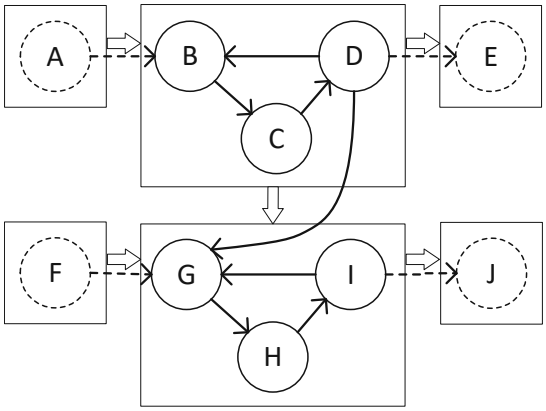


Figure 5.4: Example of the graph from Figure 5.3 being transformed to a DAG of components. Components are displayed using rectangles with the boxed arrows representing the edges between them. The original vertices, which form a component, and their edges are also included to visualise how the transformed graph has been created. In the context of the heuristic, there is no need to create a component for an already removed vertice, but it is still displayed here to provide a more extensive example of the transformation process.

thereby trivially be extended with the selection of a first cycle to remove an edge from to avoid the previously discussed problem presented in Figures 5.3 and 5.4.

Using the heuristic, a topological ordering can therefore be calculated for potentially cyclical graphs, which in turn allows calculating the latency of the stream processing application, although with only limited correctness. Furthermore, the heuristic can be adapted so that sources and sinks are not removed one by one, but rather in sets. This way, an ordering of sets of vertices would be returned rather than individual vertices. These sets contain additional information, because the order of vertices in each set is completely interchangeable and can be used to easily create alternative topological orderings. Creating multiple topological orderings can, for example, be useful when creating a greedy placement heuristic, as changing the order of operators to place can be easily adjusted, while still keeping a valid topological order. Additionally, in Apache Storm, the operators are constant and can not be added or removed after their initial creation on topology submission. Consequently, the topological order can simply be cached for any future placement calculation, until the topology is terminated.

The complete pseudocode of the modified heuristic can be seen in Algorithm 5.4. It consists of the Eades, Lyn and Smith heuristic (Lines 1-32, 35-41), as discussed in Section 5.2.1, but modified to remove all viable sources or sinks as a set rather than individually (Lines 8-19, 20-31). Additionally, the logic of how to select a vertex to resolve a cycle was modified as described in the current section. Tarjan's algorithm (Line 33) is applied to identify the first cycle in the DAG, which is the last component in the output of the algorithm. Finally, a vertex in this cycle is selected for removal based

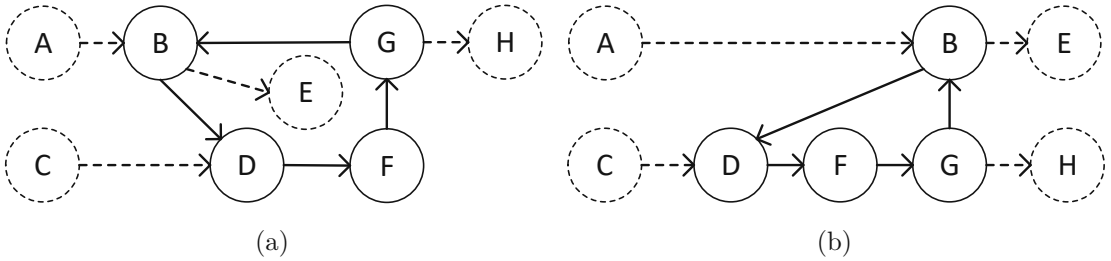


Figure 5.5: Different visualisations of the same graph to highlight potentially different interpretations of the intuitive importance of an edge.

on Eades, Lyn and Smith's heuristic with the addition of requiring it to neighbour a previously removed vertex categorised as a source (Line 34).

Finally, Figure 5.5 shows that not for every cycle an ideal edge or vertex can be selected purely based on graph theory. In both visualisations, an identical graph is displayed, but the difference in presentation may make us identify different edges as candidates to remove the cycle. In (a), the edge from G to B seems like the backwards-leading edge of the cycle, while in (b), it is the edge from B to D. Of course, arbitrary more complex problems can be created, but this simple example shows that to make a correct decision, a detailed understanding of the data to be processed, application and intended behaviour is necessary. As such, without further information from the stream processing application's developer, this problem can only be solved heuristically. Requiring such additional information for a placement is contrary to the requirement of supporting existing Apache Storm topologies and avoiding modifications to the API or internals of Apache Storm. Therefore, such an extension is out of the scope of this thesis.

In practice, if cycles are a critical feature for a stream processing framework and the intent is to fully support them and latency optimisations or requirements, then a solution would likely be similar to the use of selectivities of operators. Before an application is executed, information on the usage or importance of optimising cycles could be provided externally by a developer. Alternatively, a heuristic could be used as the last pages have described.

Similar to selectivity, once the application is running, the specific behaviour can be observed and tracked for future placements and thereby does not require any optimisation hints manually defined by developers. Apache Storm already does something similar for any events currently being processed. As a part of the efforts to guarantee message processing to ensure reliability, tuple trees are created. They are DAGs which track an event and its modifications from the data source through the operators which process it. As such, these tuple trees effectively track the pathing of events, but this information is only stored as long as necessary, only collected if guaranteed processing is utilised and only accessible within ackers, the components which track the processing acknowledgements of operators [Apa22d]. If this information could be accessed, it could be stored more

Algorithm 5.4: Modification of the Eades, Lyn and Smith heuristic to create topological orderings of cyclic graphs by ignoring edges with less impact on the pathing than the minimal feedback arc set problem [ELS93].

Input: G=directed, potentially cyclic, graph
Output: sets of vertices in topological order after minimal number of edges has been removed to break cycles

```
1  left=new Stack();
2  right=new Stack();
3  sourcesToProcess, sinksToProcess=new Queues();
4  tempQueue=new Queue();
5  add sources in G to sourcesToProcess, sinksToProcess;
6  while G is not empty do
7      set=new Set();
8      while sourcesToProcess is not empty do
9          add sourcesToProcess[0] to set;
10         remove sourcesToProcess[0] and its edges from G and sourcesToProcess;
11         add new sources to tempQueue;
12         if sourcesToProcess is empty then
13             swap tempQueue and sourcesToProcess;
14             if set is not empty then
15                 push set onto left;
16                 set=new Set();
17             end
18         end
19     end
20     while sinksToProcess is not empty do
21         add sinksToProcess[0] to set;
22         remove sinksToProcess[0] and its edges from G and sinksToProcess;
23         add new sinks to tempQueue;
24         if sinksToProcess is empty then
25             swap tempQueue and sinksToProcess;
26             if set is not empty then
27                 push set onto right;
28                 set=new Set();
29             end
30         end
31     end
       // graph has all sources and sinks removed, but has a
          cycle preventing removal of more vertices
32     if Graph is not empty then
33         components=Tarjan's algorithm(G);
34         find vertice v in the last component with max(outdegree(v)-indegree(v)),
            which is a neighbour of a previously removed source;
35         remove v and its edges from G;
36         add v to a new set;
37         push new set onto left;
38     end
39 end
40 pop entries from right and push onto left;
41 return left;
```

permanently in the form of statistics about the usage of specific edges and cycles or even as test data, of which a replay could be simulated to evaluate latency metrics. This likely provides a good solution for online placement mechanisms, but creates the question again of how or if these cycles should be optimised during the first run, when such metrics have not been collected yet.

Depending on the application, a developer might also be able to unroll the loops to create a cycle-free variant of the application. Especially for safety-critical applications, where latency is not simply a metric to optimise but an actual constraint, this might be the most viable solution.

This section has discussed topological sorting and the removal of cycles to, for example, estimate the latency of a topology. With all the information, therefore, being available for the constrained optimisation problem, the following section discusses the implementation of solving them.

5.3 Operator Placement Heuristic

To solve the constrained optimisation problem, three different iterative solvers have been implemented. These are a hill-climbing, an ant system and a hybrid approach of both to combine their different advantages.

Iterative solvers are highly desirable for online schedulers because the computation of placements is relatively cheap and can, therefore, easily be performed every ten seconds when Apache Storm triggers the placement logic. The disadvantage is that they are heuristics and, as such, do not guarantee optimal results. This is because they do not exhaustively explore the search space, but instead, attempt to iteratively improve an existing placement. These placements are then only executed if they pass a threshold of at least an 0.3 score improvement of the placement. This is because slight score variations exist purely due to changes in the measured metrics, thereby resulting in new theoretically optimal placements. In practice, the slight score improvements do not significantly affect real-world performance, especially when considering the downtime due to the enactment of the new placement, which is another reason why a heuristic is good enough rather than requiring an exact solver.

The iterative solvers used in this thesis all share similar mechanisms to limit how many iterations of placement they may perform each time Apache Storm triggers them. These are time limits, iteration limits and limits on the number of iterations since the last improvement was found. Additionally, while potentially better placements could be computed every time Apache Storm triggers the logic, there are two more conditions. These are primarily because Apache Storm takes a significant amount of time to execute placements and, as such, it is likely for the placement logic to be engaged again before the topology is being completely executed again. Even if the topology is already fully capable of processing data, new metric data likely has not been received yet or does not fully represent the current placement due to the short runtime. Therefore, a topology

may only be placed again 30 seconds after the last placement has occurred and when new metric data has already been received.

The first method that is being discussed to actually solve the constrained optimisation problem is hill-climbing.

5.3.1 Hill-Climbing

Local search is based on the idea of performing small modifications on an existing solution iteratively to find better placements [TR14]. The initial solution for the search is either the current placement or a potentially highly inefficient placement from a greedy heuristic that mainly aims to satisfy the memory and slot availability constraints. In this local search implementation, for a modification to be accepted as a better solution, the amount of constraint violations has to be reduced or at least kept equal while improving on the score. This is also often referred to as hill-climbing [TR14]. This means that a suboptimal modification will never be executed, thereby also resulting in a high risk of getting stuck in a local optimum. As discussed previously, reaching the optimal score is not considered necessary and as such local optimums are also not that problematic.

The hill-climbing implementation uses the following three potential operations to modify a placement:

1. Moving one operator to a different supervisor.
2. Swapping the placement of two operators which are not co-located.
3. Moving all operators of a supervisor to a different supervisor.

Swapping and moving all operators are, in theory, not necessary because they can be expressed through multiple individual move operations. Due to co-location being highly relevant in multiple metrics of the scoring mechanisms, it is also often the cause of local optimums. Swapping the placements is highly effective at reducing the application's latency, while moving all operators allows for the contraction of a placement when a topology is experiencing low load. In both cases, the loss of co-location would result in significantly worse scores if these modifications were to be executed in multiple steps instead, thereby resulting in significant and undesirable local optimums, which this form of local search would not overcome.

Because the topologies are relatively small, it was observed that generating and testing all potential modifications is still performant enough. For larger environments or topologies, this approach could not be used and only a random subset of the modifications should be generated. Generating the entire neighbouring solution space also provides the advantage that the search for a better solution can be aborted once a single iteration fails to improve the score. A random neighbourhood can not provide this certainty and requires multiple iterations to achieve a high certainty that a continued search would either be

too inefficient or can not improve on the score anymore. As such, the resulting score quality can vary more with random solutions constructions, which is also the case for the following approach, the ant system.

5.3.2 Ant System

The ant system is based on the real-world collaborative pathfinding of ants. The movement decisions of the ants are modelled probabilistically on a graph. Once an ant has built a complete path this way, it will deposit pheromone on it. Based on the real world, a shorter path will allow an ant to walk over it more frequently, thereby depositing more pheromones than on a longer path. The pheromone then attracts other ants to follow the path and further reinforces the pheromone trail. At the same time, the pheromone will slowly evaporate, thereby deprioritising less used or unused paths. This way, the ants can collaboratively find short paths [DMC96].

The placement problem can be mapped to the pathfinding problem by representing the operators and supervisors of a node. An assignment can then be represented by a path that starts on an operator, ends on a supervisor and alternates between supervisors and operators until all operators are included in the path exactly once. Edges from the operators to the supervisors then represent individual operator assignments. The only movement that is not allowed for an ant is an operator placement which would overload the memory of a supervisor. The edges from the supervisors to the operators do not have any meaning except for deciding the order in which the individual operator assignments are explored.

Placing pheromones on these edges leads to a bias where some order for the assignment of operators is preferred, even though it does not affect the solution quality. At the same time, a specific operator order can make finding certain solutions more difficult. An example of this is the memory constraint that can block operators which are considered later from being placed there. This can lead to the creation of local optimums. Therefore, the paths of the ants only consist of the edges from the operators to the supervisors. The reverse edges are not modelled. Instead, the operator placement order for each ant is randomised to avoid this bias, which can lead to local optimums.

Algorithm 5.5 shows the pseudocode of the ant system for the placement of operators. To start with, the pheromone of each edge (i, j) is initialised with the configurable parameter p_i. Additionally, an already existing placement also affects the initial pheromone amount by executing the pheromone placement p_c times for a path that is equivalent to the current placement (Lines 1-2). During early testing, placing five times the amount of pheromone resulted in a good balance of exploring new alternatives and remembering the current solution. In lines four to six, the ant population of m ants is generated with a random path each, the best ant that has been found is updated and the pheromone on the graph edges is modified accordingly. Similar to hill-climbing, it uses the absolute iteration count, iterations since the last score improvement and total runtime as its exit conditions (Lines 3,7). Once the algorithm finishes, it returns the best placement that

Algorithm 5.5: Ant system pseudocode [DMC96]

```
   Input: currentPlacement
   Output: new optimised placement
1  bestAnt=currentPlacement
2  pheromone=initialisePheromone(bestAnt)
3  while no exit condition fulfilled do
4  |  ants=generatePopulationOfAntsWithPaths()
5  |  bestAnt=selectBestAnt(ants,bestAnt)
6  |  pheromone=placeAndDecayPheromone(pheromone,ants)
7  |  updateExitConditions()
8  end
9  return bestAnt;
```

has been found (Line 9). The following discussion presents the underlying math of the ant system with the equations used for the path generation (Line 4), scoring (Line 5), and pheromone update (Line 6).

In an ant system, the probability of an ant k at the time t to move from a node i in the graph to j is shown in Equation 5.1. It consists of an a priori heuristic, for which a custom one is defined in Equation 5.2, and an a posteriori heuristic, the pheromone. α and β are used as parameters to weigh the importance of both elements [DMC96].

$$p_{ij}^{k} = \begin{cases} \frac{|\mathcal{T}_{ij}(t)^{\alpha}| \cdot |\eta_{ij}(t)^{\beta}|}{\sum_{k \in \text{allowed movements}} |\mathcal{T}_{ik}(t)^{\alpha}| \cdot |\eta_{ik}(t)^{\beta}|} & j \in \text{allowed movements} \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

The custom heuristic in Equation 5.2 is primarily used to find solutions that utilise fewer supervisors and achieve more co-locations faster. It acts as a factor for the existing pheromone to encourage or deter from certain decisions and thereby guide the exploration. Therefore, it usually returns the value one, but is optionally multiplied by two in two favourable conditions or divided by two each if the placement would overload the supervisor's CPU or memory. The favourable conditions are if another operator of the topology has already been placed on the supervisor or if a predecessor or successor operator has been placed on the supervisor, which would therefore achieve co-location. Because of the random order in which the ants' paths are created and operators are placed, the impact of individual decisions on the application's latency is generally unknown and, therefore, only optimised indirectly by the heuristic using co-location.

$$\eta_{ij} = 1 \cdot 2^{\text{pred or suc placed on } j} \cdot 2^{j \text{ already used}} \cdot \frac{1}{2^{\text{overloads } j\text{'s CPU}}} \cdot \frac{1}{2^{\text{overloads } j\text{'s memory}}} \tag{5.2}$$

Equation 5.3 shows the decay of the pheromone based on the parameter ρ and the newly placed pheromone of all ants for an edge (i, j) [DMC96]. In this ant system

variant, a minimum pheromone amount p_m on each edge is enforced to ensure that the random exploration of alternate paths never stops, which was inspired by max-min ant systems [SH00].

$$\mathcal{T}_{ij}(t) = \max(\rho \cdot \mathcal{T}_{ij}(t-1) + \sum_{k=1}^{m} \Delta\mathcal{T}_{ij}^{k}, p_m) \tag{5.3}$$

The pheromone placed on an edge (i, j) by the ant k is defined in Equation 5.4. It uses a constant Q to scale the placed pheromone and S_k, the score of the ant k's solution. It ensures that a lower score results in more pheromones placed, thereby attracting more ants to better solutions [DMC96].

$$\Delta\mathcal{T}_{ij}^{k} = \begin{cases} \frac{Q}{S_k} & k\text{th ant uses edge (i,j) in path} \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

To ensure that constraint violations are actively being reduced, they are considered in the placement of the heuristic by using a combined score defined in Equation 5.5.

$$S_k = score(k) + 10000 * constraintViolations(k) \tag{5.5}$$

Hill-climbing and the ant system share a lot of similarities in their interface but are very different in how they perform the optimisation. This can lead to the solvers excelling in solving different types of problems. The following section, therefore, describes a combined approach.

5.3.3 Hybrid

The hybrid approach attempts to combine the benefits of hill-climbing and the ant system. The random solution construction of the ant system is more effective at exploring the solution space by avoiding getting stuck in local minimums but also does not identify and exploit the smaller optimisations as greedily as hill-climbing. This means that after the ant system finds a good solution, there may still be some small adaptions that can be made to greedily improve the score. For this reason, the hybrid search consists of running the ant system for its effective exploration, which is then followed by hill-climbing to find and make those small adjustments. The configuration of both solvers was kept identical to their independent versions, except for the time limit, which was halved to ensure that the execution of both solvers in succession would still fulfil the same design constraints. The idea of combining local search and ant systems is not new in any way but is usually directly applied within the ant system by optimising each ant's path again before the pheromone is placed [TR14].

Table 5.1 showcases the different parameters that have been used to configure the solvers. As previously mentioned, the hybrid approach uses matching parameters except for the runtime limitation of the individual solvers.

Table 5.1: Parameters used for the solver implementations.

Parameter	**Hill-Climbing**	**Ant System**	**Hybrid**
max runtime	1000ms	1000ms	500|500ms
max iterations	1000	1000	1000|1000
max iterations without improvement	1	15	1|15
α		1	1
β		1	1
initial pheromone(p_i)		5	5
min pheromone(p_m)		0.6	0.6
pheromone decay(ρ)		0.95	0.95
pheromone placement constant(Q)		5	5
pheromone current solution placement(p_c)		5	5
ant count(m)		30	30

The following section discusses Apache Storm's scheduling API in more detail and captures some of the experiences that were made during the implementation of the scheduling heuristic and the three solvers.

5.4 The State of the Apache Storm Scheduling API

One of the major reasons for the framework selection of Apache Storm for the implementation in Section 4.3 is its general documentation and the existence as well as the accessibility of the scheduling API. This chapter has discussed how the API was used to create an online scheduler, unlike the ones provided by Apache Storm itself. In addition to changing how placements are computed, the heuristic itself differs significantly by adaptively changing placements and using runtime statistics. As such, the heuristic highlights that the API allows conceiving schedulers that significantly differ from the schedulers the API was built for. This is a clear indicator of the high quality of the API and validates the decision to use Apache Storm. At the same time, there are other properties that can affect an evaluation of the quality of the API. Therefore, this section aims to provide a more detailed look at the API and presents limitations, problems and difficulties that were discovered during this work.

In general, it is relatively easy to get an overview of the API. This is because it only uses a few classes that all directly relate to elements one would consider when solving the placement problem. Some examples are the classes *Cluster*, *SupervisorDetails*, *WorkerSlot*, *TopologyDetails*, *ExecutorDetails* and so on [Apa22e]. The *Cluster* acts as an entry point to the API from which it is then possible to navigate to and through the other elements. In general, it is possible to navigate from any object to any other related object, for example, from *WorkerSlots* to *SupervisorDetails*, but these relations are often only stored by exposing the related id rather than the objects. As such, it is often necessary to look up the relevant object by id in one of the lookup tables stored on the *Cluster* object.

The *Cluster* holds the functions to retrieve and modify assignments, among others, while the rest of the classes act as read-only structures to organise and hold data. With that said, not all the information is stored in this object-oriented approach. The *Cluster* also has a variety of lookup functions or maps storing additional information about the objects of the other classes, likely because of implementation details. As a result, for most objects, the available information is split between the *Cluster* and whatever object it belongs to. But even this has exceptions, the *TopologyDetail* stores the information on the resource request of an *ExecutorDetail*. This makes identifying all available information more difficult and accessing it more tedious. At the same time, the *Cluster* has an extensive list of functions making it more difficult to get an overview or to find something specific. Overall, this mostly raises the learning curve with the API and somewhat the readability of the resulting implementation. Unfortunately, the maps often use the id of an object as the key, rather than the object itself, which can lead to errors such as querying with ids from the wrong type of object, for example [Apa22e].

