Massively Parallel Stream Processing with Latency Guarantees

vorgelegt von
Dipl.-Inform. Björn Lohrmann
geb. in Berlin

von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzende:	Prof. Anja Feldmann, Ph.D.
Gutachter:	Prof. Dr. Odej Kao
	Prof. Johann-Christoph Freytag, Ph.D.
	Prof. Dr. Kai-Uwe Sattler

Tag der wissenschaftlichen Aussprache: 26.11.2015

Berlin 2016

Acknowledgment

Writing a thesis requires a lot of support from various sides. First and foremost, I would like to thank my advisor Odej Kao. He provided me with a great working environment in his research group and supported my work through many helpful comments and collaboration opportunities. I also would like to express my appreciation to Johann-Christoph Freytag and Kai-Uwe Sattler for their helpful comments and agreeing to review this thesis.

Much of the good atmosphere I experienced at TU Berlin is owed to the many people I had the opportunity to work with. Special thanks goes to Daniel Warneke who helped me a lot with the first publications of this thesis and whose Nephele framework is the solid foundation of the stream processing engine developed as part of this thesis. I would also like to give credit to the wonderful members of the CIT team (past and present), especially Philipp Berndt, Matthias Hovestadt, Alexander Stanik, Andreas Kliem, Lauritz Thamsen, Tim Jungnickel, Peter Janacik, Tobias Herb, as well as Dominic Battré who advised my Master's thesis. They were always open to discussions (on research and non-research related matters) and were great company during the evenings we spent working late. I would also like to thank the students Sascha Wolke and Ilya Verbitskiy for helping me with a lot of the implementation work. Furthermore, I must not forget to express my gratitude to Matthias Sax, Fabian Fier and Bruno Cadonna from DBIS at HU-Berlin, for the fruitful discussions on streaming related topics.

Last but not least, I would like to especially thank my fiancé Andrea and my parents. Thank you for your kindness and understanding as well as the support, patience, and love during this time. Finally, many thanks (and apologies) go to my friends for their understanding, when I could not attend so many events.

Abstract

A new class of stream processing engines has recently established itself as a platform for applications in numerous domains, such as personalized content- and ad-serving, online recommender systems or social media analytics. These new engines draw inspiration from Big Data batch processing frameworks (Google's MapReduce and its descendants) as well existing stream processing engines (e.g. Borealis and STREAM). They process data on-the-fly without first storing it in a file system and their core programming abstractions hide the complexity of distributed-parallel programming.

Although stream processing applications commonly need to process ingested data within time bounds, this new class of engines so far computes results "as fast as possible". As variations in workload characteristics are often hard to predict and outside the control of the application, this may quickly lead to a situation where "as fast as possible" becomes "not fast enough" for the application.

This thesis revisits the design of this new class of stream processing engines. The core question addressed by this thesis is how latency requirements can be specified and continuously enforced within these engines in a resource-efficient manner. To this end, this thesis contributes (1) a formalism and associated semantics for expressing latency requirements for stream processing applications, (2) a set of techniques for engines to enforce them and (3) an evaluation of the effectiveness of the presented techniques. The proposed techniques optimize resource efficiency by automatically adjusting the data shipping strategy between data flow tasks and adapting the mapping between tasks and execution threads at runtime. Furthermore, based on workload statistics measured at runtime, they adapt the application's parallelism by exploiting the elasticity of shared, resource-managed compute clusters. To evaluate their effectiveness, they have been implemented in a research prototype and an experimental evaluation with several application workloads has been conducted on a large commodity cluster.

Zusammenfassung

In den vergangenen Jahren hat sich eine neue Generation von Systemen zur Streamdatenverarbeitung etabliert, die ihre Anwendung zum Beispiel in Echzeit-Empfehlungssystemen, personalisiertem Online-Marketing und der Analyse von Daten aus sozialen Netzwerken findet. Diese neuartigen Systeme vereinen Eigenschaften batch-orienter Datenanalysesyssteme aus dem Big Data Bereich (z.B. Google MapReduce), mit denen klassischer Systeme zur Streamdatenverarbeitung (z.B. Borealis und STREAM). Zumeist handelt es sich hierbei um Software-Frameworks deren Programmierabstraktionen die Komplexität paralleler Programmierung kapseln und deren Fokus die Verarbeitung eingehender Daten ohne vorherige persistente Speicherung ist.

Obwohl konkrete Anwendungen der Streamdatenverarbeitung eingehende Daten für gewöhnlich innerhalb enger Zeitgrenzen verarbeiten müssen, ist der Fokus der existierenden Systeme diese Verarbeitung "so früh wie möglich" durchzuführen. Schwer vorhersagbare und unkontrollierbare Schwankungen in der Verarbeitungslast führen jedoch schnell zu einer Situation in der "so früh wie möglich" sich als "nicht früh genug" für die Anwendung erweist.

Aufbauend auf dem aktuellen Design von Systemen zur Streamdatenverarbeitung, behandelt diese Arbeit im Kern die Frage, wie sich die Latenzanforderungen von Anwendungen spezifizieren and zur Laufzeit ressourceneffizient garantieren lassen können. Die wissenschaftlichen Beiträge dieser Arbeit sind (1) ein Formalismus zur Spezifikation von Latenzanforderungen von Anwendungen der Streamdatenverarbeitung, (2) ein Satz an Verfahren, die derart spezifizierte Latenzanfordungen zur Laufzeit umsetzen und (3) eine experimentelle Evaluation dieser Verfahren. Die in dieser Arbeit beschriebenen Verfahren optimieren die Ressourceneffizienz durch Anpassung der Datenübertragungsstrategien und der Abbildung von Anwendungsteilen auf Threads zur Laufzeit. Basierend auf Messungen bestimmter Aspekte der Anwendungslast, passen sie zudem die Parallelität dieser Anwendungen zur Laufzeit an, unter Ausnutzung der Ressourcen-Elastizität aktueller Cluster Management Systeme. Die genannten Verfahren sind im Rahmen dieser Arbeit prototypisch implementiert und in mehreren Anwendungsszenarien auf einem großen Rechencluster experimentell evaluiert worden.

Contents

Acronyms

AOB Adaptive Output Batching.

AS Amnesia.

CEP Complex Event Processing.

DAG directed acyclic graph.

DSMS Data Stream Management System.

DTC Dynamic Task Chaining.

ES Elastic Scaling.

IaaS Infrastructure-as-a-Service.

NTP Network Time Protocol.

PaaS Platform-as-a-Service.

QoS Quality of Service.

RGA Runtime Graph Aggregator.

SM State Management.

SPE Stream Processing Engine.

TGA Template Graph Aggregator.

UB Upstream Backup.

UBD Upstream Backup with Duplicate Elimination.

UDF User-Defined Function.

Chapter 1: Introduction

Contents

Many computing applications in the areas of industry and science require large volumes of *streamed* data to be processed with low latency. These applications are often 'under-the-hood' components of large websites, where they provide personalized content- and ad-serving [94, 120], recommender systems [110], or social media analytics [118]. However, similar technology is also being applied to sensor data processing [79] and network security management [31]. The emergence of the *Internet of Things* with its plethora of internet-connected embedded devices is expected to be another significant driver for new applications of this type [123]. Smart meters, that measure utility consumption (electricity, gas, water), are currently being deployed in growing numbers at consumer homes are expected to provide energy utilities with a wealth of streamed data that could potentially be analyzed, e.g. for power usage forecasting and energy grid monitoring [59, 96]. Furthermore, major cloud providers begin to offer stream processing as a Platform-as-a-Service (PaaS) offering [89, 7, 43], promising to simplify the development and operation of such applications. These services are available on demand and without long-term commitment or upfront capital expenditures on the customer side, thus allowing companies that cannot afford their own data center to operate such stream processing applications.

All of these applications share a common set of requirements. They have to *continuously* ingest data streams arriving at *high and varying rates* and process them within certain time bounds. The sources of these data streams are usually external to the application, and of diverse origins ranging from mobile phones over web browsers to sensor networks. Both the volume of such data streams and the complexity of their processing is steadily increasing [112, 94]. Hence a model of horizontal scaling [88] has become popular [94] for these applications, supported by the favorable price-performance ratio of

inexpensive commodity server hardware. Since these applications usually run *continuously* or for extended periods of time, cost-efficient techniques are required to adapt to variations in the stream rate and deal with software and hardware faults. Once *inside the application*, ingested data needs to be processed within application-specific time bounds, ranging from milliseconds to seconds. In many cases, if processing is delayed beyond a certain threshold, the service quality of application degrades or the results are not useful anymore. The processing inside such applications can involve complex combinations of transformation, aggregation, analysis and storing. Hence support for *general-purpose processing* is required, that goes beyond what algebraic continuous query languages [14] and pattern-detection languages [128] can express.

The popular "process-after-store" model [112] inherent to today's large-scale batch processing engines [11, 61, 24, 20] is not a good fit for these applications. In the "process-after-store" model, incoming data is first persistently stored and its analysis is deferred to a possibly much later (minutes to days) point in time. This mismatch has led to the creation of a new type of data processing engines, that draw inspiration from batch processing engines as well as previous Data Stream Management System (DSMS) [1, 2, 90]. Examples of such engines are the Apache systems Storm [13], Flink [10], S4 [94] and Spark Streaming [132], numerous research prototypes [91, 32, 29, 101, 129, 81] and industrial systems [4, 60]. From batch processing engines they have adopted the principle of hiding the complexity of distributed-parallel programming incurred by the shift towards horizontal scaling. Specifically, they allow application developers to write pieces of sequential code in the form of User-Defined Functions (UDFs) and provide parallel and distributed execution on large clusters of commodity servers. From DSMS however, they have adopted the active model of processing, where data is processed immediately without persisting it first.

Often, such applications are part of large websites operated by Internet companies like Yahoo [120], Twitter [118] or Spotify [110]. Hence these engines run on very large clusters comprising hundreds or up to several thousands of machines [105, 36]. As it is very expensive to build and operate large clusters, it is an economic priority to maintain a *high utilization* of the cluster's resources. A common way of increasing cluster utilization is to employ a *cluster resource manager* such as YARN [122] or Mesos [52]. A cluster

resource manager increases cluster utilization by providing the possibility of *multi-tenancy*. To this end, it offers a generic interface that any data processing engine (and other systems as well) can use to allocate and use cluster resources, e.g. CPU cores, memory, storage or networking. Analogous to a Infrastructure-as-a-Service (IaaS) cloud where a large pool of servers is shared between virtual machines, the cluster resource manager shares a pool of available cluster resources between multiple competing engines. According to predefined policies it mediates between the resource demands of engines with possibly very different workload characteristics, e.g. continuous and time-sensitive stream processing and (comparably) short-lived batch processing workloads.

1.1 Problem Definition

The key question this thesis is going to address can be summarized as follows:

> *"Given a large and resource-managed commodity cluster, how can the new class of highly distributed, parallel and general-purpose stream processing engines provide latency guarantees to applications they execute?"*

Obtaining results within time bounds is a common non-functional requirement of stream processing applications and a notion of latency is the performance metric that determines, whether or not time bounds can be met. Indeed, the specific time bounds depend very much on the application itself. For an application that performs computation for dashboard-style monitoring, a latency of up to several seconds may be acceptable. An application that is part of a user-interactive system, 100 ms is the time limit for the application to be perceived as responding instantaneously [95]. Consider an application that recommends potentially interesting news articles to visitors of a large news website. If such recommendations appear too late for the user to see them, because the application cannot compute them fast enough, this will decrease the click-through rate, an important website performance metric. Delayed results thus not only negatively impact *user experience*, but may also have direct negative economic impact such as lost advertising revenue.

Today's general-purpose stream processing engines provide a *data flow* programming abstraction. Application developers are thus expected to structure

their application like a graph. Data flows along the graph's edges between UDFs that perform computation on consumed data and may produce output data. The stream processing engine executes the data flow graph in a highly parallel and distributed fashion, exploiting massive data, task and pipeline parallelism [54]. The main focus here has been the highly parallel, distributed and fault-tolerant execution of arbitrary user code. Many of these engines have so far *interpreted stream processing in the sense that results should be computed as fast as possible* [13, 10, 69, 12, 94, 60, 132]. However, situations in which the computational load overwhelms the amount of provisioned CPU resources must be resolved manually and entail notoriously difficult tuning to obtain the desired performance. Other engines can adapt to workload variations, but the proposed schemes are designed to prevent overload or optimize the execution towards a target CPU utilization [4, 104, 41, 29, 129]. However, which specific target CPU utilization leads to a suitable latency for the application must still be determined manually. Hence the first challenge this thesis addresses, is to find a suitable *formalism* for stream processing applications to express latency requirements. It should be able to express the requirement in way that is *meaningful* to the application and *measurable* as well as *enforceable* by a stream processing engine that executes the application.

Given an application that explicitly declares a latency requirement, a stream processing engine now has the possibility of organizing the application's execution towards fulfilling the requirement. Given the environment of a shared resource-managed commodity cluster and variations in application workload characteristics, the main challenges for the engine are *adaptiveness* and *resource-efficiency*.

Adaptiveness in stream processing means responding to variations in application workload characteristics, such as the arrival rate of incoming data streams and the resulting computational load inside the application. Many data streams vary heavily in their rate over the course of time. For user-facing applications, the day-night rhythm often drives stream rates. Real-world events in particular can be a significant source of rate peaks. For example, while the rate of Tweets per second averaged at ca. 5700 in 2013, Twitter cites a rate of ca. 144,000 Tweets per second as a significant peak in 2013 [119]. Traditional techniques such as *load-shedding* [17, 114], where excess data is dropped may often not be desirable for the use case at hand.

A the same time, an a priori resource provisioning for peak load may be prohibitively expensive and lead to low resource utilization.

Resource efficiency refers to the challenge of minimizing the amount of allocated cluster resources during processing. Finding the "right" amount of resources for any given workload is non-trivial, as the demands of the workload may be unknown or unknowable in advance. Even for non-streaming workloads, finding the minimum required resource allocation has proven to be very difficult [21], as evidenced by the low resource utilization even on shared, multi-tenant clusters in use at many companies. For example, the average CPU utilization of a large production cluster at Twitter has been found to be 20% [36]. Similar utilization figures ranging from 20% to 30% have been reported for a production cluster at Google [103]. This problem becomes even more pronounced for stream processing applications, due to their variations in computational load. A *static* resource provisioning may thus leave the application in a state of permanent over- or under-provisioning. Both states are equally undesirable. Over-provisioning conflicts with the goal of high cluster utilization, which is an economic priority for the operation of a large compute cluster. On the other hand, under-provisioning leads to a situation where an application's latency requirements cannot be fulfilled.

A shared, resource-managed cluster is a major opportunity to addressing these two challenges. It allows the stream processing engine to abandon static resource provisioning in favor of *resource elasticity* i.e. to adapt its allocation of cluster resources depending on the current workload characteristics. Although current engines [94, 13, 132] already provide integration with cluster resource management software such as Mesos [52] and YARN [122], they do not yet exploit resource elasticity at all [10, 69, 12, 94, 60, 132] or make limited use of it [13, 4, 104, 41, 29, 129].

1.2 Contributions

This thesis proposes a set of solutions to the challenges described above. Based on a definition of latency for UDF-heavy data flows, *latency constraints* are proposed as a formalism with clearly defined *semantics* to express an applications latency requirements. This allows application developers to annotate their application with non-functional latency requirements, so that

the engine can enforce them at execution time without manual performance tuning.

Second, the challenges of *resource-efficiency* and *adaptiveness* are addressed from two angles. First, the focus is on making the most efficient use of currently allocated resources. To this end, this thesis proposes techniques for stream processing engines to minimize continuously and automatically the resource consumption of an application, while guaranteeing the *latency constraints* that the application declares. At the heart of the proposed scheme is a *feedback-cycle*. It continuously measures the workload characteristics of a running application and then applies techniques to *optimize* the execution to better match the current workload with defined latency constraints. The optimization techniques proposed in this thesis are *adaptive output batching* and *dynamic task chaining*. They reduce the overhead of the producer-consumer model with intermediate queues, that is commonly used for communication between running tasks of an application. These two techniques enforce latency constraints by themselves as long as computational resources are sufficiently provisioned, relative to the current workload. Second, the thesis proposes a policy for *elastic scaling*, that defines *when* and *how* to adapt the application's resource allocation inside a shared compute cluster. In situations of over- (or under-) provisioning, the elastic scaling technique adapts the parallelism of the application so that the latency constraints can still (or again) be enforced. To this end, it employs a *latency model* based on queuing theory, that allows to estimate the latency of a data flow, when the degrees of parallelism of the tasks within the data flow are changed.

The above techniques have been implemented in a research prototype, derived from the Apache Flink engine. They have been experimentally evaluated with several application workloads at high degrees of parallelism on a large commodity cluster with 130 worker nodes at TU-Berlin.

Parts of this thesis have been published in the following publications:

Journal:

Björn Lohrmann, Daniel Warneke, Odej Kao
Nephele Streaming: Stream Processing under QoS constraints at Scale
In: Cluster Computing, Volume 17, Number 1, pp. 61–78, Springer, 2014

Proceedings:

Björn Lohrmann, Odej Kao
Processing Smart Meter Data Streams in the Cloud
In: 2nd IEEE PES International Conference and Exhibition on Innovative Smart Grid Technologies (ISGT Europe 2011), pp. 1–8, IEEE, 2011

Björn Lohrmann, Daniel Warneke, Odej Kao
Massively-Parallel Stream Processing under QoS Constraints with Nephele
In: Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC 2012), pp. 271–282, ACM, 2012

Björn Lohrmann, Peter Janacik, Odej Kao
Elastic Stream Processing with Latency Guarantees
In: The 35th International Conference on Distributed Computing Systems (ICDCS 2015), accepted for publication

1.3 Outline of the Thesis

The remainder of this thesis if structured as follows:

Chapter 2: Background

Chapter 2 summarizes the requirements of stream processing application, and provides an overview of models for the creation of such applications. The chapter focuses on UDF-heavy data flows, the currently predominant model for highly parallel and distributed stream processing on large and shared commodity clusters. A formal definition of *Application Graphs* is provided. Application Graphs are the formalism for describing UDF-heavy data flows within this thesis.

Chapter 3: State of the Art

Chapter 3 provides an overview of state of the art infrastructure for massively parallel stream processing. After summarizing the typical execution environment of a resource managed commodity cluster, the

chapter focuses on a broad overview of state of the art massively parallel stream processing engines that execute UDF-heavy data flows. Here, a general understanding of core functionality and cross-cutting concerns in such engines is combined with a summary of state of the art approaches.

Chapter 4: Stream Processing with Latency Constraints

Chapter 4 defines a notion of latency for UDF-heavy data flows and introduces both a formalism and semantics for latency constraints. Then a set of core techniques – adaptive output batching, dynamic task chaining and elastic scaling – is presented. They aim at enforcing latency constraints and complement each other by dealing with various aspects of adaptiveness and resource-efficiency.

Chapter 5: Experimental Evaluation

Chapter 5 experimentally evaluates the presented techniques from Chapter 4 on a large compute cluster. To this end, a set of benchmark applications is described with their relevant workload characteristics. The effects of each technique are evaluated based on these benchmark applications.

Chapter 6: Related Work

Chapter 6 surveys relevant related work with respect to the use of batching, chaining, and elastic scaling techniques, in streaming applications and beyond.

Chapter 7: Conclusion

Chapter 7 provides an outlook on future research topics in massively parallel stream processing and concludes the thesis with a summary of the contributions and results.

Chapter 2: Background

Contents

Based on the problem definition of this thesis, this chapter begins with an outline of the requirements of stream processing applications. Then it describes the currently predominant abstractions to create such applications. Further, we introduce UDF-heavy data flows as the principal abstraction for stream processing, that guide the creation of stream processing applications on today's massively parallel stream processing engines. Finally, this chapter formally defines *Application Graphs* – a concrete type of UDF-heavy data flows – in order to provide consistent terminology for use within the thesis.

2.1 Application Requirements

According to Andrade et al. [9], stream processing applications are highlighted by the following requirements:

Dealing with high-volume live (also: streamed or real-time) data from external sources
In stream processing applications, input data arrives at high volume just in time for processing. Due to the high data volume, such applications would quickly overwhelm existing relational database systems, that store and index data in relational form for later querying with SQL. In fact, the volumes of stream data in personalized content- and ad-serving in the web [120],

recommender systems [110], social media analytics [118], sensor data processing [79] and the emergent Internet of Things [123], require large-scale distributed systems to merely be *reliably persisted* on disks. Data sources are usually external to the application itself, can be of diverse type and geographically distributed. Examples are distributed environmental sensors, network monitoring equipment, servers generating logging data, cell phones, stock exchanges and RFID tags.

Data heterogeneity
The data processed by stream processing applications is in heterogeneous formats and can be of any type (e.g. structured, semi-structured or unstructured). Incoming stream data can for example be text, images, video, audio or XML data.

Complex processing to obtain actionable information
Stream processing applications perform complex computational operations on their input data including but not limited to transformation, aggregation, filtering, pattern matching or correlating data of fundamentally different types and origins (e.g. live video and textual data from a social website). Often, incoming data needs to be integrated with stored data. Such processing is performed on moving data, i.e. without storing it on disks first. Often, the goal of such processing is to yield new insights that can be acted upon. For example, an application performing security monitoring of traffic in a large enterprise network, must (1) search for anomalous patterns in live traffic and (2) mitigate network security threats by warning network administrators or autonomously initiating counter-actions. A online recommender system at a large music streaming website must be able to (1) continuously update a personalized recommendation model for each user that predicts music preferences based on past listening behavior and (2) use the model to instanteously recommend new music tracks to website users.

Fault-Tolerant Execution
Stream processing applications typically have *service characteristics*, i.e. they are long-running, continuous processes. At the same time, the high data volumes and complex processing they perform necessitate a distributed and parallel system. The combination of these two aspects, make tolerance to faults in hardware and software a requirement for reliable operation.

Latency Constraints
These applications need to be able to process and analyze data at its arrival rate, and deliver the results of this analyis within time bounds. This is a complex requirement due to (1) variations in the arrival rate of incoming data and (2) variations in the computational work performed on that data. Financial trading applications that monitor market data and buy or sell when trading opportunities are detected, are certainly at the very extreme end with microsecond latency requirements [133]. The authors of the same paper describe the highly bursty nature of option price quotes in a US options market data feed. They state 907,000 messages per second as a peak rate recorded in 2008 and highlight the fact that the peak rate has been increasing by a factor of roughly 1.9 per year since the beginning of the decade. For stream processing applications directly tied into user-interaction (for example web applications), 100 ms has been found as a time constant in human perceptual processing, so that a response of a computer system is perceived as instantaneous [95]. The same class of applications also incurs large variations in message rates due to the human day-night rhythm and real-world events. For stream processing application providing dashboard-style infrastructure monitoring (e.g. a data center) a sub-second or multi-second latency may be sufficient. The concrete notions of "high" arrival rate and "low" latency is therefore relative to the application itself and its properties e.g. computational intensity, and (user) requirements.

2.2 Abstractions for Stream Processing

For more than a decade, continuous processing of ingested data has been the subject of vivid research and development in multiple research communities, each one bringing forth its own abstractions and systems. In [86] Cugola et al. provide a comprehensive overview of abstractions and systems prior to 2012. The following subsections first summarize the established (pre 2012) abstractions and then shifts the focus on those provided by current (post 2012) massively parallel stream processing engines.

2.2.1 Data Stream Management Systems (DSMSs)

Early models came from the database community with *active databases* [38]. These systems provide so-called Event-Condition-Action (ECA) rules, where changes in a persistent database (called "events") trigger subsequent changes (called "actions") under defined conditions. Current relational database systems still provide ECA rules in the form of SQL triggers. The research on active databases eventually led to DSMS [1, 90, 30], where an asynchronous push-based model with on-the-fly data ingestion was introduced. Here, *standing queries* are continuously evaluated over streams of structured data [16] without prior persistence or indexing. Models in DSMSs are *transformation-oriented* and employ declarative (often SQL-like) or imperative languages. As an example of a declarative stream processing language we take a brief look at CQL [14] with an example inspired by the Linear Road Benchmark [15]. Consider a stream `SegSpeedStr` of tuples with attributes `(vehicleId, speed, seg, hwy)` that provides push updates of vehicle locations (segment) and speeds on a highway. The following CQL query computes a stream indicating new traffic congestions on highway segments:

```
SELECT IStream(hwy, seg, Avg(speed))
FROM SegSpeedStream [Range 5 Minutes]
GROUP BY hwy, seg
HAVING Avg(speed) < 40
```

The `FROM` clause defines that a *time-based sliding window* shall be created for every incoming tuple on `SegSpeedStr`. The window is an intermediate relation that contains all stream tuples of the last five minutes. At this point, `GROUP BY` and `HAVING` are applied to the intermediate relation with regular SQL semantics creating another intermediate relation containing all highway segments with slow traffic. *IStream* turns the resulting series of intermediate relations back into a stream that contains only new tuples. CQL further supports other window types, and SQL elements such as joins.

2.2.2 Complex Event Processing (CEP)

At around the same time that DSMSs were created, researchers from other communities proposed the idea of Complex Event Processing (CEP) [84] and

created a new breed of systems [83, 128, 18, 25]. Here, incoming data is interpreted as *events* generated by external sources. The CEP model emphasizes the detection of *complex events* that manifest themselves as *patterns* of the input events. CEP languages are *detection-oriented* and are predominantly *declarative*. As an example consider the following example using the SASE event language [128]. The example assumes a retail store scenario with RFID tagged products. RFID readers that are installed at the shelves, checkout counters and exits produce a reading whenever an item comes into their range. SASE processes typed events. The following query detects theft:

```
EVENT SEQ( SHELF-READING x,
           !(COUNTER-READING y),
           EXIT-READING z)
WHERE x.id = y.id AND x.id = z.id
WITHIN 12 hours
```

`SEQ` defines a temporal pattern, and matches items that were first detected at shelves (type `SHELF-READING`) and later at a store exit (type `EXIT-READING`) without having been checked out inbetween (type `COUNTER-READING`). The `WHERE` clause ensures that the pattern only matches events from the same item, and the `WITHIN` clause defines a time period of 12 hours in which the events must (not) occur.

Despite several efforts, standard languages with broad adoption have neither evolved for DSMS nor for CEP systems. Commercial systems often support elements of both abstractions within the same system, thus blurring the distinction between the two.