The actual naming of elements can also lead to some confusion. *WorkerSlots* are also sometimes referred to as workers or slots across functions or the documentation. The most inconsistent location seems to be the JavaDoc and related function names of the *Cluster* [Apa22e]. This can lead to uncertainty, such as if workers and *WorkerSlots* are the same concepts. A similar problem exists with *SupervisorDetails*, which are referenced as a *nodeId* in *WorkerSlots*. In addition to the different naming schemes, the related fields or functions lack a description in the documentation [Apa22e].

The same is true with functions, for which *getAssignableSlots()* and *getAvailableSlots()* are a particularly frustrating example. They are difficult to differentiate because neither their name nor their documentation clarifies the difference and may even be misdirecting depending on the interpretation [Apa22e]. The answer is that both functions retrieve the assignable slots from *getAssignablePorts()*, but *getAvailableSlots()* additionally filters out all occupied slots. As a result, the return of the assignable slots function also contains slots that have to be emptied first to allow for a new assignment. This means that, in a confusing manner, the slots from *getAvailableSlots()* are more assignable than the ones returned from *getAssignableSlots()*, because Storm only allows assignments to empty slots.

Fortunately, the naming of elements is still improving with updates. With the release of version 2.3.0, the function *getTopologyComponents()* was renamed to *getUserTopologyComponents()*, which now clarifies why some components are not returned from this function but referenced by the *ExecutorDetails* for which such a differentiation does not exist [Eth21]. Unfortunately, a similar function for presumably system components, such as ackers or IMetricsConsumer implementations, does not exist. *Components* are the equivalent of a node in the DAG, the operation, and therefore store the information on what the inputs and outputs are connected to. This is, for example, relevant to model bandwidth usage and to identify good operators to co-locate. At the moment, for system components, these relations have to be ignored or manually defined based on the name of the component, but this is, of course, error-prone. Furthermore, this only clarifies the

interconnection at the component level and does not provide easy-to-use information on how a specific type of grouping affects the routing at the operator level. Such a logic would have to be manually implemented based on the documentation of groupings.

A large limitation for adaptive placement heuristics is that the metrics API is not natively available in Nimbus and therefore requires the less maintainable workaround with the *IMetricsConsumer* implementation. With the release of version 2.3.0, the metrics API has also been nearly completely transitioned to v2, except for the *___recv-iconnection* metric, which this thesis uses to retrieve information on the bandwidth usage [Eth21]. As such, the old API does not receive most data anymore unless the *topology.enable.v2.metrics.tick* setting is configured to essentially enable something close to backwards compatibility. This critical information about the old API, as it otherwise can not be used, is not even mentioned on its documentation page but at the bottom of the metrics v2 documentation as a note on backwards compatibility [Apa22f]. An additional effect of this update is also that some metrics slightly changed their names or structures in the old API meaning that implementations need to also be updated. The documentation does also not yet reflect these changes [Apa22j]. Additionally, some bugs were introduced. Both the *execute-latency* and the *process-latency* are now always reported as zero in the old API. Most of these issues can be worked around and the update also has a benefit. The capacity metric can now be accessed in the old API via the backwards compatibility flag and does not need to be computed manually anymore [Eth21].

Another problem for adaptive placement heuristics is the recently buggy nature of Storm's worker restart mechanism. Normally, this functionality may only be triggered during rebalancing or when failures occur and, as such, is rarely used. This has, unfortunately, also caused this function to exhibit bugs in released versions. Version 2.3.0 has fixed the bug #3658, in which a worker could fail to shut down, thereby preventing restarts necessary to change the assignments. The bugs #3655 and #3677 have also been resolved in the same update to remove issues related to changed assignments [Eth21]. During the work on this thesis, a new bug has been found in version 2.3.0, where workers would sometimes begin the start-up before the previous instance has completely shut down. This resulted in the new instance running out of memory because the old one still used it or the instance failing to acquire its assigned port, resulting in a running instance that other workers could not connect to. Briefly after identifying the problem, Apache Storm version 2.4.0 was released, in which the problem could not be reproduced anymore and, as such, was resolved, likely by one of the dependency upgrades [Bip22]. Using version 2.4.0, no further issue was identified as this functionality now works reliably, but this brief history shows that the adoption of an online scheduler in a production environment has additional risks and would have struggled to be functional for at least the last year. Of course, this also showcases the rapid improvement and quick resolution of any issues once they are identified and reported.

Another large benefit of Apache Storm's community is that its JavaDoc is very extensive and documents nearly every function [Apa22e]. Generally, the function names are also intuitive enough, such that reading up additional information in the documentation is

only rarely necessary. Similarly, the rest of the Apache Storm documentation presents information in a very approachable manner. Unfortunately, there are two conflicting views in the documentation: one intended for the user and one for the developers. These views are mostly what information is described or where it exists. For example, in the previous case of the metrics API, the information on how to enable backward compatibility is placed with the new API, because that is where it is implemented and, in principle, enabled [Apa22f]. But for a user that attempts to use the old API, it is nearly impossible to find that information, because in the documentation of the old API there is no mention of this lack of compatibility [Apa22j]. A similar problem exists with the description of scheduling. The documentation entirely focuses on existing schedulers, their behaviour and how to configure them. On the other side, the documentation only mentions the possibility of adding a scheduler, one of the two possible interfaces and how to configure it. Critical information such as the build or deployment steps and constraints the scheduler and its placement have to fulfil are not mentioned. This information has to be in part learned by trial and error and is spread across the JavaDoc, the general documentation, the scheduling configuration and documentation of other schedulers and the original Storm paper [Apa22e, Apa22m, Apa22h, TTS+14].

With a better understanding of the API, the following section discusses the difficulty of implementing potential improvements from researchers.

The Growing Split between Industry Practices and Research on Apache Storm Operator Placements

To this day, the default scheduler in Apache Storm is essentially an application of round-robin scheduling of a single topology across a limited number of all computational resources [Apa22h]. This is, of course, a very basic strategy that does not make use of co-location optimisations, can cause higher latency and can easily waste resources because it does not consider the heterogeneous requirements of tasks or capabilities of computational resources. This raises the question of why this provably less optimal scheduler, from the perspective of a researcher, is still being used when an alternative like the Resource Aware Scheduler is already integrated and can be enabled with a small change in the configuration [Apa22g]. Similarly, many researchers have developed new schedulers over recent years that have not been integrated into Apache Storm [ABQ13, XCTS14, MAI21, QR21]. The following discussion aims to clarify the crucial benefits of Storm's default scheduler, why new schedulers proposed by researchers struggle to compete in these areas and what the impact of that is on scheduling in Apache Storm.

A large benefit of round-robin scheduling is the simplicity of its usage and configuration as well as the implementation and maintenance itself. It barely requires any configuration and performance problems are easy to identify and solve, but not necessarily in an efficient way. Essentially, a user can only add or increase the capacity of the computational resources, rebalance the topology to allow it to use more resources or resubmit it with more instances of operators. These three options all solve clearly defined problems, such as the cluster lacking resources as a whole or a specific operation being a bottleneck.

It makes the scheduler easy to use, the code simple to maintain and the behaviour predictable and reliable, even if the actual placements lose out in efficiency. As such, supporting this scheduler with a large base of users is not too difficult, because most questions can be resolved with minimal documentation.

In contrast to that is the Resource Aware Scheduler. It requires the user to specify the resource constraints of operators and the availabilities of supervisors [Apa22g]. This is an additional configuration effort where mistakes can be made, especially because the resource usage of an operator always correlates with the rate of data to be processed and, as such, can diverge significantly from the estimated configuration. Identifying such a problem increases the difficulty of using it correctly and similarly can the necessary fine-tuning be too bothersome for its users. As such, the Resource Aware Scheduler is integrated, but locked behind a simple configuration change [Apa22g]. This way, the potential for better scheduling is available for everyone that needs the extra efficiency and is willing to make this effort.

Online schedulers could resolve the additional configuration problem because they can collect this data themselves by tracking the current execution, but in exchange, their complexity grows significantly. This is not only the complexity of implementing it, but also maintaining it, supporting it and achieving effective placements in the variety of real-world deployments and their workloads. The previous section has already discussed that in some cases the scheduling API is designed for static scheduling and, while not being particularly limiting, still has a variety of complications that make implementing an online scheduler more difficult than it could be. And this is likely the reason why online schedulers have been developed by researchers for Apache Storm but have never been integrated because deploying, maintaining and supporting them would be too difficult. As such, while researchers keep creating new solutions, these changes do not seem to make it into Apache Storm or other stream processing frameworks. Apache Storm has been selected for this thesis, because of its excellent scheduling API, especially in comparison to other frameworks, but the API is still likely one of the major hurdles that limit the industry at large to round-robin and other simple forms of scheduling.

As such, the following chapter continues the research effort by discussing the evaluation of the heuristic placement implementation in Apache Storm.

CHAPTER 6

Evaluation

This chapter exhibits the results of the prototype scheduler's evaluation. Section 6.1 presents the overview of the used testing methodology and architecture. This is followed by details on the experimental setup in Section 6.2. Section 6.2.1 contains information on the emulation of the testing network. In Section 6.2.2, the workload applications for the benchmark are showcased. The test driver and the collection of metrics are discussed in Section 6.2.3 in more detail. Finally, the results for both the latency estimation and the placement heuristic are shown and analysed in Section 6.3.

6.1 Evaluation Methodology

The methodology of this thesis to evaluate the operator placement heuristic is primarily based on the benchmarking methodology of Karimov et al. [KRK+18]. It ensures a separation between the system under test, Apache Storm in this case, and the benchmark tool, the test driver, itself. As such, the entire system is being benchmarked, which includes the ingestion capabilities and mechanisms to handle an overloaded system, such as back-pressure. The metrics being collected are the assigned resources, the throughput and the topology's execution time to process an event, simply referred to as latency. While the general approach was replicated for the evaluation, some smaller changes were made to account for the fog computing environment and the stream operator placement itself.

While fog computing has a large impact on research, there are no readily available deployments of it yet that could be used for this research. As such, research in fog computing generally relies on simulations or emulations of fog computing networks [HGG+19, HGB21, MGG+17]. Given that this thesis aims to develop a placement heuristic for a real-world application, Apache Storm, to gain new insights, rather than relying on simulations, only emulation presents a feasible path for this thesis. At the same time, the use of emulated environments, rather than rented physical infrastructures,

helps with the consistency in the collection of results, as, for example, the network is not shared with other users, which could cause interference. Similarly, an emulated environment is easier to control for variables and reproduce in comparison to a real-world setup. Therefore, a fog network is being emulated as described in Section 6.2.1.

To test the general applicability of the operator placement, multiple topologies are used for the benchmark. Some edge cases have been manually defined, but most are randomly generated, which is a common approach to test operator placement heuristics:

- Yanif and Uğur primarily used binary trees with simulated selectivity [Ac04].
- Lambert et al. generated linear applications of varying sizes where the selectivity of an operator could either triple the date or reduce it to a third [LGI20].
- Nardelli et al. evaluated three types of structures: linear applications, a diamond pattern and replicated, where two intermediary layers of operators exist, with the first one being twice the size of the following one [NCGP19].
- Rizou et al. generated graphs with six or twelve operators. Each operator has two or three successors and is configured with a selectivity between zero and one [RDR10].
- Thoma et al. used random topologies with 40, 100 or a varying amount of operators with five potential workload levels [TLL14].
- Veith et al. evaluated four manually defined edge cases with eleven or fewer operators and two random topologies with ten or 25 operators. The CPU cost, memory usage, selectivity and data compression of operators were simulated [VdAL18].

The random generation and all tested topologies are presented in Section 6.2.2. The generation of the topologies uses an iteratively growing approach, where edges are replaced with certain patterns until the desired size has been reached. The operators in the topology emulate a workload by simulating characteristics such as their CPU cost, memory usage, selectivity and bandwidth consumption.

Section 6.2.3 explains how the experimental runs are executed in detail. For the static operator placement heuristics already in Apache Storm, the number of supervisors to be utilised can be simply configured when the topology is submitted to the cluster. With the adaptive heuristics presented in this thesis, this is more complex. A constant input rate of events is created to allow the heuristics to adjust to that load and thereby indirectly configure the number of workers to utilise. The adaptive heuristics are then disabled when any measurement takes place. First, this is done to prevent the heuristic from adjusting to the changed load during throughput testing, thereby allowing the current placement to be evaluated. Secondly, as discussed in Section 4.4.3, while Apache Storm does allow for adaptive placements, they cause large interruptions of the service in the range of seconds. Therefore, it is ensured that any topology is fully deployed and capable

of processing events before any measurements are taken. Each experimental run is also repeated ten times.

Following this overview of the evaluation methodology, the following section documents the experimental setup in detail.

6.2 Experimental Setup

This section aims to introduce the emulated network, workload topologies and the test driver in detail.

6.2.1 Network Emulation

Mininet is a network emulator to create virtual networks executing actual router, switching and application logic purely based on software. Mininet allows defining topologies of hosts interconnected by network links, switches and routers. These network links are highly configurable to be able to recreate realistic networks. For example, network links can be defined with restrictions such as their latency, bandwidth or packet loss. Similarly, the switches and routers can also be modified, which has led to the large-scale usage of Mininet for the research of SDN [HHJ+12].

For this thesis, Mininet was not used, but instead, Containernet. Containernet is an extension of Mininet, which allows using Docker containers as hosts in the topology. While this does not provide particularly new functionality, as Mininet already allows for the execution of applications, it does provide some advantages for this thesis. First off, Docker containers allow to preconfigure and -build a host by defining the well-known Dockerfiles, rather than the more manual setup for Mininet [PKvR16]. Furthermore, Docker containers are readily available for most software, thereby simplifying or even eliminating the setup and configuration of all applications involved. Simplifying the setup of the evaluation environment was, therefore, the main driver for selecting Containernet over Mininet. In particular, its simple setup and extensive documentation, including smaller examples, are noteworthy. Although, in some cases, there are currently discrepancies between the documentation and the actual application. Unfortunately, Containernet is a much less mature software than Mininet, which makes it a lot easier to run into various solvable issues.

Networks in Mininet and Containernet can be designed in three different ways and represent real-world applications. Hosts can all be directly connected to each other, which is uncommonly seen in real-world situations because of the difficulty of managing the wiring, even with only a few hosts. The hosts are, therefore, usually connected to interconnected switches, which can then route data to the correct destination. Because of administrative purposes, routers are often used, which allow for the creation of multiple networks, subnetworks, while still allowing for data exchanges between hosts of different subnetworks. Additionally, they allow for network topologies containing cycles.

Mininet and Containernet already provide network links and switches for the creation of virtual networks. Routers can, in the simplest case, be set up as a Linux host with IP-Forwarding enabled in the network stack [HHJ+12]. This works well for immediate and therefore known neighbours in the network topology, but for more distanced packet receivers, either default routes have to be defined, or the distanced host has to be added to the routing table. Manually defining a static routing table for every router is a tedious and error-prone task, even in smaller networks. To simplify the setup and avoid errors, a router which implements dynamic routing protocols can therefore be used.

Quagga is a software suite supporting various dynamic routing protocols and can be used as a router by deploying it in a Docker container [Pau18]. Dynamic routing protocols are used to exchange information in the routing table with other routers. As such, it is enough to define the immediate neighbours of a router and this information will be shared with other routers, while also receiving information about more distant routes [JL14]. The main benefit of using routers instead of switches in the context of such an experiment would, therefore, only be the ability to create network topologies with cycles and not the administrative features or benefits. As such, simpler-to-set-up routing protocols like RIP or OSPF are preferable over more complicated ones like BGP if routers are used [Pau18].

With the different ways of designing the network topologies, it raises the question of what fog networks they might represent. Switch-based networks are used for smaller to medium-sized networks, such as a single factory or office building which could have a private deployment of a cloud or fog. Routers are used for larger networks and, thereby, would best represent community-based, public or hybrid deployments, such as city-wide installations to monitor traffic.

Switch-based network topologies have been selected for this thesis, with the consideration that the existence of cycles in network topologies does not significantly affect the involved latencies and evaluation. While more direct paths between certain hosts would reduce their latencies, this does not change the fact that some hosts will still have smaller or larger latencies to each other and should therefore be considered during operator placement. Similarly, because the network and resources are emulated and not simulated, resource constraints already prevent experimentation on large-scale networks where routers would be used. At the same time, switch-based networks are a lot simpler to set up and barely require any configuration. In contrast, applications may need to be reconfigured if routers and multiple networks were to be used, because, for example, many applications reject connections from hosts outside of the current subnetwork by default. While in principle the existence of cycles in the network topology does not create large differences, it does allow for multiple paths and, therefore, avoidance of bottlenecks in the network. The placement heuristic does not specifically consider the pathing in the network and, therefore, potential bottlenecks, which should be a more significant problem in switch-based networks, because no alternative paths exist. In the end, both switch- and router-based networks are potential deployments of the fog with various use cases, but only a switch-based network is being evaluated. This likely correlates with private, community or hybrid deployments of fog computing, as discussed in Section 2.4.

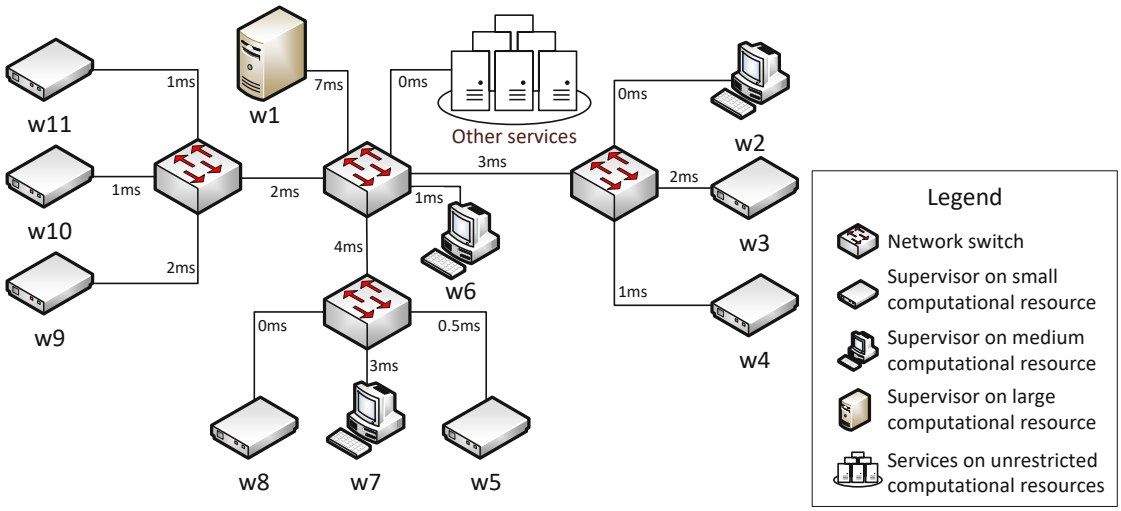


Figure 6.1: Emulated network topology for the evaluation.

These differences, in turn, mean that any collected results may not be fully representative of large router-based networks. While there are, of course, still differences, the insights gained should largely transfer between the types of networks.

In Figure 6.1, the emulated network topology is shown. It consists of $w1, ..., w11$, which are the Apache Storm supervisors. Additionally, the symbol *other services* refers to all other necessary services as discussed for the architecture in Section 5.1, such as Nimbus, Zookeeper or Redis. The resource limitations of the worker's containers are shown in Table 6.1 and the other services have not been limited, because their actual consumption is negligible and easily covered by leftover resources. Furthermore, because the other services are not directly involved in the processing of data, they do not affect metrics such as throughput or latency. The CPU capacity of many of the supervisors is intentionally small to model IoT devices with lower computational performance because they are being considered to be used in fog computing environments, which is discussed in Section 2.4. At the same time, the network provides the challenge of heterogeneous resources in which the largest one is five times more capable than the smallest ones. The specific values have been decided based on the hardware limitation of 8 available logical cores. As such, if all supervisors are fully utilised, there are still 1.3 logical cores of capacity for other services and the operating system. This is primarily to avoid an overutilisation of the real-world resources and has, in practice, never been observed. The lower bound of memory limitations has been based on the default memory configuration of Apache Storm for workers. While smaller limitations were still functional, the limit could not be set too low because otherwise, the default scheduler would often create placements that would fail to execute. Because of this, the memory limitations are far less significant in this evaluation than the CPU capacities. This should not distort the results, because the default scheduler of Apache Storm does not consider memory limitations, which indicates

Table 6.1: The configuration of the resource constraints for the Docker containers of the respective supervisors.

Supervisors	**CPU**	**Memory**
w1	1.5 logical cores	2048MB
w2, w6, w7	1 logical core	1024MB
w3, w4, w5, w8, w9, w10, w11	0.3 logical cores	768MB

that they are not considered a common problem in real-world environments. The smaller resource provisions have also been chosen to be able to emulate a larger network with only 16GB of memory, as otherwise, the placement problem would be trivial with only a few hosts.

The latency between network links is based on other available work, because concrete data on fog network performance capabilities does not seem to be available. Emulators for fog computing generally use latencies in the single-digit ms range for individual network links as examples for their evaluations [HGG+19, HGB21, MGG+17]. A study of existing cloud and edge services has found that edge servers exist within ten milliseconds for 55% of users. Additionally, for nearly half of all users accessing a cloud service takes less than ten milliseconds more than an edge server [CAG20]. Based on these considerations, link latencies in the low single-digit millisecond range were selected, and $w1$, which represents a more powerful instance, like potentially a cloud or a better server, has a seven millisecond link latency to the central switch.

To ensure that these latencies are not included in the measured latency by the test driver, a second emulated test network exists. In this second network, no latencies are simulated and it is only used to facilitate the services of the test driver and the test driver's connection to the supervisors. As such, the test driver measures latencies that are respective to it running on the same machine as the data source or sink and does not include additional network latency between them.

For the evaluation of the latency estimation component, the network has been extended by adding two switches to each of the outer switches and attaching four hosts to each switch. This results in a tree-shaped network with ten switches and 40 hosts.

The network emulation and benchmark were executed on an Intel i7-4770k with 3.5Ghz, four cores and eight threads. The memory consisted of 16GB DDR3 with 1600MHz. Ubuntu 20.04.4 LTS was used as the operating system and version 2.4.0 of Apache Storm was evaluated. Docker version 20.10.16 and Mininet version 2.3.0 emulated the network and hosts. A Containernet installation using the version from the 21st of March 2022 was used and because the emulated network does not rely on routers there was no need to install Quagga.

The following section starts with the discussion on how the workload is being emulated.

6.2.2 Test Data and Application

Twelve different topologies are used for the evaluation and are listed in Table 6.2 for an overview. The complete configuration for each operator of the topologies can be found in the appendix. The topologies consist of three manually defined topologies to test edge cases, seven randomly generated topologies to evaluate the general cases and two additional random topologies that contain a cycle to also cover this special case. Each of the topologies has only a single source and sink operation to allow for an easier comparison during the benchmarking, as otherwise, potentially multiple input and output rates would have to be considered for each topology.

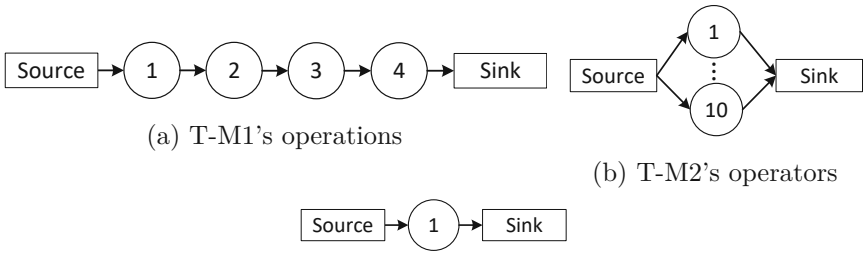
Table 6.2: Overview of the main properties of the topologies being tested. The operations and operators include the data source and sink in their count.

Topology	Randomly Generated	Contains a Cycle	Operations	Operators
T-M1	✗	✗	6	6
T-M2	✗	✗	3	12
T-M3	✗	✗	3	3
T-R1	✓	✗	16	30
T-R2	✓	✗	7	12
T-R3	✓	✗	12	22
T-R4	✓	✗	15	28
T-R5	✓	✗	11	20
T-R6	✓	✗	11	20
T-R7	✓	✗	22	42
T-R8	✓	✓	8	14
T-R9	✓	✓	10	18

Three basic edge cases have been considered for the manually defined topologies, which are shown in Figure 6.2. The topology M1 is merely a sequence of operations. M2 is a single operation for which ten instances exist and thereby tests a fan out of data and the load balancing of the individual instances. M3 represents the minimal possible topology with a single operation as a workload with only one instance.

For the random topologies Java's pseudorandom generator *Random* has been initialised with a seed of 3141592653589793238, which are the first 19 digits of π and the maximum that fits into a long. Before using the random number generator initialised like this, an additional $(n-1) * 1000$ random numbers are being generated for the nth topology to ensure a different random state for each one.

The first step is to select a random operation count between three and 24 with a discrete uniform distribution. The DAG generation starts with a single source and sink operation connected by an edge. Afterwards, one of the edges is randomly selected and either one operation or two, in the case of the diamond pattern, are inserted and replace the original edge, as shown in Figure 6.3. The single operation insertion is generated with a 60%



(a) T-M1's operations

(b) T-M2's operators

(c) T-M3's operations

Figure 6.2: Visualisations of manually defined topologies' operations or its instances in the special case of T-M2.

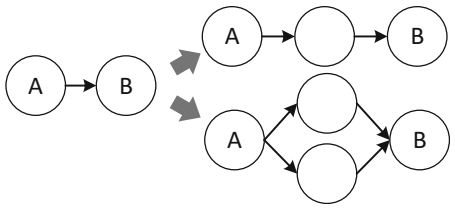
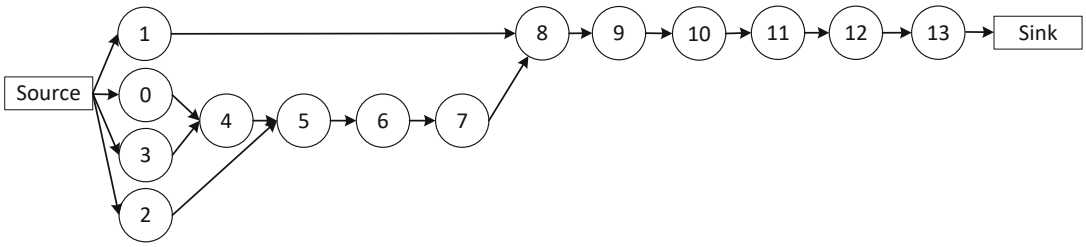


Figure 6.3: Iterative random edge replacement step to generate the test load topologies.

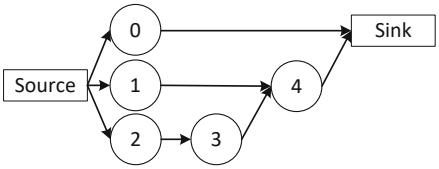
probability and the diamond pattern with a 40% probability. This is mostly because equal probabilities tended to create too wide graphs. Additionally, Apache Storm sends emitted events to all following operations and, as such, a split is always a duplication of the output rate. The decision was also made with the selectivity of the individual operations in mind. This was balanced to ensure that applications are being generated where some output overall more and others fewer events than they receive as input, to represent both kinds in the evaluation. It also has to be considered with the selectivity of the individual operations to ensure that applications which output more and fewer events than they receive as input are being generated. The topology is then iteratively grown until it reaches the exact operation count. The resulting DAGs of the topologies' operations can then be seen in Figures 6.4 and 6.5.

To generate the cyclic topologies, the same random generation is used, but an additional edge is added afterwards to create the cycle. Additionally, only between five and 19 operations are being generated to account for the increased workload in a cyclic topology. First, a random node which is neither the sink, the spout nor has the spout as a predecessor is selected as the start of the edge. For this node, the set of all predecessors, excluding the spout, is iteratively collected. A random node in this set is then selected as the end for the new cyclic edge. This edge is then inserted into the DAG, thereby creating a cycle. The two resulting cyclic topologies are shown in Figure 6.6.

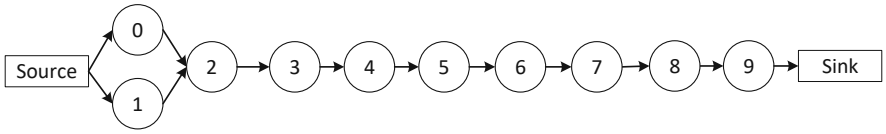
Afterwards, individual attributes, such as their CPU and memory load, are assigned to each operation and two instances are created for each one that is neither the source nor the sink. For the manually defined topologies, these have also been manually defined.



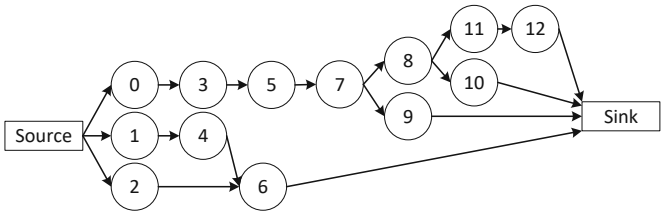
(a) T-R1's operations



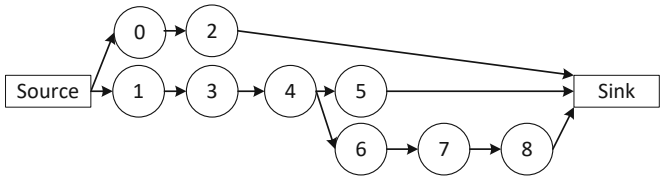
(b) T-R2's operations



(c) T-R3's operations



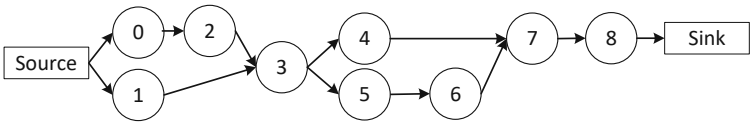
(d) T-R4's operations



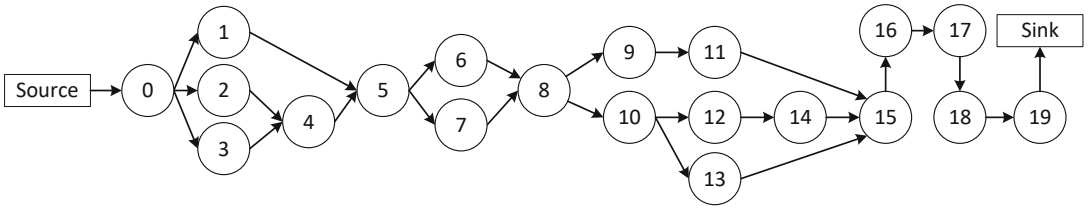
(e) T-R5's operations

Figure 6.4: Visualisations of the randomly generated topologies' operations.

As a workload, the generation of the nth random number using the *Random* number generator is used. This number is then also the output of the operators. n is generated using a normal distribution with a mean of 90 and a standard deviation of 75 and a minimum of one, finally, it is also scaled by 25. The intention was to create some very cheap operations, like, for example, some basic filtering conditions, and more expensive ones. The generation parameters were tuned to create both feasible and varied operations

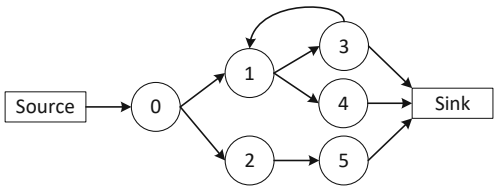


(a) T-R6's operations

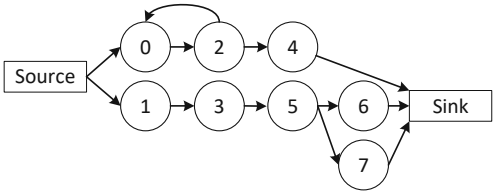


(b) T-R7's operations

Figure 6.5: Visualisations of the randomly generated topologies' operations.



(a) T-R8's operations



(b) T-R9's operations

Figure 6.6: Visualisations of the randomly generated cyclic topologies' operations.

on the hardware in use.

Selectivity uses a normal distribution with a mean of 0.9 and a standard deviation of 0.3 with a minimum of 0.3. The main idea is to create a selectivity around one, but biased to smaller numbers to equalise the output duplication of Apache Storm when a diamond pattern is being created. The memory requirement of an operator is twice the amount necessary to keep a random set of additional characters in memory, plus an additional ten megabytes are requested for the actual execution of any logic or buffering of events. The amount of characters is defined by the absolute value of a normal distribution with a mean of ten and a standard deviation of 12.5 and scaled by $1024 * 1024$ to create a value in the range of megabytes. The memory requirements mostly exist to make the placement more challenging, but it also can not be too difficult to fulfil because Storm's default scheduler is not aware of heterogeneous resources and requirements and, as such, would often place topologies in ways they could not actually execute. The final parameter is additional bandwidth waste, which is the length of a sequence of characters that is appended to any output of the operator. This is to emulate the data being transformed by operators like in real-world topologies, where they can have different data formats

and data sizes, which affects the bandwidth usage. It is 20 characters plus the absolute value of a random sample of a normal distribution with a standard deviation of 100.

The same operators are used for the cyclic and non-cyclic topologies, although one small adjustment was made. In a cyclic topology, each event also contains a cycle count. Whenever it is sent along the cyclic edge, this count is increased by one. Once an event has been sent three times across the cyclic edge, it will be removed the next time it is sent across the cyclic edge. This is to avoid potentially infinitely cycling events which, as discussed in Section 5.2, would never be fully acknowledged and get replayed after a timeout because Storm would consider it to have failed its processing. In addition to the operator's output with the bandwidth waste and the cycle count in a cyclic topology, a timestamp of the original input data creation of the test driver is included to allow tracking of the application's latency. The test driver is discussed in the following section.

6.2.3 Test Driver

The test driver acts as the data generator for the stream processing application. Each input event that is being generated for the application only contains the current timestamp. During the processing within the topology, this timestamp is always copied and, as such, contained in the output without modification. When the test driver receives an output tuple, it compares the timestamp to the current time to compute the latency. Additionally, it computes and records metrics such as the throughput, the current computational resource utilisation using Storm UI's REST API and the information on the scheduling process from Redis.

The test driver is structured into four threads, a data generator, a thread each to send and receive events from the stream processing application via sockets and, finally, one to aggregate and output the metrics. It has been benchmarked to ensure that it is capable of handling a significantly larger throughput than is experienced in any of the actual experiments. Additionally, it has been verified that it does not significantly impact the measured latency. During development, it has been verified that for small topologies, like T-M3, the minimum latency measured is at or less than one millisecond, but such a performance is only achievable if the entire topology is co-located on the same host. The methodology to measure the latency, which does include the transmission to and from the stream processing application and the processing within the test driver, has therefore been shown to be precise and introduces only a negligible amount of latency compared to the actual application and placement.

The methodology by Karimov et al. utilised sustainable throughput as a metric. The main idea is that the throughput is sustainable if the application can process the data at the given rate and, as such, the latency does not grow indefinitely, because events have to queue up to be processed. The maximum sustainable throughput is then simply the highest input rate that is still sustainable. Karimov et al. used the idea of overloading a topology with a too-high input rate and decreasing it until the latency stabilised to find the maximum sustainable throughput [KRK+18]. Lambert et al. instead went the

opposite way of increasing the input rate until the output does not scale linearly with it anymore [LGI20]. The maximum sustainable throughput effectively describes the highest possible throughput at which the service can offer consistently low latency and, as such, is highly relevant when optimising for the latency.

In practice, there were difficulties in accurately and reproducibly measuring the maximum sustainable throughput. Both methodologies rely on accepting some margin of error in their acceptance of what can be identified as sustainable. This is particularly problematic with the upscaling methodology of Lambert et al., as it was observed that output rates could more than double in comparison to the previous second at identical input rates. While the methodology of Karimov et al. can avoid this by relying on latency and thereby being more consistent, the actual stream processing application is being overloaded with events queued at every bottleneck. As such, the application being tested has overall more potential to see bursts of throughput and is, therefore, less consistent with its throughput, which can then affect the measured latency. Additionally, the accuracy with which the metric can be measured is directly related to how slowly the input is being adjusted. Overall this means that while the maximum sustainable throughput can be measured, the actual metric has a high level of variance and can sometimes be inaccurate because the process of measuring it has the potential for errors itself. This also resulted in difficulties in reproducing measurements within the same experiment. The accuracy can be improved by slowing the measurement process, but it also significantly increases the time and, therefore, the cost needed to run a single experiment.

In conclusion, the simpler measurement of the maximum throughput based on the output rate was chosen as the primary metric. Sustainable throughput is still being considered in the evaluation, but with the caveat that its assessment is less accurate, because of a relatively quick ramp-up in the input rate. To measure both metrics, the input rate is increased until the application is overloaded, at which point the maximum throughput is measured. For the sustainable throughput, the linear upscaling is considered to be violated if the average latency exceeds the minimum observed average latency by 100ms or more, resulting in the previous input rate being considered as the maximum sustainable one. To account for the large variance in the output rate, the median of the maximum output rates is measured over 20 seconds, while the application is overloaded, which is referred to as the throughput of the application for this evaluation. An application is considered as being overloaded if the minimum latency measured is above a threshold. For this evaluation, 300ms was empirically selected as a suitable threshold.

Across the experiments, there was not a significant difference detected between the relative performance of schedulers when comparing the maximum output rate and the median maximum output rate. In other words, none of the schedulers was more effective in one or the other metric but instead achieved similar relative performances. As such, any observation made should apply to both metrics similarly. Still, the median output rate provided more consistent measurements and is therefore used as the definition for throughput considerations in this evaluation.

A single experimental run is executed in stages. This is done for two reasons. First, an

overloaded topology results in infinitely growing latency, which is why the maximum throughput and latency measurements are not performed in parallel. Secondly, the adaptive online scheduling approaches implemented in this thesis need time to adapt the placement to a certain level of work being performed. As such, any measurement is preceded by a constant rate of input events to allow for potential adjustments of placements. The input rates, therefore, indirectly decide how many computational resources should be utilised in the placement. As such, they were set based on observations in the specific experimental environment and the performance of each topology. For the static approaches which already exist in Apache Storm, the maximum and intended number of workers to utilise is directly configured at the submission of the topology.

An experiment, therefore, primarily consists of a constant throughput rate, followed by a latency measurement and then a throughput measurement. To measure the latency, it is enough to ensure that the application has processed all events and then supplying a low constant rate of input events. With Storm's existing static schedulers, this process is performed once and different scales are investigated by increasing the worker number at topology submission and the resource requests for the operators. To evaluate the adaptive schedulers, this process is repeated three times, specifically once with a low constant input rate, followed by a larger one to test the scale-out behaviour and, finally, the initial input rate again to test the scale-in.

The dynamic schedulers are given two minutes to adjust the placement with each constant input rate, except during the scale-out phase, as a longer ten-minute time period is used there to also test the stability of the placements. The two-minute period was chosen in part because the test driver slowly adjusts to the intended throughput level. Additionally, metrics at the given throughput level have to be collected and propagated to the scheduler and then the new placement has to be executed by Storm. Adjusting a placement with Storm is not a fast process and was observed to need around at least 20 seconds where the service is being interrupted for a majority of the time. 30 seconds of effective downtime and even much longer ones are very common and mostly independent of the actual amount of changes made.

As such, it is critical to ensure that Storm has finished executing a placement modification and the application is fully functional again before measuring its metrics. This is done by adjusting the input rate and waiting for the output rate to match because Storm does not directly provide information about ongoing topology changes. Repeating a pattern of providing no input followed by some input twice ensures that the application is not overloaded, has started or finished any current topology change and all operators are operational again. Similarly, the test driver waits for the application to have processed all events after it has measured the throughput or at the start up to ensure that all queued-up events have been processed before the next constant input rate stage is started. Additionally, the adaptive placement heuristics are disabled outside of the simulation of a constant input rate.

Finally, for topologies T-R8 and T-R9, only their latency is measured and not the throughput. This is because they are cyclic and a throughput measurement can trigger

the back-pressure mechanism of Apache Storm. In a cyclic topology, the mechanism can lead to a deadlock, because an entire cycle of operators can detect an overload and stop processing data until their successor notifies them that it can accept data again. There seems to be no option to effectively disable the mechanism to avoid this deadlock and, as such, collecting throughput measurements is problematic. Of course, as discussed previously, there is no indication that cyclic topologies are being supported and recent Storm versions even warn about detecting a cycle at the time of submission. As such, the cyclic topologies are mostly a curiosity and not an essential part of this evaluation. With a full understanding of the experimental setup, the following section presents the results.

6.3 Experimental Results

The evaluation is split into two parts. First, the latency estimation is discussed and then the placement heuristics.

6.3.1 Latency Estimation Evaluation

For the latency estimation, a slightly larger network with 40 hosts and ten switches was used, as discussed in Section 6.2.1. The larger network could be used because all the other stream-related services were not needed and thereby freed up some computational resources. The estimations are compared to the mean of 100 ping measurements that have each been collected from each host to all others in both directions.

For the comparison, the mean absolute error, mean squared error and mean relative error are used. If x_i represents the measurement and y_i the prediction, then mean absolute error is defined as $E_{MAE} = \frac{1}{n}\sum_{i=1}^{n}|x_i - y_i|$, the mean squared error as $E_{MSE} = \frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2$ and the mean relative error as $E_{MARE} = \frac{1}{n}\sum_{i=1}^{n}\frac{|x_i - y_i|}{|x_i|}$ [Bot19].

The evaluation of the latency estimation primarily aims to record the observed performance because it is used as an input for the placement heuristic, which is the focus of the thesis. In Section 4.5, some possible variations have been discussed, primarily the selections of peers and the linear or log-based error and force calculations. These are compared to each other and an implementation of the simple Vivaldi algorithm using a constant δ, because of the large similarities of the approaches [DCKM04]. The following sections showcase direct comparisons of the selected design variations.