2.2.3 UDF-Heavy Data Flows

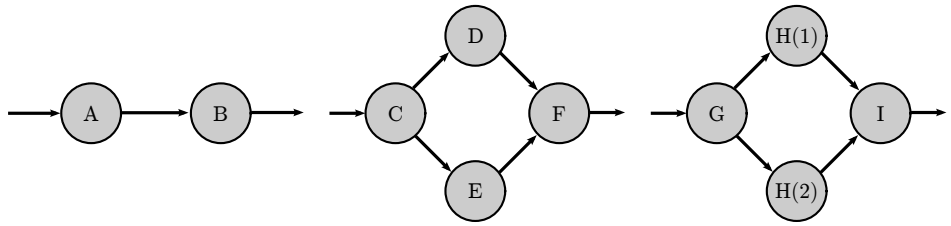
In recent years, a new class of stream processing systems has emerged, driven by the demand of growing amounts of streamed data. Inspired by distributed batch-processing models such as MapReduce [35], their models are centered around data flows consisting of UDFs. To create a stream processing application, an application developer writes UDFs in an object-oriented programming language by implementing a system-provided interface. Afterwards, the UDFs are programmatically assembled into a data flow graph.

The following subsections first describe the underlying design principles that guide UDF-heavy data flows. Afterwards, the currently existing variants of this model are discussed.

Design Principle I: Black-Box UDFs
Being inspired by the MapReduce programming model [35], these engines are centered around the execution of opaque UDFs. A UDF is a piece of – often, but not always – sequential code that is called by the engine following the inversion of control paradigm. Within this call, the UDF processes input data and may produce output data. There is no conceptual limit on the *type* of data that can be processed. In general UDFs

- perform complex operations such as transformation, aggregation, filtering, analysis and storing,
- process and correlate data of fundamentally different types and origins, e.g. live video and textual data from a social network website,
- utilize existing libraries of the programming language they were written in,
- perform stateful computation with side-effects, such as interaction with databases, caches, file systems or messaging middleware.

Design Principle II: Data Flow Graphs
All applications *are* modeled as data flow graphs, whether directly in a programmatic fashion or indirectly through a declarative language [53]. A data flow graph consists of vertices and edges, where vertices represent computation and encapsulate the application's functionality in the form of a UDF. Edges represent a directional flow of data between UDFs. Data flow graphs are directed graphs and some systems allow cycles, others do not. Data flow graphs provide the basis for parallel execution, by encapsulating three types of parallelism shown in Figure 2.1, called pipeline-, task- and data-parallelism [54]. *Pipeline parallelism* is concurrency between pipelined UDFs, i.e. between one that produces data and one that consumes it. *Task parallelism* is concurrency between two UDFs that are not pipelined. And finally, *data-parallelism* is concurrency between distinct instances of the same UDF, where each instance processes a distinct part of the data.



(a) Pipeline parallel UDFs (A & B) (b) Task parallel UDFs (D & E) (c) Data-Parallel UDF instances (H(1) & H(2))

Figure 2.1: Types of UDF parallelism found in data flows (adopted from [54]).

Design Principle III: Scalability by Massive Data Parallelism
The UDF-heavy data flow model is designed to naturally scale with growing stream data volume and computational load. Easy-to-use data parallelism is the tool employed to achieve this goal, causing the data flow graphs to be typically much wider than long.

Variations of Programming Abstractions
Similar to DSMS and CEP systems, there is no consent on a standard abstraction for writing stream processing applications. The models of of currently availabe systems however seem have converged to three types. In order of decreasing generality and increasing ease-of-use and semantic richness, the model variants are as follows.

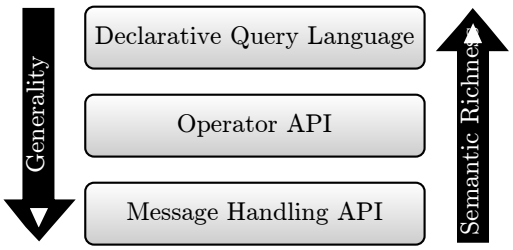


Figure 2.2: Variants of models for massively parallel stream processing.

Message Handling API – All UDFs implement the same interface with a message-handling method. This method takes a single message and produces zero or more output messages. The framework invokes the handler method for each arriving message and routes its output messages. Examples of frameworks with this type of API are Apache Storm [13] ("Bolt API") and Samza [12], Google's MillWheel [4] as well as the prototype developed as part

of this thesis.

Operator API – Each UDF implements an interface specific to an operator type. It is then wrapped in a framework-provided higher order function that enforces operator semantics by deciding when and how to call the UDF. Operator APIs usually provide stream-relational operators (project, filter, join, windows, aggregations) or functionally-inspired operators (map, flatMap, reduce). Examples of frameworks with this type of API are Apache Storm [13] ("Trident API"), Apache Flink [10] and mupd8 [70].

Declarative Query Language with embedded UDFs – In the spirit of DSMSs, in this model a declarative language allows the user to express queries, with the option to embed code written in an imperative programming language. The only system providing such a model is Microsoft's TimeStream [101] prototype that uses the declarative LINQ query language [87].

Finally, it shall be noted that many engines follow a "stacking" approach, where a semantically richer abstraction is implemented via a more low-level one. The technical realization of this approach exploits the black-box nature of UDFs and has the benefit of simplifying the engine's runtime that is concerned with organizing the efficient execution of an abstraction. Depending on whether an abstraction is designed for public or just internal use, this may lead to a system with multiple available abstractions. An example of this is Apache Storm that has stacked public abstractions, i.e. a Message-Handling API call "Bolt API" and an Operator API called "Trident", which is implemented using the Message-Handling API. Another example is Apache Flink, that has one public Operator API called "Flink Streaming" and an internal – therefore unnamed – Message-Handling API. In both cases the Message-Handling API forms the basis of execution.

2.3 Application Graphs

The following formally introduces *Application Graphs*, a type of UDF-heavy data flow used in the remainder of this thesis. Application Graphs as defined in the following are an instantiation of the "Message-Handling API" abstraction from Section 2.2.3. Therefore, Application Graphs offer a maximum in generality and a minimum in semantic richness, while maintaining the possibility of creating semantically richer model on top of it. Our definition is

derived from models found in real-world stream processing engines [13, 10]. Within this thesis, Application Graphs are the abstraction that is optimized by the techniques for stream processing under latency constraints. To this end, reducing them to only the strictly necessary concepts is useful for clarity of presentation. After a definition of formalisms and terminology, this section also introduces a graphical notation for Application Graphs.

2.3.1 Definition

An Application Graph is a data flow graph that resembles a directed acyclic graph (DAG), but is not a pure DAG. It captures the application at two different levels of detail, the *template level* and the *runtime level*. On the template level, the graph's vertices are *tasks*, and its edges are *streams*. On the runtime level, the graph's vertices are *subtasks* and edges are *channels*. The template level is fully specified by the application developer and defines the overall structure of the data flow. It provides sufficient information to generate the runtime level in an automated fashion (hence the name "template").

To formally define the Application Graph we first need a few prerequisite definitions:

Definition: Data Item
A *data item* is a discrete portion of data of an arbitrary but fixed size and arbitrary type. A data item must be serializable into a byte sequence, so that it can be stored or sent over a network.

Examples of data items are integer numbers, strings, lists, hash tables and images.

Definition: UDF and UDF instance
A UDF is a function provided by the application developer, that processes data items. We shall distinguish three types of UDFs:

- *Source-UDFs* take no input and output zero or more data items per invocation.
- *Inner-UDFs* take one data item and output zero or more data items per invocation.

- *Sink-UDFs* take one data item per invocation and produce no output.

For each produced data item a UDF specifies one or more UDF instances (see below) that the data item shall be delivered to. A UDF is not a pure function because it may have side-effects. A UDF may retain *state* between invocations and has access to an *environment*, from which it can obtain information about the available subtasks it can send data items to. A UDF has full API access to the standard libraries of the language it is written in and may come with its own custom libraries. It can use the capabilites of the underlying hardware and software platform, e.g. access local files or open network sockets.

While a UDF encapsulates functional behavior in the shape of code, a *UDF instance* combines the behavior of its UDF with a concrete state and environment. The relationship between a UDF and a *UDF instance* is thus analogous to the one between class and object in object-oriented programming.

Definition: Tasks and Subtasks
A *task* is a function that provides the "glue" between a UDF and the stream processing engine. It reads incoming bytes from a queue, deserializes the bytes into a data item, and invokes a UDF with it. It then serializes any data items resulting from the UDF invocation, and routes the resulting bytes to the receiver subtasks designated by the UDF. A *subtask* is a concrete instance of a task and wraps a UDF-instance.

Definition: Application Graph
An *Application Graph* consists of two directed acyclic graphs: the *template graph* $\mathbf{G}$ and the *runtime graph* G. The template graph $\mathbf{G} = (\mathbf{V}, \mathbf{E}, p, w)$ consists of

- a set of vertices $\mathbf{V} = \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$, representing *tasks*,
- a set of edges $\mathbf{E} \subseteq \mathbf{V}^2$ representing *streams*,
- a function $p : \mathbf{V} \to \mathbb{N}^+$ that specifies a task's degree of data parallelism,
- and a wiring function $w : \mathbf{V}^2 \times \mathbb{N}^2 \to \{T, F\}$ that specifies which subtasks shall be allowed to send messages to each other. For example, if $w(\mathbf{v}_1, \mathbf{v}_2, 1, 2) = T$ then $\mathbf{v}_1$'s first subtask shall be able to send data items to $\mathbf{v}_2$'s second subtask.

Now, let $v_{k,i}$ denote the i-th subtask of the k-th task $\mathbf{v}_k$ in $\mathbf{V}$. A given template graph $\mathbf{G} = (\mathbf{V}, \mathbf{E}, p, w)$ induces the runtime graph $G = (V, E)$ consisting of a set of subtasks

$$V = \bigcup_{k \in |\mathbf{V}|} \{v_{k,1}, \ldots, v_{k,p(\mathbf{v}_k)}\}$$

that are the graph's vertices, and a set of channels

$$E = \bigcup_{(\mathbf{v}_k, \mathbf{v}_l) \in \mathbf{E}} \{(v_{k,i}, v_{l,j}) | w(\mathbf{v}_k, \mathbf{v}_l, i, j) = T\}$$

that are the graph's edges. Channels indicate the *possibility* of shipping data items between subtasks and exhibit FIFO behavior. If and when a channel is *actually* used to transfer data items is up to the UDF instance of the sending subtask. To distinguish elements of the template and runtime graph in a formal context visually, we shall always use a bold, sans-serif font for template graph elements ($\mathbf{G}, \mathbf{V}, \mathbf{E}, \mathbf{v}, \mathbf{e}$) and a regular, italic font for runtime graph elements (G, V, E, v, e).

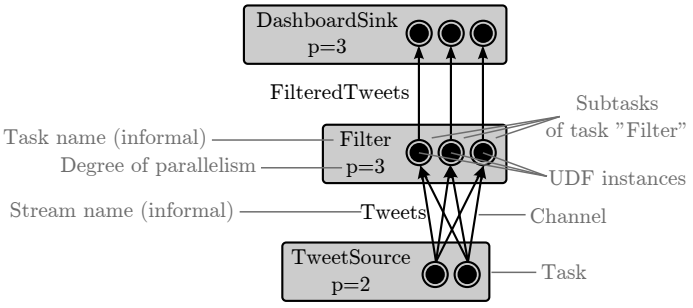


Figure 2.3: Graphical notation for Application Graphs.

2.3.2 Graphical Notation

Depending on the context, we may also describe Application Graphs in a graphical way. Figure 2.3 shows a graphical example of a simple Application Graph with three tasks and two streams. In a graphical context, tasks and streams will be given informal names that capture the functionality of the task and the data that flows through the stream respectively. The graphical representation omits the explicit display of streams as lines between tasks, because the lines representing channels already convey the same information.

Chapter 3: State of the Art

Contents

This chapter provides an overview of state of the art stream processing engines that execute UDF-heavy data flows on resource managed clusters. The contributions of this thesis presented in Chapter 4 assume some pre-existing engine functionality. Hence, it is the goal of this chapter to establish a general understanding of such core functionality, i.e. to describe the involved challenges and summarize state of the art approaches.

First, Section 3.1 provides a high-level overview of the layered architecture current massively parallel stream processing have converged to. Proceeding in a bottom-up manner, Section 3.2 takes a look at cluster resource management. It summarizes state of the art systems and describes the specific approach of Apache Hadoop YARN as an example. Section 3.3 gives an overview of massively parallel stream processing engines for UDF-heavy data flows. Based on state of the art systems, it summarizes engine core functionality, cross cutting concerns such as fault tolerance and workload adaptivity.

3.1 Overview

Despite differences between conrete systems, the new class of stream processing systems has converged to a common layered architecture. The converged architecture can be divided into *layers* and *functional components*. Figure 3.1 provides an overview of the typically available layers and components.

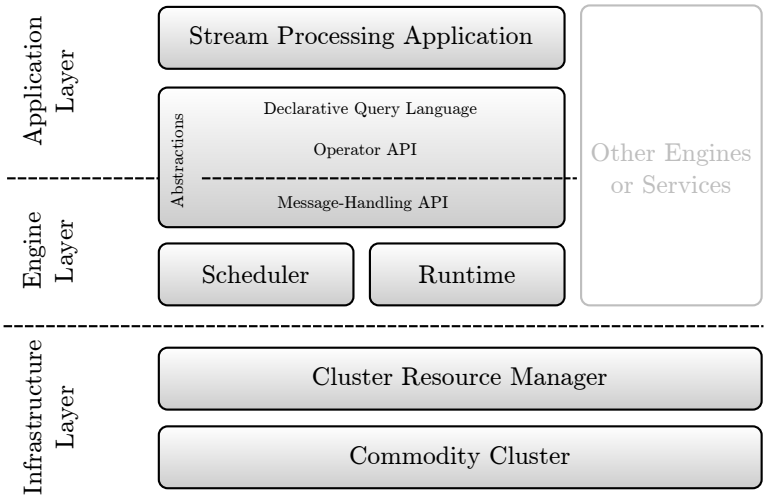


Figure 3.1: Layers involved in massively parallel stream processing.

On the *Infrastructure Layer* a compute cluster comprising up to hundreds or thousands of commodity servers provides the basis for both streaming and non-streaming workloads. *Cluster Resource Managers* [122, 52, 36, 105] facilitate multi-tenancy on very large compute clusters and thus are considered important for their economic operation. A Cluster Resource Manager keeps track of the available resources such as servers, CPU cores, RAM, disk, and network and allows them to be leased and released through an API. Section 3.2 describes this layer in more detail.

The *Engine Layer* is concerned with organizing the highly parallel and distributed execution of UDF-heavy data flows. Typically, there will at least be a *Scheduler*, and a *Runtime*. The Scheduler prepares the application's execution by mapping the elements of the data flow graph to resources. Based on this mapping, the Runtime executes the data flow, monitors the execution and deals with software- or hardware faults that may occur. Section 3.3 describes this layer in more detail.

On the *Application Layer*, software engineers or domain experts such as Data

Scientists [34, 37] create stream processing applications using the concrete abstraction offered by an engine. The possible abstraction variants have already been described in Section 2.2.3 and will not be discussed further at this point.

3.2 Cluster Resource Management

Since the advent of the UNIX operating system in the early 1970s, *compute clusters* consisting of computers connected via packet-switched networks have been a platform for data processing in research and industry [99]. Today's compute clusters comprise up to many thousands of servers and are dominated by low-end shared-nothing servers [19], due to their favorable price-performance ratio. As of this writing, this implies servers with multi-core x86 CPUs, memory in the two-digit GB range as well as local hard disks or flash drives [19]. Typically, these servers run a Unixoid operating system such as Linux and communicate via commodity packet-switched networks such as Ethernet and TCP/IP or UDP/IP. The networking fabric inside these clusters is typically a hierarchical structure, that arranges switching and routing elements in multiple layers [19, 5]. This thesis assumes a commodity cluster such as the one just described as the execution environment for massively parallel stream processing.

Due to their size, these clusters are expensive to build and operate, hence it is an economic priority to maintain a *high utilization* of the cluster's resources. The commonly accepted solution to this is to *share* the cluster between multiple tenant applications. *Cluster resource managers* solve this by providing a middleware that arbitrates cluster resources between the resource demands of multiple tenant applications.

This thesis assumes a cluster resource manager as a piece of existing software infrastructure installed on a compute cluster. While a comprehensive overview of cluster management is therefore out of scope, it is still important to understand which aspects of the *functionality* provided by cluster managers are relevant to this thesis. Section 3.2.1 thus describes the relevant functionality this thesis assumes to be provided by a cluster manager, and Section 3.2.2 gives an brief overview of state of the art systems. Section 3.2.3 summarizes Apache YARN [122] as a concrete example of a suitable cluster

resource manager.

3.2.1 Key Functionality

A cluster resource manager is a middleware that facilitates sharing a compute cluster between multiple tenant applications. In the context of commodity compute clusters, the term *resource* typically refers to CPUs (or cores thereof), RAM, disk space and network bandwidth. To this end, a cluster resource manager is mainly concerned with (i) *resource monitoring* (ii) *resource allocation*, (iii) *resource assignment* as well as (iv) facilitating the execution of applications on top of assigned cluster resources.

Resource allocation is the process of determining the *amount* of resources to use for a tenant application. The main challenge here is to enforce operational policies e.g. a notion of resource fairness. Resource assignment deals with assigning concrete resources to an allocation, e.g. specific cluster nodes. The challenges in resource assignment are

1. resource heterogeneity, due to multiple server generations being deployed within the same cluster,
2. placement constraints such as data locality, because applications often need to access data stored at specific locations in the cluster,
3. and interference minimization, due to applications being colocated on the same resource.

Once a resource has been assigned to an application, the cluster resource manager is responsible for triggering and monitoring the application's execution on the resource. Resource monitoring is the process of maintaining an up-to-date view of the resource use inside the cluster and detecting resource failures, e.g. defective cluster nodes.

3.2.2 State of the Art

Early cluster resource managers such as Condor [78], NQS [66], EASY [75], LoadLeveler [58], Torque [3] and others have long been used to organize *batch job* computation, i.e. jobs that perform computation and then terminate. In recent years, the growing popularity of data processing frameworks such as Hadoop [11] led to the creation of new commodity cluster resource managers, such as the Mesos [52], YARN [122], Omega [105] systems or

research-prototypes such as Quasar [36]. These newer resource managers are capable of colocating many different application workloads within the same cluster, e.g. batch jobs and services that run indefinitely or for extended periods of time. While a Hadoop MapReduce job fits in the former category, stream processing applications typically have the characteristics of a *service*, because they process unbounded streams of data. In all of the newer systems, existing allocations can be changed at runtime of the application, thus allowing for resource elasticity.

Managers like YARN [122] and Mesos [52] take a *reservation-centric* approach to resource allocation. Here the manager provides an API that allows applications to reserve amounts of resources. These reservations are however subject to limitations defined by a policy. YARN supports *fairness* and *capacity* policies. In the former case, the resource manager allows each tenant allocation to use a roughly equal share of the cluster, while in the latter case applications are guaranteed a minimum share plus an option to obtain excess resources that would otherwise remain unused. Mesos also supports fair allocations and additionally application prioritization. Mesos is *offer-based*, while YARN is *request-based*. In the former case, the manager only offers resources that are in line with the defined policy and lets the application decide on whether it takes the allocation or not. In the latter case, applications issue resource requests, and the manager honors them only within the boundaries of the policy. Other managers such as Quasar [36] do not use reservations. Instead, applications define performance goals e.g. completion time (for batch job applications) or request latency (for a web server application). Quasar then infers a suitable resource allocation by profiling and a classification technique based on collaborative filtering [73]. In Google's Omega [105] the resource manager is a thin abstraction that provides a transactional interface over shared cluster state, and hence does not enforce allocation policies. Using the transaction mechanism, application-specific schedulers can then access this state and allocate resources to their applications as needed.

With respect to resource assignment, Mesos delegates challenges (1) and (2) (see Section 3.2.1) to an application-specific scheduler, that decides which conrete offers to accept and which not, whereas challenge (3) is mainly addressed by the use of lightweight operating-system level virtualization [109] techniques, such as Linux Containers [74]. YARN addresses (1) and (2) by preferences declared by the application via the resource-request mech-

anism and (3) by optional operating-system level virtualization. Quasar's techniques for workload classification address (1), (2) and (3). In Google's Omega, resource assignment is delegated to application-specific schedulers that handle issues (1)-(3) internally.

3.2.3 Example: Apache Hadoop YARN

Apache Hadoop YARN [11, 122] is an open-source cluster resource manager. It is the result of decoupling resource management functionality from the MapReduce batch computation framework, that used to be integrated in the first versions of the Hadoop platform. YARN has evolved into a generic cluster resource manager, thus "demoting" the MapReduce framework to a *tenant application*. As YARN supports arbitrary tenant applications, many existing distributed computation frameworks such as Apache's Spark, Storm and Flink implement a binding to YARN's resource management API and can therefore be deployed as tenant applications.

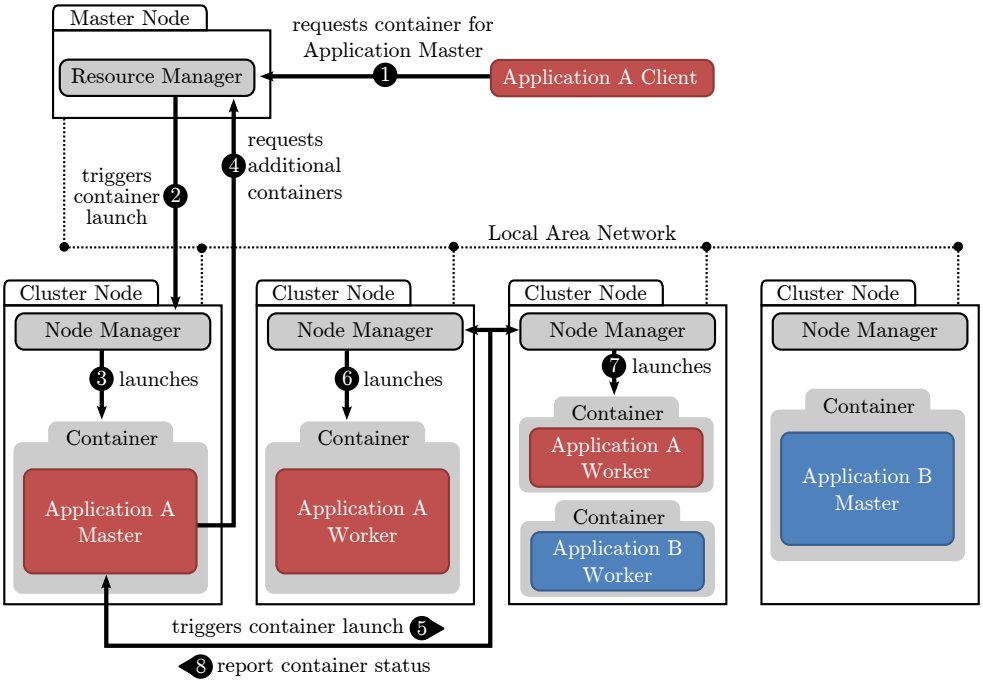


Figure 3.2: Application deployment process in YARN.

A YARN cluster has a central *Resource Manager (RM)* process running on a cluster head node, and one *Node Manager (NM)* process per cluster node.

The NM describes the resource properties (hostname, rack location, CPU cores, RAM) of its cluster node to the RM and sends regular heartbeats to indicate liveness. Figure 3.2 shows the steps of running a tenant application on top of a YARN-managed cluster. In step ❶ an application client registers the application with the RM and issues a *resource request* and a *launch description* to start an *Application Master (AM)* process. Via the resource request it describes desired properties of the cluster node that shall run the AM. A resource request must specify desired hardware properties (CPU cores and amount of RAM) and may specify additional locality properties such as hostname or rack location. The launch description contains all information necessary to launch the AM process, e.g. shell commands, environment variables and files that must be made available to the process. In step ❷ the RM allocates the requested resources, assigns them to a suitable cluster node and orders the NM to launch the AM. As already mentioned, resource allocation and assignment in YARN are governed by configurable policies. By default, YARN has a *capacity policy*, where the resource requests of each application are put into one of possibly several queues. Each queue is a guaranteed a (configurable) percentage of the cluster's resources. Resources may also be allocated to a queue beyond its configured percentage, as long as they are unallocated. In step ❸ the NM launches the AM process inside a *container*. By default, a container in YARN is only a reservation of CPU cores and RAM on a cluster node. The NM may terminate the AM if it consumes more RAM than reserved to its container (CPU limits are currently not enforced). Once the AM is running, in step ❹ it requests additional cluster resources from the RM in order to launch worker processes. As usual the RM checks the resource requests against its policy and assigns containers on cluster nodes as soon as possible. Once this is done, in step ❺ the AM can directly order the respective NMs to launch worker processes inside these containers. In steps ❻ - ❽ NMs start the containers and report their status to the AM. Due to cluster sharing between multiple tenant applications, it is likely that there are containers of other tenant applications on the cluster at the same time. This may result in colocation of containers on cluster nodes as shown in Figure 3.2.

3.3 Stream Processing Engines

This section surveys state of the art massively parallel stream processing engines that execute UDF-heavy data flows. First, this section summarizes eastablished *approaches* to providing engine core functionality i.e. deployment (Section 3.3.1), scheduling (Section 3.3.2) and execution (Section 3.3.3) of UDF-heavy data flows. Further, this section summarizes challenges and established approaches in addressing the major cross-cutting concerns fault tolerance (Section 3.3.4) and workload adaptiveness (Section 3.3.5). For clarity of exposition, whenever possible, the terminology introduced for Application Graphs is employed. Finally, since each of the existing systems and research prototypes occupies a different place in the design space, Section 3.3.6 summarizes state of the art systems in tabular form.

3.3.1 Deployment Architecture

A deployment architecture provides a high-level assignment of *engine functionality* to engine processes running on the cluster. The state of the art engines I surveyed follow the *master-worker* pattern or are *decentralized*, i.e. all processes of the running engine share the same functionality.

Master-Worker

The *master-worker* deployment architecture is predominant in the engines I surveyed [13, 10, 69, 12, 4, 101, 104, 60, 129]. It consists of a *master* process that coordinates the execution of the application, and a set of *worker* processes that perform the execution (see Section 3.3.3). The functionality of the master process always involves at least the following:

1. Scheduling (see Section 3.3.2)
2. Monitoring, which involves collecting and storing monitoring data about worker liveness and at least some application workload characteristics (e.g. stream rates or CPU utilization).
3. In fault tolerant engines (see Section 3.3.4), the master coordinates the engine's response to failures.
4. In elastic engines (see Section 3.3.4) the master defines the elastic behavior and if necessary (de)allocates cluster resources for worker processes via the cluster resource manager.

Figure 3.3 provides an overview of the steps involved in running a stream processing application on such a deployment architecture.