Error and Force Calculation

First off, Figure 6.7 showcases how the linear error calculation converges a lot faster and with a lower error overall when compared to the log-based forces. In addition to performing worse, the log-based forces also aim to minimise the relative error. This results in an increased mean absolute error to the closest peers, as Figure 6.8 shows, which is particularly undesirable for the application in this thesis. Interestingly, the linear error calculations also outperform the log-based ones when considering the relative error. As a result, it is very clear that the linear calculation is preferable, although the

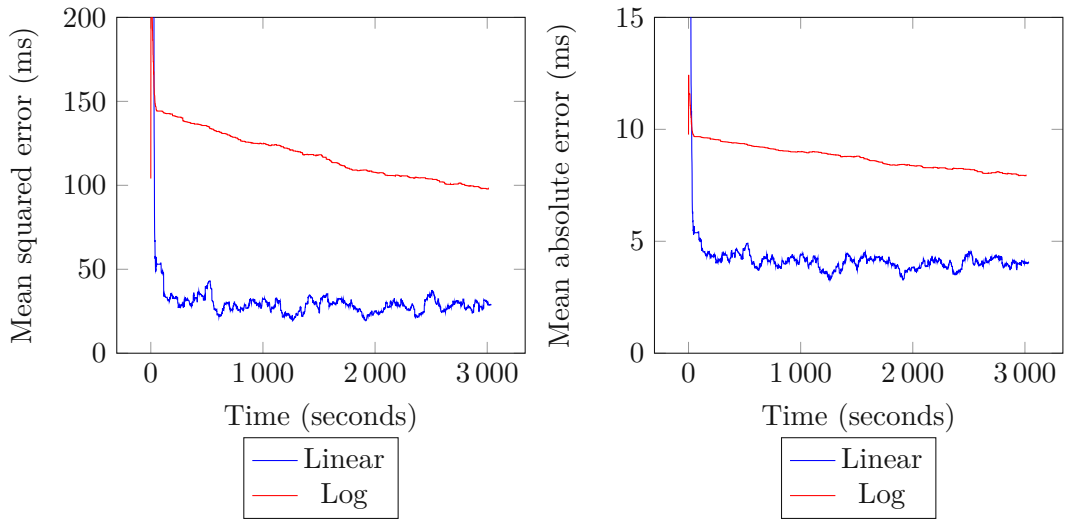


Figure 6.7: Comparison of the mean squared and mean absolute latency error across time for the linear and log-based error and force calculations.

log-based calculations provide one advantage, which is being a lot more stable in the estimations.

Peer Selection

The second major variant to compare is the selection of peers. This could happen completely randomly or it could prefer nearby peers in an attempt to reduce the local error, as these are the peers most likely to interact during stream processing when optimising for latency. As one would expect, the strategy to always reuse the three closest peers results in a general increase in the mean error, which is shown in Figure 6.9. The actual difference is very minor and only observable towards the end of the experiment. Over the last ten minutes of the experiment, preferring nearby peers results in a mean error of 4.08ms, while a fully random peer selection has a mean error of 3.62ms.

Preferring nearby peers results in a slightly reduced standard deviation of 0.1993 compared to 0.2 for the mean error values. This is because reusing the nearby peers results in more stable position estimations, which is likely also the cause of the slightly slower reduction of the error during start up. The comparison of the absolute and squared error metrics also highlights that the estimation errors are relatively even spread and not dominated by few estimates with particularly large errors.

Figure 6.10 shows the expected result that the latency estimation is improved for nearby peers and, as such, has smaller local errors when they are regularly preferred as peers. The actual difference in the metrics is mostly noticeable because the closest peers cause above-average errors with a random peer selection. In contrast, with the nearby peer selection, the estimation errors are similar to the random peer selection for the other

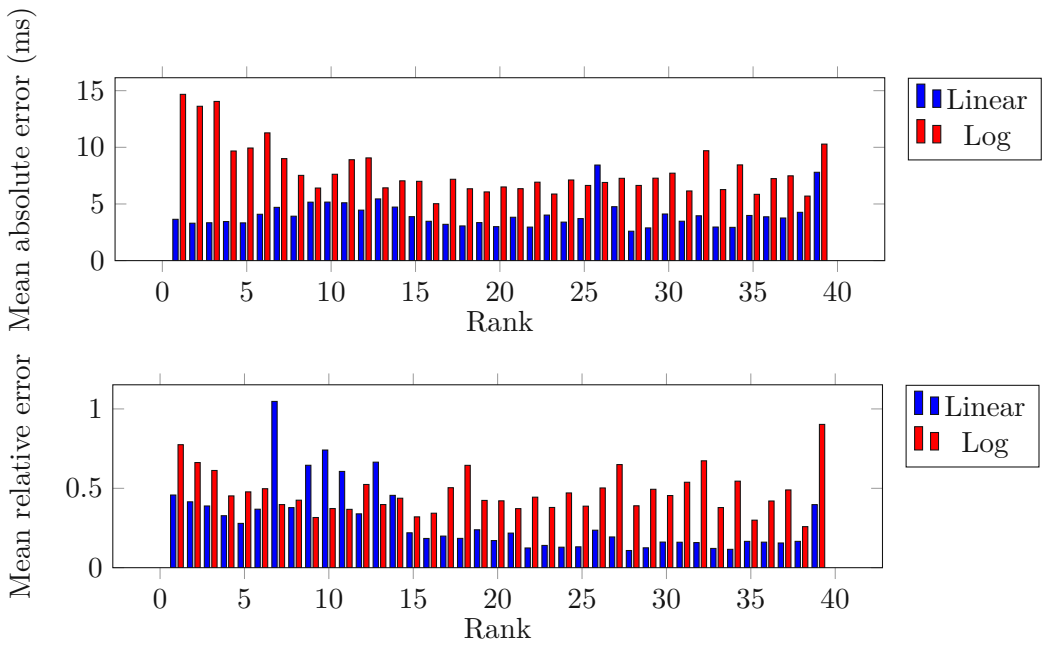


Figure 6.8: Mean absolute and mean relative errors of the estimations to the *nth* closest host ranked by their measured latency after 3000 seconds of runtime.

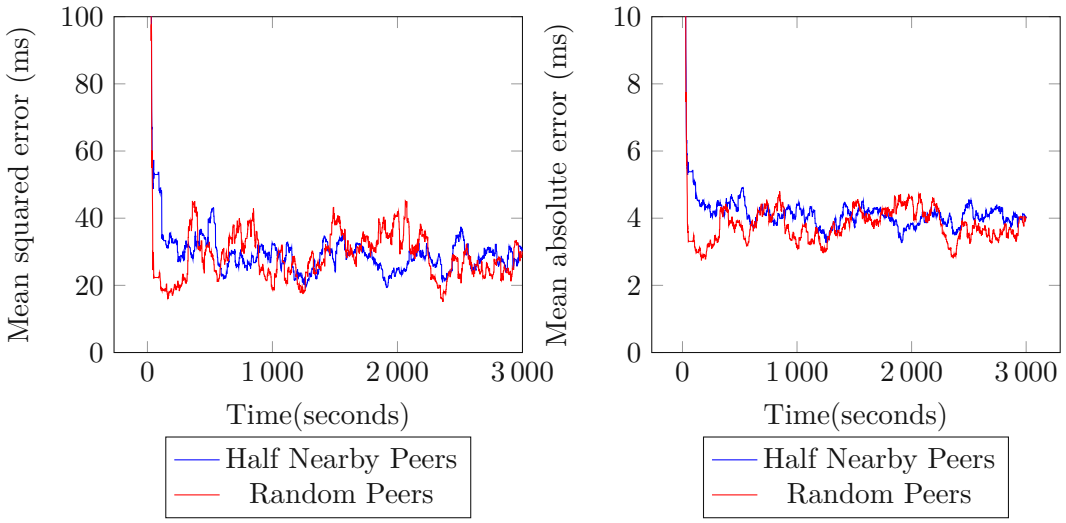


Figure 6.9: Comparison of the mean squared and mean absolute latency error across time for a completely random or nearby peer preferring selection.

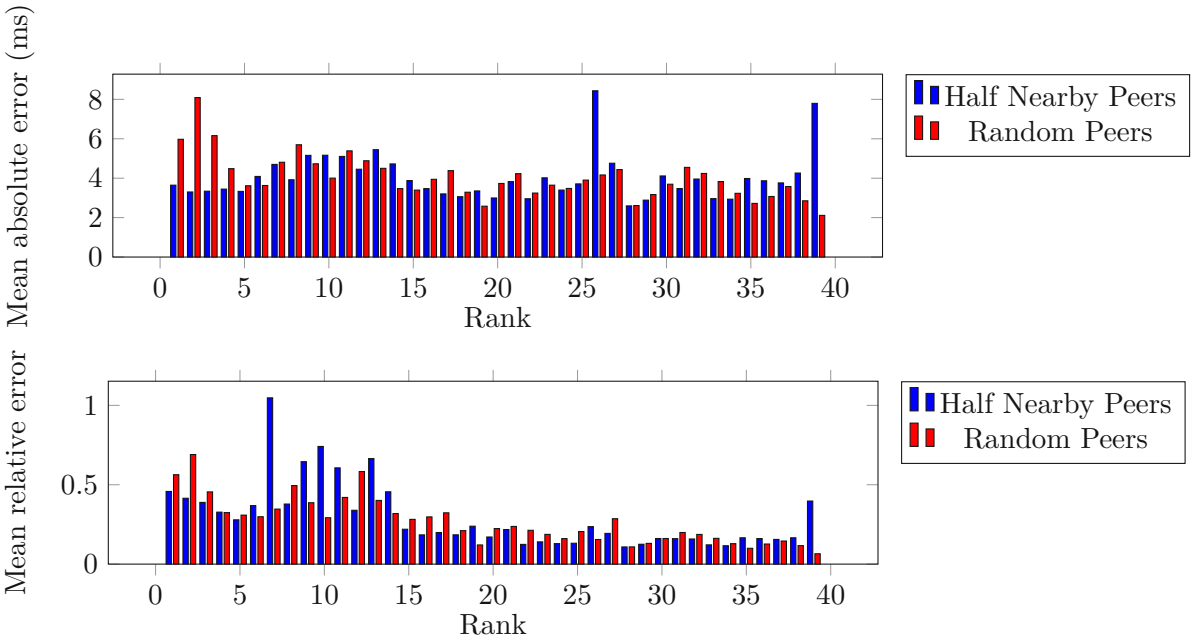


Figure 6.10: Mean absolute and mean relative errors of the estimations to the *nth* closest host ranked by their measured latency after 3000 seconds of runtime.

peers, but large divergences for some estimations are more common.

While preferring nearby peers does accomplish the intended trade-off of local and global estimation errors, the mechanism to accomplish this is both very simplistic and an extreme case. Always reusing the three closest peers does lead to significant error reductions for these peers, which, interestingly, are also the ranks with the highest mean error for a fully random selection. An approach that does not use a specific number of peers constantly but, for example, has a set of nearby peers from which three are being picked every round is likely to generalise better and potentially increases global estimation errors less. As such, while the value of selecting some nearby peers could be showcased, the current algorithm or implementation is less refined and likely problematic in or not tuned for more general cases. For this reason, a fully random peer selection has been used for the operator placement heuristic evaluation rather than the, in this specific case, better-performing nearby peer selection that was proposed.

Median Filter

The final design decision to validate is the usage of the median filter. Instead of filtering the data through a median filter and then performing multiple force iterations based on the median, the ping samples can be used directly for the force calculations. The simple Vivaldi algorithm does this and, essentially, does a single force calculation for each peer sample. Considering the previous sections, an implementation with linear forces

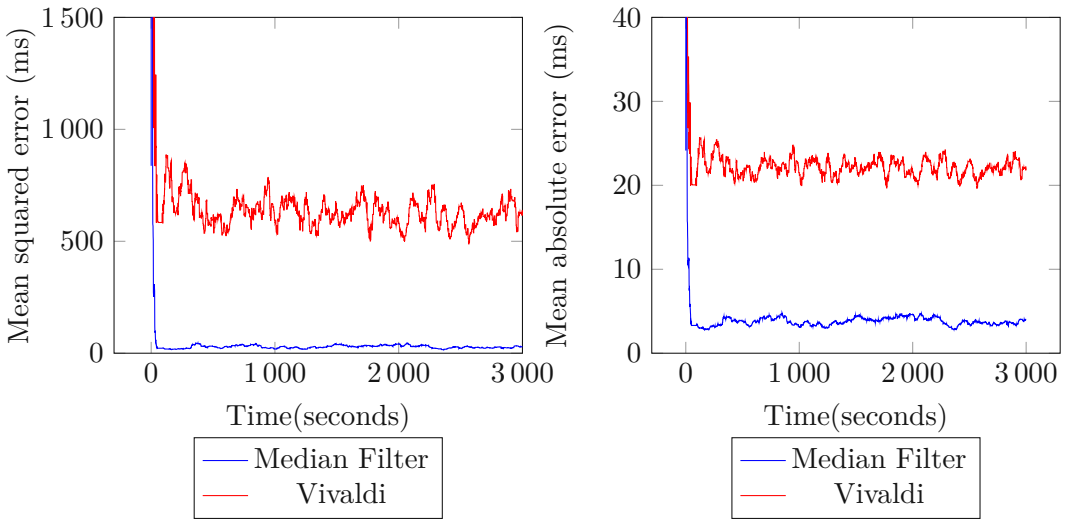


Figure 6.11: Comparison of the mean squared and mean absolute latency errors across time between using a media filter and Vivaldi applying the force calculation directly with each ping sample.

is, therefore, identical to the simple Vivaldi algorithm, except for how the samples are utilised. The Vivaldi algorithm only specifies how the ping samples are utilised and not which peers to choose. As such, the same random peer selection strategy is chosen. The evaluation of Vivaldi has shown that at least five per cent of peers should be distant to ensure lower estimation errors. The special case of half of the peers being distant peers is stated to lead to fast convergence, which is also the configuration of the proposed solution, which was based on empirical observations during the development [DCKM04].

In Figure 6.11, their performance is compared. Considering that the approaches are nearly identical, the large difference in the performance is slightly surprising but easy to explain. The median filter protects against outliers in the ping measurement. With Vivaldi, a single outlier sample will immediately result in a large error and, therefore, movement force being applied. This mistaken movement can then require multiple samples to undo and achieve a similarly accurate position again.

This is particularly problematic in emulated networks with Mininet. During development, it has been observed that while Mininet does accomplish a high accuracy when emulating latency, there are occasional outliers. The exact cause could not be identified and various configurations have been attempted to eliminate the issue. Additionally, the occurrence is not caused by some sort of resource overload, as it could reliably be observed in minimal networks at near idle. Overall, the problem has never been observed during the stream processing evaluation, but instead only with the latency estimation. As such, the hypothesis for this problem is related to a network link being idle and, in rare cases, the actual transfers then being delayed, potentially because of scheduling, unloaded caches or some wakeup procedure. As such, the emulated network causes rare outlier ping

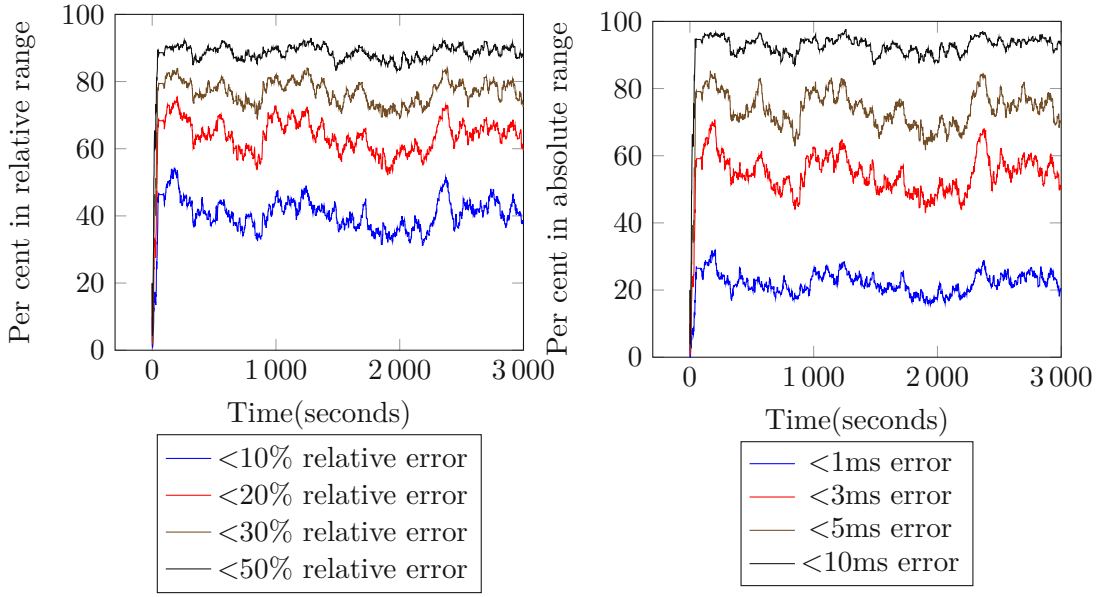


Figure 6.12: Percentage of estimations within certain error ranges across time.

measurements, which the Vivaldi algorithm does not handle well.

This could be considered as Mininet not being suited for such an emulation task, but this is likely an issue with most available network emulators for the fog. Emufog is based on MaxiNet, which is an extension of Mininet, and, as such, likely has the same issue [MGG+17]. Mockfog and Mockfog2 use the Linux Traffic Control utility tc to emulate the network delay, which is also the case in Mininet and likely additional network emulators [HGG+19, HGB21, HHJ+12]. Therefore, they probably share this issue, as it has only been observed in Mininet when tc is utilised with a configured delay.

One might assume that these outliers also significantly affect the mean measurement of the pings that are used as the ground truth in this evaluation. Because of their rarity and the large sample size, the impact on the mean is very minimal. As a result, the mean measurement is still primarily defined by the high consistency at which Mininet can emulate the latency. For most network links, the difference between the minimum and mean measured latency is less than 200μs. Even in individual measurements where outliers have been observed, the mean is still only offset by less than one millisecond for measured latencies around 30ms. The difference between the mean to the minimum is so insignificant that the already presented graphs can not showcase this difference effectively at their current scales. Therefore, using the mean is still a valid and highly accurate metric as the ground truth for the evaluation because the skew caused by rare outliers is negligible.

Figure 6.12 presents the absolute and relative error ranges across time. Just like with the mean and mean squared error, these slightly improve over time, but the difference is

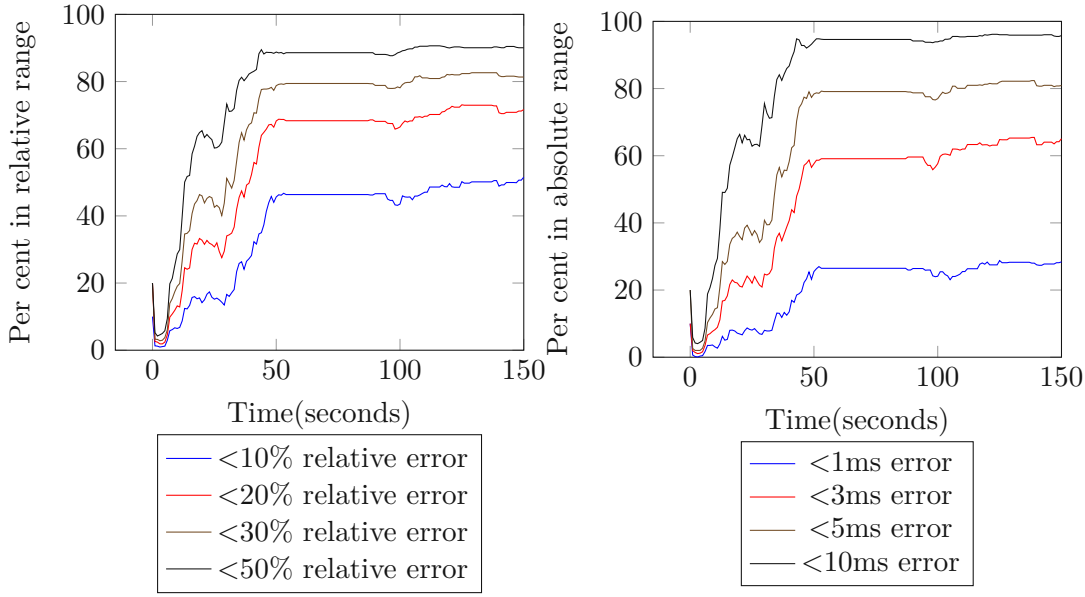


Figure 6.13: Percentage of estimations within certain error ranges during start up.

barely noticeable and overshadowed by the variance in the estimations. In other networks, Vivaldi has been reported to achieve median relative errors of around eleven per cent depending on the network and configuration [DCKM04]. A similar performance to that is shown in Figure 6.12 for the proposed solution. As such, the conclusion from this is that both solutions can have near identical overall performance, but the proposed median filter reduces the estimation errors in less consistently performing networks, which is, of course, desirable.

To conclude the evaluation of this component, there are also some general observations that have been made. The latency estimation tends to underestimate compared to the mean measurements. This is because the mean measurements, of course, include the occasional spike in latency, while the estimation, which uses a median filter, is closer to the actual minimum network latency. Another reason is that the initially random coordinates are selected in a smaller area than the estimators later position themselves in. As such, the network slowly expands over time, leading to a reduction in estimation errors.

The experiment showcases a worst-case scenario in which all estimators join the network in less than four seconds. As such, most estimators have no or only a few peers as landmarks to identify their initial position, which results in a mostly random placement. Figure 6.13 shows how various error ranges across time change during the start up of all estimators. Because the first estimation rounds occur every four seconds, the estimation errors can be quickly reduced. After 50 seconds, the estimated positions have mostly stabilised. This is in part because the estimators have switched to 60-second wait times between estimation rounds at this point in time, but also because each estimation round

does not cause significant movements anymore. This can be seen by the far less significant changes after the 100-second mark. The convergence speed is deemed sufficient for evaluating the stream processing placement heuristic, which restarts the cluster for each experiment. In practice, the speed of convergence could likely be further improved by using the local error estimation of both peers during the force calculation, as has been shown with the adaptive δ step variant of Vivaldi [DCKM04].

Overall, the estimation has similarly low estimation errors even when only estimating positions in two dimensions rather than with three dimensions, but this is likely only the case because the network itself is trivial to lay out in a two-dimensional space. With that said, the actual estimation accuracy is not as important for the placement heuristic as one might initially believe. During the development and tuning of parameters, it was observed that even wildly inaccurate estimations were useful for the placement heuristic. As long as nearby and far hosts could be differentiated with a better than random accuracy, it is an information gain and, as such, helped with placements.

The following section continues with the evaluation of the placement heuristic.

6.3.2 Placement Heuristic Evaluation

This section compares the performance of the proposed adaptive placement heuristics to Apache Storm's default scheduler and the integrated Resource Aware Scheduler, previously also known as R-Storm [Apa22h, Apa22g, PHH+15]. Both schedulers perform static operator placements. The default scheduler uses a round robin-scheduler across a specified amount of worker slots. The Resource Aware Scheduler aims to maximise resource utilisation by packing operators based on their requirements onto workers with the most available resources. Additionally, it aims to reduce latency by reducing a Euclidean distance metric based on a hierarchy of, for example, racks and sub-clusters. As such, it is not aware of any actual latencies. Unfortunately, as the following evaluation shows, the default scheduler, with its essentially random placements, does not perform as well as other solutions. For this reason, the focus of the evaluation is primarily a comparison between the proposed solutions to each other and their performance to the Resource Aware Scheduler.

One difficulty in comparing the performance of the Resource Aware Scheduler to others is that it can be difficult with it to create a variety of placements in the evaluation environment. This is because it aims to utilise the most capable computational resources, while the resource requests of an operator are treated as a requirement and the number of operators to utilise is repurposed as an upper limit. In the heterogeneous evaluation environment, the most capable resource offers five times more CPU than the least capable ones. This leads to somewhat of a limitation of the approach. In the current scheduler implementation, the CPU request of an operator is a requirement to be fulfilled. As such, it has to be less than the resource it is to be placed on. At the same time, to optimise the placement for an intended amount of workers, these requests should be maximised, as otherwise, the scheduler will not spread the operators around but consolidate them on the

Table 6.3: The table shows the average throughput factor increase over the default scheduler at the same CPU reservation for each topology and their operator count. If a direct equivalent does not exist with the Default Scheduler, then linear interpolation between its closest data points is used. If the CPU usage is outside the Default Scheduler's utilisation range, then the next closest value of the Default Scheduler is used. Some entries, primarily for the Resource Aware Scheduler, have only a single throughput factor increase because all placements share the same CPU capacity and therefore have a standard deviation of zero. Additionally, the cyclic topologies T-R8 and T-R9 are not included, because the throughput has not been measured for them.

Topology	**Ops**	**Hill-Climbing**	**Ant System**	**Hybrid**	**Resource Aware Scheduler**
T-M1	6	3.35 (σ=1.37)	4.10 (σ=0.29)	4.15 (σ=0.18)	3.45 (σ=0.00)
T-M2	12	4.32 (σ=4.08)	2.68 (σ=3.28)	3.19 (σ=2.03)	1.94 (σ=0.00)
T-M3	3	7.24 (σ=2.55)	5.28 (σ=4.36)	8.46 (σ=0.00)	10.44 (σ=0.00)
T-R1	30	1.79 (σ=0.62)	1.70 (σ=0.67)	1.68 (σ=0.64)	1.81 (σ=0.55)
T-R2	12	2.19 (σ=1.03)	2.19 (σ=0.95)	2.08 (σ=0.97)	2.91 (σ=0.00)
T-R3	22	2.04 (σ=0.58)	1.90 (σ=0.50)	2.24 (σ=0.60)	2.37 (σ=0.60)
T-R4	28	2.60 (σ=1.01)	2.07 (σ=1.24)	2.06 (σ=0.59)	1.80 (σ=0.91)
T-R5	20	2.10 (σ=1.02)	1.51 (σ=0.97)	2.15 (σ=0.90)	2.13 (σ=1.30)
T-R6	20	1.80 (σ=0.46)	1.71 (σ=0.71)	1.95 (σ=0.70)	2.09 (σ=0.32)
T-R7	42	2.08 (σ=0.72)	1.75 (σ=0.57)	1.92 (σ=1.00)	1.63 (σ=0.79)

largest resource. This leads to somewhat of a dilemma where the resulting placement has to be understood and anticipated to correctly set the resource request. Alternatively, a lot of trial and error has to be used. For this reason, the resource requests of a single operator were limited to the capabilities of the smallest computational resources to guarantee a possible placement, rather than repeated attempts to find a placement that fails. This mostly affects smaller topologies, but in practice, those get consolidated onto a few computational resources anyway. In the original publication, CPU was only considered as a soft constraint and allowed to be over-utilised, thereby avoiding this issue [PHH+15]. Additionally, it is the only scheduler that is deterministic. As such, when compared to the other solutions, its results are based on few but often repeated placements. This also results in usually lower standard deviations than the other approaches for many of the reported metrics because the placements and, therefore, resulting measures are more consistent.

Table 6.3 provides a brief overview of the general throughput that was observed. Because the Resource Aware Scheduler only creates placements with few specific resource utilisations, there is often not a placement from other placement heuristics that can be directly compared. For this reason, the default scheduler is used as a comparison baseline for all solutions because its random placements result in the highest variety. The main idea

of the table is to calculate the average throughput of all placements created by one of the heuristics at a specific resource usage. This average can then be put into relation to the default scheduler to calculate the factor the throughput has been improved by. If a direct reference throughput does not exist, then linear interpolation is used or the closest value if it is outside the range of results. By averaging all these throughput improvement factors, a rough overview of the performance of a heuristic on a specific topology can be provided. Of course, a lot of nuances are lost in this transformation, which is why scatter plots will be used to directly highlight specifics further on in the evaluation.

First off, there is a stark difference between the manually defined topologies and the randomly generated ones. In general, they are much smaller and, as such, the Resource Aware Scheduler places them in all configurations on the largest computational resource, resulting in effectively a single placement. In contrast, the adaptive placement heuristics are more varied, as they also consider scaling down to a less capable resource. Additionally, generally larger throughput increases relative to the default scheduler are measured because all other schedulers optimise for co-location and are aware of the heterogenous operator requirements and resource capabilities.

When the solutions are compared on the random topologies, then a trend can be seen where the Resource Aware Scheduler ($\mu = 2.11$) generally outperforms hill-climbing ($\mu = 2.09$), followed by the hybrid approach ($\mu = 2.01$) and, finally, the ant system ($\mu = 1.83$). Secondly, of the proposed heuristics, the average standard deviations are overall very similar, although, for the ant system, it is around 0.03 larger than the averages of the other proposed solutions indicating slightly more varied placements and performance. Another observable trend is that the proposed solutions perform better in this comparison on topologies with more operators, specifically T-R4 and T-R7, and comparable performance on T-R1 and T-R5. The inverse is true for smaller topologies, where the Resource Aware Scheduler achieves more significant gains. Particularly on the smaller topologies, the actual throughput tends to be higher as there are fewer operators competing for resources with more potential for co-location.

A potential explanation for this is the optimisations of Apache Storm's ackers, which are a part of the optional guaranteed message processing mechanism. The proposed solutions handle them like all the other operators, while the Resource Aware Scheduler and default scheduler spread them evenly. This can lead to the case with the proposed solutions, that not all operators are co-located with ackers. In higher throughput situations, this could then create additional messaging overhead. As such, this oversight may explain the lower performance in some experiments.

Table 6.4 shows instead a head-to-head comparison of placements where identical amounts of resources are being allocated. This shows overall a similar pattern with the proposed solutions performing better on larger topologies, but also highlights that there are some placements in T-R7 where the Resource Aware Scheduler performed unexpectedly worse, with even the default scheduler achieving higher throughput. This is discussed in more detail further on in the chapter when the results of individual topologies are being analysed. Additionally, it shows a more consistent performance for the ant system and

Table 6.4: The table shows the average of the average throughput factor increase over the Resource Aware Scheduler at the same CPU reservation for each topology and their operator count. Some entries are empty, because no direct equivalent placement to compare are available. In some cases, only individual CPU reservation throughput averages are compared and therefore have a standard deviation of zero. Additionally, the cyclic topologies T-R8 and T-R9 are not included, because the throughput has not been measured for them.

Topology	**Ops**	**Hill-Climbing**	**Ant System**	**Hybrid**	**Default Scheduler**
T-M1	6	1.45 (σ=0.00)	1.15 (σ=0.00)		0.29 (σ=0.00)
T-M2	12	0.89 (σ=0.00)		0.73 (σ=0.00)	
T-M3	3	0.89 (σ=0.00)			0.10 (σ=0.00)
T-R1	30	0.98 (σ=0.24)	1.08 (σ=0.02)	1.14 (σ=0.00)	0.70 (σ=0.00)
T-R2	12	0.61 (σ=0.00)	0.97 (σ=0.00)	0.89 (σ=0.00)	0.34 (σ=0.00)
T-R3	22	0.94 (σ=0.04)	0.92 (σ=0.11)	0.96 (σ=0.08)	0.42 (σ=0.13)
T-R4	28	1.11 (σ=0.03)	0.84 (σ=0.28)	1.03 (σ=0.32)	0.83 (σ=0.70)
T-R5	20	0.68 (σ=0.14)	0.72 (σ=0.07)	0.95 (σ=0.19)	0.98 (σ=0.89)
T-R6	20	0.87 (σ=0.19)	0.75 (σ=0.23)	0.96 (σ=0.09)	0.42 (σ=0.00)
T-R7	42	2.61 (σ=2.67)	0.96 (σ=0.45)	1.27 (σ=0.46)	2.31 (σ=2.61)

hybrid approach relative to the Resource Aware scheduler's placements. On average, hill-climbing ($\mu = 1.11$) and hybrid search ($\mu = 1.03$) manage to outperform the Resource Aware Scheduler in this comparison and the ant system achieves a lower average relative performance of 0.89.

If similar comparisons are made with the maximum sustainable throughput, then hybrid search ($\mu = 8.46$) outperforms the Resource Aware Scheduler ($\mu = 7.76$), followed by hill-climbing ($\mu = 7.64$) and the ant system ($\mu = 7.56$). In this comparison, the average standard deviations are particularly high, close to the actual averages for all solutions, indicating very inconsistent maximum sustainable throughput for the default scheduler that forms the basis of this comparison. In the head-to-head comparison to the Resource Aware Scheduler, this throughput ordering repeats, but with much closer results: hybrid search ($\mu = 1.01$) outperforms the Resource Aware Scheduler followed by hill-climbing ($\mu = 0.92$) and the ant system ($\mu = 0.87$). For the proposed solutions, the average standard deviations are between 0.15 and 0.20.

To summarise the considerations on the throughput, hybrid search tends to outperform the Resource Aware Scheduler on average across the random topologies by one to three per cent on the median maximum throughput and maximum sustainable throughput. Hill-climbing performs better on the median max throughput by eleven per cent and worse on the maximum sustainable throughput by 8 per cent and the ant system has eleven to 13 per cent less. More generally, the proposed solutions achieve slightly higher throughput metrics on larger topologies, less on smaller ones and T-R7 is a topology on

which the Resource Aware Scheduler did not perform well on.

The measured latency performance of the Resource Aware Scheduler to all proposed solutions was generally very similar across most topologies. In direct comparisons, hill-climbing achieves a two per cent lower average latency, while hybrid search has a four per cent increased average latency. Ant system again performs worse with a 15 per cent increased average latency. The maximum latency is increased by five, seven and 30 per cent, respectively. In contrast, the minimum latency is reduced by 36, 23 and 22 per cent. In all the comparisons, the ant system has a higher average standard deviation between 0.25 and 0.34, while the other proposed solutions are more consistent with average standard deviations between 0.11 and 0.18.

The difference in minimum and maximum latency behaviour points to similar average latency behaviour, but with a different prioritisation of the multiple paths of a topology. This is contradicted by the results of T-R3 and, for example, T-R6, which are highly linear topologies and still show this pattern. The difference is likely not caused by the latency component in the scoring function, because it only optimises the maximum network latency and has a minimal weighting. Additionally, the network latency is overall only a smaller part of the average or maximum latencies being reported, because of the very short delays in the network topology and the few workers being utilised. Instead, the reduction of network overhead by co-locating data-intensive operators is likely the cause. Processing bottlenecks can also be excluded because of the adaptive placements and the already low throughput while measuring latencies. As such, this points to communication overhead or synchronisation costs being the cause. Depending on the fog network, it might therefore be more impactful to reduce the number of network hops in a path rather than the accumulated latency or, ideally, a combination of both. Alternatively, the weighting of the latency scoring function could also be adjusted to increase its importance, but this may not be desirable because maximising co-location and reducing the number of workers involved has generally been observed to have a far larger positive impact on throughput and latency.

The worse performance of the ant system is likely caused by the random order in which the solution is constructed. This makes the consideration of latency and especially co-location difficult for which the integrated heuristic could not account sufficiently. More iterations might have helped because of the memory capabilities of the pheromones, but this adds cost.

Table 6.5 compares the average runtime in milliseconds to compute a single placement. The default scheduler, unfortunately, does not report such a statistic, but its runtime is likely lower than the Resource Aware Scheduler. Overall, the Resource Aware Scheduler provides by far the lowest runtimes, except on the smallest topologies. This is expected, because the proposed heuristics perform iterative adjustments and additionally consider latency and the exchange of data in the placement problem. On most topologies, the hybrid solution requires the time of hill-climbing in addition to the ant system, which is also expected because it consists of running both solutions after each other. The ant system has a higher constant cost, because it has a high minimum iteration count to

Table 6.5: The table shows the average runtimes of the proposed heuristics in milliseconds.

Topology	Ops	Hill-Climbing	Ant System	Hybrid	Resource Aware Scheduler
T-M1	6	9 (σ=2.25)	75 (σ=7.88)	78 (σ=4.67)	16 (σ=2.08)
T-M2	12	45 (σ=5.94)	99 (σ=18.35)	130 (σ=19.84)	21 (σ=3.39)
T-M3	3	4 (σ=0.55)	35 (σ=1.82)	38 (σ=2.83)	16 (σ=2.70)
T-R1	30	558 (σ=125.68)	450 (σ=38.86)	841 (σ=31.95)	25 (σ=4.34)
T-R2	12	65 (σ=14.52)	131 (σ=21.14)	197 (σ=26.40)	20 (σ=3.16)
T-R3	22	231 (σ=49.60)	285 (σ=36.29)	534 (σ=60.07)	23 (σ=3.32)
T-R4	28	179 (σ=59.91)	447 (σ=80.54)	717 (σ=80.49)	24 (σ=3.59)
T-R5	20	138 (σ=46.62)	306 (σ=56.16)	428 (σ=32.54)	22 (σ=3.03)
T-R6	20	198 (σ=75.98)	274 (σ=67.17)	472 (σ=40.59)	21 (σ=3.47)
T-R7	42	1029 (σ=28.00)	661 (σ=84.76)	1020 (σ=26.41)	26 (σ=9.38)
T-R8	14	185 (σ=61.83)	377 (σ=41.18)	355 (σ=63.65)	19 (σ=2.60)
T-R9	18	263 (σ=77.06)	408 (σ=125.23)	480 (σ=140.60)	21 (σ=3.00)

counteract the probabilistic solution construction. In contrast, it scales better than the other proposed solutions, because of the constant amount of ants or solutions being computed rather than the quadratically growing neighbourhood in hill-climbing. The pheromone can also focus the search on a smaller search space over time. Only on T-R7 were hill-climbing and the hybrid search limited by the runtime limit defined as one second, but, as discussed previously, it did not have a particularly noticeable effect on the quality of the placements. Additionally, the Resource Aware Scheduler uses a similarly configurable timeout that is defined as 60 seconds by default. As such, this may have been an overly strict design requirement. But this shows that for increasingly larger topologies generating the full move neighbourhood for hill-climbing will be a performance issue and should, in these cases, be replaced by smaller or randomised neighbourhoods.

To get a better understanding, some of the individual results are discussed. Figure 6.14 shows the median maximum throughput for T-R3. It is used to describe the general behaviour in more detail and representative of the average case. To start with, the figure can be considered to consist of two major clusters in the scatter plot. The default scheduler is consistently located near the bottom and forms a cluster of its own. All other heuristics form a cluster near the centre. Additionally, clusters on the same vertical line from individual heuristics are recognisable. These are essentially similar or identical placements and show that even with identical placements, there is a large variance in the measured performance. They are most notable for the Resource Aware Scheduler, which only produces a few placement variations, and the default scheduler, which has a line in the bottom right that is the utilisation of all eleven computational resources.

It can be seen that the round-robin placement of the default scheduler results in relatively consistent, but worse, performance. A large part of this is that at practically any resource

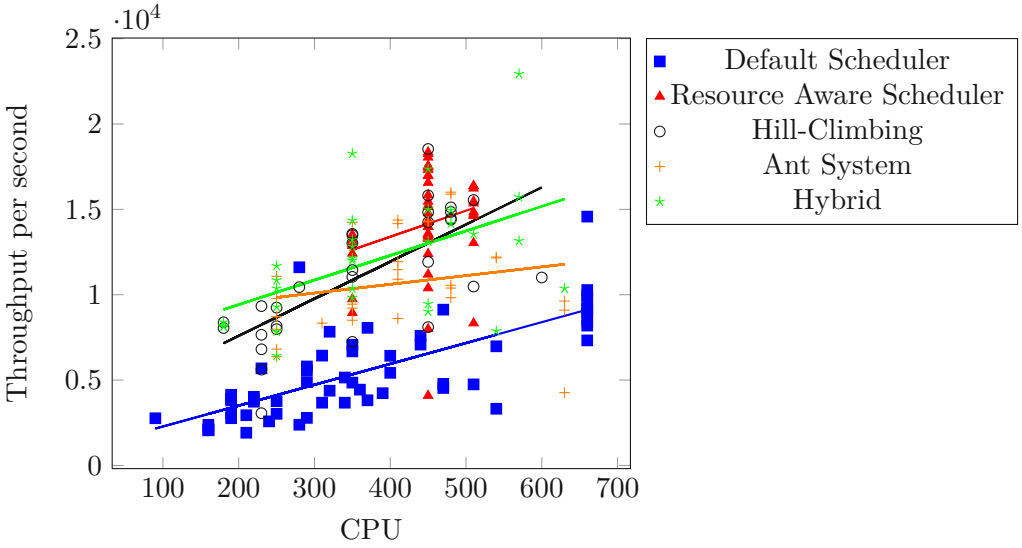


Figure 6.14: Scatter plot of the median maximum throughput of individual placements for all placement heuristics with trend lines for T-R3.

utilisation, co-location of operators is unlikely. Most placements are, therefore, highly inefficient, but more consistent. The Resource Aware Scheduler shows the contrast of similarly dense groupings of placements, but ones which largely benefit from co-location in addition to avoiding resource bottlenecks, because of its awareness of heterogeneity. While the Resource Aware Scheduler usually does not produce the best performing placements, they are consistently among the best performing, resulting in its generally above-average performance. The most varied performance is showcased by the adaptive heuristics, primarily the current resource utilisation introduces a lot of variance in the placements.

The least resource-consuming placements outside of the default scheduler are generally created by hill-climbing and, in turn, the hybrid approach. In contrast, the ant system that does not perform such a greedy search is far more spread in both its performance and resource utilisation.

In Figure 6.15, the minimum measured latency is shown for the same topology. It is, again, straightforward to tell that the default scheduler performs significantly worse. This is in part because most operators are not co-located and therefore use more network links. Another reason is that the default scheduler can struggle on some topologies with achieving a minimum throughput. The figure shows many similarities to the throughput.

Now that the general case is well understood, some special cases will be discussed. In T-M3, only three operators are in the topology, as such when accounting for co-locality their placements are essentially identical. Still, the Resource Aware Scheduler achieves higher throughput. The reason for this is likely the additional metrics consumer that has to be deployed with the proposed solutions to collect the topologies' metrics and

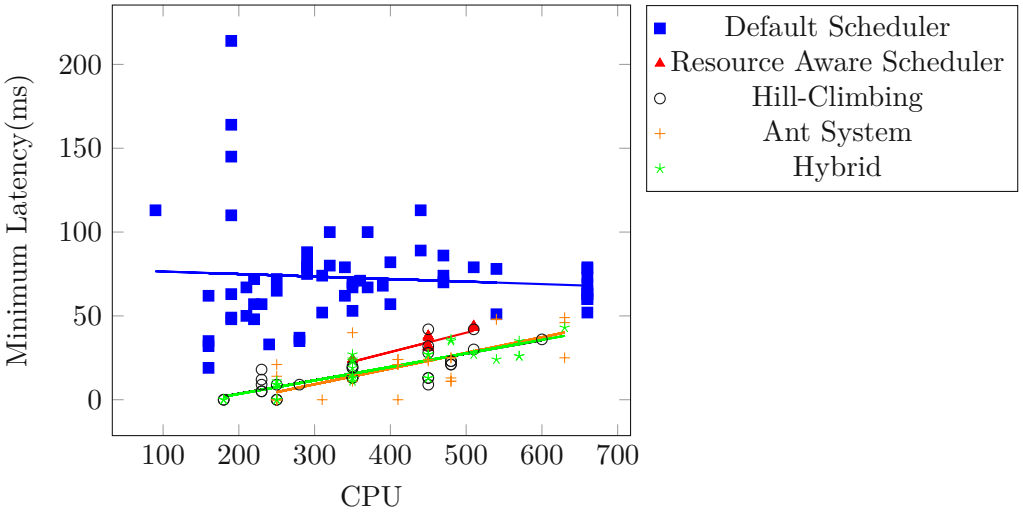


Figure 6.15: Scatter plot of the minimum latency of individual placements for the placement heuristics with trend lines for T-R3.