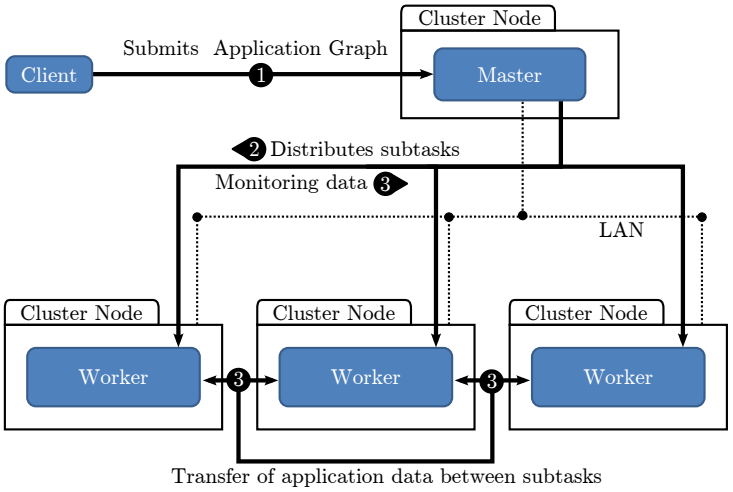


Figure 3.3: Cooperation between client, master and workers of a stream processing engine.

First, an external *client* process submits the stream processing application to the *master*. The submission contains the Application Graph with all of the UDF code, configuration data as well as any required libraries referenced by the UDFs. After computing a schedule (see Section 3.3.2), the master forwards subtasks including UDF code and configuration data to the *worker* processes. Now, the workers start to execute the application and are responsible for monitoring their subtasks for failures and efficiently shipping data between them along channels, which usually requires establishing pairwise network connections. During execution, the master process collects monitoring monitoring data and uses it to detect and mitigate failures or further optimize the execution.

Decentralized

A decentralized deployment architecture is employed by the S4 [94] engine. Here, all engine processes have a worker role and also additional responsibilities to provide automatic failover. S4 requires a cluster coordination service such as Zookeeper [56] to coordinate the worker processes. A client submits the application data to an arbitrary engine process, that puts it into a repository and notifies the other workers of the application. All of them pull

the application data and start executing the data flow. Failure monitoring and mitgation during the execution is also organized via the coordination service.

3.3.2 Scheduling

A scheduler assigns *subtasks* to worker processes on cluster nodes. A schedule for the example from Section 2.3.2 could look as displayed in Figure 3.4. Depending on the *scheduling strategy* used by an engine, schedules may however look very different. In the current state of the art engines that I surveyed, I found several scheduling strategies described in the following subsections.

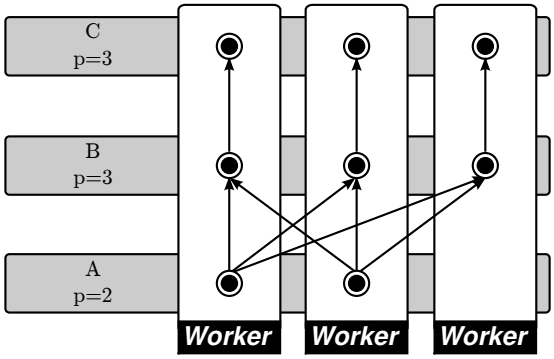


Figure 3.4: Exemplary schedule for Application Graph from Figure 2.3.

Round-Robin Scheduling
Round-Robin scheduling is widely used by the stream processing engines I surveyed [13, 12, 4, 104]. Given a template graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ and the corresponding runtime graph $G = (V, E)$, the scheduler enumerates all tasks $\mathbf{v}_1, \ldots, \mathbf{v}_n$ and builds a list of subtasks $[v_{1,1}, \ldots, v_{1,p(\mathbf{v}_k)}, \ldots, v_{n,1}, \ldots, v_{n,p(\mathbf{v}_n)}]$. It then performs round-robin assignment of the subtasks to a set of workers.

Round Robin scheduling has the benefit of simplicity, but may exhibit bad *locality* properties, because it does not consider the communication structure of the runtime graph. This may unnecessarily increase latency and lead to congested network interfaces.

Constrained Scheduling
Some engines allow application developers to constrain the scheduler's degrees of freedom by annotating the application with scheduling constraints.

Scheduling constraints found in the literature belong to the following categories:

- *Co-location constraints* specify that certain subtasks must be executed within the same worker process or cluster node. Subtask co-location can be used to ensure access to resources shared between subtasks on the same node, such as disks or memory segments. Additionally co-location can reduce communication overhead, because inter- and intra-process communication is considerably more efficient than communication via network.

- *Ex-location constraints* specify that certain subtasks must not be executed within the same worker process or cluster node. Ex-location can be used to guard against worker or node resource limits such as RAM or disk space, and to improve reliability in the face of failures.

- *Capability constraint* specify that certain subtasks must be scheduled for execution on cluster nodes with certain capabilities, e.g. specific hardware such as GPUs. In case of heterogeneous cluster nodes, this can be used to ensure that subtasks which require access to such resources are always executed on suitable nodes.

At the time of this writing, co-location constraints can be expressed in the two stream processing engines Apache Flink [10] and IBM Infosphere Streams [60]. Ex-location and capability constraints are specific to only IBM Infosphere Streams.

Hash-Based Scheduling

Some engines perform no explicit a-priori scheduling of subtasks to workers. Such engines [94, 69] rely on a richer data model, where every data item has a *key*. Here, *every* worker is potentially capable of executing *every* subtask. The concrete "schedule" is determined at runtime by consistent use of a hash function, that hashes an emitted data item's key to a destination worker. When the destination worker receives a data item with a specific key for the first time, it creates the consuming subtask on-the-fly by instantiating the respective UDF.

3.3.3 Runtime

The runtime of a stream processing engine executes a scheduled application in a distributed-parallel fashion. Figure 3.5 provides an overview of the typical components found in many engine runtimes. A *network transport* component abstracts away the details connection management. This includes finding the IP addresses and TCP/UDP ports of other worker processes as well as the low-level details of sending and receiving application data over TCP and UDP sockets between workers. Each data item that arrives via the network is dispatched to a queue for processing. At a later point in time, a thread dequeues the data item and invokes the message-handling method of a subtask with it. Any resulting data items from the message-handling method are then given to the dispatcher. Depending on whether the recipients of a data item are local or not, the dispatcher either locally enqueues it or sends it over the network to the worker process executing the respective recipient subtask.

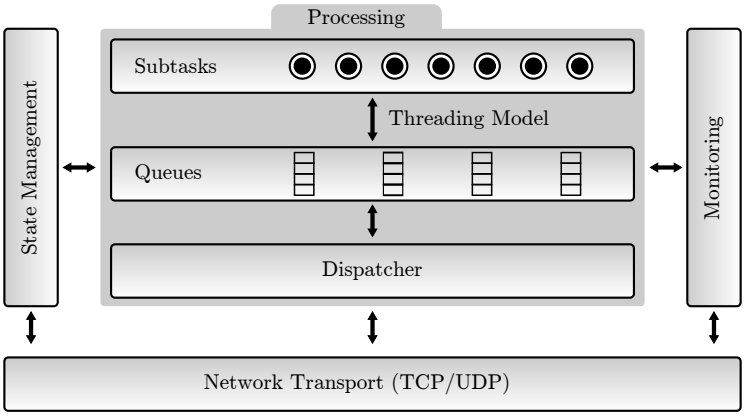


Figure 3.5: Anatomy of a stream processing engine runtime.

Depending on the engine, there may be one or multiple queues per worker, one or multiple threads per queue, and one or mutiple subtasks bound to each thread. A full per-system overview of this aspect is given in Section 3.3.6. With the exception of Samza [12], all surveyed engines use multiple queues per worker process and at least one thread per queue. This approach makes the best use of current multi-core CPUs. Since inter-thread communication has much less overhead than inter-process communication, which is again much cheaper than network communication, this structure also efficiently exploits data locality of colocated subtasks. Moreover, it gives the underlying

operating system various degrees of freedom in scheduling the threads among the individual CPU cores. For example, if a thread cannot fully utilize its assigned CPU resources or is waiting for an I/O operation to complete, the operating system can assign the idle CPU time to a different thread/process. The surveyed systems slightly differ in the way that subtasks are bound to threads. In [13, 12, 94] the subtasks associated with a thread all belong to the same task from the template graph, i.e. their output is always handed to the dispatcher. Conversely, in [10, 60] if a thread is associated with multiple subtasks, then they form a pipeline. This means that output produced by the first subtask of the pipeline can directly be handed to the second without dispatching, thus reducing the number of data items dispatched between threads (via queues) and worker processes (via network).

Thus the communication model of stream processing engines follows the *producer-consumer* pattern. Commonly, implementations of this pattern employ *queues* to achieve asynchronous message passing between threads. As depicted in Figure 3.6, the data items produced by subtasks are passed to and consumed by successor subtasks along channels.

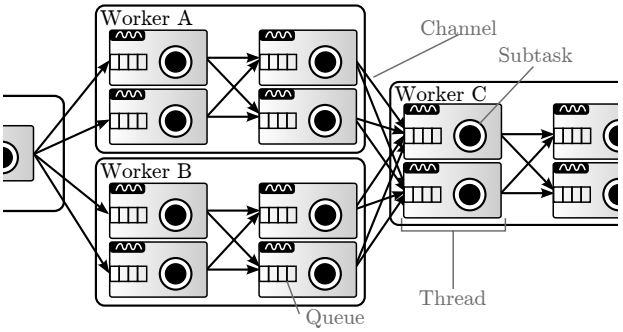


Figure 3.6: Producer-Consumer pattern in massively parallel stream processing.

Furthermore, a typical engine runtime has a local monitoring component that monitors the status of subtasks and creates statistics about them, e.g. processed data items or CPU usage. This data is usually shipped to the master to create a global view of the application status.

Finally, some engines include a component for *state management*. State management is a technique that allows subtasks to externalize their internal state in order to better deal with failures and elasticity. State management is covered in more detail in the next section.

3.3.4 Fault Tolerant Execution

Due to their continuous nature and highly-parallel and distributed execution, stream processing applications are particularly sensitive to node failures, software crashes and network outages. Although individual elements such as nodes, worker processes or subtasks have a small probability of failing, the probability that at least one such part fails is not so small. The ability to provide some degree of fault tolerance is therefore an important non-functional requirement for any stream processing engine. Current state of the art engines provide different *recovery guarantees* to applications. Recovery guarantees have been broadly categorized by Hwang et al. in [57] as *precise recovery*, *rollback recovery* and *gap recovery*. In engines providing precise recovery guarantees, the post-failure output of an application is identical to the output without failure, thus the effects of a failure are masked except for a temporary increase in latency. In engines with a rollback recovery guarantee, the application experiences no data loss in case of failures, but its output may differ from the output without failure, e.g. it may contain duplicates. Finally, in engines with a gap recovery guarantee, the application may experience data loss. To provide recovery guarantees, several *recovery techniques* have been developed. Techniques such as active or passive standby nodes described in [57] effectively double the resource consumption of an application and are rarely used in massively parallel engines. The techniques I found to be in use are summarized in the following.

Amnesia (AS)
The amnesia technique introduced in [57] detects and subsequently restarts each failed subtask on a different worker process. The advantage of this approach is its simplicity and that it introduces no overhead at runtime. However, amnesia recovery implies the loss of data items that have not been fully processed at the time of failure as well as the loss of the internal state of failed subtasks. Due to this, the amnesia technique is only applicable to applications that are tolerant to temporary data and state loss (or maintain no state). The amnesia technique is limited to providing gap recovery guarantees. S4 [94] and mupd8 [69] employ the amnesia technique.

Upstream Backup (UB) / Upstream Backup with Duplicate Elimination (UBD)
In the upstream backup technique introduced in [57], upstream subtasks log their output before shipping it to downstream subtasks. The logged data items are retained until they have been fully processed by downstream subtasks. This requires an acknowledgement protocol between neighboring subtasks to determine when logged data items can be safely discarded. If subtasks, workers or cluster nodes fail, the affected subtasks are restarted with an empty state and the logged data items are replayed by upstream subtasks in order to reconstruct the subtask state at the time of failure. Upstream backup introduces some runtime overhead due to the continuous logging of data items. Upstream backup alone provides rollback recovery. If subtasks are deterministic, i.e they produce the same output every time they have a particular state, upstream backup can be combined with the elimination of duplicate outputs to provide precise recovery. The efficiency of precise recovery via upstream backup is however dependent on how much logged data needs to be replayed.

State Management (SM)
State management is a technique that externalizes the internal state of subtasks. Externalized state is then periodically checkpointed to a location considered safe. In case of a failure, restarted subtasks can reconstruct their internal state using a checkpoint. Some engines persist state to a distributed and replicated key-value store [4, 69] while others use a shared filesystem [94, 101, 104, 60]. The SEEP system [29] transfers state checkpoints to upstream workers. State management alone only provides gap recovery because data items in transit may still be lost due to the use of the amnesia technique [94, 69]. In careful combination with upstream backup including duplicate elimination, state management can provide efficient precise recovery [29, 101, 4, 13] even for subtasks whose state depends on the whole history of input data. Without duplicate elimination, state management still improves the efficiency of rollback recovery [12].

3.3.5 Workload Adaptiveness

Many stream processing applications ingest data from sources at rates that may be unknown or unknowable in advance, and which may significantly

vary over the course of time. Additionally, even under steady ingestion rate, the cost (processor cycles or time) of processing data items can change due to complex behavior exhibited by UDFs. Unforeseen changes in workload can lead to overload where a stream processing application cannot keep up with the arrival rate of new input data. In consequence, the application's results may be become less useful or even useless.

Due to this, several stream processing engines have been proposed [13, 4, 101, 104, 41, 129, 29] that are *runtime elastic*, i.e. they adapt the amount of allocated cluster resources to the current workload without requiring a restart or redeployment of the application.

Horizontal and Vertical Elasticity
A horizontally elastic system is able to seamlessly, i.e without restart of the engine or the application, change the number of running worker processes and ad-hoc schedule subtasks on them. Since starting new worker processes requires allocation of new servers or containers, this also horizontally scales the amount of allocated cluster resources. Horizontal elasticity is the predominantly found form [13, 4, 101, 104, 41, 129, 29] of elasticity. Conversely, a vertically elastic stream processing system is able to seamlessly increase and make use of cluster resources (CPU cores, memory) allocated to already running worker processes, e.g. utilize more CPU cores sharing the same virtual memory space. Chronostream [129], which is built on a cluster resource manager with operating-system level virtualization (e.g. Linux containers) that provides ad-hoc resizable containers, is currently the only vertically elastic engine I am aware of. In the context of cloud computing, the act of allocating more/less servers has been called *scaling out/in*, while the act of using more/less powerful servers has been called scale up/down [88].

Resource Pools for Horizontal Elasticity
Both horizontal and vertical elasticity require a managed pool of resources (servers, virtual machines or containers) that can be allocated and deallocated on demand. The availability of a cloud or a large resource-managed cluster is therefore a requirement for elastic stream processing. For common IaaS clouds such as Amazon EC2 or the Rackspace cloud, a 2012 study [85] has shown that provisioning and starting a new VM takes time on the order of minutes, assuming common filesystem image sizes in the lower

gigabyte range. While the SEEP [29] engine builds on the Amazon EC2 IaaS cloud, the authors note that VM instances had to preallocated to deal with the minute-range provisioning time. The other elastic systems I surveyed [13, 4, 101, 104, 41, 129] relied on resource-managed clusters where container provisioning times are in the second range, because no filesystems have to be copied, and no operating systems need to boot.

Scaling Policies

An elastic stream processing engine must define a *scaling policy* that determines *when* elastic scaling is triggered and *how much* to scale in/out or up/down. Current state of the art engines employ the following scaling polices with the goal of preventing overload and resolving bottlenecks. Note that Section 6.3 discusses scaling policies found in in the broader thematic scope of cloud computing.

CPU Utilization-Based A scaling policy based on CPU utilization targets, monitors the CPU utilization of subtasks and/or worker processes and triggers scaling actions when the measured CPU utilization significantly diverges from a predefined utilization target. SEEP [29] continuously measures the CPU utilization of subtasks and scales *out* if a predefined utilization threshold is crossed. A policy for scaling *in* is however not specified. Google's Millwheel [4] paper hints towards a CPU target based policy to adjust data parallelism, but does not provide specifics other than it being based on the measured CPU utilization of worker processes.

Rate-Based A scaling policy based on stream rates monitors stream rates and triggers scaling actions when the consumption rate of tasks cannot keep up with the arrival rate of new data items. Inspired by [124], the AnduIN engine [104] uses this technique to detect bottlenecks and then rewrites the application graph using rewriting patterns that adjust data parallelism or pipeline-parallelism.

Congestion-Based A congestion based scaling policy detects the effects of subtasks that are not able to process data as fast as it comes in, and triggers scaling actions to alleviate congestion. In a recent paper Gedik et al. [41] propose such a scaling policy with SASO [51] properties (stability, accuracy, short settling time, overshoot avoidance) for stream processing applications running on IBM Infosphere Streams [60]. Given an upstream and a down-

stream task, the policy measures and memorizes for each degree of scale of the downstream task, how often an upstream task is blocked due to congestion (backpressure) and what throughput was achieved. If the congestion frequency is too high, the downstream task will be scaled out (accuracy property) unless scaling out from the current point yielded no improvement in throughput in the past (stability property). If the congestion frequency is too low, the downstream task will be scaled in (overshoot avoidance), unless scaling in from the current point has resulted in congestion in the past (stability property). Further, past measurements are discarded if workload characteristics change. To provide short settling times, the downstream task is not scaled out/in one subtask at a time, but in superlinearly increasing steps.

External An external scaling policy requires an entity other than the engine to determine how the stream processing application shall be scaled. The Storm [13] engine currently supports this simple type of policy by via command-line tooling[1] that allows users or other processes to scale out/in as well as scale up/down. In the TimeStream[101] engine, scaling policies can be defined by the stream processing application.

3.3.6 State of the Art Systems

Table 3.1 provides a tabular overview of the state of the art stream processing engines that execute UDF-heavy data flows. All terminology and abbreviations used in the table have been introduced in the previous sections. To briefly describe the thread model, the table uses a notation of the form $q/t/s$ with $q, t, s \in \{1, n\}$ where

- q is the number of queues per worker,
- t is the number of threads consuming from each queue,
- s is the number of subtasks bound to each thread.

For example, a $n/1/n$ thread model implies that there are multiple queues per worker, one consumer thread per queue and multiple subtasks bound to each thread.

[1] `http://storm.apache.org/documentation/Command-line-client.html`

Table 3.1: Properties of state of the art massively parallel stream processing engines executing UDF-heavy data flows.

	Apache Storm v0.9.2 [13, 117]	Apache Flink Streaming v0.9 [10]	WalmartLabs mupd8 2.0 [69]	Apache Samza [12]	S4 [94]	Google Mill-Wheel [4]	TimeStream [101]	AnduIN v2 [104]	IBM Infos-phere Streams v3.2.1 [60]	Elastic Sys-tem S [41]	SEEP [29]	ChronoStream [129]
Provenance	industrial	academic	industrial	industrial	industrial	industrial	academic	academic	industrial	academic	academic	academic
Open-Source	✔	✔	✔	✔	✔	✗	✗	✗	✗	✗	✔	✗
Status	Production	Prototype	Production	Production	Abandoned	Production	Prototype	Prototype	Production	Prototype	Prototype	Prototype
Architecture	Ma-Wo	Ma-Wo	Ma-Wo	Ma-Wo	Decentr.	Ma-Wo	Ma-Wo	Ma-Wo	Ma-Wo	Ma-Wo	Ma-Wo	Ma-Wo
Scheduler	Round Robin	Constrained (Co-Loc.)	Hash-Based	Round Robin	Hash-Based	Round Robin	Unknown	Unknown	Constrained Co & Ex Loc., Capab.	Constrained Co & Ex Loc., Capab.	Unknown	Round Robin
Thread Model	n/1/n	n/1/1	n/1/n	1/1/n	n/n/n	Unknown	Unknown	?/1/n	n/1/n	n/1/n	n/1/1	n/1/n
API												
– Msg-Handling	Bolt	✗	✗	✔	✔	✔	✗	✗	✔	✔	✗	✔
– Operator	Trident	✔	MapUpdate	✗	✗	✗	✗	✔	✗	✗	✔	✗
– Declarative	✗	✗	✗	✗	✗	✗	LINQ [87]	✗	SPL [53]	SPL [53]	✗	✗
Res. Manager Integration	YARN, Mesos	YARN, Mesos	✗	YARN	✗	Unknown	✗	Mesos	YARN [92]	Unknown	EC2	YARN, Mesos
Fault Tolerance												
– Guarantee	Rollback (Bolt)/ Precise (Trident)	✗	Gap	Rollback	Gap	Precise	Precise	Rollback	Gap/ Precise	Unknown	Precise	Precise
– Techniques	UB/ UBD+SM	✗	SM	UB+SM	SM	UBD+SM	UBD+SM	UB+SM	SM / AS	Unknown	UBD+SM	UBD+SM
Elasticity												
– Model	Horiz.	✗	✗	✗	✗	Horiz.	Horiz.	Horiz.	✗	Horiz.	Horiz.	Horiz.+Vert.
– Scaling Pol.	External	✗	✗	✗	✗	CPU Util.	External	Rate	✗	Congestion	CPU Util.	Unknown

3.3.7 Requirements Review

This section briefly reviews how the requirements of stream processing applications described in Section 2.1 are addressed by the state of the art engines surveyed in this chapter.

Requirement 1: Dealing with high-volume stream data from external sources
All state of the art systems provide the necessary horizontal scalability to ingest high volumes of data, i.e. they are capable of exploiting pipeline-, task- and data-parallelism as well as scheduling and exuting parallelized data flows on large compute clusters. UDF-heavy data flows faciliate the ingestion of data from *external* sources. In many cases engines provide ready-to-use adapters for external data sources, e.g. Storm provides standard connectors[2] for popular data sources such as the Kafka[3] messaging middleware. By encapsulating the required functionality within UDFs it is generally possible to ingest data from arbitrary external sources.

Requirement 2: Data heterogeneity
The surveyed engines are capable of processing and transmitting arbitrary data, whether its structured, semi-structured or unstructured. The only condition the object representation of data items that UDFs operate with must be serializable to a byte stream representation.

Requirement 3: Complex Processing
UDF-heavy data flows cover a wide range of application classes. Stream-relational operators such as projection, selection, as well as windowed aggregations and joins are generally expressible within this model, as evidenced by the powerful declarative SPL language of IBM Infosphere Streams [60] and several Operator APIs the include stream-relational operators [10, 13]. It shall however be noted that the open-source engines currently lack convenient declarative frontends. Beyond purely stream-relational processing, UDFs allow developers to integrate existing libraries to address complex application requirements, e.g. text mining, video de- and encoding, image manipulation to name just a few.

[2] https://github.com/apache/storm/tree/master/external/
[3] http://kafka.apache.org/

Requirement 4: Fault-Tolerant Execution
Most engines offer a type of fault-tolerance guarantee, such as rollback [13, 12, 104] or precise recovery [13, 4, 101, 60, 29, 129]. To achieve precise recovery guarantees, UDF state needs to be restored in the event of failues. For this reason all engines with precise recovery checkpoint UDF state and can recover from failures based on those checkpoints.

Requirement 5: Latency Constraints
As pointed out in Section 2.1, any notion of "low" or "high" latency is entirely application-specific. Despite that, none of the surveyed engines has the capability to define or enforce any type of application-specific latency constraints. Conversely, their goal is to provide the "lowest latency possible". Generally, it is the developer or operator of a particular stream processing application, who must estimate the effects of engine configuration and cluster resource provisioning on application latency. Some of the state of the art engines are elastic [13, 4, 101, 129, 104, 29], while some are not (yet) [10, 70, 94]. The unelastic ones require cluster resources to be permanently provisioned for peak-load in order to remain low latency in the face of varying and bursty application workload. Furthermore, the particular workload characteristics of an application are often unknown or unkowable in advance, which makes peak-load provisioning both expensive and inefficient. The elastic engines either do not specify a scaling policy [13, 101], or employ a policy that is based on CPU utilization, stream rates or congestion. While all of these policies prevent situations of *overload*, managing the execution towards fulfillment of application-specific latency constraints is not in their scope. In order to obtain a desired latency, the parameters for these scaling policies , e.g. the target CPU utilization, must be obtained by estimated guesses or trial-and-error.

Chapter 4: Stream Processing with Latency Constraints

Contents

Today's stream processing engines as presented in Section 3.3 excel at the highly parallel, fault-tolerant and low latency execution of UDF-heavy data flows. Although latency constraints are a common requirement of stream processing applications [9], state of the art engines currently do not provide facilities to formally express and enforce application-specific latency constraints.

This chapter addresses this gap and constitutes the principal contribution of this thesis. The overarching question of this chapter is "How can latency constraints be enforced with minimal resource footprint?" The challenge lies in the conflicting nature of the goals postulated in this question. Latency constraints are much simpler to enforce if the resource footprint does not matter. Conversely, if application latency is not much of a concern for the application, there are no detrimental effects to resource underprovisioning.

The general approach in this chapter is to build on – and occasionally modify – several engine core functionalities that were summarized in Section 3.3. First, Section 4.1 and Section 4.2 define a formal notion of *latency* and *latency constraints* for UDF-heavy data flows. Section 4.3 introduces a distributed *feedback cycle* that optimizes the execution of a stream processing application towards fulfillment of application-defined latency constraints. Each such feedback cycle begins with the collection and aggregation of workload statistics, which is covered in Section 4.4. The collected statistics provide input for three optimization techniques that complete the feedback cycle. The first one is Adaptive Output Batching (AOB), which is covered in Section 4.5. The AOB technique reduces the often substantial data item shipping overhead without violating latency constraints. The second technique is Elastic Scaling (ES) and covered in Section 4.6. The ES technique adjusts the parallelism of an UDF-heavy data flow so that application-specific latency constraints are fulfilled. The third and final technique is Dynamic Task Chaining (DTC) and covered in Section 4.7. The DTC technique reduces data flow latency by dynamically switching the communication between subtasks from queue-based asynchronous message passing to synchronous message passing at selected locations in the data flow.

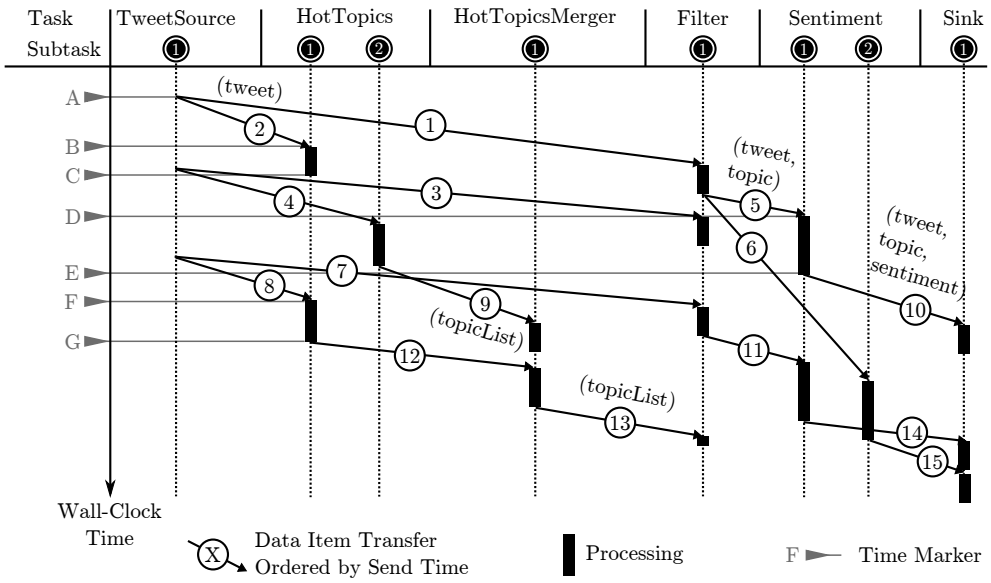


Figure 4.1: Exemplary flow of data items in the Twitter Sentiments application from Section 5.1.3.