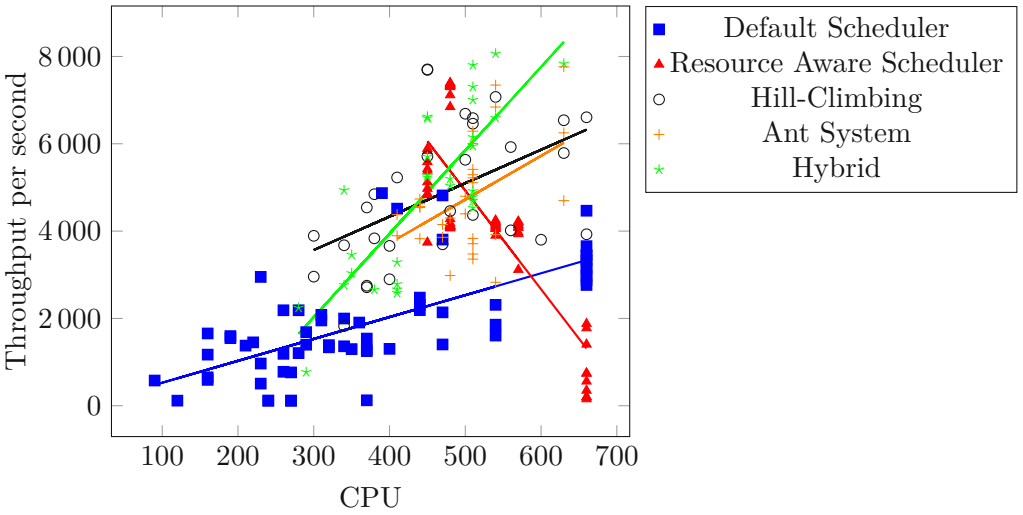


Figure 6.16: Scatter plot of the median maximum throughput of individual placements for all placement heuristics with trend lines for T-R7.

indirectly forward them to the placement heuristic. While the actual workload should be minor, it is still an extra operator compared to the static heuristics, where Storm has not been configured to include the extra operator in the topologies. The actual performance difference seems to be around an eleven per cent throughput loss. In the larger topologies, this difference is less noticeable and is mostly lost in the high variance the metric usually has.

TR-7 is interesting because as the heuristics scale out, the Resource Aware Schedulers

throughput tends to drop, as can be seen in Figure 6.16. Unfortunately, no exceptional cause could be identified with certainty as the source of this behaviour. Similar throughput reductions, but far less significant, occur for the Resource Aware Scheduler on T-R4, T-R5 and potentially T-R1 as well. They occur at the point when the smaller resources are starting to be utilised, which could indicate a problem with communication overhead, but at the same time, this is not observed on T-R6. In contrast, of the proposed solutions, only the ant system has consistent reductions in the throughput as it scales out, but those are less severe and affect fewer topologies. Another explanation is that these are the largest topologies and they highlight the limitations of the greedy heuristic used. At the same time, this could also be a case of misconfiguration, which the adaptive solutions can avoid.

Finally, concerning the cyclic topologies T-R8 and T-R9, it is difficult to directly compare them because the placement heuristics operated at different scales. As such, only the functionality itself could be asserted for all placement heuristics, with the caveat that Apache Storm can deadlock any processing if a cyclic topology is overloaded, turning this into a relatively insignificant feature.

The scaling itself performed reasonably well on all topologies by making significant adjustments within two minutes successfully. The 20-second or longer downtime of the service would, of course, be a problem in practice, especially because data will queue up during that time leading to an even more significant overload. In general, the scale-in performs better than the scale-out. This is in part because all information to estimate the performance is already available before the placement is changed. The adaptive heuristics are purely reactive and do not predict how a scale-out will affect other operators. The only reactive behaviour has the risk of a bottleneck being slowly pushed downstream with each scale-out and, therefore, only being resolved very slowly. The thresholds regarding the utilisation of a worker seem to have prevented this issue insofar that no problematic cases have been noticed.

The stability of a placement mostly depends on the stability of the CPU utilisation of the operators. While multiple placements have often been necessary to identify the correct scale, once found, they were usually stable. One of the problems with Apache Storm's capacity metric is that it is relatively inaccurate. For example, it is not unusual for the reported capacity usage to exceed the total capacity by significant margins. This is because capacity is a partially estimated metric. While the level of accuracy has been sufficient, this is an area that should be improved for actual usage. For example, the latency estimations running on each supervisor could additionally report the actual CPU usage, which could then be used to scale the reported capacity usage estimations of operators to more accurate scales.

A slight oddity was noticed with the Resource Aware Scheduler. As discussed in Section 4.6.1, there is no scheduling benefit to utilising multiple slots on a single worker. In fact, this can even be detrimental to the performance because it can introduce overhead in addition to needlessly blocking an extra slot. In the topologies T-R1, T-R3, T-R4, T-R7 and T-R9, during some of the deployments, the Resource Aware Scheduler utilised

one extra slot. The reason could not be identified and seems unintended. Interestingly, this does not occur deterministically across similarly configured topologies, but seemingly randomly, with an increased tendency with certain topologies. For example, an extra slot was utilised in all configurations of T-R1, but with T-R9 only one run used the correct slot count with three workers, while two runs did not with six workers. All other configurations for T-R9 produced slots counts which matched the supervisor counts as expected. As the evaluation shows, this is not a significant issue, but overall it still seems to be unintended. The extra slot utilisation was also only observed with the Resource Aware Scheduler and, as such, is specific to it and likely not an issue in Apache Storm itself.

To summarise the evaluation, the benefits and feasibility of an adaptive placement heuristic based on estimations of latency have been successfully shown. In all metrics, except for the runtime, the hybrid approach or hill-climbing have been able to perform close to or better than the Resource Aware Scheduler and this without the risk of misconfiguration. While the Resource Aware Scheduler generally achieves better throughput, it is not able to do so as consistently. If these cases are considered, then hill-climbing achieves eleven per cent higher maximum throughput, while the hybrid approach has one to three per cent better maximum sustainable and median maximum throughput. Similarly, average latency improved by two per cent for hill-climbing and worsened by four per cent for the hybrid search. The largest differences are the minimum latency, where all proposed solutions achieved reductions of at least 22 per cent on average, at the cost of increases to the maximum latency. Hill-climbing has a five per cent larger maximum latency, with hybrid search at seven per cent.

The performance generalises reasonably well to all tested topologies and no problematic cases could be identified. Both the hill-climbing and hybrid heuristic performed particularly well, but the hill-climbing seems to be preferable because of a far simpler implementation that has fewer parameters to tune and offers better runtime performance. Additionally, while the ant system generally performed worse in most experiments, it does offer better scalability. Still, many practical limitations and difficulties make creating, maintaining and deploying such a scheduler for a productive system difficult or unviable, such as, for example, the extensive downtime when changing a placement.

CHAPTER 7

Conclusion and Future Work

This section summarises the work of this thesis in Section 7.1 and presents potential future work and research questions based on this work in Section 7.2.

7.1 Conclusion

Stream processing is a software engineering paradigm that offers low latency and high scalability and reliability by splitting an application into smaller operators that can be distributed across computational resources. With the advent of the IoT, the need to process data continues to grow. Fog computing is a new paradigm that aims to move computations closer to the sources of data or the devices relying on it and thereby offer lower latency and reduce the demand on the network caused by long-distance transmissions. As such, stream processing is highly applicable to the processing of IoT data and can be deployed in fog computing environments. Ensuring the best performance with stream processing is therefore of interest, of which a significant part is how the operators are distributed: the stream processing operator placement problem.

The first steps were a discussion of the necessary background of this thesis, which also contains the collection of the different ideas, concepts and understandings of fog computing. Additionally, an extensive survey of the state of the art in the placement of stream processing operators and related domains was made. This information defined requirements for a placement heuristic and the assumptions on fog computing this thesis relies on. Particularly the heterogeneous capabilities of computational resources and awareness of the capabilities of the network grow in importance with fog computing. Additionally, the aim to realise an online scheduler to adapt to changing conditions was defined. Apache Storm was then selected as a suitable modern framework for this thesis, in part for its easy accessibility of the scheduling API and its usage by previous researchers. A latency estimation component was designed, Apache Storm's operator placement model was simplified and a COP to solve was formulated. The COP is generally applicable

to stream processing, except for the Apache Storm-specific constraint of only a limited amount of topologies being allowed to execute on individual computational resources. In addition to the constraints to avoid exceeding a resource's capacities, it contains a scoring function to rate potential placements. The scoring function considers the number of co-located operators, as this allows for additional optimisations, the number of exchanged events over the network, to minimise them, the overall usage of computational resources, to consolidate the usage and allow unused resources to be shut down, the latency and finally a penalty for excessive utilisation.

For the implementation, a hill-climbing heuristic and an ant system meta-heuristic were implemented and integrated into Apache Storm, as well as a hybrid approach combining both solutions. For the edge case of cyclic topologies, a new heuristic to select edges in cyclic graphs to remove and thereby turn them into a DAG was designed and implemented. In contrast to the previous heuristics it was based on, it additionally considers the paths within the topology to minimise the impact of the removed edge on them. It thereby preserves a more accurate cycle-free representation of the cyclic topology for the use case, but at the cost of increased computational complexity. It is used to estimate the latency at the data sinks of a stream processing topology in Apache Storm for cyclic topologies, even though support for cyclic topologies is not necessarily intended or maintained in Apache Storm. Both the architecture of the solution and the process of the implementation, as well as the difficulties with the placement heuristics and the other components, have been documented.

Additionally, it has been discussed why the gains made in research rarely progress into production use, but instead, methods such as round-robin or random assignments are still commonly used, leading to an ever-growing split between research and actual use. Particularly, a lack of documentation, not easily accessible APIs and the actual difficulty in implementing a solution have been identified as causes that significantly raise the complexity more than one would expect for such a task. This forms both a large barrier of entry and difficulty in verifying the correctness and maintaining implementations in the long term.

Finally, the implementation was evaluated with benchmarks of manually defined and randomly generated stream processing topologies on an emulated network. The implementation was compared against Apache Storm's default scheduler and the Resource Aware Scheduler. The Resource Aware Scheduler represents the state of the art with Apache Storm and is already fully integrated. The default scheduler was significantly outperformed in any metric, except for the increased computational cost of scheduling. In comparison to the Resource Aware Scheduler, a much more comparable performance was achieved with a trade-off that often reduces the minimum latency at the cost of the maximum latency. Additionally, on larger topologies, a higher average throughput was measured, but on smaller topologies, the Resource Aware Scheduler generally outperforms the placements of the proposed solution at similar scales. In general, across the various topologies and configurations, good performance was achieved more consistently with the proposed solutions. The capability to adapt placements to the currently observed

conditions was verified and can reduce resource consumption or increase processing capabilities when needed. Unfortunately, the solution provides little use in production, in part due to it being a prototype and because Apache Storm creates significant service interruptions when a placement changes. This is a known problem that researchers have previously been able to solve but has not yet been integrated.

Co-location of operators has also been shown to be highly beneficial to optimise in Apache Storm, and likely other stream processing engines, but may have limited applicability to topologies where a single computational resource may not be capable of hosting multiple operators. Co-location is very significant to the optimisation of throughput and can be used as a cheaper heuristic to minimise latency indirectly in smaller networks. Between hill-climbing and the ant system, the placement problem, as defined and implemented, is better suited for hill-climbing. It allowed for cheaper computations in the evaluation and is more effective at optimising latency and co-location, resulting in improved throughput. An ant system may be more effective if the order of the solution construction of ant only considers operators with already placed neighbours, but this would also limit the exploration of the search space and potentially result in a greater difficulty in escaping local optima. As such, while the results have been shown to provide various improvements and a variety of observations, there is still more work left to be done, which the following section discusses.

7.2 Future Work

The placement problem could be expanded to account for other properties that may become more relevant with fog computing, such as the mobility of devices, their battery availability or the availability of the device and reliability of the stream processing application. Additionally, the network usage could be better modelled as Apache Storm does allow to provide custom definitions to which successor operators the data should be sent, which is an edge case that was not considered. In general, new metrics or constraints can be easily added, because of the definition of the problem as a COP. For example, enactment costs or the operating costs of the computational resources can be easily added by extending the scoring function if such metrics are actually available.

Another limitation is that the current placement is a centralised process, while fog computing is largely decentralised. This mismatch could be resolved by adjusting the heuristic, but it is less of a problem in smaller deployments, which could be private ones. For example, the current centralised placement could be used to manage individual regions, such as cloudlets or geographic areas, rather than the entire cluster. To fully decentralise the placement, large modifications would be necessary and especially the scoring function would be problematic as it requires essentially global knowledge of the topology. Alternatively, a model could be attempted where with each topology, a scheduler dedicated to it is created on any of the resources, thereby providing the centralised knowledge, but avoiding a single resource being the bottleneck for all topologies.

A problem closely related to operator placement is deciding the replication or, in other

words, the number of instances of an operator. In Apache Storm, this is static and, as such, was not a relevant problem, but is available in other stream processing engines and, therefore, is a capability that could be added to further improve elasticity.

The largest improvement in the field of operator placement would likely not come from developing yet another scheduler but instead from improving the environment of this domain. Most schedulers use relatively similar information, resource request, network usage rates, selectivity, network link latencies and so on, while also working with DAGs that form the basis for stream processing. This means that there is the potential to attempt to create a more standardised API that could then be either integrated directly into stream processing engines or as an adapter to existing APIs. While this would mean losing some specificity by creating placements independent of specific stream processing engines and likely miss some specific information, because of new limitations, it would externalise the placement of operators from the actual stream processing engines. This would mean that solutions would not have to be specific to an engine and, as such, could be far easier to reproduce results or compare them among researchers. Additionally, an external component is easier to release, especially when the API is standardised, intended for this usage and documented. This would not only allow for a standardisation of emulators or simulators to reduce the need for costly benchmarks, but also reduce the barrier of entry and difficulty of developing and maintaining a scheduler. The largest problem with the operator placement problem at this time seems not to be developing better schedulers anymore, but having any sort of improvement moving into production use, because there the state of the art is still often random scheduling or round-robin assignments. As much as such a design and integration into various projects would be a monumental effort, so would the gains likely be in the long term.

APPENDIX A

Configurations of the Evaluation Topologies

Table A.1: Parameters of T-M1's operators

Id	Selectivity	CPU	CPU Request for 3/4/5/6/7/11 workers	Memory	Bandwidth	Instances
spout	1.0	10	10.0/10.0/10.0/10.0/10.0/10.0	50.0	0	1
sink	0.0	10	45.0/45.0/45.0/45.0/45.0/45.0	50.0	0	1
1	1.0	90	30.0/30.0/30.0/30.0/30.0/30.0	11.0	100	1
2	1.0	90	30.0/30.0/30.0/30.0/30.0/30.0	11.0	100	1
3	1.0	90	30.0/30.0/30.0/30.0/30.0/30.0	11.0	100	1
4	1.0	90	30.0/30.0/30.0/30.0/30.0/30.0	11.0	100	1

Table A.2: Parameters of T-M2's operators

Id	Selectivity	CPU	CPU Request for 3/4/5/6/7/11 workers	Memory	Bandwidth	Instances
spout	1.0	10	10.0/10.0/10.0/10.0/10.0/10.0	50.0	0	1
sink	0.0	10	20.0/20.0/20.0/20.0/20.0/20.0	50.0	0	1
1	1.0	90	28.0/30.0/30.0/30.0/30.0/30.0	11.0	100	10

Table A.3: Parameters of T-M3's operators

Id	Selectivity	CPU	CPU Request for 3/4/5/6/7/11 workers	Memory	Bandwidth	Instances
spout	1.0	10	10.0/10.0/10.0/10.0/10.0/10.0	50.0	0	1
sink	0.0	10	20.0/20.0/20.0/20.0/20.0/20.0	50.0	0	1
1	1.0	90	30.0/30.0/30.0/30.0/30.0/30.0	11.0	100	1

Table A.4: Parameters of T-R1's operators

Id	Selectivity	CPU	CPU Request for 3/4/5/6/7/11 workers	Memory	Bandwidth	Instances
spout	1.0	10	10.0/10.0/10.0/10.0/10.0/10.0	50.0	0	1
sink	0.0	10	20.0/20.0/20.0/20.0/20.0/20.0	50.0	0	1
0	1.25	119	5.9/8.0/8.5/9.1/9.6/11.7	108.0	189	2
1	0.66	75	3.7/5.0/5.4/5.7/6.0/7.4	61.0	72	2
2	1.32	1	2.0/2.7/2.9/3.0/3.2/3.9	39.0	72	2
3	1.4	1	2.0/2.7/2.9/3.0/3.2/3.9	46.0	73	2
4	1.28	58	7.6/10.3/11.0/11.7/12.4/15.1	79.0	56	2
5	0.54	104	24.3/30.0/30.0/30.0/30.0/30.0	25.0	53	2
6	1.12	231	29.3/30.0/30.0/30.0/30.0/30.0	86.0	49	2
7	1.54	1	2.0/2.7/2.9/3.0/3.2/3.9	37.0	210	2
8	0.71	1	2.0/2.7/2.9/3.0/3.2/3.9	213.0	25	2
9	0.41	168	30.0/30.0/30.0/30.0/30.0/30.0	26.0	145	2
10	0.97	113	8.3/11.2/11.9/12.7/13.4/16.4	36.0	106	2
11	0.77	118	8.4/11.4/12.2/12.9/13.7/16.7	20.0	114	2
12	1.18	130	7.2/9.7/10.4/11.0/11.7/14.2	18.0	23	2
13	0.62	113	7.3/10.0/10.6/11.3/11.9/14.6	19.0	80	2

Table A.5: Parameters of T-R2's operators

Id	Selectivity	CPU	CPU Request for 3/4/5/6/7/11 workers	Memory	Bandwidth	Instances
spout	1.0	10	10.0/10.0/10.0/10.0/10.0/10.0	50.0	0	1
sink	0.0	10	20.0/20.0/20.0/20.0/20.0/20.0	50.0	0	1
0	0.99	108	25.0/30.0/30.0/30.0/30.0/30.0	119.0	72	2
1	0.46	80	18.5/25.1/26.8/28.4/30.0/30.0	67.0	35	2
2	0.97	148	30.0/30.0/30.0/30.0/30.0/30.0	52.0	74	2
3	1.22	237	30.0/30.0/30.0/30.0/30.0/30.0	89.0	42	2
4	1.12	23	9.3/12.6/13.4/14.2/15.0/18.4	20.0	29	2

Table A.6: Parameters of T-R3's operators

Id	Selectivity	CPU	CPU Request for 3/4/5/6/7/11 workers	Memory	Bandwidth	Instances
spout	1.0	10	10.0/10.0/10.0/10.0/10.0/10.0	50.0	0	1
sink	0.0	10	20.0/20.0/20.0/20.0/20.0/20.0	50.0	0	1
0	1.16	61	4.2/5.8/6.1/6.5/6.9/8.4	193.0	40	2
1	0.97	43	3.0/4.1/4.3/4.6/4.9/5.9	41.0	33	2
2	0.66	291	30.0/30.0/30.0/30.0/30.0/30.0	28.0	77	2
3	1.11	142	13.8/18.7/19.9/21.2/22.4/27.3	57.0	36	2
4	1.22	191	20.5/27.9/29.7/30.0/30.0/30.0	77.0	145	2
5	1.35	91	11.9/16.2/17.3/18.3/19.4/23.7	99.0	158	2
6	1.19	29	5.2/7.0/7.5/7.9/8.4/10.2	58.0	58	2
7	1.36	54	11.4/15.5/16.5/17.5/18.5/22.6	74.0	139	2
8	1.2	53	15.3/20.7/22.1/23.4/24.8/30.0	31.0	154	2
9	0.67	34	11.7/15.9/17.0/18.0/19.1/23.3	12.0	55	2

Table A.7: Parameters of T-R4's operators

Id	Selectivity	CPU	CPU Request for 3/4/5/6/7/11 workers	Memory	Bandwidth	Instances
spout	1.0	10	10.0/10.0/10.0/10.0/10.0/10.0	50.0	0	1
sink	0.0	10	20.0/20.0/20.0/20.0/20.0/20.0	50.0	0	1
0	0.48	223	30.0/30.0/30.0/30.0/30.0/30.0	67.0	25	2
1	0.69	158	24.7/30.0/30.0/30.0/30.0/30.0	89.0	56	2
2	1.26	5	6.2/8.5/9.0/9.6/10.1/12.4	13.0	79	2
3	0.52	65	6.2/8.5/9.0/9.6/10.1/12.4	65.0	181	2
4	0.92	142	15.2/20.7/22.0/23.4/24.7/30.0	61.0	22	2
5	1.07	41	6.2/8.5/9.0/9.6/10.1/12.4	59.0	93	2
6	0.37	31	9.1/12.4/13.2/14.0/14.9/18.1	95.0	100	2
7	0.69	47	6.2/8.5/9.0/9.6/10.1/12.4	59.0	135	2
8	0.83	51	6.2/8.5/9.0/9.6/10.1/12.4	28.0	24	2
9	0.62	59	6.2/8.5/9.0/9.6/10.1/12.4	15.0	139	2
10	0.38	71	6.2/8.5/9.0/9.6/10.1/12.4	62.0	29	2
11	0.66	68	6.2/8.5/9.0/9.6/10.1/12.4	76.0	87	2
12	0.62	143	6.2/8.5/9.0/9.6/10.1/12.4	73.0	129	2

Table A.8: Parameters of T-R5's operators

Id	Selectivity	CPU	CPU Request for 3/4/5/6/7/11 workers	Memory	Bandwidth	Instances
spout	1.0	10	10.0/10.0/10.0/10.0/10.0/10.0	50.0	0	1
sink	0.0	10	20.0/20.0/20.0/20.0/20.0/20.0	50.0	0	1
0	1.19	128	30.0/30.0/30.0/30.0/30.0/30.0	21.0	192	2
1	0.92	29	10.8/14.7/15.6/16.6/17.5/21.4	18.0	45	2
2	1.02	42	13.5/18.3/19.5/20.7/21.9/26.8	21.0	127	2
3	0.69	1	10.8/14.7/15.6/16.6/17.5/21.4	14.0	78	2
4	0.3	159	27.2/30.0/30.0/30.0/30.0/30.0	72.0	54	2
5	0.42	87	10.8/14.7/15.6/16.6/17.5/21.4	30.0	116	2
6	0.3	86	10.8/14.7/15.6/16.6/17.5/21.4	11.0	156	2
7	1.13	273	10.8/14.7/15.6/16.6/17.5/21.4	29.0	107	2
8	0.99	263	10.8/14.7/15.6/16.6/17.5/21.4	12.0	63	2

Table A.9: Parameters of T-R6's operators

Id	Selectivity	CPU	CPU Request for 3/4/5/6/7/11 workers	Memory	Bandwidth	Instances
spout	1.0	10	10.0/10.0/10.0/10.0/10.0/10.0	50.0	0	1
sink	0.0	10	20.0/20.0/20.0/20.0/20.0/20.0	50.0	0	1
0	0.93	1	7.0/9.5/10.2/10.8/11.4/13.9	11.0	83	2
1	0.63	81	14.2/19.3/20.6/21.8/23.1/28.2	60.0	35	2
2	0.3	70	11.4/15.5/16.6/17.6/18.6/22.7	131.0	135	2
3	1.06	190	30.0/30.0/30.0/30.0/30.0/30.0	15.0	88	2
4	1.38	17	7.0/9.5/10.2/10.8/11.4/13.9	64.0	36	2
5	1.08	123	20.9/28.4/30.0/30.0/30.0/30.0	134.0	76	2
6	0.6	109	19.9/27.1/28.8/30.0/30.0/30.0	40.0	173	2
7	0.74	64	22.0/29.9/30.0/30.0/30.0/30.0	104.0	186	2
8	0.77	9	7.0/9.5/10.2/10.8/11.4/13.9	137.0	46	2

Table A.10: Parameters of T-R7's operators

Id	Selectivity	CPU	CPU Request for 3/4/5/6/7/11 workers	Memory	Bandwidth	Instances
spout	1.0	10	10.0/10.0/10.0/10.0/10.0/10.0	50.0	0	1
sink	0.0	10	20.0/20.0/20.0/20.0/20.0/20.0	50.0	0	1
0	1.05	206	10.8/14.7/15.7/16.6/17.6/21.5	23.0	162	2
1	0.76	134	7.4/10.0/10.7/11.3/12.0/14.6	21.0	70	2
2	0.56	29	2.1/2.9/3.0/3.2/3.4/4.2	60.0	65	2
3	0.98	106	5.8/7.9/8.5/9.0/9.5/11.6	61.0	64	2
4	0.95	161	13.7/18.5/19.8/21.0/22.2/27.1	36.0	114	2
5	0.5	43	5.3/7.2/7.6/8.1/8.6/10.5	79.0	52	2
6	0.89	148	9.1/12.3/13.1/13.9/14.8/18.0	42.0	58	2
7	1.21	126	7.7/10.5/11.2/11.9/12.6/15.3	58.0	25	2
8	0.63	73	9.4/12.8/13.6/14.5/15.3/18.7	19.0	307	2
9	1.06	199	16.1/21.8/23.3/24.7/26.1/30.0	54.0	26	2
10	0.73	1	2.1/2.9/3.0/3.2/3.4/4.2	69.0	305	2
11	1.25	145	12.4/16.8/17.9/19.0/20.1/24.5	12.0	124	2
12	1.19	188	11.1/15.1/16.1/17.1/18.1/22.1	20.0	108	2
13	0.46	1	2.1/2.9/3.0/3.2/3.4/4.2	64.0	117	2
14	1.03	51	3.6/4.9/5.2/5.5/5.9/7.1	74.0	184	2
15	0.71	21	4.3/5.9/6.3/6.7/7.1/8.6	88.0	65	2
16	1.01	1	2.1/2.9/3.0/3.2/3.4/4.2	74.0	268	2
17	0.75	1	2.1/2.9/3.0/3.2/3.4/4.2	44.0	71	2
18	1.11	1	2.1/2.9/3.0/3.2/3.4/4.2	55.0	29	2
19	0.68	87	10.6/14.4/15.4/16.3/17.3/21.0	14.0	188	2

Table A.11: Parameters of T-R8's operators

Id	Selectivity	CPU	CPU Request for 3/4/5/6/7/11 workers	Memory	Bandwidth	Instances
spout	1.0	10	10.0/10.0/10.0/10.0/10.0/10.0	50.0	0	1
sink	0.0	10	20.0/20.0/20.0/20.0/20.0/20.0	50.0	0	1
0	1.35	1	12.1/16.4/17.4/18.5/19.6/23.9	81.0	103	2
1	0.84	64	26.1/30.0/30.0/30.0/30.0/30.0	52.0	86	2
2	0.96	62	25.3/30.0/30.0/30.0/30.0/30.0	41.0	69	2
3	1.21	84	28.8/30.0/30.0/30.0/30.0/30.0	21.0	51	2
4	1.05	58	19.9/27.0/28.8/30.0/30.0/30.0	42.0	54	2
5	0.83	71	27.8/30.0/30.0/30.0/30.0/30.0	72.0	45	2

Table A.12: Parameters of T-R9's operators

Id	Selectivity	CPU	CPU Request for 3/4/5/6/7/11 workers	Memory	Bandwidth	Instances
spout	1.0	10	10.0/10.0/10.0/10.0/10.0/10.0	50.0	0	1
sink	0.0	10	20.0/20.0/20.0/20.0/20.0/20.0	50.0	0	1
0	0.3	120	28.6/30.0/30.0/30.0/30.0/30.0	23.0	53	2
1	0.97	166	30.0/30.0/30.0/30.0/30.0/30.0	73.0	204	2
2	0.94	95	9.5/13.0/13.8/14.7/15.5/18.9	103.0	26	2
3	0.8	21	9.5/13.0/13.8/14.7/15.5/18.9	23.0	32	2
4	0.66	136	9.5/13.0/13.8/14.7/15.5/18.9	90.0	36	2
5	0.85	42	9.5/13.0/13.8/14.7/15.5/18.9	113.0	84	2
6	0.76	54	9.5/13.0/13.8/14.7/15.5/18.9	54.0	208	2
7	1.74	153	24.0/30.0/30.0/30.0/30.0/30.0	96.0	134	2

List of Figures

List of Tables

List of Algorithms

Acronyms

API Application Programming Interface. 6, 21, 37, 51, 57, 58, 67, 70, 71, 80, 87–91, 103, 123, 124, 126

CaaS Containers as a Service. 6, 7, 20

CDN content delivery network. 39, 45

COP constrained optimisation problem. 32, 41, 47, 48, 123, 125

CSP constraint satisfaction problem. 32, 42

DAG directed acyclic graph. 9–11, 23, 33, 38–42, 72–74, 78–80, 88, 99, 100, 124, 126, 134

IaaS Infrastructure as a Service. 6, 8, 20

ILP Integer Linear Programming. 41

IoT Internet of Things. xi, xiii, 3–5, 8, 13–22, 29, 53, 97, 123, 133

MaCE Maximum Cumulative Excess. 42, 49

MCC Mobile Cloud Computing. 22

MEC Mobile Edge Computing. 22

MOPA Multi-operator Placement Algorithm. 39, 46, 47

NIST National Institute of Standards and Technology. 17, 18

PaaS Platform as a Service. 6, 7, 20

QoS Quality of Service. 34, 39, 45–48, 59

REST Representational State Transfer. 70, 103

SaaS Software as a Service. 6, 20

SBON Stream-based Overlay Network. 39, 46–48, 60–62

SDN Software-Defined Networking. 21, 95

VNF Virtualised Network Function. 45, 49, 50

Bibliography

[AA12] Najah AbuAli and Mervat Abu-Elkheir. Data management for the internet of things: Green directions. In *Workshops Proceedings of the 31st Global Communications Conference, GLOBECOM 2012, 3-7 December 2012, Anaheim, California, USA*, pages 386–390. IEEE, 2012.

[AA20] Adam A. Alli and Muhammad Mahbub Alam. The fog cloud of things: A survey on concepts, architecture, standards, tools, and applications. *Internet of Things*, 9:100177, 2020.

[AAB19] Asad-ur-rehman, Rui L. Aguiar, and João Paulo Barraca. Network functions virtualization: The long road to commercial deployments. *IEEE Access*, 7:60439–60464, 2019.

[ABQ13] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in Storm. In *The 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13, Arlington, TX, USA, June 29 - July 03, 2013*, pages 207–218. ACM, 2013.

[Ac04] Yanif Ahmad and Uğur Çetintemel. Network-aware query processing for stream-based applications. In *Proceedings of the 30th International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, 2004*, volume 30, page 456–467. VLDB Endowment, 2004.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 26th ACM SIGMOD International Conference on Management of Data, SIGMOD 2000, Dallas, Texas, USA, May 16-18, 2000*, pages 261–272. ACM, 2000.

[Ama22] Amazon Webservices. Amazon Kinesis data streams. `https://aws.amazon.com/kinesis/data-streams/`, 2022. Accessed: 2022-11-28.

[Apa18] Apache Software Foundation. Apache Spark news: Spark release 2.3.0. `https://spark.apache.org/releases/spark-release-2-3-0.html`, 2018. Accessed: 2022-11-28.

[Apa19] Apache Storm PMC. Apache Storm news: Apache storm 2.0.0 released. https://storm.apache.org/2019/05/30/storm200-released.html, 2019. Accessed: 2022-11-28.

[Apa21] Apache Software Foundation. Apache Spark documentation: Continuous processing. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#continuous-processing, 2021. Accessed: 2022-11-28.

[Apa22a] Apache Software Foundation. Apache Pulsar. https://pulsar.apache.org/, 2022. Accessed: 2022-11-28.

[Apa22b] Apache Software Foundation. Apache Samza. https://samza.apache.org/, 2022. Accessed: 2022-11-28.

[Apa22c] Apache Software Foundation. Apache Storm documentation: Faq. https://storm.apache.org/releases/2.4.0/FAQ.html, 2022. Accessed: 2022-11-28.

[Apa22d] Apache Software Foundation. Apache Storm documentation: Guaranteeing message processing. https://storm.apache.org/releases/2.4.0/Guaranteeing-message-processing.html, 2022. Accessed: 2022-11-28.

[Apa22e] Apache Software Foundation. Apache Storm documentation: JavaDoc. https://storm.apache.org/releases/2.4.0/javadocs/index.html, 2022. Accessed: 2022-11-28.

[Apa22f] Apache Software Foundation. Apache Storm documentation: Metrics reporting API v2. https://storm.apache.org/releases/2.4.0/metrics_v2.html, 2022. Accessed: 2022-11-28.

[Apa22g] Apache Software Foundation. Apache Storm documentation: Resource aware scheduler. https://storm.apache.org/releases/2.4.0/Resource_Aware_Scheduler_overview.html#Enhancements-on-original-DefaultResourceAwareStrategy, 2022. Accessed: 2022-11-28.

[Apa22h] Apache Software Foundation. Apache Storm documentation: Scheduler. https://storm.apache.org/releases/2.4.0/Storm-Scheduler.html, 2022. Accessed: 2022-11-28.

[Apa22i] Apache Software Foundation. Apache Storm documentation: Setting up a storm cluster. https://storm.apache.org/releases/2.4.0/Setting-up-a-Storm-cluster.html, 2022. Accessed: 2022-11-28.

[Apa22j] Apache Software Foundation. Apache Storm documentation: Storm metrics. https://storm.apache.org/releases/2.4.0/Metrics.html, 2022. Accessed: 2022-11-28.

[Apa22k] Apache Software Foundation. Apache Storm documentation: Storm state management. https://storm.apache.org/releases/2.4.0/State-checkpointing.html, 2022. Accessed: 2022-11-28.

[Apa22l] Apache Software Foundation. Apache Storm documentation: Storm UI REST API. https://storm.apache.org/releases/2.4.0/STORM-UI-REST-API.html, 2022. Accessed: 2022-11-28.

[Apa22m] Apache Software Foundation. Apache Storm documentation: Understanding the parallelism of a Storm topology. https://storm.apache.org/releases/2.4.0/Understanding-the-parallelism-of-a-Storm-topology.html, 2022. Accessed: 2022-11-28.

[Apa22n] Apache Software Foundation. Apache Storm github: Version 2.4.0 scheduler implementations. https://github.com/apache/storm/tree/a432e99bca526886655cc1d5b2453a09b302b5ca/storm-server/src/main/java/org/apache/storm/scheduler, 2022. Accessed: 2022-11-28.

[ATB+19] Cristian Axenie, Radu Tudoran, Stefano Bortoli, Mohamad Al Hajj Hassan, Carlos Salort Sánchez, and Goetz Brasche. Dimensionality reduction for low-latency high-throughput fraud detection on datastreams. In *18th IEEE International Conference On Machine Learning And Applications, ICMLA 2019, Boca Raton, FL, USA, December 16-19, 2019*, pages 1170–1177. IEEE, 2019.

[AUK+15] Sahar Arshad, Saeed Ullah, Shoab Ahmed Khan, M. Daud Awan, and M. Sikandar Hayat Khayal. A survey of cloud computing variable pricing models. In *Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2015, Barcelona, Spain, 29-30 April, 2015*, pages 27–32. SciTePress, 2015.

[AVA19] Mattia Antonini, Massimo Vecchio, and Fabio Antonelli. Fog computing architectures: A reference for practitioners. *IEEE Internet Things Magazine*, 2(3):19–25, 2019.

[BGM+20] Maycon Viana Bordin, Dalvan Griebler, Gabriele Mencagli, Cláudio F. R. Geyer, and Luiz Gustavo Leão Fernandes. DSPBench: A suite of benchmark applications for distributed data stream processing systems. *IEEE Access*, 8:222900–222917, 2020.

[Bip22] Bipin Prasad. Apache Storm news: Apache storm 2.4.0 released. https://storm.apache.org/2022/03/25/storm240-released.html, 2022. Accessed: 2022-11-28.

[BMZA12] Flavio Bonomi, Rodolfo A. Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the 1st edition of the workshop on Mobile cloud computing, MCC@SIGCOMM 2012, Helsinki, Finland, August 17, 2012*, pages 13–16. ACM, 2012.

[Bot19] Alexei Botchkarev. A new typology design of performance metrics to measure errors in machine learning regression algorithms. *Interdisciplinary Journal of Information, Knowledge, and Management*, 14:045–076, 2019.

[BR03] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[CAG20] Batyr Charyyev, Engin Arslan, and Mehmet Hadi Gunes. Latency comparison of cloud datacenters and edge servers. In *IEEE Global Communications Conference, GLOBECOM 2020, Virtual Event, Taiwan, December 7-11, 2020*, pages 1–6. IEEE, 2020.

[CDE+16] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Peng, and Paul Poulosky. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*, pages 1789–1792. IEEE Computer Society, 2016.

[CGB+11] Badrish Chandramouli, Jonathan Goldstein, Roger S. Barga, Mirek Riedewald, and Ivo Santos. Accurate latency estimation in a distributed event processing system. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, Hannover, Germany, April 11-16, 2011*, pages 255–266. IEEE Computer Society, 2011.

[CGPN15a] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Distributed QoS-aware scheduling in Storm. In Frank Eliassen and Roman Vitenberg, editors, *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015*, pages 344–347. ACM, 2015.

[CGPN15b] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. On QoS-aware scheduling of data stream applications over fog computing infrastructures. In *2015 IEEE Symposium on Computers and Communication, ISCC 2015, Larnaca, Cyprus, July 6-9, 2015*, pages 271–276. IEEE Computer Society, 2015.

[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. *IEEE Data Engineering Bulletin*, 38(4):28–38, 2015.

[CNL16] Valeria Cardellini, Matteo Nardelli, and Dario Luzi. Elastic stateful stream processing in Storm. In *14th International Conference on High Performance Computing & Simulation, HPCS 2016, Innsbruck, Austria, July 18-22, 2016*, pages 583–590. IEEE, 2016.

[CRC16] Paolo Cappellari, Mark Roantree, and Soon Ae Chun. A scalable platform for low-latency real-time analytics of streaming data. In *Data Management Technologies and Applications - 5th International Conference, DATA 2016, Lisbon, Portugal, July 24-26, 2016, Revised Selected Papers*, volume 737 of *Communications in Computer and Information Science*, pages 1–24. Springer, 2016.

[CSI+20] Anne Collin, Afreen Siddiqi, Yuto Imanishi, Eric Rebentisch, Taisetsu Tanimichi, and Olivier L. de Weck. Autonomous driving systems hardware and software architecture exploration: optimizing latency and cost under safety constraints. *Systems Engineering*, 23(3):327–337, 2020.

[CZM20] Ankit Chaudhary, Steffen Zeuch, and Volker Markl. Governor: Operator placement for a unified fog-cloud environment. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, pages 631–634. OpenProceedings.org, 2020.

[dAVB18] Marcos Dias de Assunção, Alexandre Da Silva Veith, and Rajkumar Buyya. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *Network and Computer Applications*, 103:1–17, 2018.

[DCKM04] Frank Dabek, Russ Cox, M. Frans Kaashoek, and Robert Tappan Morris. Vivaldi: a decentralized network coordinate system. In Raj Yavatkar, Ellen W. Zegura, and Jennifer Rexford, editors, *Proceedings of the ACM Conference on Applications, SIGCOMM '04, Technologies, Architectures, and Protocols for Computer Communication,Portland, Oregon, USA, August 30 - September 3, 2004*, pages 15–26. ACM, 2004.

[DLL+16] Ruilong Deng, Rongxing Lu, Chengzhe Lai, Tom H. Luan, and Hao Liang. Optimal workload allocation in fog-cloud computing toward balanced delay and power consumption. *IEEE Internet of Things Journal*, 3(6):1171–1181, 2016.

[DM74] Jack B. Dennis and David Misunas. A preliminary architecture for a basic data flow processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture, Houston, TX, USA, December 1974*, pages 126–132. ACM, 1974.

[DMC96] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 26(1):29–41, 1996.

[DSX+19] Rong Du, Paolo Santi, Ming Xiao, Athanasios V. Vasilakos, and Carlo Fischione. The sensable city: A survey on the deployment and management for smart city monitoring. *IEEE Communications Surveys and Tutorials*, 21(2):1533–1560, 2019.

[DTD19] Michele De Donno, Koen Tange, and Nicola Dragoni. Foundations and evolution of modern computing paradigms: Cloud, IoT, edge, and fog. *IEEE Access*, 7:150936–150948, 2019.

[Ead84] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[ELS93] Peter Eades, Xuemin Lin, and W. F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, 1993.

[Eth21] Ethan Li. Apache Storm news: Apache storm 2.3.0 released. `https://storm.apache.org/2021/09/27/storm230-released.html`, 2021. Accessed: 2022-11-28.

[FMIF18] Manoel C. Silva Filho, Claudio C. Monteiro, Pedro R. M. Inácio, and Mário M. Freire. Approaches for optimizing virtual machine placement and migration in cloud environments: A survey. *Journal of Parallel and Distributed Computing*, 111:222–250, 2018.

[FMXY12] Xi Fang, Satyajayant Misra, Guoliang Xue, and Dejun Yang. Smart grid - the new and improved power grid: A survey. *IEEE Communications Surveys and Tutorials*, 14(4):944–980, 2012.