4.1 Latency in UDF-heavy Data Flows

This section first discusses the challenges that need to be addressed in order to obtain a useful notion of latency in UDF-heavy data flow graphs. Afterwards, it provides a definition designed to address these challenges.

Let us first take a look at the data flow of the exemplary Twitter Sentiments application from Section 5.1.3, that will also used during the experimental evaluation in Chapter 5. Figure 4.1 maps a possible flow of data items to wall-clock time. The *TweetSource* subtask emits each tweet twice into the data flow: one copy to the branch that performs sentiment analysis, and one to the hot topics branch, that computes the most popular topics people are tweeting about. On the sentiment analysis branch, both *Filter* and *Sentiment* subtasks operate on a per-data-item basis. On the hot topics branch, both the *HotTopics* and *HotTopicsMerger* subtasks perform windowed aggregation, hence data items they produce are based on a history of data items consumed in the past.

How can we define a notion of *latency* for such an application? A first observation is that the Application Graph model for UDF-heavy data flow graphs as described in Section 2.3 is time-agnostic. UDFs can choose to embed

timestamps in data items if relevant to application semantics. For stream-relational queries such as those found in CQL [14], notions of application time have been described by Srivastava et al. in [111]. However, such semantics are out of scope for an engine that executes UDF-heavy data flows, due to the opaqueness of the application payload. In order to maintain the generality of the UDF-heavy model, a more natural notion of time is *wall-clock time*. Consequently, latency can be defined via the difference between points in wall-clock time. In Figure 4.1 the time difference $B - A$ is an example of latency caused by data item transfer, including any network and queueing delay. For subtasks such as *Sentiment* and *Filter*, that operate one a per-data-item basis, a time difference such as $E - D$ in Figure 4.1 adequately captures the latency caused by processing. Still, several issues need to be addressed for a useful notion of latency:

1. *How to define latency for UDFs that perform aggregation operations?* UDFs such as those in the *HotTopics* and *HotTopicsMerger* subtasks buffer consumed data items in order to produce an aggregate result at a possibly much later point in time. Figure 4.1 shows an example of this. Data item transfer 12 contains aggregated information from transfers 2 and 8. While the time difference $C - B$ captures the amount of time the first *HotTopics* task is busy with computation, it does not adequately capture *latency*. Rather, the data item from transfer 2 incurs a latency of $G - B$ in the *HotTopics* subtask, while the data item from transfer 8 incurs a latency of $G - F$. Hence any attempt at a proper definition of latency must not only account for time the UDF is busy, but also be able to capture time intentionally "lost" during aggregation.

2. *What is a useful notion of end-to-end latency?* End-to-end latency is an important performance metric for stream processing applications. Hence a latency constraint should be defined with respect to a notion of end-to-end latency. Since UDF-heavy data flows can be arbitrarily shaped DAGs, the semantic relationship between data items emitted at sources and consumed at sinks is generally unknowable to the engine, without making strong assumptions on application semantics. In the Twitter Sentiments application, end-to-end latency for the hot topics branch intuitively starts at the *TweetSource* and "ends" at the *Filter* subtask, not at the *Sink* subtask. To complicate matters further, we

may not only have multiple data parallel source and sink *subtasks* in the application's runtime graph, but also multiple task parallel source and sink *tasks* in the template graph.

3. *How to identify time critical parts of an application?* A notion of end-to-end latency is necessary but not sufficient to enforce latency constraints. Any notion of end-to-end latency useful towards this goal must be a *composite* value, that allows to clearly identify the time critical portions of the data flow graph.

In summary, we need a composite definition of end-to-end latency for UDF-heavy data flow graphs, that captures the delays caused by (i) data item transfer (including message-passing and serialization overhead) and (ii) UDF behavior (per-data-item vs. aggregation).

The remainder of this section first defines *channel* and *subtask latency*, that capture delay caused by data item transfer and UDF behavior respectively. From these two definitions, *sequence latency* is defined as a composite notion of end-to-end latency.

Definition: Channel Latency
Given two subtasks $v_i, v_j \in V$ connected via channel $e = (v_i, v_j) \in E$, we define the *channel latency* $l_e(d)$ as the difference in wall-clock time between the data item d exiting the UDF of v_i and entering the UDF of v_j. Channel latency may vary significantly between data items on the same channel due to differences in item size, network congestion, environmental effects (e.g. JVM garbage collection) and queues that need to be transited on the way in between.

Definition: Subtask Latency
Let us assume a subtask v with an inbound channel e_{in} and an outbound channel e_{out}. We define the *subtask latency* $l_v(d, e_{in}, e_{out})$ incurred by data item d that is consumed from channel e_{in} by task v as either *Read-Ready* or *Read-Write task latency.*

Read-Ready (RR) subtask latency is the difference in wall-clock time between (1) data item d being consumed from channel e_{in} and (2) subtask v becoming ready to read the next data item from *any* of its input channels. This defini-

tion is a good match for subtasks executing UDFs that perform per-data-item computation. RR subtask latency is only defined for subtasks with ingoing channels, which rules out source subtasks.

Read-Write (RW) subtask latency is the difference in wall-clock time between (1) data item d being consumed from channel e_{in} and (2) the next time subtask v writes any data item to e_{out}. This definition is a good match for UDFs that perform computation based on more than one previously consumed data item, e.g. aggregation. This definition has several implications. First, RW subtask latency is undefined on source and sink subtasks because they lack incoming or, respectively, outgoing channels. RW subtask latency can be infinite if the subtask never emits for certain combinations of ingoing and outgoing channels. Moreover, RW subtask latency can vary significantly between subsequent data items, for example, if the task reads two items but emits only one item after it has read the last one of the two. In this case the first item will have experienced a higher task latency than the second one.

A UDF is expected to *declare* to the engine, which one of the two options shall be used to measure subtask latency. By convention, we will quietly assume RR subtask latency as the default and only explicitly mention the use of RW subtask latency.

Definition: Runtime Sequence
A *runtime sequence* $S = (s_1, \ldots, s_n)$ with $n \geq 1$ shall be defined as an n-tuple of connected subtasks and channels within the same runtime graph, i.e. each s_i is either a subtask or a channel. For example, if s_2 is a subtask, then s_1 needs to be one of its ingoing and s_3 one of its outgoing channels.

The first element of a runtime sequence is allowed to be either a non-source subtask or a channel. The last element is allowed to be any type of subtask or a channel.

Definition: Runtime Sequence Latency
Let us assume a runtime sequence $S = (s_1, \ldots, s_n)$, $n \geq 1$ of connected subtasks and channels. For a data item d entering s_1, we can define the *runtime sequence latency* $l_S(d)$ that the item d incurs as $l^*_S(d, 1)$ where

$$l_S^*(d,i) = \begin{cases} l_{s_i}(d) + l_S^*(s_i(d), i+1) & \text{if } i < n \\ l_{s_i}(d) & \text{if } i = n \end{cases}$$

If s_i is a subtask, then $l_{s_i}(d)$ is equal to its subtask latency $l_{s_i}(d, s_{i-1}, s_{i+1})$ and $s_i(d)$ is the next data item emitted by s_i to be shipped via the channel s_{i+1}. If s_i is a channel, then $l_{s_i}(d)$ is its channel latency and $s_i(d) = d$.

Runtime sequence latency hence provides a notion of end-to-end latency with respect to a runtime sequence. Further, runtime sequence latency is fully decomposable into channel and subtask latencies.

4.2 Latency Constraints

This section defines latency constraints as a formal expression of a non-functional stream application requirements. In order to specify latency constraints, an application developer must be aware how much latency his or her application can tolerate in order to still be useful.

First, this section introduces *runtime constraints* in order to obtain runtime graph semantics for latency constraints. The subsequent definition of *template constraints* utilizes these semantics to express latency requirements for a template graph.

Definition: Runtime Constraint
A *runtime constraint* is a tuple $c = (S, \ell, t)$, where

- S is a runtime sequence,
- ℓ is a desired latency bound in an arbitrary but fixed time unit (e.g. milliseconds),
- and t is a time span in an arbitrary but fixed time unit (e.g. seconds).

A runtime constraint expresses a desired upper latency bound ℓ for the arithmetic mean of the runtime sequence latency $l_S(d)$ over all the data items $d \in D_t$ that enter S during any time span of t time units:

$$\frac{\sum_{d \in D_t} l_S(d)}{|D_t|} \leq \ell \tag{4.1}$$

For runtime graphs with high degrees of data parallelism, many runtime constraints may need to be specified to cover all runtime sequences. Therefore, runtime constraints are neither directly specified nor explicitly materialized at execution time. They are however relevant to the semantics of *template constraints*, which we will define next.

Definition: Template Sequence
Analogous to a runtime sequence, a *template sequence* $\mathbf{S}$ shall be defined as an n-tuple of connected streams and tasks within a template graph. The first element can be either a non-source task or a stream, while the last element can be any type of task or a stream. Analogous to the way the template graph fully induces the runtime graph (see Section 2.3), each template sequence $\mathbf{S}$ induces a set of runtime sequences $\{S_1, \ldots, S_n\}$.

By $tasksOf(\mathbf{S})$ we shall refer to the set of tasks within the template sequence $\mathbf{S}$. For example, if $\mathbf{S} = (\mathbf{v}_1, \mathbf{e}_1, \mathbf{v}_2, \mathbf{e}_2)$, then $tasksOf(\mathbf{S}) = \{\mathbf{v}_1, \mathbf{v}_2\}$.

Definition: Template Constraint
A *template constraint* $\mathbf{c} = (\mathbf{S}, \ell, t)$ induces a set of runtime constraints $\{c_1, \ldots, c_n\}$, where $c_i = (S_i, \ell, t)$ is a runtime constraint on the i-th runtime sequence S_i induced by template sequence $\mathbf{S}$.

Figure 4.2 presents an example that clarifies the relationship between template and runtime constraints. Template constraints can be *attached* to a template graph by an application developer. Attaching *multiple* template constraints shall be possible under the condition that (i) their template sequences do not share any streams and (ii) the time span t is the same for all template constraints.

Discussion
Usually, the degree of parallelism of a large scale stream processing applications is so high, that manual specification or even just the explicit materialization of all possible *runtime constraints* is prohibitively expensive. The reason for this is the combinatorial "explosion" of possible runtime sequences within such graphs. Hence, runtime constraints are strictly a *backend* concept, and provide required semantics, but are not specified by an application developer directly. Conversely, template sequences and constraints are *frontend* concepts. An application developer can use template sequences to conveniently

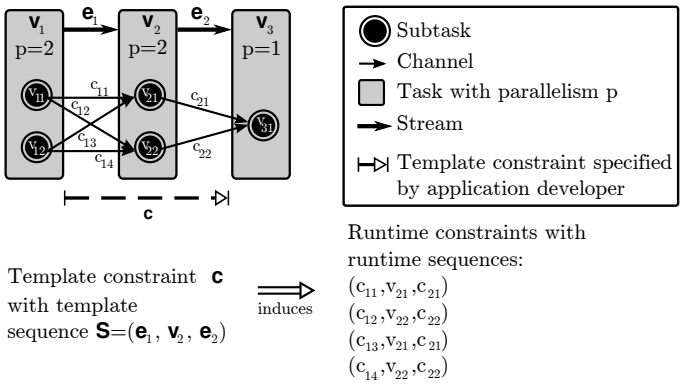


Figure 4.2: Relationship between template constraints, runtime constraints and their respective sequences.

identify latency critical portions of the template graph, and template constraints to specify their concrete latency requirements. The engine can then take steps to enforce the induced runtime constraints.

Runtime constraints do not specify a hard upper latency bound for every single data item, but a statistical upper bound on mean runtime sequence latency. In this context the purpose of the time span t is to provide a time frame, in which constraint violations can be detected. With $t \to \infty$ a runtime constraint would cover all data items to ever pass through the runtime sequence. In this case, it is not possible to evaluate during the execution of the application, whether or not a runtime constraint has been violated. The time frame t should be chosen low enough to ensure fast detection of constraint violations and changes in workload characteristics, but also high enough so that a reasonable amount of measurement data can be gathered. In practice, choosing $t = 5\ s$ produced reasonable results.

It may be desirable to enforce harder latency bounds than *mean latency*, e.g. a 99th latency percentile. While such performance metrics are relevant, it is questionable whether meaningful hard upper bounds can be enforced in practice, considering the complexity of most real-world setups and the violent changes in workload characteristics, e.g. bursty stream rates, that real-world applications have to deal with.

4.3 Latency Constraint Enforcing: Overview

As described in the previous section, latency constraints can express the non-functional latency requirements of a stream processing application. This section first outlines the assumptions on engine architecture that the techniques for constraint enforcing (AOB, ES and DTC) depend on. The remainder of the section gives an overview of the three techniques and how they collaboratively optimize the execution of a stream processing application towards application-defined latency constraints.

4.3.1 Assumptions on Engine Architecture

As previously mentioned, a deliberate choice in the design of the AOB, ES and DTC techniques is to build on – and occasionally modify – established approaches to provide engine core functionalities as described in Section 3.3. Unlike a clean-slate design approach, this has been benefit of facilitating the integration of the techniques into existing engines. In this spirit, the techniques make the following assumptions on engine architecture:

- **Deployment Architecture (see Section 3.3.1):** A Master-Worker deployment architecture is assumed. Not only is it the most common deployment architecture in today's engines, it also allows to create a consistent view of the whole application on the master, which is useful to some of the techniques presented in the remainder of this chapter.

- **Scheduler (see Section 3.3.2):** A scheduler with round-robin scheduling or – preferably – colocation constraints shall be assumed. The DTC technique relies on a schedule with colocated and pipeline-parallel subtasks. While such a schedule *can* be the result of a round-robin scheduler, colocation constraints can be used to consciously *create* such a schedule. Furthermore, the scheduler shall be able to (de)commission new workers on-the-fly from a cluster resource manager, which is assumed by the ES technique.

- **Runtime (see Section 3.3.3):** A runtime with a n/1/m thread model to execute subtasks shall be assumed, i.e. n input queues per worker, one thread per input queue and m adjacent, pipeline-parallel subtasks bound to each thread. Despite n/1/m *support*, the initial *default* during

execution shall always be m=1. The DTC technique in Section 4.7 adaptively adjusts the value of m during execution. Data items shall be passed via asynchronous message passing between all threads, i.e. via queues between subtasks on the same worker and via TCP/IP between subtasks on different workers.

- **Elasticity (see Section 3.3.5):** The engine shall be able to horizontally scale tasks at runtime. The ES technique described in Section 4.6 provides a scaling policy that controls horizontal scaling and relies on the underlying scheduler to start and stop subtasks accordingly.

- **Fault-Tolerance (see Section 3.3.4):** Fault-tolerance functionality is not assumed but considered orthogonal, as it is out of the scope of this chapter.

4.3.2 Distributed Latency Constraint Enforcing

Let us assume an arbitrary stream processing application with a set of attached latency constraints as defined in Section 4.2. Figure 4.3 gives a bird's eye overview of how latency constraints are enforced at runtime. It shows a distributed *feedback cycle* that starts with measurements of workload statistics on workers. These workload statistics go through several aggregation steps. The output of each step serves as input to the techniques for constraint enforcing. The techniques trigger actions that optimize the execution of a stream processing application towards constraint fulfillment. This feedback cycle is completed once every configurable *adjustment interval*, in order to obtain a system that adapts quickly to variations in application workload. The adjustment interval should be chosen low enough so as to obtain a system that adapts quickly, and high enough so that there is a sufficient amount of fresh workload statistics to act upon. In practice, an adjustment interval of 5 s yielded a reasonably responsive system. In the following, we will take a brief look at the components from Figure 4.3 in the order as they appear in the feedback cycle.

Statistics Reporters

Statistics Reporters measure a set of *workload statistics* and report then to Runtime Graph Aggregators (RGAs) once per measurement interval. Each Statistics Reporter is tied to a subtask and its measurement logic is invoked

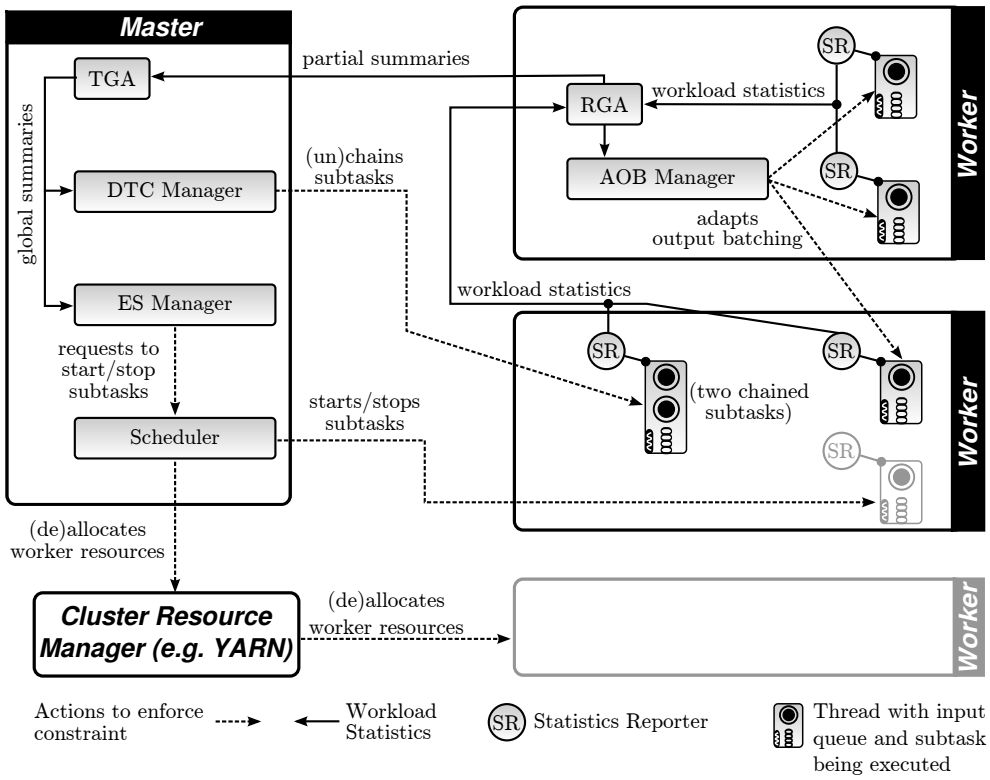


Figure 4.3: Overview of distributed latency constraint enforcing.

from the subtask via callbacks. Statistics Reporters are described in more detail in Section 4.4.

Runtime Graph Aggregators (RGAs) and Adaptive Output Batching (AOB) Managers

The task of tracking and enforcing runtime constraints on any central node may impose undue overhead for this node or overwhelm it, especially for large data flows with many subtasks and channels. Therefore, this task is divided between workers. RGAs collect and cache workload statistics for all subtasks and channels of a specific set of runtime constraints. The workload statistics within each RGA serve two purposes. First, they are used by an AOB Manager on the same worker to reduce the often substantial data item shipping overhead without violating latency constraints. The respective AOB technique as well as the placement of RGAs and AOB Managers is described in detail in Section 4.5. The second purpose is *summarization* to obtain runtime statistics for template graph entities (tasks and streams).

Each RGA forwards so-called partial-summaries to the Template Graph Aggregator (TGA) on the master. The process of summarization is described in detail in Section 4.4.2.

Template Graph Aggregators (TGAs)
On the master, the TGA collects and caches the partial summaries received from RGAs. Once per adjustment interval it merges the partial summaries into one global summary per template constraint (see Section 4.4.2).

The Elastic Scaling (ES) Manager
Once per adjustment interval, the Elastic Scaling (ES) manager uses the global summaries to determine whether scaling actions need to be taken in order maintain a minimal but sufficient resource footprint to fulfill the template constraints of the application. The details of the ES technique are described in Section 4.6. While the ESs manager determines *when* and *how much* to scale, the engine's scheduler determines *where* to start or stop specific subtasks. This may require the allocation of new workers from the Cluster Resource Manager, or may lead to empty workers that can be terminated to free unused cluster resources.

The Dynamic Task Chaining (DTC) Manager
Analogous to the ES Manager, a DTC Manager runs on the master and acts upon the global summaries obtained once per adjustment interval. It can decide to *chain* or *unchain* subtasks in order to reduce data flow latency. Chaining removes intermediate queues between connected subtasks and causes those subtasks to be executed within the same thread. Section 4.7 describes the Dynamic Task Chaining (DTC) technique in detail.

4.3.3 The Nephele Stream Processing Engine

The Nephele Stream Processing Engine (SPE) was developed as part of this thesis and implements the overall engine architecture described in the previous sections, as well as the techniques for latency constraint enforcing described in this chapter. Its name is derived from the parallel data processing framework Nephele [20], that executes DAG-based data flow programs on compute clouds and clusters. Although the Nephele SPE shares some of the codebase with the Nephele framework, it has been heavily modified and

extended for latency-constrained stream processing. In particular, the scheduler of the Nephele SPE is capable of reserving cluster resources on-demand from the YARN cluster resource manager, using the deployment process already described in Section 3.2.3.

Note that the focus of this chapter is on concepts rather than engine-specific implementations. Despite that, occasional implementation aspects will be mentioned in this chapter, that intend to shed light on practical issues that arose when creating the Nephele SPE.

4.4 Workload Statistics and Summaries

Each feedback cycle begins with the collection of workload statistics, that are subsequently reported and summarized. They provide the input that the AOB, ES and DTC techniques require to optimize the execution of a stream processing application towards constraint fulfillment.

This section describes the process of workload statistics measurement, reporting and summarization in a bottom up fashion. Section 4.4.1 describes how Statistics Reporters create workload statistics, while Section 4.4.2 describes how workload statistics are summarized by RGAs and the TGA.

4.4.1 Statistics Reporters and Workload Statistics

A *Statistics Reporter* measures and reports a set of workload statistics to RGAs once per adjustment interval. Conceptually, each Statistics Reporter belongs to a subtask from which it receives callbacks to perform measurements. Inside these callbacks it measures several aspects of the subtask as well as of its channels. Once per adjustment interval, it aggregates those measurements into *workload statistics*.

A *workload statistic* is a statistic (e.g. the mean) of a measurable aspect of a subtask or channel and computed over the measurements performed during an adjustment interval. For example, the *mean channel latency* l_e is the mean of the channel latencies over all the data items sent through e during the last adjustment interval. Additional workload statistics – mostly means and variances – are required by each of the three techniques (AOB, ES, DTC). Their full description is out of context at this point, as often their use can

only be motivated in the context of the respective technique. Appendix A lists and describes all collected workload statistics and shall serve as a point of reference.

All workload statistics are measured using the same methodology. Each measurable aspect of the workload is considered a random variable with generally unknown probability distribution. Measurements are obtained by *random sampling*, in order to minimize the measurement overhead. The sampling procedure uses a Bernoulli design, where each candidate is sampled with the same configurable probability. The workload statistic is estimated by computing the respective statistic over the measured samples using numerically stable algorithms, such as Knuth's online algorithm for computing mean and variance [67].

Implementation Notes
The sampling procedure for subtask statistics (e.g. mean subtask latency) can be implemented by having subtasks invoke callback functions on their respective Statistics Reporter. Via those callbacks, the Statistics Reporter encapsulates the sampling procedure without any changes to the subtask code. For example, let us assume that a data item is read from a channel by subtask with Read-Ready subtask latency. Right after reading the data item, the subtask invokes the callback of the Statistics Reporter, whose sampling procedure might decide that a subtask latency sample should be taken. In this case the Statistics Reporter memorizes the current system time. After the UDF has processed the data item and all resulting data items have been sent, the subtask invokes another callback. Now, the Statistics Reporter can complete the sample by computing the difference between the previously memorized and the current system time and adding the difference to the mean subtask latency statistic. Channel statistics such as mean channel latency can be estimated using *tagged* data items. A tag is a small piece of data that contains a creation timestamp and a channel identifier. Before the sending subtask ships a data item it invokes a callback of the Statistics Reporter, whose sampling procedue might decide to tag the data item. In this case the tag is evaluated in a callback of a Statistics Reporter at the receiving subtask, just before the data item enters the receiving UDF. While the sender decides which data items to tag (sample), the receiver computes the statistic. If the sending and receiving subtasks run on different workers,

clock synchronization is required.

To further reduce overhead, the workload statistics collected on a worker are grouped by the receiving RGA. Statistics Reporters on the same worker merge their statistics into the same reports, so that only one report per receiving RGA is sent per adjustment interval.

4.4.2 Workload Statistics Summaries

A so-called *summary* is an aggregation of workload statistics for the tasks and streams of a template constraint. Once per adjustment interval each RGA aggregates the workload statistics it has received into one *partial summary* per template constraint. All partial summaries are shipped to the TGA on the master and merged into one *global summary* per template constraint. Global summaries are consumed by the ES and DTC techniques described in Section 4.6 and Section 4.7.

Let us assume a template graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, a runtime graph $G = (V, E)$ and a template constraint $\mathbf{c} = (\mathbf{S}, \_, \_)$. If stream $\mathbf{e}$ is part of the template sequence $\mathbf{S}$, then the global summary for constraint $\mathbf{c}$ contains

$$l_{\mathbf{e}} = \frac{1}{n} \sum_{i=1}^{n} l_{e_i}, \tag{4.2}$$

where $e_1, \ldots, e_n$ are the n channels of $\mathbf{e}$ and l_{e_i} is the mean channel latency of e_i. The global summary contains several other values that are computed from their respective workload statistics. Analogous to workload statistics, the concrete member values in summaries can only be properly motivated in the context of the respective technique. For this reason, the description of each technique in Section 4.6 and Section 4.7 briefly lists the values required from the global summary.

Partial summaries are structurally identical to the respective global summary created by the TGA. However, each RGA has workload statistics only for a subset of all constrained subtasks and channels and hence computes a partial summary from that data. In the above example, the value for $l_{\mathbf{e}}$ in each partial summary is computed over a subset of of the channels of $\mathbf{e}$. The TGA merges those values to obtain the result of Equation 4.2.

Note that RGAs are not deployed on every worker. Instead, they are deployed in pairs with AOB Managers, as these require direct access to workload statistics. The placement of AOB Managers (and therefore also RGAs) is described in detail in Section 4.5.4.

4.5 Adaptive Output Batching (AOB)

Current state of the art stream processing engines such as Apache Storm [13] ship data items immediately between producing and consuming subtasks. This shipping strategy is natural fit with their goal to offer the lowest possible latency. However, it is not optimal when designing a system that can enforce latency constraints, while at the same time reducing cluster resource consumption.

The "immediate shipping" strategy does not account for the often high overhead of network transport in many parts of the commodity technology stack, especially for small data items. Shipping small data items one-by-one via TCP/IP causes high overhead due to data item metadata, memory management, thread synchronization, ISO/OSI layer headers, system calls and hardware interrupts.

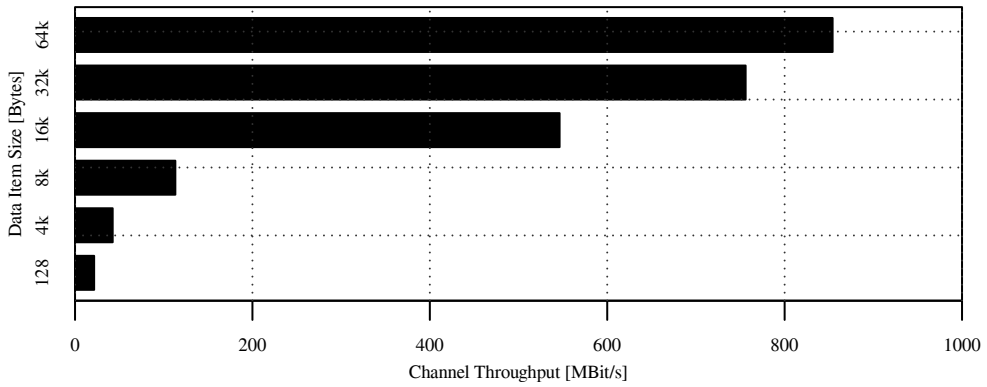


Figure 4.4: Effect of different data item sizes on achievable channel throughput.

To illustrate the magnitude of this effect, consider a small stream processing application that consists of only two subtasks: a sender and a receiver subtask connected by a single channel. The sender creates data items of fixed size as fast as possible and sends them to the receiver subtask on a remote

worker. The experiment was run several times for a range of data item sizes. The results are depicted in Figure 4.4 that shows the *effectively* achievable throughput in MBit/s for various data item sizes. The figure shows that the achievable throughput grows with data item size. With large data items of 64 or 32 KB in size, we are able to almost fully saturate the 1 GBit/s network link between the sender and the receiver. However, with small data item sizes such as 128 bytes, the engine is unable to attain a data item throughput of more than 10 MBit/s. This behavior is inherent to TCP/IP networks, which is why the Nagle algorithm [93] has been designed to lower the amount of small packets sent over the network. The mechanism by which this is achieved – buffering up outgoing data before shipping it – increases achievable throughput at the cost of increased latency.

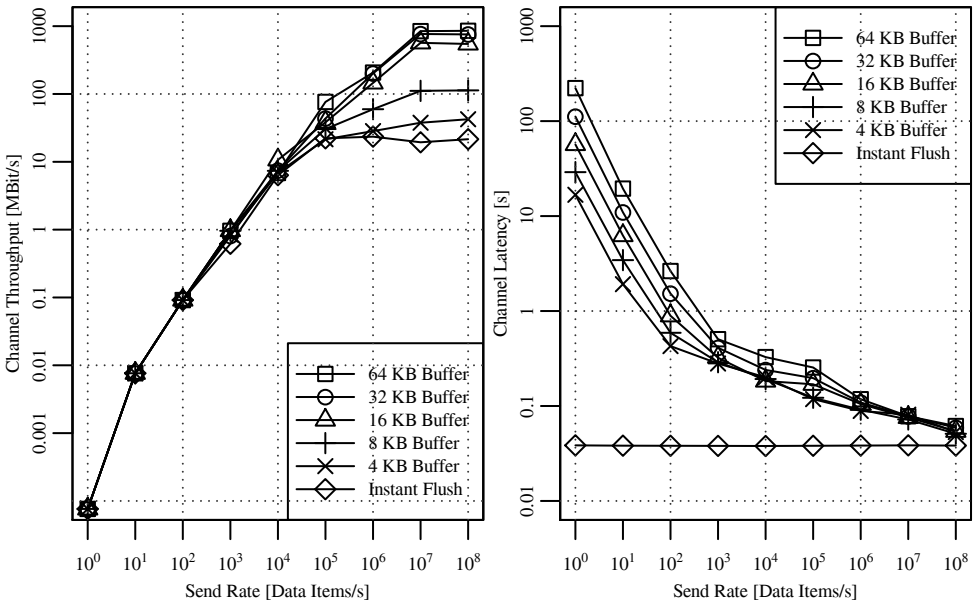


Figure 4.5: The effect of different output buffer sizes on channel throughput and latency.

In the spirit of the Nagle algorithm, we can use this behavior towards enforcing latency constraints – if the constraints leave "slack time". Sending data items in larger batches trades off latency against achievable throughput. To demonstrate this effect, consider a modified version of the previous example application. Now, both the send rate as well as the data item size[1] are fixed,

[1]Data item size is fixed at 128 bytes.

and the sending subtask employs a simple batching strategy, that puts data items into an output buffer of fixed size. The buffer is shipped only once it is full. Figure 4.5 shows the achieved channel latency and throughput for various send rates. The average latency from the creation of a data item at the sender until its consumption at the receiver depends heavily on the sending rate and the size of the output buffer. With only one created data item per second and an output buffer size of 64 KB, it takes more than 222 seconds on average before a data item arrives at the receiver. At low send rates, the size of the output buffer has a significant effect on the latency. With increasing send rate, the latency converges towards a lower bound. At a rate of 10^8 data items per second, the system is again saturated resulting in an average data item latency of approximately 50ms, almost independent of the output buffer size. As a baseline, we also executed separate runs of the application, where data items are shipped immediately. As a result, the average data item latency was uniformly 38ms, independent of the sending rate.

In summary, the simple batching strategy highlights an interesting trade-off that exists in distributed stream processing: Output batching can be used to trade off channel against achievable channel throughput. The resources freed by the lower overhead of data shipping can then be used to lower the resource footprint of the stream processing application, while still fulfilling constraints. Putting the application developer in charge of this trade-off assumes that (a) the engine allows him to specify output buffer or batch sizes, (b) he can estimate the expected sending rates for all channels in the application, and (c) the expected sending rates do not change over time. For complex applications with many subtasks and channels this is not a realistic expectation. Therefore, an automated and adaptive strategy for output batching could alleviate this burden from the application developer.

4.5.1 Overview

The AOB technique trades off shipping throughput against shipping latency in a workload-adaptive manner, so that the latency constraints of an application are met with the lowest possible overhead. It consists of four parts, each of which is covered in its own subsection:

- A modified data item shipping strategy for subtasks, where data items

are batched until a certain *output batch lifetime* is reached. This part is described in Section 4.5.2.

- The AOB Manager, that analyzes the workload statistics collected by an RGA in order to set the output batch lifetime of latency constrained channels. This part is covered in Section 4.5.3.
- A setup algorithm, that splits a runtime graph into subgraphs along latency constrained runtime sequences. Each subgraph is assigned to an AOB Manager, in order to distribute the overhead of latency constraint enforcing. This part is described in Section 4.5.4.

4.5.2 The Output Batching Mechanism

The output batching mechanism shown in Figure 4.6 is a low-level mechanism of the engine runtime to ship data items in a deadline-driven manner. At any time, each channel e shall have up to one *output batch* as well as a configured *output batch lifetime* $oblt_e \geq 0$, with an initial default of $oblt_e = 0$ (immediate shipping). An output batch is a container data structure for serialized data items and has a shipping *deadline*. Each subtask has an associated *Flush Thread*, where output batches can be scheduled for shipping. Once the deadline of a batch has expired, the Flush thread ships the batch by handing it to the dispatcher of the engine runtime. Thus, shipping a data item via a channel encompasses the following steps:

1. Check whether the channel has an existing output batch and, if not, open a new batch. If $oblt_e > 0$, schedule the new batch at the Flush Thread with deadline $t + oblt_e$, where t is the current system time. If no new batch was opened, or $oblt_e = 0$, proceed with the next step.
2. Serialize the data item and add it to the output batch.
3. If $oblt_e = 0$, ship the batch immediately.

Note that the output batch lifetime of a channel is controlled by one or more AOB Managers. In case of conflicting output batch lifetimes requested by two or more AOB Managers, the output batching mechanism always chooses the *lowest* output batch lifetime.

Output batches can be efficiently implemented by serializing data items into a large, fixed-size byte buffer. In this case, step (3) needs to be modified to also ship an output batch if its underlying byte buffer is full, which may

happen before its flush deadline expires. In general, a reasonable default size for these byte buffers is in the 32 – 64 KB range, because larger batches do not yield a significant increase in throughput (see Figure 4.4). However, if data item sizes and rates can be accurately estimated prior to execution, lower buffer sizes may yield similar results at lower memory consumption.

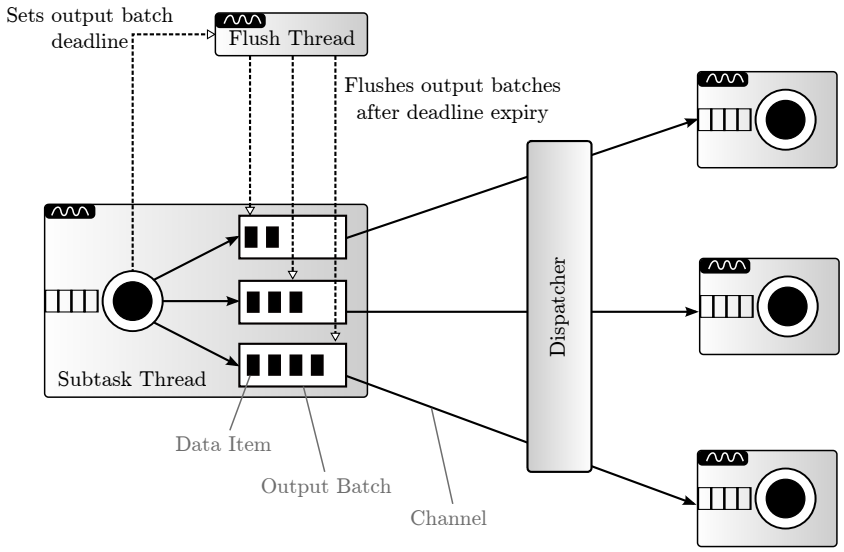


Figure 4.6: Overview of the output batching mechanism.

4.5.3 Managing Output Batching

An AOB Manager controls the output batch lifetime of the channels in a *subgraph* of a runtime graph. Its purpose is to enforce all runtime constraints in that subgraph, while at the same time minimizing shipping overhead. An AOB Manager analyzes the workload statistics collected by an RGA on the same worker and configures the output buffer lifetime of channels in its subgraph. The discussion of where to place AOB Managers and which subgraphs to assign to them is deferred until Section 4.5.4.

For the remainder of this section, let us assume a template graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ with induced runtime graph $G = (V, E)$, as well as a template constraint $\mathbf{c} = (\mathbf{S}, \ell, t)$ with induced runtime constraints C. An AOB Manager is *responsible* for enforcing a subset $C' \subseteq C$ of all runtime constraints. It is assumed that its RGA receives workload statistics for a subgraph $G' = (V', E')$ of G that contains all subtasks and channels part of the runtime constraints in C'. At

the end of each adjustment interval, the RGA holds the following workload statistics:

$$\begin{aligned} l_v &\quad \forall v \in V' \\ (l_e, obl_e) &\quad \forall e \in E', \end{aligned}$$

where

- l_v is the *mean subtask latency* of v,
- l_e and obl_e are the *mean channel latency* and *mean output batch latency* of e.

Please see Appendix A for a detailed description of the above workload statistics.

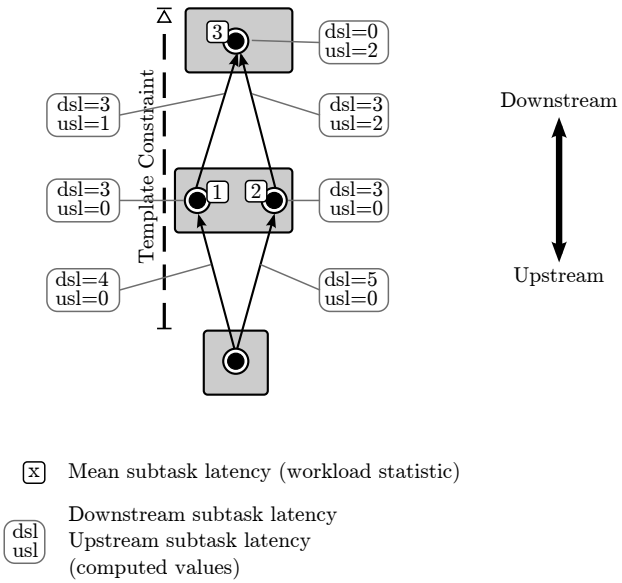


Figure 4.7: Example of the downstream/upstream subtask latency values computed by Algorithm 1 and Algorithm 2. These values values are computed by backward- and forward-propagation of subtask latency along the template constraint.

At the end of every adjustment interval, the AOB Manager executes Algorithm 3, which sets an output buffer lifetime for each channel in G'. First, it computes the *downstream and upstream subtask latency* for each constrained element $x \in E' \cup V'$. The downstream subtask latency $x.dsl$ of element x is defined as the maximum amount of subtask latency, that data items experience *after* having passed through x. Note that each element x may be part of several runtime sequences, each one with its own total subtask latency. As we do not know beforehand which way data items take, the downstream subtask latency definition provides a worst case calculation. Downstream subtask latency (dsl) is computed in Algorithm 1 by backward-propagation of subtask latency through the subtasks and channels in G'. In the context of the algorithm several helper functions are used:

- $isStream(\mathbf{x}_i)$ tests whether element $\mathbf{x}_i$ is a stream or not.
- $channels(\mathbf{x}_i, E')$ returns the set of all channels in E' that belong to stream $\mathbf{x}_i$.
- $subtasks(\mathbf{x}_i, V')$ returns the set of all subtasks in V' that belong to task $\mathbf{x}_i$.

Upstream subtask latency is defined in an analogous manner and Algorithm 2 computes it by forward-propagation of subtask latency. Figure 4.7 shows an example of downstream/upstream subtask latency values as computed by Algorithm 1 and Algorithm 2.

Once usl/dsl values have been computed, Algorithm 3 proceeds by setting a target output buffer lifetime $e.oblt$ for each channel e of each stream in $\mathbf{S}$. By setting output batch *lifetime* $e.oblt$, we want to influence the workload statistic mean output batch *latency* obl_e, which is the mean delay data items incur on the channel due to output batching.

Given a channel, line 6 first computes a desired *mean output batch latency* $targetObl$ for the channel, by subtracting the channel's precomputed dsl and usl from the available "time budget" given by the constraint ℓ, and equally distributing the remainder over the channels of the constrained sequence. This yields the available time budget *per channel*. In order to account for the fact that channel latency also includes queueing and possibly network transfer, we multiply this value with a factor $0 \leq w_{ob} \leq 1$. The so called *output batching weight* w_{ob} is a configuration constant, that determines the fraction of the time budget that should be spent on output batching.

Algorithm 1 SetDownstreamSubtaskLatencies(G', $\mathbf{S}$)

Require: Runtime subgraph $G' = (V', E')$ and template sequence $\mathbf{S} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$

1: $x.dsl \leftarrow 0$ for $\forall x \in channels(\mathbf{x}_n, E') \cup subtasks(\mathbf{x}_n, V')$
2: **for all** $i = n-1, \ldots, 1$ **do**
3: **if** $isStream(\mathbf{x}_i)$ **then**
4: **for all** $e = (v_a, v_b)$ in $channels(\mathbf{x}_i, E')$ **do**
5: $e.dsl \leftarrow v_b.dsl + l_{v_b}$
6: **if** $v_a.dsl = undef$ **then**
7: $v_a.dsl \leftarrow e.dsl$
8: **else**
9: $v_a.dsl \leftarrow \max\{v_a.dsl, e.dsl\}$
10: **end if**
11: **end for**
12: **end if**
13: **end for**

Algorithm 2 SetUpstreamSubtaskLatencies(G', $\mathbf{S}$)

Require: Runtime subgraph $G' = (V', E')$ and template sequence $\mathbf{S} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$

1: $x.usl \leftarrow 0$ for $\forall x \in channels(\mathbf{x}_1, E') \cup subtasks(\mathbf{x}_1, V')$
2: **for all** $i = 2, \ldots, n$ **do**
3: **if** $isStream(\mathbf{x}_i)$ **then**
4: **for all** $e = (v_a, v_b)$ in $channels(\mathbf{x}_i, E')$ **do**
5: $e.usl \leftarrow v_a.usl + l_{v_a}$
6: **if** $v_b.usl = undef$ **then**
7: $v_b.usl \leftarrow e.usl$
8: **else**
9: $v_b.usl \leftarrow \max\{v_b.usl, e.usl\}$
10: **end if**
11: **end for**
12: **end if**
13: **end for**

Algorithm 3 AdjustOutputBufferLifetimes(G', $\mathbf{C}$)

Require: Runtime subgraph $G' = (V', E')$ and template constraint $\mathbf{C} = (\mathbf{S}, \ell, \_)$ with template sequence $\mathbf{S} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$

```
1:  SetDownstreamSubtaskLatencies(G', S)
2:  SetUpstreamSubtaskLatencies(G', S)
3:  for all x_i in x_1, ..., x_n do
4:    if isStream(x_i) then
5:      for all e ∈ channels(x_i, E') do
6:        targetObl ← max{0, w_ob × (ℓ − e.dsl − e.usl)/countStreams(S)}
7:        Δ ← targetObl − obl_e
8:        e.oblt ← max{0, min{2 × targetObl, e.oblt + Δ}}
9:      end for
10:   end if
11: end for
```

Directly choosing the correct value of $e.oblt$ that achieves $obl_e \approx targetObl$ requires us to know the underlying distribution of data item inter-emission times, which however is generally unknown and may change over time. Lines (7) and (8) of Algorithm 3 address this challenge in an adaptive way. They exploit the following property of the output batching mechanism. Let us assume that output batches get flushed after an unknown number of n data items have been written to it. Given that $d_1, \ldots, d_n$ are the individual delays that the data items incur due to output batching, the equality

$$\frac{1}{n}\sum_{i=1}^{n}(\Delta + d_i) = \Delta + \underbrace{\frac{1}{n}\sum_{i=1}^{n} d_i}_{\approx obl_e} \tag{4.3}$$

holds. Adjusting the output batch lifetime by Δ, changes the delay of each data item from d_i to $d_i + \Delta$. The workload statistic obl_e gives us an estimation of $\frac{1}{n}\sum_{i=1}^{n} d_i$. Hence, from Equation 4.3 follows that adjusting $e.oblt$ by $\Delta = targetObl - obl_e$ can be expected to yield the desired mean output batch latency $targetObl$. Algorithm 3 thus finds the correct value of $e.oblt$ to achieve $obl_e \approx targetObl$ within two adjustment intervals. The first interval starts with the default $e.oblt = 0$ (immediate shipping) which results in $obl_e = 0$ being measured. After the first interval, the algorithm sets $e.oblt = targetObl$. After the second interval, the algorithm will com-

pute the correct Δ to achieve $obl_e \approx targetObl$. In subsequent intervals, $e.oblt$ only needs to be adjusted if the workload changes, e.g. the data item inter-emission time distribution or subtask latencies.

As mentioned in Section 4.5.2, some channels may not be able to batch data items up to a desired output batch deadline, e.g. because the underlying byte buffer of the batch is full before the deadline expires. In this case the workload statistic obl_e is insensitive to changes in $e.oblt$ – no matter how high we set $e.oblt$, we will never obtain $obl_e \approx targetObl$. This case is dealt with in line (8) of Algorithm 3 that limits $e.oblt$ to a sensible range.

In terms of runtime complexity, Algorithm 3 is a multi-pass algorithm. The channels of a runtime subgraph $G' = (V', E')$ are traversed twice to precompute upstream/downstream subtask latencies. Afterwards, the channels are traversed once more to set output batch lifetimes. The asymptotic runtime complexity is however $O(|E'|)$.

4.5.4 AOB Setup

Each AOB Manager enforces a subset of the runtime constraints in a runtime graph. This is a deliberate design decision to make the AOB technique itself horizontally scalable, as the work of tracking and enforcing runtime constraints can overwhelm any central node for large runtime graphs[2]. This section describes a setup algorithm to determine how many AOB Managers to instantiate and on which workers to place them.

Problem Definition

Let us assume a runtime graph $G = (V, E)$, a template constraint **c** and its set of induced runtime constraints $C = \{c_1, \ldots, c_n\}$. It is the goal of the AOB setup algorithm to split the runtime graph into m subgraphs $G_i = (V_i, E_i)$ with $i = 1, \ldots, m$ each of which is to be assigned to an AOB Manager/RGA pair. We shall say that G_i *contains* a runtime constraint $c = (S, \_, \_)$, if and only if $e \in E_i$ and $v_a, v_b \in V_i$ for each channel $e = (v_a, v_b)$ in S. Lastly, by $constr(G_i, C)$ we shall denote the set of those runtime constraints in C that G_i contains. The chosen subgraphs must fulfill the following conditions:

[2] An early version of the AOB technique published in [80] employed a single AOB Manager that ran on the master, which caused the master to be overwhelmed for applications with very high degrees of parallelism.

1. For each runtime constraint $c \in C$ there is *exactly* one subgraph G_i with $c \in constr(G_i, C)$, which implies

$$\bigcup_{1 \leq i \leq m} constr(G_i, C) = C.$$

2. A subgraph G_i does not contain any subtasks or channels that are not part of the constraints in $constr(G_i, C)$.

The subgraphs shall be optimized towards the following objectives:

1. The number m of subgraphs is maximized. This objective ensures that the amount of work to be done by each AOB Manager is minimized and thus reduces the impact on the application.

2. The number of common subtasks between subgraphs should be minimized:

$$\underset{G_1,\ldots,G_m}{\text{minimize}} \sum_{1 \leq i \leq m} \sum_{j \neq i} |V_i \cap V_j|$$

   If a subtask is part of more than one subgraph G_i, multiple AOB Managers require workload statistics for it and its channels. This objective reduces the amount of workload statistics that have to be shipped between workers.

For some runtime graphs, objectives (1) and (2) are contradictory. Since the network traffic caused by the Statistics Reporters can be assumed to be negligible, the AOB setup algorithm prioritizes objective (1).

Setup Algorithm

After the engine has scheduled all subtasks, the master computes the subgraphs $G_i = (V_i, E_i)$ and sends each one to a worker so that it can set up the AOB Manager/RGA pair. Algorithm 4 describes a heuristic algorithm to compute the subgraphs. It computes a set of AOB Manager *allocations* in the form of tuples (w_i, G_i), where w_i is the worker supposed to run an AOB Manager responsible for the runtime subgraph G_i. For each given template constraint, it invokes $GetAllocationsForSequence()$ that computes AOB Manager allocations for a single constraint. If allocations from different

constraints affect the same worker, the subgraphs of the two allocations are merged using *mergeGraphs*().

Algorithm 4 ComputeAOBSetup(**G**, **C**)

Require: Template graph **G** and a set of template constraints **C** = $\{\mathbf{c}_1, \ldots, \mathbf{c}_n\}$

```
 1: allocs ← ∅
 2: for all c_i = (S, _, _) in C do
 3:   for all (w_i, G_i) in GetAllocationsForSequence(G, S) do
 4:     if ∃(w_i, G_i*) ∈ allocs then
 5:       allocs ← (allocs − {(w_i, G_i*)}) ∪ {(w_i, mergeGraphs(G_i*, G_i))}
 6:     else
 7:       allocs ← allocs ∪ {(w_i, G_i)}
 8:     end if
 9:   end for
10: end for
11: return allocs
```

Algorithm 5 describes *GetAllocationsForSequence*() that computes AOB Manager allocations for the template sequence of a single template constraint. First, it invokes Algorithm 6 to determine an *anchor task* on the template sequence. An AOB Manager will be allocated to each worker executing a subtask of the anchor task. The function *PartitionByWorker*() partitions the subtasks of the anchor task into disjoint subsets V_i of subtasks that run on the same worker. With *RuntimeGraphExpand*() each such set V_i of subtasks is then expanded to the runtime subgraph that the AOB Manager on that worker shall be responsible for. *RuntimeGraphExpand*() can be implemented by traversing the runtime graph in forward and reverse direction in a depth-first manner, starting from the subtasks in V_i. Every constrained subtask and channel encountered during traversal is then added to the runtime subgraph.

Finally, Algorithm 6 describes *GetAnchorTask*() that employs a simple heuristic to pick an anchor task for a given template sequence. It starts with an initial candidate set that contains all tasks covered by the template sequence, which is determined by *tasksOf*() (see Section 4.2). The heuristic then reduces the candidate set in two steps. First, it retains only tasks whose subtasks are executed on the highest number of workers. Then it invokes *cntChannels*() to count the number of ingoing and outgoing channels each

Algorithm 5 GetAllocationsForSequence(**G**, G, **S**)

Require: Template graph **G**, runtime graph G and template sequence **S**
1: $\mathbf{v} \leftarrow GetAnchorTask(\mathbf{G}, \mathbf{S})$
2: $allocs \leftarrow \emptyset$
3: **for all** V^* **in** $PartitionByWorker(\mathbf{v})$ **do**
4: $\quad allocs \leftarrow allocs \cup \{(worker(V^*[0]), RuntimeGraphExpand(G, V^*, \mathbf{S}))\}$
5: **end for**
6: **return** $allocs$

candidate task has on the constrained sequence. It retains only the candidates with the lowest number of channels. Finally, one of the remaining candidate tasks is chosen as anchor task in a random but deterministic fashion. The rationale of this algorithm, is that the first reduction of the candidate set maximizes the number of AOB Managers, which is the primary optimization objective. The second reduction heuristically minimizes the overlap between subgraphs of different AOB Managers, which is the secondary optimization objective from Section 4.5.4. Choosing tasks with low numbers of channels yields smaller and possibly less overlapping runtime subgraphs during expansion by *RuntimeGraphExpand*() in Algorithm 4.