[FRA+19] Muhammad Shoaib Farooq, Shamyla Riaz, Adnan Abid, Kamran Abid, and Muhammad Azhar Naeem. A survey on the role of IoT in agriculture for the implementation of smart farming. *IEEE Access*, 7:156237–156271, 2019.

[fTISA19] Alliance for Telecommunications Industry Solutions (ATIS). ATIS telecom glossary (ATIS-0100523.2019). `https://glossary.atis.org/`, 2019. Accessed: 2022-11-28.

[FW09] Yongquan Fu and Yijie Wang. HyperSpring: Accurate and stable latency estimation in the hyperbolic space. In *15th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2009, Shenzhen, China, December 8-11, 2009*, pages 864–869. IEEE Computer Society, 2009.

[Goo22] Google Cloud. Google cloud dataflow. `https://cloud.google.com/dataflow`, 2022. Accessed: 2022-11-28.

[GPFS02] Anastasios Gounaris, Norman W. Paton, Alvaro A. A. Fernandes, and Rizos Sakellariou. Adaptive query processing: A survey. In *Advances in Databases, 19th British National Conference on Databases, BNCOD 19, Sheffield, UK, July 17-19, 2002, Proceedings*, volume 2405 of *Lecture Notes in Computer Science*, pages 11–25. Springer, 2002.

[HGB21] Jonathan Hasenburg, Martin Grambow, and David Bermbach. MockFog 2.0: Automated execution of fog application experiments in the cloud. *IEEE Transactions on Cloud Computing*, pages 1–1, 2021.

[HGG+19] Jonathan Hasenburg, Martin Grambow, Elias Grünewald, Sascha Huk, and David Bermbach. MockFog: Emulating fog computing infrastructure in the cloud. In *IEEE International Conference on Fog Computing, ICFC 2019, Prague, Czech Republic, June 24-26, 2019*, pages 144–152. IEEE, 2019.

[HHJ+12] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *Conference on emerging Networking Experiments and Technologies, CoNEXT 2012, Nice, France,December 10-13, 2012*, pages 253–264. ACM, 2012.

[HKH+19] Thomas Hiessl, Vasileios Karagiannis, Christoph Hochreiner, Stefan Schulte, and Matteo Nardelli. Optimal placement of stream processing operators in the fog. In *3rd IEEE International Conference on Fog and Edge Computing, ICFEC 2019, Larnaca, Cyprus, May 14-17, 2019*, pages 1–10. IEEE, 2019.

[HLR+13] Kirak Hong, David J. Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. Mobile fog: a programming model for large-scale applications on the internet of things. In *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing, MCC@SIGCOMM 2013, Hong Kong, China, August 16, 2013*, pages 15–20. ACM, 2013.

[HMA19] Mohamed-K. Hussein, Mohamed-H. Mousa, and Mohammed A. Alqarni. A placement architecture for a container as a service (CaaS) in a cloud environment. *Journal of Cloud Computing*, 8:7, 2019.

[HPS+15] Hugo Hromic, Danh Le Phuoc, Martin Serrano, Aleksandar Antonic, Ivana Podnar Zarko, Conor Hayes, and Stefan Decker. Real time analysis of sensor data for the internet of things by means of clustering and event processing. In *2015 IEEE International Conference on Communications, ICC 2015, London, United Kingdom, June 8-12, 2015*, pages 685–691. IEEE, 2015.

[HSS+13] Martin Hirzel, Robert Soulé, Scott Schneider, Bugra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys*, 46(4):46:1–46:34, 2013.

[HV19] Cheol-Ho Hong and Blesson Varghese. Resource management in fog/edge computing: A survey on architectures, infrastructure, and algorithms. *ACM Computing Surveys*, 52(5):97:1–97:37, 2019.

[HVSD16] Christoph Hochreiner, Michael Vögler, Stefan Schulte, and Schahram Dustdar. Elastic stream processing for the internet of things. In *9th IEEE International Conference on Cloud Computing, CLOUD 2016, San Francisco, CA, USA, June 27 - July 2, 2016*, pages 100–107. IEEE Computer Society, 2016.

[HVWD16] Christoph Hochreiner, Michael Vögler, Philipp Waibel, and Schahram Dustdar. VISP: an ecosystem for elastic data stream processing for the internet of things. In *20th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2016, Vienna, Austria, September 5-9, 2016*, pages 1–11. IEEE Computer Society, 2016.

[HYA+15] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. The rise of "big data" on cloud computing: Review and open research issues. *Information Systems*, 47:98–115, 2015.

[IC91] Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. In *Proceedings of the 17th ACM SIGMOD International Conference on Management of Data, SIGMOD 1991,Denver, Colorado, USA, May 29-31, 1991*, pages 268–277. ACM Press, 1991.

[IDR07] Zachary G. Ives, Amol Deshpande, and Vijayshankar Raman. Adaptive query processing: Why, how, when, and what next? In *Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB 2007, University of Vienna, Austria, September 23-27, 2007*, pages 1426–1427. ACM, 2007.

[IEE18] IEEE. *IEEE Standard 1934-2018 for Adoption of OpenFog Reference Architecture for Fog Computing*, August 2018.

[IEE20] IEEE. *IEEE Standard 2413-2019 an Architectural Framework for the Internet of Things (IoT)*, March 2020.

[IGF+18] Michaela Iorga, Nedim S. Goren, Larry Feldman, Robert Barton, Michael J. Martin, and Charif Mahmoudi. Fog computing conceptual model. Special Publication (SP) 500-325, National Institute of Standards and Technology (NIST), March 2018.

[JL14] Paul Jakma and David Lamparter. Introduction to the Quagga routing suite. *IEEE Networks*, 28(2):42–48, 2014.

[JMLL16] Jingjie Jiang, Shiyao Ma, Bo Li, and Baochun Li. Symbiosis: Network-aware task scheduling in data-parallel frameworks. In *35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-14, 2016*, pages 1–9. IEEE, 2016.

[JRNR20] Motahareh Nazari Jahantigh, Amir Masoud Rahmani, Nima Jafari Navimirour, and Ali Rezaee. Integration of internet of things and cloud computing: a systematic survey. *IET Communications*, 14(2):165–176, 2020.

[Kag17] Kaggle. New york city taxi trip duration. `https://www.kaggle.com/c/nyc-taxi-trip-duration/overview`, 2017. Accessed: 2022-11-28.

[Kah62] Arthur B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[KDKZ17] Kuljeet Kaur, Tanya Dhand, Neeraj Kumar, and Sherali Zeadally. Container-as-a-Service at the edge: Trade-off between energy efficiency and service availability at fog nano data centers. *IEEE Wireless Communications*, 24(3):48–56, 2017.

[Kob12] Stephen G. Kobourov. Spring embedders and force directed graph drawing algorithms. *CoRR*, abs/1201.3011, 2012.

[KRK+18] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. Benchmarking distributed stream data processing systems. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 1507–1518. IEEE Computer Society, 2018.

[LGI20] Thomas Lambert, David Guyon, and Shadi Ibrahim. Rethinking operators placement of stream data application in the edge. In *The 29th ACM International Conference on Information and Knowledge Management, CIKM 2020, Virtual Event, Ireland, October 19-23, 2020*, pages 2101–2104. ACM, 2020.

[LGM+15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

[LK19] Manisha Luthra and Boris Koldehofe. ProgCEP: A programming model for complex event processing over fog infrastructure. In *Proceedings of the 2nd International Workshop on Distributed Fog Services Design, DFSD@Middleware 2019, Davis, CA, USA, December 9-13, 2019*, pages 7–12. ACM, 2019.

[LLS08] Geetika T. Lakshmanan, Ying Li, and Robert E. Strom. Placement strategies for internet-scale data stream systems. *IEEE Internet Computing*, 12(6):50–60, 2008.

[LLZM17] Wenzhuo Li, Chuang Lin, Puheng Zhang, and Mao Miao. Probe sharing: A simple technique to improve on sparrow. In *22nd IEEE Symposium on Computers and Communications, ISCC 2017, Heraklion, Greece, July 3-6, 2017*, pages 863–870. IEEE Computer Society, 2017.

[LMT+05] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 31st ACM SIGMOD International Conference on Management of Data, SIGMOD 2005, Baltimore, Maryland, USA, June 14-16, 2005*, pages 311–322. ACM, 2005.

[LT19] Abdelquoddouss Laghrissi and Tarik Taleb. A survey on the placement of virtual resources and virtual network functions. *IEEE Communications Surveys and Tutorials*, 21(2):1409–1434, 2019.

[LYCW18] Ximeng Liu, Yang Yang, Kim-Kwang Raymond Choo, and Huaqun Wang. Security and privacy challenges for internet-of-things and fog computing. *Wireless Communications and Mobile Computing*, 2018:9373961:1–9373961:3, 2018.

[MAI21] Asif Muhammad, Muhammad Aleem, and Muhammad Arshad Islam. TOPStorm: A topology-based resource-aware scheduler for stream processing engine. *Cluster Computing*, 24(1):417–431, 2021.

[MCA+17] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, María S. Pérez-Hernández, Radu Tudoran, Stefano Bortoli, and Bogdan Nicolae. Towards a unified storage and ingestion architecture for stream processing. In *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*, pages 2402–2407. IEEE Computer Society, 2017.

[MG11] Peter Mell and Timothy Grance. The NIST definition of cloud computing. Special Publication (SP) 800-145, National Institute of Standards and Technology (NIST), September 2011.

[MGG+17] Ruben Mayer, Leon Graser, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. EmuFog: Extensible and scalable emulation of large-scale fog computing infrastructures. In *IEEE Fog World Congress, FWC 2017, Santa Clara, CA, USA, October 30 - November 1, 2017*, pages 1–6. IEEE, 2017.

[MMA+17] Eva Marín-Tordera, Xavier Masip-Bruin, Jordi Garcia Almiñana, Admela Jukan, Guang-Jie Ren, and Jiafeng Zhu. Do we all really know what a fog node is? current trends towards an open definition. *Computer Communications*, 109:117–130, 2017.

[NCGP19] Matteo Nardelli, Valeria Cardellini, Vincenzo Grassi, and Francesco L. Presti. Efficient operator placement for distributed data stream processing applications. *IEEE Transactions on Parallel and Distributed Systems*, 30(8):1753–1767, 2019.

[OKRR13] Beate Ottenwälder, Boris Koldehofe, Kurt Rothermel, and Umakishore Ramachandran. MigCEP: operator migration for mobility driven distributed complex event processing. In *The 7th ACM International Conference on*

Distributed Event-Based Systems, DEBS '13, Arlington, TX, USA - June 29 - July 03, 2013, pages 183–194. ACM, 2013.

[Ora22] Oracle. JavaDoc: InetAddress.isReachable. `https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/net/InetAddress.html#isReachable(int)`, 2022. Accessed: 2022-11-28.

[OWZS13] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *24th Symposium on Operating Systems Principles, SOSP 2013, Farmington, PA, USA, November 3-6, 2013*, pages 69–84. ACM, 2013.

[Pau18] Paul Jakma. Quagga routing suite. `https://www.nongnu.org/quagga/`, 2018. Accessed: 2022-11-28.

[PB19] Tobias Pfandzelter and David Bermbach. IoT data processing in the fog: Functions, streams, or batch processing? In *IEEE International Conference on Fog Computing, ICFC 2019, Prague, Czech Republic, June 24-26, 2019*, pages 201–206. IEEE, 2019.

[PCSA18] Laurent Prosperi, Alexandru Costan, Pedro Silva, and Gabriel Antoniu. Planner: Cost-efficient execution plans placement for uniform stream analytics on edge and cloud. In *Proceedings of the 2nd IEEE/ACM Workflows in Support of Large-Scale Science, (WORKS) 2018, Dallas, TX, USA, November 11, 2018*, pages 42–51, 2018.

[PHH+15] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy H. Campbell. R-Storm: Resource-aware scheduling in storm. In *Proceedings of the 16th Annual Middleware Conference, Middleware 2015, Vancouver, BC, Canada, December 7 - 11, 2015*, pages 149–161. ACM, 2015.

[PKvR16] Manuel Peuster, Holger Karl, and Steven van Rossem. MeDICINE: Rapid prototyping of production-ready network services in multi-pop environments. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks, NFV-SDN 2016, Palo Alto, CA, USA, November 7-10, 2016*, pages 148–153. IEEE, 2016.

[PLS+06] Peter R. Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo I. Seltzer. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 49. IEEE Computer Society, 2006.

[PRB20] Frank Pallas, Philip Raschke, and David Bermbach. Fog computing as privacy enabler. *IEEE Internet Computing*, 24(4):15–21, 2020.

[PZ17] Davy Preuveneers and Elisabeth Ilie Zudor. The intelligent industry of the future: A survey on emerging trends, research challenges and opportunities in industry 4.0. *Journal of Ambient Intelligence and Smart Environments*, 9(3):287–298, 2017.

[QR21] Tianyu Qi and Maria Rodriguez. A traffic and resource aware online Storm scheduler. In *ACSW '21: 2021 Australasian Computer Science Week Multiconference, Dunedin, New Zealand, 1-5 February, 2021*, pages 8:1–8:10. ACM, 2021.

[RAWY15] Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, and Minlan Yu. Hopper: Decentralized speculation-aware cluster scheduling at scale. *Computer Communication Review*, 45(5):379–392, 2015.

[RDR10] Stamatia Rizou, Frank Dürr, and Kurt Rothermel. Solving the multi-operator placement problem in large-scale operator networks. In *Proceedings of the 19th International Conference on Computer Communications and Networks, IEEE ICCCN 2010, Zürich, Switzerland, August 2-5, 2010*, pages 1–6. IEEE, 2010.

[Red21] Redis Labs. Redis. `https://redis.io/`, 2021. Accessed: 2022-11-28.

[RPD06] Florian Rosenberg, Christian Platzer, and Schahram Dustdar. Bootstrapping performance and dependability attributes of web services. In *2006 IEEE International Conference on Web Services, ICWS 2006, Chicago, Illinois, USA, September 18-22, 2006*, pages 205–212. IEEE Computer Society, 2006.

[RZH+20] Ju Ren, Deyu Zhang, Shiwen He, Yaoxue Zhang, and Tao Li. A survey on end-edge-cloud orchestrated network computing paradigms: Transparent computing, mobile edge computing, fog computing, and cloudlet. *ACM Computing Surveys*, 52(6):125:1–125:36, 2020.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *Proceedings of the 5th ACM SIGMOD International Conference on Management of Data, SIGMOD 1979, Boston, Massachusetts, USA, May 30 - June 1, 1979*, pages 23–34. ACM, 1979.

[SAG+19] Surbhi Saraswat, Vishal Agarwal, Hari Prabhat Gupta, Rahul Mishra, Ashish Gupta, and Tanima Dutta. Challenges and solutions in software defined networking: A survey. *Journal of Network and Compututer Applications*, 141:23–58, 2019.

[SBCD09] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Cáceres, and Nigel Davies. The case for VM-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[SH00] Thomas Stützle and Holger H. Hoos. MAX-MIN ant system. *Future Generation Computer Systems*, 16(8):889–914, 2000.

[Sim19] Yogesh Simmhan. Big data and fog computing. In *Encyclopedia of Big Data Technologies*. Springer, 2019.

[SNSD17] Olena Skarlat, Matteo Nardelli, Stefan Schulte, and Schahram Dustdar. Towards QoS-aware fog service placement. In *1st IEEE International Conference on Fog and Edge Computing, ICFEC 2017, Madrid, Spain, May 14-15, 2017*, pages 89–96. IEEE Computer Society, 2017.

[SPPvS08] Michal Szymaniak, David L. Presotto, Guillaume Pierre, and Maarten van Steen. Practical large-scale latency estimation. *Computer Networks*, 52(7):1343–1364, 2008.

[SZ20] Elham Ali Shammar and Ammar Thabit Zahary. The internet of things (IoT): a survey of techniques, operating systems, and trends. *Library Hi Tech*, 38(1):5–66, 2020.

[Tar72] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[TDJ11] Kostas Tzoumas, Amol Deshpande, and Christian S. Jensen. Lightweight graphical models for selectivity estimation without independence assumptions. *Proceedings of the VLDB Endowment*, 4(11):852–863, 2011.

[TKMN19] Artem Thofimov, Igor E. Kuralenok, Nikiga Marshalkin, and Boris Novikov. Delivery, consistency, and determinism: rethinking guarantees in distributed stream processing. *CoRR*, abs/1907.06250, 2019.

[TLL14] Cory Thoma, Alexandros Labrinidis, and Adam J. Lee. Automated operator placement in distributed data stream management systems subject to user constraints. In *Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014*, pages 310–316. IEEE Computer Society, 2014.

[TR14] Chun-Wei Tsai and Joel J. P. C. Rodrigues. Metaheuristic scheduling for cloud: A survey. *IEEE Systems Journal*, 8(1):279–291, 2014.

[TR18] Ruchi Tripathi and Ketan Rajawat. Dynamic network latency prediction with adaptive matrix completion. In *12th International Conference on Signal Processing and Communications (SPCOM), Bangalore, India, July 16-19, 2018*, pages 407–411. IEEE, 2018.

[TTS+14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy V. Ryaboy.

Storm@twitter. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 147–156. ACM, 2014.

[VdAL18] Alexandre Da Silva Veith, Marcos Dias de Assunção, and Laurent Lefèvre. Latency-aware placement of data stream analytics on edge computing. In *16th International Conference on Service-Oriented Computing, ICSOC 2018, Hangzhou, China, November 12-15, 2018*, volume 11236 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2018.

[VGT14] Federico Vicentini, Matteo Giussani, and Lorenzo Molinari Tosatti. Trajectory-dependent safe distances in human-robot interaction. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation, ETFA 2014, Barcelona, Spain, September 16-19, 2014*, pages 1–4. IEEE, 2014.

[VS20] Prateeksha Varshney and Yogesh Simmhan. Characterizing application scheduling on edge, fog and cloud computing resources. *Software: Practice and Experience*, 50(5):558–595, 2020.

[WKB+19] Rich Wolski, Chandra Krintz, Fatih Bakir, Gareth George, and Wei-Tsung Lin. CSPOT: portable, multi-scale functions-as-a-service for IoT. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing, SEC 2019, Arlington, Virginia, USA, November 7-9, 2019*, pages 236–249. ACM, 2019.

[WvG93] Annita N. Wilschut and Stephan A. van Gils. A model for pipelined query execution. In *MASCOTS '93, Proceedings of the International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems, January 17-20, 1993, La Jolla, San Diego, CA, USA*, pages 225–232. The Society for Computer Simulation, 1993.

[WZR19] Patrick Wiener, Philipp Zehnder, and Dominik Riemer. Towards context-aware and dynamic management of stream processing pipelines for fog computing. In *3rd IEEE International Conference on Fog and Edge Computing, ICFEC 2019, Larnaca, Cyprus, May 14-17, 2019*, pages 1–6. IEEE, 2019.

[XBLK18] Luna Xu, Ali Raza Butt, Seung-Hwan Lim, and Ramakrishnan Kannan. A heterogeneity-aware task scheduler for spark. In *IEEE International Conference on Cluster Computing, CLUSTER 2018, Belfast, UK, September 10-13, 2018*, pages 245–256. IEEE Computer Society, 2018.

[XCTS14] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. T-Storm: Traffic-aware online scheduling in Storm. In *IEEE 34th International Conference on Distributed Computing Systems, ICDCS 2014, Madrid, Spain, June 30 - July 3, 2014*, pages 535–544. IEEE Computer Society, 2014.

[YFN+19] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P. Jue. All one needs to

know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, 98:289–330, 2019.

[YHM15] Shaoyi Yin, Abdelkader Hameurlain, and Franck Morvan. Robust query optimization methods with respect to estimation errors: A survey. *SIGMOD Record*, 44(3):25–36, 2015.

[YZXS11] Jing Yuan, Yu Zheng, Xing Xie, and Guangzhong Sun. Driving with knowledge from the physical world. In *17th International Conference on Knowledge Discovery and Data Mining, KDD 2011, San Diego, CA, USA, August 21-24, 2011*, pages 316–324. ACM, 2011.

[YZZ+10] Jing Yuan, Yu Zheng, Chengyang Zhang, Wenlei Xie, Xing Xie, Guangzhong Sun, and Yan Huang. T-drive: driving directions based on taxi trajectories. In *18th International Symposium on Advances in Geographic Information Systems, ACM-GIS 2010, San Jose, CA, USA, November 3-5, 2010*, pages 99–108. ACM, 2010.

[ZDL+12] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *4th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'12, Boston, MA, USA, June 12-13, 2012*. USENIX Association, 2012.

[ZJW+19] Zhou Zhang, Peiquan Jin, Xiaoliang Wang, Ruicheng Liu, and Shouhong Wan. N-Storm: Efficient thread-level task migration in Apache Storm. In *21st IEEE International Conference on High Performance Computing and Communications; 17th IEEE International Conference on Smart City; 5th IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2019, Zhangjiajie, China, August 10-12, 2019*, pages 1595–1602. IEEE, 2019.

[ZLN+17] Rui Zhu, Bang Liu, Di Niu, Zongpeng Li, and Hong Vicky Zhao. Network latency estimation for personal devices: A matrix completion approach. *IEEE/ACM Transactions on Networking*, 25(2):724–737, 2017.