Algorithm 6 GetAnchorTask(**G**, **S**)

Require: Template graph **G** and a template sequence **S**
1: $ret \leftarrow tasksOf(\mathbf{S})$
2: $maxWorkers \leftarrow \max\{cntWorkers(\mathbf{v}) | \mathbf{v} \in ret\}$
3: $ret \leftarrow ret \setminus \{\mathbf{v} \in ret | cntWorkers(\mathbf{v}) < maxWorkers\}$
4: $minChannels \leftarrow \min\{cntChannels(\mathbf{v}, \mathbf{S}) | \mathbf{v} \in ret\}$
5: $ret \leftarrow ret \setminus \{\mathbf{v} \in ret | cntChannels(\mathbf{v}, \mathbf{S}) > minChannels\}$
6: **return** $ret[0]$

In summary, the AOB setup algorithm splits the latency constrained part of a runtime graph into a maximum number runtime subgraphs, whose overlap is heuristically minimized. This is achieved by choosing an anchor vertex and then building one runtime subgraph for each worker the anchor vertex has been scheduled on. If there are multiple constraints, the procedure is repeated once per constraint.

To determine the runtime complexity of the AOB setup, let us assume k template constraints and a schedule that executes the application on m

workers. The asymptotic worst case runtime complexity of Algorithm 4 is $O(k(|\mathbf{V}| + m(|V| + |E|)))$. The worst case occurs when the constraints cover the whole template graph and the subtasks of adjacent tasks are connected in an all-to-all manner. In this case the subgraphs of the AOB Managers mostly overlap, which results in a complexity of $O(|V| + |E|)$ per $RuntimeGraphExpand()$ invocation.

4.5.5 Summary

It is well known that the transmission latency and achievable throughput in commodity TCP/IP networks are linked via the amount of bytes that are sent per packet [93].

This section proposes a strategy to ship data items in batches that are as large as possible, while still meeting latency constraints. The hypothesis is that batching makes it possible to run stream processing applications on fewer cluster resources at a higher utilization (compared to immediate data item shipping), while still meeting application-defined latency constraints.

Moreover, this section has presented a setup algorithm that distributes the overhead of the AOB technique over as many AOB Managers as possible, in order to make the technique itself horizontally scalable. The setup algorithm splits the runtime graph into subgraphs and assigns each subgraph to an AOB Manager, heuristically optimizing for a maximum number of subgraphs with minimal overlap.

4.6 Elastic Scaling (ES)

So far, this chapter has discussed stream processing under the assumption of a static but sufficient cluster resource provisioning. The additional exploitation of resource elasticity is necessary to support application workloads, where a static, a-priori resource provisioning is impossible or prohibitively expensive. In the following, we will motivate the ES technique based on experimentation with the *PrimeTest* application, that showcases the challenges of stream processing under varying workload. Figure 4.8 shows the Application Graph and respective schedule. The *Source* subtasks shall produce data items at a step-wise varying rate, so that we can observe steady-state workload statis-

tics at each step. Please refer to Section 5.1.2 for a full description of the *PrimeTest* application.

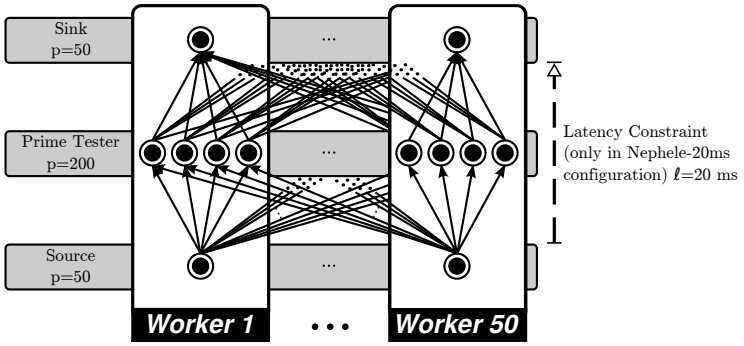


Figure 4.8: Application graph and schedule of the *PrimeTest* application.

Experimental Setup

The PrimeTest application was implemented for Apache Storm [13] and the Nephele SPE. Each run of the application was performed on the cluster testbed described in Appendix B, with a static resource provisioning of 50 cluster nodes (200 CPU cores). The application was run with the following four *configurations*:

- **Storm:** The application was run on Apache Storm[3] with default settings and deactivated fault tolerance functionality. Storm is optimized towards the *lowest latency possible*, because it ships produced data items immediately.
- **Nephele-IS:** The application was run on the Nephele SPE configured with immediate shipping (no output batching). This configuration is equivalent to the Storm configuration and ensures comparability between the Storm configuration and those based on the Nephele SPE, as the engines have different codebases.
- **Nephele-16KiB:** The application was run on the Nephele SPE modified to use a simple fixed output batching strategy. Output batches are implemented using 16KiB byte buffers on each channel. Each data item is serialized into the buffer, and each buffer is only shipped when full. This configuration is optimized towards *maximum throughput* by

[3]Version `v0.9.2-incubating`

sending data items in large batches. Larger buffer sizes than 16KiB had no measurable impact on throughput.

- **Nephele-20ms:** The application was run on the Nephele SPE extended with the AOB technique as described in Section 4.5. In this configuration, the PrimeTest application declares a latency constraint $(\mathbf{S}, 20ms, 5s)$, where the template sequence $\mathbf{S}$ is chosen as shown in Figure 4.8. This configuration trades off latency against maximum throughput, by batching as much as possible while attempting to guarantee the 20 ms constraint.

Discussion of Results

Figure 4.9 shows the results of running the PrimeTest application with the previously described configurations. It plots the performance metrics *latency* and *throughput* for each configuration (see description of Figure 4.9).

In the *Warm-Up* phase, all configurations can keep up with the attempted throughput. Storm and Nephele-IS have the lowest latency of 1-2ms, due to their use of the immediate shipping strategy. Nephele-20ms batches data items as necessary to guarantee the 20 ms latency constraint. Nephele-16KiB has a latency of almost 3s, because the 16KiB buffers on each channel take a long time to be filled.

During the first steps of the *Increment* phase, each configuration has a stepwise increasing but otherwise steady latency. This latency is caused by the input queues of *Prime Tester* subtasks, that grows with each step. At later steps the *Prime Testers* subtasks eventually turn into bottlenecks and their input queues loose steady-state and grow until full. At this point *backpressure* throttles the *Source* subtasks and each configuration settles at its individual throughput maximum. Backpressure is an effect that starts at overloaded subtasks and propagates backwards through the runtime graph via input queues and TCP connections.

When looking at the latency plot, Storm and Nephele-IS are the first configurations to loose steady-state queue waiting time at roughly 180 s, as evidenced by a sharp increase in latency. They are followed by Nephele-20ms at 300s and by Nephele-16KiB at 360s. Once bottlenecks have manifested, the latency of a configuration is mainly affected by how long the input queues can become. Since resources such as RAM are finite, both Nephele and Storm

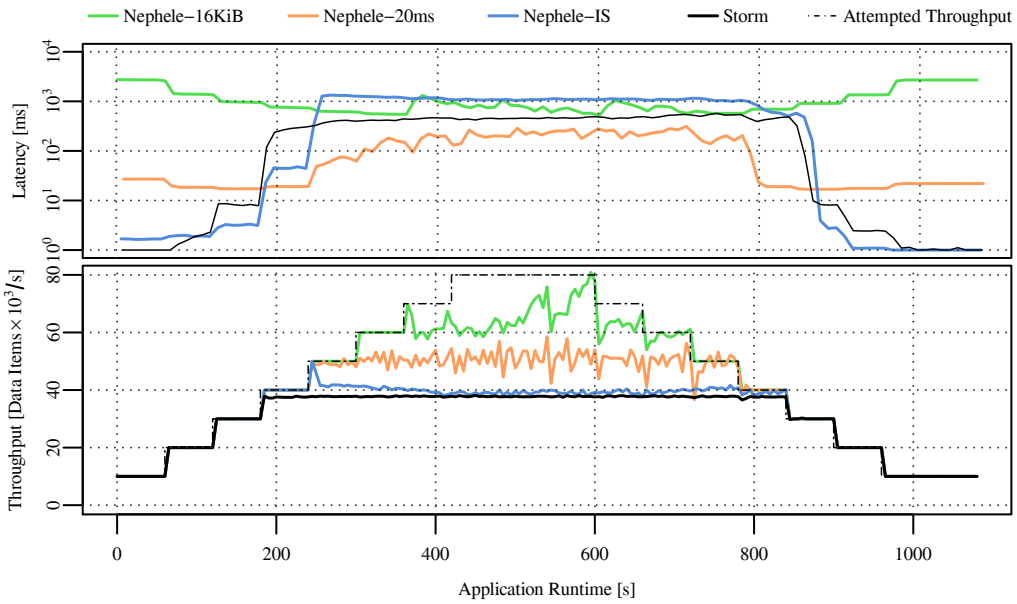


Figure 4.9: Experimental results of running the *PrimeTest* application. The plotted latency is the mean time that elapses between the emission of a data item at a *Source* subtask and its consumption at a *Sink* subtask. The plotted throughput is defined as the mean rate at which the *Source* subtasks emitted data items. In the throughput plot, solid lines indicate the *effectively achieved* throughput. Conversely, the dotted line indicates the *attempted* throughput as dictated by the current phase step.

limit the maximum input queue length that bounds maximum latency. It is noteworthy that a linear increase in throughput leads to a *super-linear* increase in input queue waiting time at the *Prime Tester* subtasks. This effect is present in every configuration, but *dominates* latency in configurations geared towards low latency. Moreover, it is important to note for Nephele-20ms, that at the 240s threshold, the AOB technique fails to enforce the constraint, because the steady-state queue waiting time has become dominant.

When looking at the throughput plot, one can see that the configurations with immediate shipping peak at ca. 40×10^3 data items per second. The configurations using batching show a higher throughput, i.e. Nephele-20ms peaks at 52×10^3 data items per second (a 30% improvement), and Nephele-16KiB peaks at 63×10^3 data items per second (a 58% improvement).

In conclusion, while the AOB technique has a measurable impact on maximum effective throughput, it can only guarantee latency constraints up to a certain point. This point is reached when either the steady-state input queues have become so long, that the resulting waiting times make the constraint impossible to enforce, or when queues have lost steady-state length and are growing due to bottlenecks. Thus, an engine that enforces latency constraints has to also address the issue of *queue waiting time*. Preventing bottlenecks is equally important, but rather mandatory than sufficient.

4.6.1 Overview

The remainder of this section describes the ES technique that adjusts the data parallelism of tasks with the goal of managing queue waiting time so that latency constraints are fulfilled. It complements the AOB and DTC techniques, because it ensures that a sufficient amount of cluster resources is provisioned at any time. It consists of two parts, each of which is covered in its own subsection:

1. Once per adjustment interval (see Section 4.3), the TGA computes a global summary per template constraint (see Section 4.4). The ES Manager on the master uses the global summaries to initialize a *latency model*. Its purpose is to estimate the amount of queueing latency of a UDF-heavy data flow, when the degrees of parallelism of the tasks in

the data flow are changed. The latency model and its assumptions are described in Section 4.6.2.

2. The ES Manager then initiates the *Elastic Scaler*, that leverages the latency model to decide how the parallelism of elastically scalable tasks shall be adjusted. As a result, the Elastic Scaler issues *scaling actions* to the scheduler of the engine. It is up to the scheduler to decide which specific subtasks to start and stop on which workers. The Elastic Scaler is covered in Section 4.6.3.

4.6.2 The Latency Model

The latency model is a core component of the ES Manager. Once initialized with a global summary, it can be used to estimate the effects of scaling actions on queue waiting time within a latency constrained template sequence. It is based on the following assumptions:

1. **Homogeneous Worker Nodes** Workers that execute subtasks of the same task must be sufficiently similar in their main performance properties, i.e. CPU core speed and NIC bandwidth. Worker nodes that are significantly slower than their peers will introduce *hot spot subtasks* that lag behind.

2. **Effective Stream Partitioning** The exploitation of *data-parallelism* requires a way of partitioning data streams, so that each data item is assigned to one or more partitions. A partitioning shall be *effective*, if it avoids *hot spot subtasks* that incur a significantly higher computational load than other subtasks of the same task. Per the definition of Application Graphs in Section 2.3, an upstream UDF instance decides which downstream UDF instance(s) a data item shall be delivered to. Hence, UDFs decide on the partitioning scheme to use, which is a natural fit since an effective stream partitioning strategy is often application-specific. As this is a load-balancing problem, existing solutions may be applicable. A random or round-robin partitioning often delivers good load-balancing properties. Some applications however require a stream partitioning with *grouping* behavior, where data items have a key of some sort and each downstream UDF processes only the data items

with certain keys. In [46] Gulisano et al. present a load-balancing approach for data-parallel, stream-relational operators such as grouped aggregates and equijoins. While not in the scope of the ES technique presented in this section, such approaches are orthogonal.

3. **Elastically Scalable Tasks** A task shall be *elastically scalable*, when changing the number of its subtasks does not impact the correctness of the results it produces. Data-parallel subtasks may depend on their stream partitioning scheme not only for load-balancing, but also to maintain the correctness their results. During a scaling action, the mapping between stream partitions and downstream subtasks has to be adaptable ad-hoc. For stateless UDFs, e.g. simple *mappers* where the input stream is partitioned in a round-robin fashion, this is not an issue. The input of stateful UDFs, such as windowed joins, is usually partitioned by a key and all data items with a particular key need to be sent to the same subtask. An approach for data-parallel, stream-relational operators can again be found in [46]. For stateful UDFs, ChronoStream [129] describes a transparent state migration protocol with low overhead. Again, these approaches are orthogonal to the ES technique, as they offer the underlying mechanisms to safely execute scaling actions with stateful tasks.

Let us assume a template graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, its induced runtime graph $G = (V, E)$, a template constraint $\mathbf{c} = (\mathbf{S}, \_, \_)$ and a global summary for constraint $\mathbf{c}$. Given that $\mathbf{v}_1, \ldots, \mathbf{v}_n$ are the constrained tasks in $\mathbf{S}$, the global summary shall contain

$$(\overline{A_{\mathbf{v}}}, \overline{S_{\mathbf{v}}}, c_{A_{\mathbf{v}}}, c_{S_{\mathbf{v}}}, w_{\mathbf{v}}, l_{\mathbf{v}}) \quad \forall \mathbf{v} \in \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$$

where

- $\overline{A_{\mathbf{v}}}$ is the mean inter-arrival time of data items at the input queues of the subtasks in $\mathbf{v}$,
- $\overline{S_{\mathbf{v}}}$ is the mean service time[4] of the subtasks in $\mathbf{v}$,

[4]Service time is a queueing theoretic term, which describes the mean time a subtask is

- $c_{A_\mathbf{v}}$ is the mean coefficient of variation of the data item inter-arrival time of the subtasks in $\mathbf{v}$,
- $c_{S_\mathbf{v}}$ is the mean coefficient of variation of the service time of the subtasks in $\mathbf{v}$,
- $w_\mathbf{v}$ is the mean waiting time of data items at the input queues of the subtasks in $\mathbf{v}$,
- and $l_\mathbf{v}$ is the mean subtask latency of the subtasks in $\mathbf{v}$[5].

All values in the global summary are computed from their respective workload statistics described in Appendix A.

Each subtask shall now be modeled as a so-called single-server queueing system. The assumptions of *homogeneous worker nodes* and *effective stream partitioning* minimize the differences between workload statistics of subtasks of the same task. Hence, we can apply available formulae from queueing theory to the values in the global summary. Specifically, each subtask shall be modeled as a GI/G/1 qeueing system, i.e. the probability distributions of data item inter-arrival and service times are unknown. For this case, Kingman's formula [65] approximates the queue waiting time of the *average* subtask in task $\mathbf{v}$ as

$$W_\mathbf{v}^K = \left(\frac{\overline{S_\mathbf{v}}\rho_\mathbf{v}}{1-\rho_\mathbf{v}}\right)\left(\frac{c_{A_\mathbf{v}}^2 + c_{S_\mathbf{v}}^2}{2}\right), \tag{4.4}$$

where $\rho_\mathbf{v} = \frac{\overline{S_\mathbf{v}}}{\overline{A_\mathbf{v}}}$ is the mean *utilization* of the subtasks in $\mathbf{v}$. The *actual* queue waiting time $w_\mathbf{v}$ of the average subtask in $\mathbf{v}$ has been obtained from the global summary. Given that

$$w_\mathbf{v} = \underbrace{\frac{w_\mathbf{v}}{W_\mathbf{v}^K}}_{:=e_\mathbf{v}} W_\mathbf{v}^K \tag{4.5}$$

$$= e_\mathbf{v} W_\mathbf{v}^K, \tag{4.6}$$

busy with a data item it has read from its input queue.

[5]This value is not required by the latency model, but by the Elastic Scaler later in this section.

we can use the newly introduced coefficient $e_{\mathbf{v}}$ to "fit" Kingman's approximation to the measured workload statistics. The core idea of the latency model is to predict $w_{\mathbf{v}}$ when $\mathbf{v}$'s degree of parallelism changes. In this context, $e_{\mathbf{v}}$ ensures that we at least obtain the currently measured queue waiting time for the current degree of parallelism. Without $e_{\mathbf{v}}$ the model might lead to a scale-down, when a scale-up would actually be necessary (or vice versa).

From the assumption of an effective stream partitioning, it follows that changing the parallelism of a task, anti-proportionally changes the average data item arrival rate at its subtasks. Hence the utilization $\rho_{\mathbf{v}}$ becomes a function

$$\rho_{\mathbf{v}}(p^*) = \frac{\overline{S_{\mathbf{v}}} p(\mathbf{v})}{\overline{A_{\mathbf{v}}} p^*}$$

where $p^* \in \mathbb{N}^{>0}$ is a new degree of parallelism for $\mathbf{v}$ and $p(\mathbf{v})$ is its current degree of parallelism for which we have measurements in the global summary. Since the utilization $\rho_{\mathbf{v}}$ is part of the Kingman formula in Equation 4.4, we can estimate the queue waiting time as a function of p^*:

$$\begin{aligned} W_{\mathbf{v}}(p^*) &:= e_{\mathbf{v}} \left(\frac{\overline{S_{\mathbf{v}}} \rho_{\mathbf{v}}(p^*)}{1 - \rho_{\mathbf{v}}(p^*)} \right) \left(\frac{c_{A_{\mathbf{v}}}^2 + c_{S_{\mathbf{v}}}^2}{2} \right) \\ &:= e_{\mathbf{v}} \left(\frac{\overline{S_{\mathbf{v}}}^2 p(\mathbf{v})}{\overline{A_{\mathbf{v}}} p^* - \overline{S_{\mathbf{v}}} p(\mathbf{v})} \right) \left(\frac{c_{A_{\mathbf{v}}}^2 + c_{S_{\mathbf{v}}}^2}{2} \right) \end{aligned}$$

Without loss of generality we shall assume all constrained tasks $\mathbf{v}_1, \ldots, \mathbf{v}_n$ in $\mathbf{S}$ to be elastically scalable. By $p_i^{min} \leq p_i^* \leq p_i^{max}$, $i = 1, \ldots, n$ we shall denote their respective new degrees of parallelism to be chosen within user specified limits p_i^{min} and p_i^{max}. The total estimated queue waiting time in $\mathbf{S}$ can therefore be modeled as

$$W_{\mathbf{S}}(p_1^*, \ldots, p_n^*) = \sum_{i=1}^{n} W_{\mathbf{v}_i}(p_i^*). \tag{4.7}$$

We can now use $W_{\mathbf{S}}$ as a rough predictor of queue waiting time in a latency constrained template sequence $\mathbf{S}$, when varying the degrees of parallelism of

the tasks within.

Note, that this makes the reasonable assumption that the service time $\overline{S_{\mathbf{v}}}$ and its coefficient of variation $c_{S_{\mathbf{v}}}$ are unaffected by the change in parallelism. The same assumption is made for $c_{A_{\mathbf{v}}}$. One might argue that by changing the data item inter-arrival time $\overline{A_{\mathbf{v}}}$, its coefficient of variation $c_{A_{\mathbf{v}}}$ will also change. Since the GI/G/1 queueing formula itself is an approximation, I have chosen to neglect this effect and accept it as a source of prediction error.

4.6.3 The Elastic Scaler

After initialization of the latency model, the ES Manager triggers the Elastic Scaler that decides how much the tasks of a stream processing application shall be scaled. The Elastic Scaler consists of a set of algorithms that leverage the latency model to determine such scaling actions.

In a bottom up fashion we will first take a look at the *Rebalance* and *Resolve-Bottlenecks* algorithms. The top-level algorithm *ScaleReactively* finalizes the description of the Elastic Scaler, by showing when and how to employ *Rebalance* and *ResolveBottlenecks*.

The *Rebalance* Algorithm

Based on an initialized latency model, the *Rebalance* algorithm shown in Algorithm 7 chooses new degrees of parallelism for the tasks of a latency-constrained template sequence. The goal is to minimize parallelism, while satisfying the constraint. *Rebalance* is applicable if we have a global summary for $\mathbf{S}$ and no bottlenecks exist, i.e. the utilization $\rho_{\mathbf{v}}$ of each constrained task $\mathbf{v}$ is sufficiently smaller than 1. See the *ResolveBottlenecks* algorithm for a discussion of bottlenecks and their removal.

For a *single* template constraint $(\mathbf{S}, \ell, \_)$ with constrained tasks $\mathbf{v}_1, \ldots, \mathbf{v}_n$ in $\mathbf{S}$, the goal of *Rebalance* can be formulated as an optimization problem, with a linear objective function and several side conditions. We must choose new degrees of parallelism $p_1^*, \ldots, p_n^* \in \mathbb{N}$ so that the *total parallelism*

$$F(p_1^*, \ldots, p_n^*) = \sum_{i=1}^{n} p_i^* \tag{4.8}$$

is minimized and the side conditions

$$W_{\mathbf{S}}(p_1^*, \ldots, p_n^*) \leq (1 - w_{ob}) \left(\ell - \sum_{i=1}^{n} l_{\mathbf{v}_i} \right) \tag{4.9}$$

$$p_i^* \leq p_i^{max}, \forall i = 1, \ldots, n \tag{4.10}$$

$$p_i^* \geq p_i^{min}, \forall i = 1, \ldots, n \tag{4.11}$$

hold, where the *output batching weight* $\leq w_{ob} \leq 1$ is a configuration constant, that determines the fraction of available channel latency that should be spent on output batching. The output batching weight has been previously introduced as part of the AOB technique in Section 4.5.3. Figure 4.10 shows an example of the above optimization problem. While the example has multiple optimal solutions, note the problem may also not be solvable due to the side conditions.

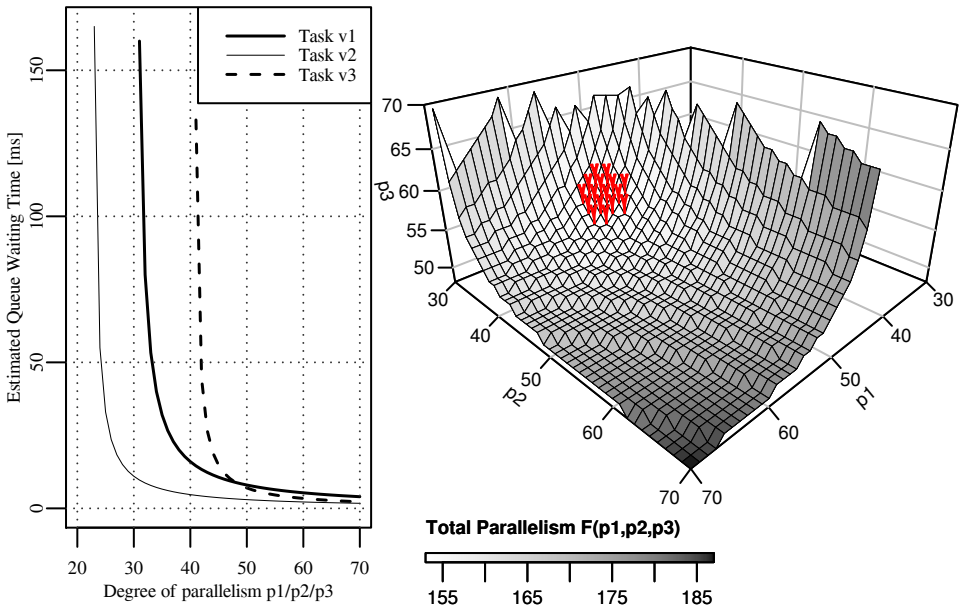


Figure 4.10: Example of the *Rebalance* optimization problem with three exemplary tasks $\mathbf{v}_2, \mathbf{v}_2, \mathbf{v}_3$ with degrees of parallelism p_1, p_2, p_3. The left plot shows a hypothetical $W_{\mathbf{v}_i}(p_i)$ for each of the exemplary tasks. For the right plot, let us assume that the tasks are in a latency-constrained sequence $\mathbf{S} = (\mathbf{v}_2, \_, \mathbf{v}_2, \_, \mathbf{v}_3)$, so that Inequation (4.9) is $W_{\mathbf{S}}(p_1, p_2, p_3) \leq 15\ ms$. The right plot shows a surface consisting of points (p_1, p_2, p_3) where Inequation (4.9) holds and p_3 is minimal for given values of p_1 and p_2. Hence all points below the surface do not fulfill Inequation (4.9), while all points above the surface do not minimize the total parallelism $F(p_1, p_2, p_3)$. The lighter the coloration of the surface, the lower the total parallelism. Optimal solutions in the example have $F(p_1, p_2, p_3) = 153$ and are marked in red. Note that *multiple optimal solutions* exist.

The *Rebalance* algorithm in Algorithm 7 solves the above optimization problem. It is given a template constraint $\mathbf{c}$ and assumes that the values from the global summary are available. Additionally, it is handed a set P_{min}, that contains a minimum required parallelism for each constrained task. Since each invocation of *Rebalance* only deals with a single constraint, the parameter P_{min} ensures that the chosen degrees of parallelism of a $Rebalance(\mathbf{c}_1, ...)$ invocation, are not overwritten with lower degrees of parallelism by another $Rebalance(\mathbf{c}_2, ...)$ invocation. See the top-level *ScaleReactively* algorithm for further details.

Algorithm 7 Rebalance($\mathbf{c}, P_{min}$)

Require: A template constraint $\mathbf{c} = (\mathbf{S}, \ell, \_)$ where sequence $\mathbf{S}$ has tasks $\mathbf{v}_1, \ldots, \mathbf{v}_n$, and a set P_{min} with minimum degrees of parallelism.

1: $\hat{W} \leftarrow (1 - w_{ob})\,(\ell - \sum_{i=1}^{n} l_{\mathbf{v}_i})$
2: $p \leftarrow \text{Array}[p_1^{max}, \ldots, p_n^{max}]$
3: **if** $W_{\mathbf{S}}(p[1], \ldots, p[n]) \leq \hat{W}$ **then**
4: $\quad p[i] \leftarrow p$ for $\forall (\mathbf{v}_i, p) \in P_{min}$
5: $\quad$ **while** $W_{\mathbf{S}}(p[1], \ldots, p[n]) > \hat{W}$ **do**
6: $\quad\quad C = \{i | i = 1, \ldots, n \text{ where } p[i] < p_i^{max}\}$
7: $\quad\quad \Delta_i \leftarrow W_{\mathbf{v}_i}(p[i] + 1) - W_{\mathbf{v}_i}(p[i])$ for $\forall i \in C$
8: $\quad\quad \Delta_{min} \leftarrow \min\{\Delta_i | i \in C\}$
9: $\quad\quad idx \leftarrow \min\{i \in C | \Delta_i = \Delta_{min}\}$
10: $\quad\quad p[idx] \leftarrow p[idx] + 1$
11: $\quad$ **end while**
12: **end if**
13: **return** $\{(\mathbf{v}_i, p[i]) | i = 1, \ldots, n\}$

Lines (2) and (3) in Algorithm 7 check whether it is possible to solve the optimization problem, by testing if Inequation (4.9) can be fulfilled at maximum parallelism. If so, it searches for a solution in a greedy manner, starting with the minimum degrees of parallelism given by P_{min}. The general idea of the while-loop is to increment the parallelism of the task that yields the highest decrease in queue waiting time, i.e. where the increment in parallelism is most effective. More formally, a task $\mathbf{v}_{idx}$ is chosen, so that incrementing its parallelism $p[idx]$ yields the highest decrease Δ_{idx} in queue waiting time.

As previously described, the goal of the *Rebalance* algorithm is to find an *optimal solution* $p_1^*, \ldots, p_n^*$, that minimizes the total parallelism $F(p_1^*, \ldots, p_n^*)$ under certain side conditions. *Rebalance* achieves this goal, if the optimization problem is solvable and $P_{min} = \{(\mathbf{v}_1, p_1^{min}), \ldots, (\mathbf{v}_n, p_n^{min})\}$, i.e. P_{min} contains the user-specified minimum parallelism for each task. In this case, line (4) begins the search for an optimal solution with $p[i] = p_i^{min}$ for $i = 1, \ldots, n$. With each iteration of the loop, the parallelism of a task is incremented that delivers the current maximum possible decrease of the total queue waiting time $W_{\mathbf{S}}$. The monotonously decreasing nature of $W_{\mathbf{S}}$ and the $W_{\mathbf{v}_i}$ functions, renders the achievable decrease of $W_{\mathbf{S}}$ smaller with each iteration. Since each iteration chooses the task with the maximum achievable decrease of $W_{\mathbf{S}}$, we obtain an optimal solution at the last

iteration. The presence of multiple optimal solutions is due to the discrete nature of the p_i^* values, which may result in the existence of an alternative optimal solution $\hat{p}_1, \ldots, \hat{p}_n$ with $F(\hat{p}_1, \ldots, \hat{p}_n) = F(p_1^*, \ldots, p_n^*)$ and $\hat{W} \geq W_{\mathbf{S}}(\hat{p}_1, \ldots, \hat{p}_n) \geq W_{\mathbf{S}}(p_1^*, \ldots, p_n^*)$. If P_{min} does *not* contain the user-specified minimum for each task, *Rebalance* degrades to a heuristic.

Implementations of *Rebalance* can exploit the fact that all but one value of C and Δ_i can be reused between iterations of the while-loop. This enables implementations to use standard data structures such as sorted sets, that offer *add*, *remove* and *peek* operations with $log(n)$ runtime complexity. Hence, *Rebalance* can be implemented with a worst-case runtime complexity of $O(n \log(n) m)$, where n is the number of tasks in $\mathbf{S}$ and m is their highest degree of parallelism.

The *ResolveBottlenecks* Algorithm

The previously presented *Rebalance* algorithm is only applicable if no bottlenecks exist. Conversely, the *ResolveBottlenecks* algorithm is applicable when we have a fresh global summary and at least one bottleneck task exists. Its goal is to resolve the bottleneck by increasing the parallelism of the respective task, so that *Rebalance* becomes applicable again at a later feedback cycle. A task $\mathbf{v}$ shall be a bottleneck if it has a utilization $\rho_{\mathbf{v}} \geq \rho_{max}$, where ρ_{max} is a value close to 1. For each such task we shall choose a new degree of parallelism

$$p^* = \min\{p^{max}, 2p(\mathbf{v}) \times \max\{1, \rho_{\mathbf{v}}\}\}. \tag{4.12}$$

The new degree of parallelism chosen by Equation 4.12 can be expected to have a new utilization lower or equal to 50%. *ResolveBottlenecks* returns a set, which contains a tuple $(\mathbf{v}, p^*)$ for each bottleneck task $\mathbf{v}$ it has found.

In general, *ResolveBottlenecks* is supposed to be a last resort. A bottleneck task causes input queue growth, that eventually leads to backpressure, which subsequently throttles upstream tasks. This degrades the usefulness of the collected workload statistics due to several anomalies. First, while intermediate queues are not yet full, an upstream task can emit at a rate higher than the service rate of the bottleneck task. This leads to a utilization higher than 100% at the bottleneck task, rendering the queueing formulae of the latency

model unusable (see Equation 4.4). Second, once intermediate queues are full, backpressure sets in and forces upstream tasks to wait before sending. This manifests itself in an artificial increase of service time and subtask latency at upstream tasks. In both cases, *Rebalance* is either not applicable or will exhibit erratic scaling behavior, as accurate values in the global summary are not available. Due to this, *ResolveBottlenecks* will at least double the parallelism of the bottleneck task, hopefully alleviating the bottleneck.

Putting it all together: *ScaleReactively*

ScaleReactively is the top-level algorithm, that decides when to apply *Rebalance* and *ResolveBottlenecks*. Therefore, it encapsulates the core logic to enforce latency constraints at a minimal degree of parallelism.

Algorithm 8 describes *ScaleReactively* in pseudo-code. Given a set of template constraints, it iterates over the constraints in a random but deterministic order. For each constraint, line (3) tests whether the template sequence of the constraint contains a bottleneck task. If necessary, line (4) invokes the *ResolveBottlenecks* algorithm to alleviate bottlenecks. If no bottlenecks were found, the *Rebalance* algorithm is invoked. Line (6) ensures that the set P_{min} given to *Rebalance* contains all tasks of the current template sequence $\mathbf{S}$, and that the user-specified minimum parallelism as well as the scaling decisions of previous *Rebalance* invocations are respected. After all constraints have been considered, the scaling actions resulting from the new degrees of parallelism are triggered with *doScale*().

For applications with a single constraint or multiple non-overlapping constraints, *Rebalance* can find the minimum degrees of parallelism. In case of template constraints that have a task in common, *Rebalance* degrades to a heuristic, because P_{min} will not contain the user-specified minimum for each task. Template constraints that have streams in common are forbidden as mentioned in Section 4.2.

The asymptotic worst-case runtime complexity of *ScaleReactively* is dominated by the invocations of *Rebalance*. It becomes $O(|\mathbf{C}|n\log(n)m)$ when *Rebalance* is invoked for each constraint, where n is the highest number of tasks among the constraints in $\mathbf{C}$, and m is the highest possible degree of parallelism among the tasks constrained by $\mathbf{C}$.

Algorithm 8 ScaleReactively($\mathbf{C}$)

Require: A set of template constraints $\mathbf{C}$

1: $P \leftarrow EmptyMap()$
2: **for all** $\mathbf{c} = (\mathbf{S}, \ell, \_) \in \mathbf{C}$ **do**
3: **if** $hasBottleneck(\mathbf{S})$ **then**
4: $P.\mathbf{v} \leftarrow \max\{P.\mathbf{v}, p^*\}$ for $\forall(\mathbf{v}, p^*) \in ResolveBottlenecks(\mathbf{S})$
5: **else**
6: $P_{min} \leftarrow \{(\mathbf{v}, \max\{p_{\mathbf{v}}^{min}, P.\mathbf{v}\}) | \mathbf{v} \in tasksOf(\mathbf{S})\}$
7: $P.\mathbf{v} \leftarrow p^*$ for $\forall(\mathbf{v}, p^*) \in Rebalance(\mathbf{c}, P_{min})$
8: **end if**
9: **end for**
10: $doScale(P)$

4.6.4 Implementation Considerations

The ES technique as presented above has been implemented inside the Nephele SPE. Like any implementation of a complex strategy in a dynamic system, it needs to cope with some technical corner cases when scaling up and down.

Nephele's scheduler can start new subtasks in parallel, hence large scale-ups can be executed quickly, even if this requires the allocation of several new workers from the cluster manager. However, the desired *effects* of scale-ups are not instantly visible in workload statistics, especially after load bursts have already caused long queues. Also, starting new tasks may initially worsen channel latency, because new TCP/IP connections need to be established. To give the application time to work off the backlog in queues and stabilize workload statistics, the Nephele implementation of the ES technique remains inactive for three adjustment intervals after scale-ups have been completed.

Conversely, Nephele's scheduler can terminate subtasks only one-by-one, because intermediate queues need to be drained and the termination of each subtask needs to be signaled to and confirmed by upstream as well as downstream subtasks. However, no inactivity phase is required afterwards, because the effects of scale-downs have proven to be almost instantly visible in workload statistics.

ES technique trades off *utilization* against *queue waiting time*, due to its queueing theoretic nature. Latency constraints with a lot of "slack" time

hence allow for very long queues, which implies very high utilization. Unfortunately, a utilization close to 100% makes the application however susceptible to bottlenecks caused by minor fluctations in load and system dynamics. The implementation of the ES technique deals with this issue, by setting the minimum parallelism p^{min} of each task to either (a) the user-specified minimum, or (b) the degree of parallelism that yields a utilization of 90%, whichever value is higher. While this utilization threshold is configurable, choosing it as 90% gives satisfactory results.

4.6.5 Summary

Both the AOB and DTC techniques depend on sufficiently provisioned cluster resources. In the context of enforcing latency constraints, a resource provisioning is *sufficient* only if it prevents long input queues at subtasks, that lead to constraint violations. It is a well known result from queueing theory, that the waiting time at input queues depends on the queueing theoretic notion of *utilization*, which is the ratio of service time to data item inter-arrival time.

This section has proposed the ES technique that employs elastic scaling as a mean to manage utilization in a way that prevents long input queues from violating the latency constraints declared by a stream processing application. The ES technique therefore provides a *scaling policy* that triggers scaling actions to be executed by the scheduler of an engine. The ES technique is built around a queuing theoretic latency model to estimate the effects of scaling actions on queue waiting time. At the end of each adjustment interval, the latency model is initialized from summarized workload statistics. At this point, an optimization algorithm (*Rebalance*) uses the latency model to find the minimal degree of parallelism for each constrained task, so that latency constraints are still fulfilled. Although the queueing formulae of the latency model are meant for steady state systems, the hypothesis is that the repeated execution of *Rebalance* (once per adjustment interval) makes the overall approach reasonably adaptive to changes in application workload. Despite that, it is still possible that bottleneck tasks occur due to abrupt changes in application workload. As the resulting *backpressure* negatively impacts the correctness of workload statistics, the *Rebalance* algorithm cannot be applied. In this case the ES technique employs a simple fallback procedure to resolve bottlenecks.

4.7 Dynamic Task Chaining (DTC)

State of the art stream processing engines as described in Section 3.3 typically map different subtasks to different threads and operating system processes. While this facilitates natural scalability and load balancing between CPUs and CPU cores, it also raises the communication overhead between subtasks. In the most lightweight case, where subtasks are mapped to different threads within the same process and communication is performed via shared memory, the overhead consists of data item (de)serialization as well as the synchronization and and scheduling of threads. Depending on the computational complexity of the subtasks, the communication overhead can account for a significant fraction of the overall processing time.

A common approach to address this form of communication overhead is to chain (also: fuse, group) communicating subtasks by executing them in a single thread. This approach sacrifices pipeline parallelism to reduce communication overhead. Examples can be found in numerous engines, such as IBM's System S [40] or COLA [64], an optimizer for System S stream processing applications. A popular example in the area of large scale parallel data processing are also the chained mapper functions in Hadoop MapReduce.

With regard to stream processing, chaining is an interesting approach to reduce overhead and processing latency at the same time. Typically, chains are constructed *before* the application executes, so once it is running each chain is statically bound to a single thread. In applications with steady load this might be beneficial since communication overhead is decreased and latency constraints can be met more easily. However, when the workload varies over the lifetime of the application, static chaining prevents the underlying operating system from distributing the application workload over several CPU cores. As a result, task chaining can also be disadvantageous if (a) the complexity of the chained subtasks is unknown in advance or (b) the workload the application has to handle is unknown or changes over time. This section describes an approach for *dynamic* chaining, where chains are established and dissolved at runtime.

4.7.1 Overview

The Dynamic Task Chaining (DTC) technique reduces the latency of a stream processing application by dynamically changing the binding of subtasks to execution threads. It consists of the following parts:

1. A *chaining mechanism*, which is implemented on workers and provides the underlying functionality to chain and unchain subtasks. This part is described in Section 4.7.2.
2. Once per adjustment interval (see Section 4.3), the TGA on the master computes global summaries (see Section 4.4). The *DTC Manager* uses the summaries to compute how the current chaining setup should be adjusted and informs workers of the new chaining setup. After introducing formal prerequisites in Section 4.7.3, an algorithm for dynamic chaining is described in Section 4.7.4.

4.7.2 The Chaining Mechanism

By default, workers execute each subtask in its own thread of execution. Each thread has an input queue that conceptually serves two purposes. First, it provides the desired FIFO behavior of channels, i.e. it ensures that the data items of each channel are consumed in the same order they were sent in. Second, it enables thread-safe asynchronous message passing between subtasks. The chaining mechanism replaces the queue between two adjacent subtasks with a synchronous function call, which eliminates the cost of asynchronous message passing, while maintaining FIFO behavior.

(a) Unchained execution (b) Chained execution

Figure 4.11: Chained and unchained execution of subtasks.

Figure 4.11 illustrates the difference between chained and unchained subtasks for a worker running two pipeline-parallel subtasks. At runtime, workers shall not only support chaining communicating subtasks, but also *merging*

and *splitting* of chains. Adjacent chains on the same worker can be *merged*, which requires the worker to take care of the input queue between them. Simply dropping the content of the queue may not be acceptable, depending on the nature of the application. Hence, the worker shall temporarily halt the execution of the first chain and wait until the input queue of the second chain has been drained. The thread of the second chain and its queue can now be removed without loss of application data. This will temporarily increase latency in this part of the runtime graph due to a growing input queue at the first chain, that needs to be worked off after the chains have been merged. Splitting a chain also requires the worker to briefly halt the respective thread in order to move the subtasks after the split point to their own thread. This is however a quick operation with negligible impact on latency.

Note that data items are passed by reference between the subtasks of a chain. In order to make this safe and fully transparent to subtasks, we shall assume data items to be *immutable*, i.e they cannot be altered after creation.

4.7.3 Formal Prerequisites

For the following definitions let us assume a stream processing application with a *template graph* $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, induced runtime graph $G = (V, E)$ and a set of template constraints $\mathbf{C}$. By v_{ij} we shall denote the j-th subtask of task $\mathbf{v}_i$, whereas by $w(v_{ij})$ we shall denote the worker that v_{ij} has been scheduled on.

Definition: Chain
A *chain* shall be a tuple $(\mathbf{v}_1, \ldots, \mathbf{v}_n) \in \mathbf{V}^n$ with $n \geq 1$ where

- $\mathbf{v}_i \in \bigcup_{(\mathbf{S},\_,\_)\in\mathbf{C}} tasksOf(\mathbf{S})$ for $1 \leq i \leq n$,
- $(\mathbf{v}_i, \mathbf{v}_{i+1}) \in \mathbf{E}$ for $1 \leq i < n$,
- $\nexists(\mathbf{v}_i, \mathbf{v}) \in \mathbf{E}$ with $\mathbf{v} \neq \mathbf{v}_{i+1}$ for $1 \leq i < n$,
- $\nexists(\mathbf{v}, \mathbf{v}_i) \in \mathbf{E}$ with $\mathbf{v} \neq \mathbf{v}_{i-1}$ for $1 < i \leq n$,
- $p(\mathbf{v}_i) = p(\mathbf{v}_j)$ for $1 \leq i, j \leq n$,
- and for all $1 \leq i < n$ and $1 \leq j \leq p(\mathbf{v}_i)$
  - $|\{(v_{ij}, v) | (v_{ij}, v) \in E\}| = 1$
  - and $\exists (v_{ij}, v) \in E$ with $w(v_{ij}) = w(v)$.

Less formally, a chain is a latency constrained path in the template graph, where the inner tasks only have one ingoing and outgoing stream. All tasks on the path have the same degree of parallelism, their subtasks are connected by channels in a one-to-one pattern and run on the same worker.

Figure 4.12 shows an example of an application with multiple chains. Note that for each constrained task, multiple chains may exist, depending on the graph structure, constraints and decisions made by the scheduler. Moreover, there is at least one chain per constrained task, since a chain can by definition consist of only one task.

Definition: Primary Chain
The *primary chain* $pc(\mathbf{v})$ of a latency constrained task $\mathbf{v}$ is the *longest possible* chain the task is a part of.

Note that we can uniquely determine the primary chain for each constrained task. This property follows directly from the definition of chains. Consider a task $\mathbf{v}$ with two chains of equal length, e.g. $(\mathbf{v}_0, \mathbf{v})$ and $(\mathbf{v}, \mathbf{v}_1)$. In this case, $(\mathbf{v_0}, \mathbf{v}, \mathbf{v}_1)$ is also a chain for $\mathbf{v}$ and is longer. Unless $\mathbf{v}$ is part of another chain with length three, $(\mathbf{v_0}, \mathbf{v}, \mathbf{v}_1)$ is its primary chain.

Definition: Primary Chain Partitioning
A *primary chain partitioning* for a constrained task $\mathbf{v}$ is a tuple $(\mathbf{cn}_1, \ldots, \mathbf{cn}_n)$ where

- $\mathbf{cn}_1, \ldots, \mathbf{cn}_n$ are chains that do not have any tasks in common,
- and $\mathbf{cn}_1 \bowtie \mathbf{cn}_2 \bowtie \ldots \bowtie \mathbf{cn}_n = pc(\mathbf{v})$, where $\bowtie$ merges two chains into a new chain.

Less formally, a primary chain partitioning splits the primary chain of a task into smaller chains. The longer the primary chain, the larger the space of possible primary chain partitionings. The dynamic chaining algorithm presented in the next section searches for a partitioning that best matches certain optimization criteria.

4.7.4 Dynamic Chaining

The dynamic chaining algorithm presented in this section decides when and which constrained tasks of a stream processing application to chain. Us-

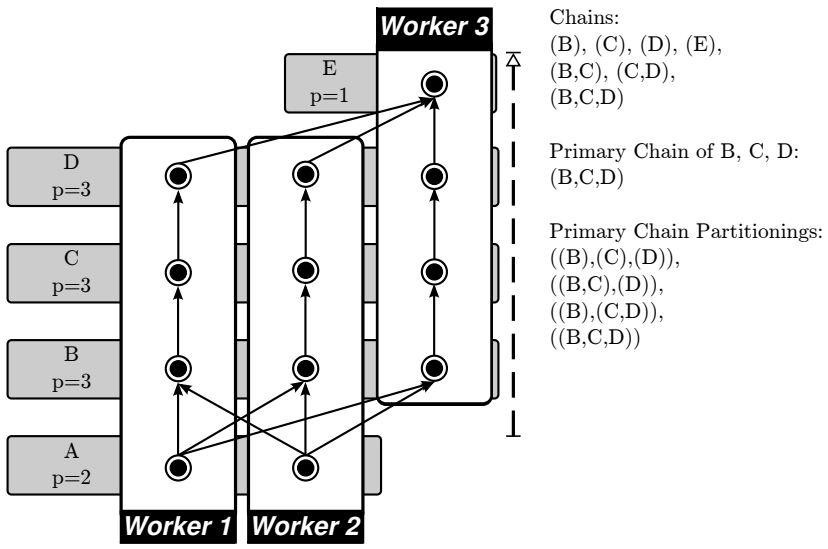


Figure 4.12: Example of chains, primary chains and primary chain partitionings.

ing the terminology of the previous section, the algorithm selects a suitable primary chain partitioning for each primary chain in the template graph.

Problem Definition

The dynamic chaining algorithm shall optimize each primary chain partitioning towards the following objectives:

1. Minimize the number of chains in each primary chain partitioning without introducing bottlenecks.

2. Minimize the difference in *utilization* between the chains in each primary chain partitioning.

The first objective ensures that chains are as long as possible without introducing bottlenecks, which minimizes the number of intermediate queues. The second objective ensures that the application workload is distributed as evenly as possible over the chains in each primary chain partitioning. This objective is motivated by the known behavior of queueing systems, where the queue waiting time rises superlinearly with utilization. For example, it is generally better to have two chains with medium utilization than to have one

with low and one with high utilization. In some situations, the two objectives may be contradictory, hence the first objective shall be the primary goal.

Dynamic Chaining Algorithm

Once per adjustment interval the DTC Manager on the master invokes Algorithm 9 to compute the primary chain partitionings for a given template graph and constraint set. First, $findPrimaryChains()$ computes the set of all primary chains within the template graph. Primary chains can be computed by enumerating the tasks of constrained template sequences and testing for the chain properties described in Section 4.7.3. Subsequently, Algorithm 10 is invoked in order to determine a primary chain partitioning for each primary chain. The function $establishChains()$ issues requests to workers to chain or unchain subtasks and merge or split existing chains, so that the computed primary chain partitioning is established. The required primitives for merging and splitting have been previously described in Section 4.7.2.

Algorithm 9 chainTasks(**G**, G, **C**)

Require: A template graph **G** with induced runtime graph G and set of template constraints **C**.

1: **for all** **pc** $\in findPrimaryChains(\mathbf{G}, G, \mathbf{C})$ **do**
2: $\quad establishChains(computePrimaryChainPartitioning(\mathbf{pc}))$
3: **end for**

Algorithm 10 computePrimaryChainPartitioning(**pc**)

Require: A primary chain **pc** $= (\mathbf{cn}_1, \ldots, \mathbf{cn}_n)$

1: $stdPcp \leftarrow ((\mathbf{cn}_1), \ldots, (\mathbf{cn}_n))$
2: **for all** $i = 1, \ldots, n-1$ **do**
3: $\quad$ **for all** $pcp \in enumeratePartitionings(\mathbf{pc}, i)$ **do**
4: $\quad\quad$ **if** $maxUtil(pcp) \leq \rho_{max}$ and $utilVar(pcp) < utilVar(stdPcp)$ **then**
5: $\quad\quad\quad$ **return** pcp
6: $\quad\quad$ **end if**
7: $\quad$ **end for**
8: **end for**
9: **return** $stdPcp$

For a given primary chain, Algorithm 10 finds a primary chain partitioning that is heuristically optimized towards the objectives stated in the prob-

lem definition at the beginning of this section. The algorithm relies on $enumeratePartitionings(\mathbf{pc}, i)$ to enumerate all theoretically possible primary chain partitionings with i chains for a given primary chain $\mathbf{pc}$. The pseudo-code of this function is omitted for brevity. A simple way of enumerating all possible partitionings is as follows. To partition a primary chain of length n into i subchains, we need to choose $i-1$ distinct split indices from $n-1$ available split indices. For example, if $n=5$ and $i=3$, then the split indices can be enumerated as tuples $(1,2)$, $(1,3)$, $(1,4)$, $(2,3)$, $(2,4)$, $(3,4)$. Line (4) in Algorithm 10 checks whether the current partitioning is (i) safe with regard to bottlenecks and (ii) has a utilization variance better than that of *stdPcp*. To this end the *utilization of a chain* $\mathbf{cn} = (\mathbf{v}_1, \ldots, \mathbf{v}_k)$ shall be defined as

$$util(\mathbf{cn}) = \sum_{i=1}^{k} \rho_{\mathbf{v}_i}^{max}, \tag{4.13}$$

where $\rho_{\mathbf{v}_i}^{max}$ is the *maximum* utilization among the subtasks of $\mathbf{v}_i$ and expected to be part of the global summaries. Please see Appendix A for a full description of the related workload statistic, from which $\rho_{\mathbf{v}_i}^{max}$ is computed. Now, we can define the *maximum utilization* of a primary chain partitioning $pcp = (\mathbf{cn}_1, \ldots, \mathbf{cn}_p)$ as

$$maxUtil(pcp) = \max\{util(\mathbf{cn}_i) | i = 1, \ldots, p\}. \tag{4.14}$$

Hence, $maxUtil(pcp)$ estimates the maximum chain utilization to be expected, if the partitioning pcp were to be established. The value $0 < \rho_{max} < 1$ on the left side of line (4) shall be a configuration constant that sets an upper bound for maximum chain utilization. Its purpose is to provide a margin of safety, so that minor variations in application workload do not immediately cause bottlenecks. Choosing $\rho_{max} = 0.9$ has been empirically determined to be a safe setting.

On the right side of line (4) the *utilization variance* of a primary chain par-

titioning $pcp = (\mathbf{cn}_1, \ldots, \mathbf{cn}_p)$ shall be defined as

$$utilVar(pcp) = \frac{1}{p}\sum_{i=1}^{p}\left(util(\mathbf{cn}_i) - \frac{1}{p}\sum_{j=1}^{p} util(\mathbf{cn}_j)\right)^2 \tag{4.15}$$

In summary, an enumerated partitioning will only be chosen as a solution if (1) the partitioning does not introduce any bottlenecks (including a margin of safety) and (2) it its utilization variance is lower than that of the default partioning where each task is its own chain. The latter condition optimizes the partitioning towards the second optimizaton objective, i.e. minimization of the utilization difference between chains. At the same time, the order in which partitionings are enumerated optimizes towards the first and primary optimization objective, i.e. as few chains as possible.

The runtime complexity of Algorithm 10 is dominated by the number of enumerated partitionings. The number of partitionings of a primary chain of length n into i subchains can be determined using a combinatorial formula. The number of partitionings equals the number of subsets with size $i-1$ of a set with $n-1$ elements, which is $\binom{n-1}{i-1}$. At the same time, computing the functions $maxUtil()$ and $utilVar()$ has complexity $O(n)$. Thus an invocation of Algorithm 10 takes

$$\begin{aligned}\sum_{i=1}^{n-1} n\binom{n-1}{i-1} &= n\left(\sum_{i=0}^{n-1}\binom{n-1}{i} - \binom{n-1}{n-1}\right)\\ &= n2^{n-1} - n\end{aligned}$$

computational steps. Hence Algorithm 10 has an asymptotic runtime complexity of $O(2^{|\mathbf{V}|})$ in the worst-case. Algorithm 9 has therefore an asymptotic worst-case runtime complexity of $O(|\mathbf{E}| + 2^{|\mathbf{V}|})$. Note that in practice, the exponential runtime complexity of Algorithm 10 should not be an issue. Common UDF-heavy data flows have rather low numbers of tasks, e.g. the largest Storm-based stream processing application at Twitter has eight tasks [117]. Furthermore, in most real-world template graphs, the longest primary chain will most likely be significantly shorter than $|\mathbf{V}|$.

4.7.5 Summary

The overhead of data item (de)serialization as well as the synchronization and and scheduling of threads can be a significant source of latency, especially for computationally lightweight subtasks. This section has proposed the DTC technique, which is designed to reduce such overhead for subtasks running on the same worker, that exclusively communicate with each other.

The decision where to establish a chain is made based on global summaries and hence a chain is modeled with tasks (not subtasks). In order to *establish* such a chain, workers employ the chaining mechanism described in Section 4.7.2, that changes the local mapping of subtasks to threads. The dynamic chaining algorithm described in Section 4.7.4 decides which chains to establish by heuristically trading-off two optimization goals. The primary goal is to produce chains that are as long as possible without introducing bottleneck tasks. The secondary goal is to distribute the workload as fairly as possible over the chains, which minimizes the input queue length of the threads that execute chained subtasks. The underlying hypothesis is that this approach does lead to an overall reduction in latency, that makes latency constraints easier to enforce.

4.8 Chapter Summary

This chapter presents a novel approach to express and enforce application-specific latency constraints in UDF-heavy data flows.

The first challenge addressed in this chapter is to find a notion of latency that (i) is capable of capturing the rich semantics of UDF-heavy data flows, (ii) is measureable with low overhead, and (iii) provides sufficient detail to be instrumental in bounding latency. The proposed notion is based on *average* latency, which makes it possible to independently measure the latency of subtasks and channels and compose an estimation of end-to-end (i.e. sequence) latency from these values. While measuring average latency may not achieve the same significance as measuring a high latency percentile, it also provides several advantages. First, unlike online percentile calculation [63], online average calculation has linear runtime and constant space complexity. This has the benefit of minimizing the measurement overhead. Second, since queues are a major source of latency, the focus on average latency permits us

to reason about queueing latency using the existing body of work in queueing theory, which is instrumental in estimating the effects of elastic scaling.

This chapter further introduces a notion of *latency constraints*. Latency constraints can be attached to an application by its developer and explicitly declare the otherwise implicit latency requirements of the application to the stream processing engine.

The main contribution of this chapter are however three techniques – Adaptive Output Batching (AOB), Dynamic Task Chaining (DTC) and Elastic Scaling (ES) – that collaboratively enforce latency constraints. Based on workload statistics that are continuously measured at runtime, they are designed to autonomously optimize important execution parameters towards constraint fulfillment.

The AOB technique decides when and how to ship data items in batches, which trades off latency against shipping overhead. Its goal is to batch data items as much as possible without violating the application specific latency constraints.

The ES technique is a novel elastic scaling policy for distributed stream processing engines. Based on a queueing theoretic model of the application it scales-up or -down as required to bound queue waiting time, so that application-defined latency constraints are fulfilled. The ES technique optimizes the parallelism of the application in manner that minimizes both over- and underprovisioning with respect to what is required to enforce latency constraints.

The DTC technique reduces the communication overhead between subtasks scheduled on the same worker, by dynamically adapting the subtask-to-thread mapping depending on measured workload statistics.

While each technique optimizes a separate aspect of the execution of a massively-parallel stream processing application, they are designed to complement each other. In particular, the ES and the AOB technique are tightly coupled as they manage channel latency towards a common goal.

Chapter 5: Experimental Evaluation

Contents

This chapter conducts an experimental evaluation of the techniques for latency constraint enforcing described in Chapter 4. The Nephele stream processing engine, that has been developed as part of this thesis, as well as a resource-managed commodity compute cluster at TU-Berlin are the basis of all experiments in this chapter.

As a prerequisite and later point of reference, Section 5.1 describes three exemplary stream processing applications that are used to investigate the effectiveness of the techniques. Each of the following sections then shifts the focus to one of the techniques, beginning with the AOB technique in Section 5.2 and followed by the ES and DTC techniques in Section 5.3 and Section 5.4 respectively. Section 5.5 summarizes the most important results and concludes the evaluation.

The recommended reading order for this chapter is to start with the introduction of Section 5.1 and then proceed with the evaluation of the techniques, starting with Section 5.2. The detailed descriptions of the applications used

during the evaluation can serve as a point of reference while progressing through the evaluations of the three techniques.

5.1 Experimental Stream Processing Applications

To experimentally evaluate the effectiveness of the techniques described in Chapter 4, three stream processing applications were implemented on top of the Nephele SPE. The table below gives a short overview of their defining properties. Each of the following sections describe an application in more detail.

	Livestream	**PrimeTest**	**TwitterSentiments**
Input	Video feeds	Random numbers	JSON data (tweets)
Workload	Multimedia (Compute-intensive)	Numeric (Compute- and I/O intensive)	Natural Language Processing & top-k aggregation (Compute-intensive)
Load Pattern	Steady	Stair-case	Day/night & bursts
Constraints	One (Src to Sink)	One (Src to Sink)	Two
Elastic	No	Yes	Yes
Chainable	Yes	No	No

5.1.1 Application 1: *Livestream*

The first example application is motivated by the idea of "citizen journalism", where website users produce and broadcast live media content to a large audience. The *Livestream* application groups related video feeds (e.g. by GPS location), merges each group of video feeds into a single feed, and augments the merged feeds with additional information, such as Twitter posts or other social network content. The idea is to provide the audience of the merged feeds with a broader view of a situation by automatically aggregating related information from various sources. Figure 5.1 shows the structure of the Application Graph.

The application consists of six distinct types of tasks. The first task is *VideoSource*. Each of its subtasks acts as a TCP/IP server over which H.264 encoded video feeds enter the data flow. Each video feed has a resolution of

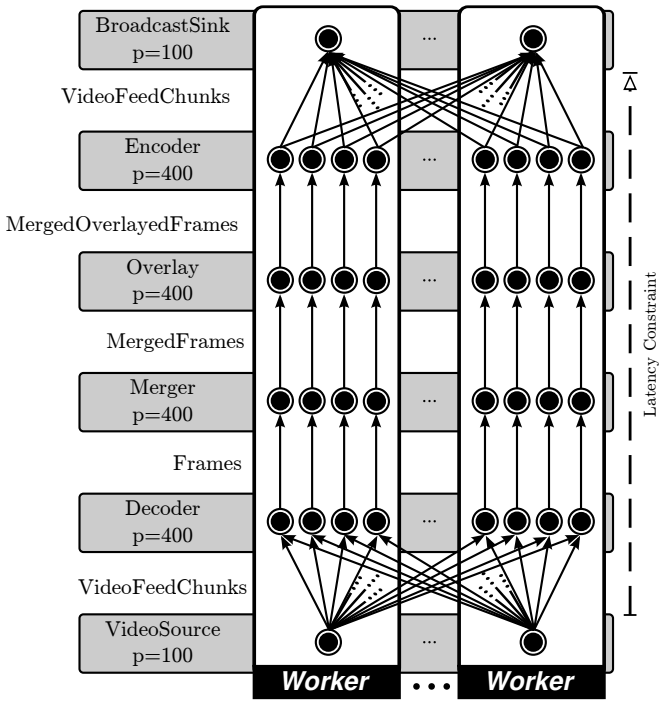


Figure 5.1: Application Graph and schedule of *Livestream* application.

320×240 pixels and is H.264 encoded. *VideoSource* subtasks assign each feed to a *group* of feeds, e.g. by GPS coordinates attached to the stream as meta data. They chunk the binary data of each video feed into data items and forward them to the *Decoder* subtask responsible for the group of the feed. *Decoder* subtasks decompress the H.264 encoded video feeds into frames using the ffmpeg library[1] and forward the frames to a *Merger* subtask. Here, the frames of each group are merged into a single frame, e.g. by tiling the individual input frames in an output frame. After each merge, a *Merger* subtask sends the merged frame to an *Overlay* subtask. An *Overlay* subtask augments each frame with information from additional related sources, e.g. it draws a marquee of Twitter feeds inside the merged video feed, which are picked based on keywords or locations close to the GPS coordinates attached to the merged video feeds. The resulting frames are then forwarded to an *Encoder* subtask, that encodes them back to a H.264 video feed, resulting in one H.264 encoded feed peer group. Finally, each of the encoded feeds is forwarded in chunks to a *BroadcastSink* subtask, for distribution to a general

[1] `http://www.ffmpeg.org/`

audience.

The *Livestream* application specifies co-location constraints to the scheduler of the Nephele SPE so that the depicted schedule is produced. As depicted in Figure 5.1, the *Decoder*, *Merger*, *Overlay* and *Encoder* tasks have the same degree of parallelism and their adjacent subtasks are scheduled on the same worker. Each worker additionally runs one *VideoSource* and one *BroadcastSink* subtask. As shown in Figure 5.1 the *Livestream* application has one latency constraint that covers all tasks and streams between *VideoSource* and *BroadcastSink*. As the *Merger* subtasks perform *aggregation*, their latency shall be measured as Read-Write subtask latency (see Section 4.1).

Workload Characteristics
The *Livestream* application provides a workload with a *steady* computational intensity. It provides an opportunity to observe the effects of the AOB technique on data item shipping latency efficiency. Moreover, the co-location constraints lead to a primary chain that begins at the *Decoder* task and ends at the *Encoder* task, thus creating an optimization opportunity for the DTC technique. Due to implementation limitations, the Livestream application is however not elastically scalable, as this would require the internal state of the ffmpeg de- and encoders to be migrated.

Application Parameters
Experiments that involve adaptive output batching use an output buffer size of 32 KB in the implementation of the output batching mechanism. The degree of parallelism of the latency constrained tasks is always 400. The following parameters need to be set for each experiment:

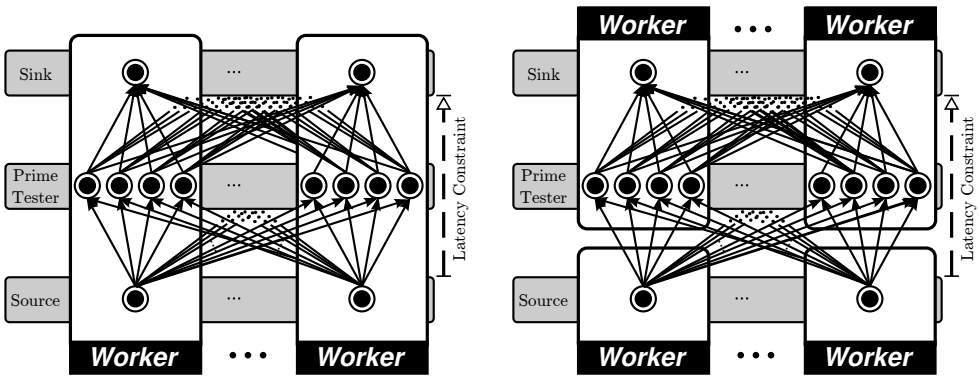
- The number of incoming video feeds, which determines the load put on *Decoder* subtasks.

- The *group size*, i.e. the number of video feed that are merged into the same output video feed. Lower group sizes lead to more output video feeds, which increases the load on *Overlay* and *Encoder* subtasks.

- The The upper runtime sequence latency bound ℓ of the constraint.

5.1.2 Application 2: *PrimeTest*

The *PrimeTest* application produces a step-wise varying computational load, so that we can observe steady-state workload statistics at each step. Hence, it is intended to be a micro-benchmark that clearly exposes certain aspects of stream processing under varying load.



(a) Schedule in unelastic experiments. (b) Schedule in elastic experiments.

Figure 5.2: Application graph and schedules of the *PrimeTest* application.

Figure 5.2 shows the application graph of the *PrimeTest* application. The *Source* subtasks produce random numbers at a rate that varies over time, and send them in a round-robin manner to *Prime Tester* subtasks, where they are tested for probable primeness. The tested numbers are sent in a round-robin manner to the *Sink* subtasks to collect the results. Testing for probable primeness is a compute intensive operation if done many times. Hence, we can control the computational load at the *Prime Tester* subtasks by controlling the send rate of the *Source* subtasks. The application progresses through several *phases* and *phase steps*, each of which lasts one minute. During each step, the *Source* subtasks send at the same constant rate, so we can briefly observe a steady-state workload. The phases are defined as follows:

1. *Warm-Up:* Numbers are sent at a low rate. This phase has one step and serves as a baseline reference point.
2. *Increment*: Numbers are sent at step-wise increasing rates.
3. *Plateau*: Numbers are sent at the peak rate of the Increment phase for the duration of one step.
4. *Decrement*: Numbers a sent at step-wise decreasing rates, until the Warm-Up rate is reached again.

The *PrimeTest* application shall have one template constraint as shown in Figure 5.2, where the adjustment interval is set to $t = 5$s, and the constraint time ℓ is varied between experiments.

Workload Characteristics
The *PrimeTest* application creates a workload with a *stair-case* load pattern. The generated data items are small, as the only hold a 32 bytes random number and some meta-data. Hence, the application generates a workload where small data items are sent at high rates. Each incoming data items triggers a compute intensive operation. Therefore, the overall computational intensity of the application changes proportionally with the send rate of each step. Since the UDF of the *PrimeTester* task is stateless, the task can be elastically scaled if necessary for the purpose of the experiment.

Application Parameters
Several aspects of the PrimeTest workload can be varied between experiments. For each experiment that involves adaptive output batching, the output buffers used in the implementation of the output batching mechanism shall have a fixed size of 16 KB. Note that choosing a larger size did not have any effect.

The following parameters are fixed for each experiment, but can be varied between experiments:

- The number of steps in the *Increment* and Decrement phases, as well as the respective send rates.

- The degree of parallelism of all tasks and the type of schedule. We shall use either an *elastic* or *unelastic* schedule, depending on the experiment. The non-elastic schedule co-locates one *Source* and *Sink* subtask as well as four *PrimeTester* subtasks on each worker. In contrast to that, the elastic schedule only co-locates a *Sink* subtask with four *PrimeTester* subtasks on each worker.

- The upper runtime sequence latency bound ℓ of the constraint.

5.1.3 Application 3: *TwitterSentiments*

The third example application is inspired by the many services that analyze social media data to identify sentiments on specific topics, brands and companies that may result in marketable trends[2][3][4]. The *Merriam-Webster Dictionary*[5] defines a *sentiment* as "*an attitude, thought, or judgment colored or prompted by feeling or emotion.*" According to [98], *sentiment analysis* "*deals with the computational treatment of [...] opinion, sentiment, and subjectivity in text.*" In the particular case of our example application, sentiment analysis is used to characterize a tweet's sentiment with respect to a particular topic as positive, neutral or negative. Figure 5.3 shows the structure of the Application Graph. The application consists of six distinct types of tasks.

The *TweetSource* task receives JSON-encoded tweets via TCP/IP from outside the data flow and forwards each tweet twice: The first copy is sent round-robin to a *HotTopics* subtask, that extracts popular topics such as hashtags from the tweet. Each *HotTopics* subtask maintains its own list of currently popular topics, sorted by popularity, and periodically forwards this list to the *HotTopicsMerger* subtask. The HotTopicsMerger subtask merges the partial lists it receives into a global one. The global list is periodically *broadcasted* to all *Filter* tasks. The second copy of each incoming tweet is sent round-robin to a *Filter* subtask that matches it with its current list of popular topics. Only if the tweet concerns a currently popular topic, it is forwarded to a *Sentiment* subtask, that attempts to determine the tweet's sentiment on the given topic. Our implementation of the Sentiment UDF delegates sentiment analysis to a library[6] for natural language processing. Finally, the result of each sentiment analysis (positive, neutral, negative) is forwarded to the *SentimentSink* subtask, that tracks the overall sentiment on each popular topic.

As depicted in Figure 5.3, the *TitterSentiment* application has two latency constraints. The first constraint convers the branch of the data flow that detects popular topics, whereas the second one covers the branch that performs sentiment analysis. The HotTopics and HotTopicsMerger subtasks perform

[2] `http://www.buzztalkmonitor.com/`
[3] `https://semantria.com/`
[4] `http://www.webmd.com/`
[5] `http://www.merriam-webster.com`
[6] `http://alias-i.com/lingpipe/`

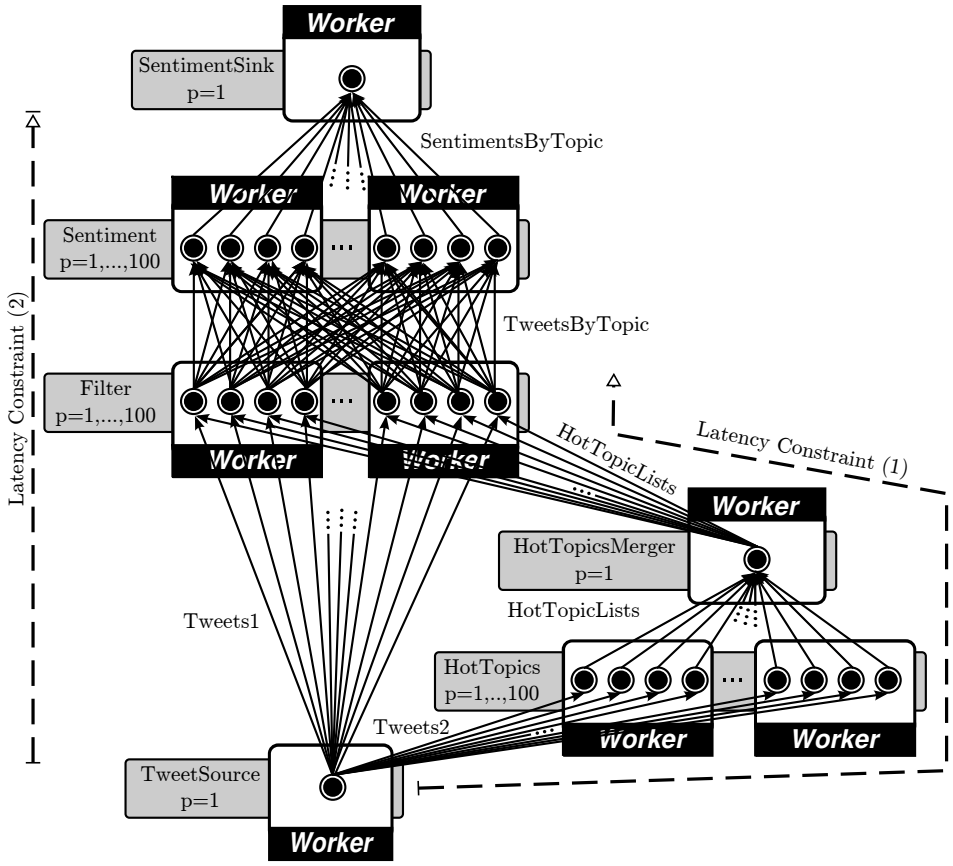


Figure 5.3: Application Graph structure and schedule of the *TwitterSentiments* application with elastic tasks.

aggregation, hence their latency is measured as Read-Write subtask latency (see Section 4.1).

The application contains three elastically scalable tasks, i.e. *HotTopics*, *Filter* and *Sentiment*. To simplify subtask placement during scaling, only subtasks of the same type are co-located on a worker. This increases the probability that workers can be deallocated afer scale-downs.

Workload Characteristics

The workload of the *TwitterSentiments* application is driven by the rate of arriving tweets. The *TweetSource* task replays a 69 GB dataset of JSON-encoded, chronologically ordered English tweets from North America logged via the public Twitter Streaming API during a two-week period in August

2014. The tweet rate of the original data set is depicted in Figure 5.4. As visible from the diagram, the tweet rate is highly variant with significant daily highs and lows and contains occasional bursts, such as the one on August 13th. Note that during the experiment the tweet rate is 200x that of the original tweet rate.

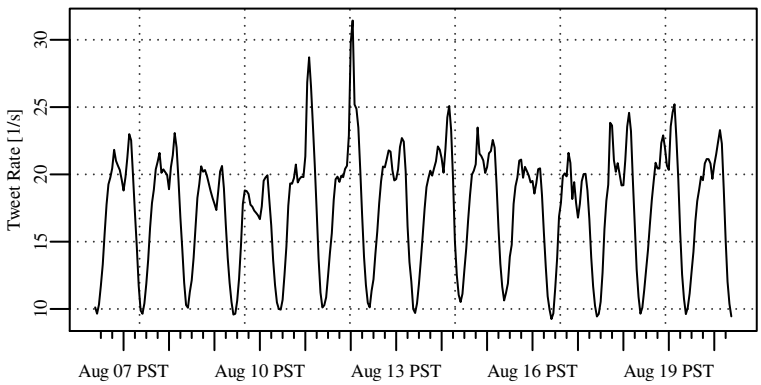


Figure 5.4: Tweet rate of the 69GB data set over a two week period. The data set is replayed at 200x speed, so that the data from two weeks is replayed within 100 minutes.

Application Parameters
Experiments that involve adaptive output batching use an output buffer size of 16 KB in the implementation of the output batching mechanism. The elastic tasks (Filter, Sentiment and HotTopics) have a minimum, maximum and initial parallelism of 1, 100 and 4 respectively. While the slack times of the two latency constraints are chosen individually for each experiment, the adjustment interval is always $t = 5$ s.

5.2 Evaluation of the AOB Technique

This section conducts an experimental evaluation of the AOB technique described in Section 4.5. To gain a better understanding of its effects, the AOB technique is employed in isolation in this section. In Section 5.2.1 we begin by an initial experiment that demonstrates the overall effect of the AOB technique. In Section 5.2.2 and Section 5.2.3 we evaluate the effectiveness and limitations of the of the AOB technique for a range of application workloads.

5.2.1 Experiment 1: Livestream

In the first experiment we take a look at the Livestream application running on the Nephele SPE with only the AOB technique (no DTC or ES). For this experiment, cluster resources were manually provisioned to be sufficient. Figure 5.5 shows the results and describes the setup parameters of the Livestream application with a 100 ms template constraint.

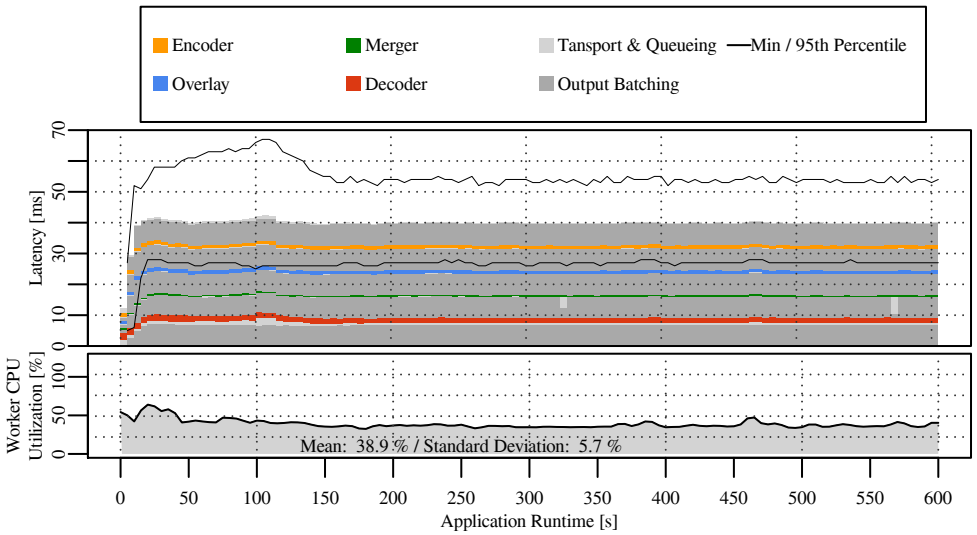


Figure 5.5: Latency and worker CPU utilization of Livestream application with 3200 video feeds, group size 4 and a 50 ms constraint. Please see Section 5.1 for further details. Latency is plotted as one bar per adjustment interval, based on values from the respectiv global summary. Hence, the bars and their subars capture mean latencies. Additionally, the sink subtasks track per-data item latency and compute the minimum and 95-th latency percentile.

As visible in Figure 5.5 the engine starts with the default immediate shipping strategy. The AOB technique determines the correct parameters for the output batch lifetime after a few intervals and which stabilizes mean latency after around 20 seconds. After stabilization, the average runtime sequence latency (height of bars) settles at around 40 ms and remains there until the application is terminated. This is due to two effects. The first is that the AOB technique uses a fixed *output batching weight* to allow for some queueing latency on each channel without endangering the constraint (see Algorithm 3 in Section 4.5.3). However due to low load, queueing latency is negligible in this configuration, therefore some of the "slack time" of the 50 ms constraint

is not used. The second reason is the implementation strategy of the output batching mechanism, which relies on fixed size output buffers of 32 KB in this experiment. These buffers fill up very quickly on channels that transport decoded images, especially between the *Decoder* and *Merger* subtasks, which results in these buffers being shipped ahead of their deadline.

Note that this experiment does not yet show any *benefits* of adaptive output batching over the default immediate shipping strategy. However, it shows that the AOB technique works as expected, i.e. that it batches emitted data items as much as application-specific latency constraints allow. In some cases, output batches are shipped ahead of their deadline because the underlying fixed size buffer was full. While this is an effect of the implementation of the output batching mechanism, it has the positive side effect of preventing batching beyond a point that will under no conditions improve throughput. As already shown in Figure 4.4 in Section 4.5, buffers larger than 32 KB show little impact on achievable channel throughput, hence making 32 KB a reasonable choice.

5.2.2 Experiment 2: Load and Constraints (Livestream)

In a second experiment, we will vary the load and latency constraints of the *Livestream* application in order to gain a better understanding of the effects of the AOB technique under various conditions.

The results shown in Figure 5.6 compare the effects of the immediate shipping strategy against adaptive output batching. Choosing the number of ingoing video feeds and the group size affects the load of the application, hence each subplot shows the results of a certain load level. Configurations with more ingoing video feeds put a higher load on *Decoder* subtasks, whereas configurations with lower group sizes put a higher load on the *Overlay* and *Encoder* subtasks. While resources were sufficiently provisioned in all the configurations displayed in the plot, it can be observed that higher load leads to higher variation in latency, regardless of whether constraints are set or not. In fact, the figure shows that adaptive output batching does not offer any benefit over immediate shipping in the Livestream application. For each load level (subplot), immediate shipping does give a lower and steadier latency. This is likely due to the fact that data item sizes in the Livestream application are already too large to reap the benefits of output batching. Data items in this

application are either compressed video frames, or uncompressed bitmaps. In the first case, due to the way H.264 video compression works, key frames appear in regular time intervals (e.g. every one or two seconds) in each video feed. A key frame contains all the necessary information to decode on full resolution image of the video feed and is hence several KB large. In the second case, once video frames have been decompressed by a *Decoder* subtask, they are bitmaps of approx. 300 KB size, thus each bitmap spans several 32KB output buffers, leaving little optimization potential for adaptive output batching.

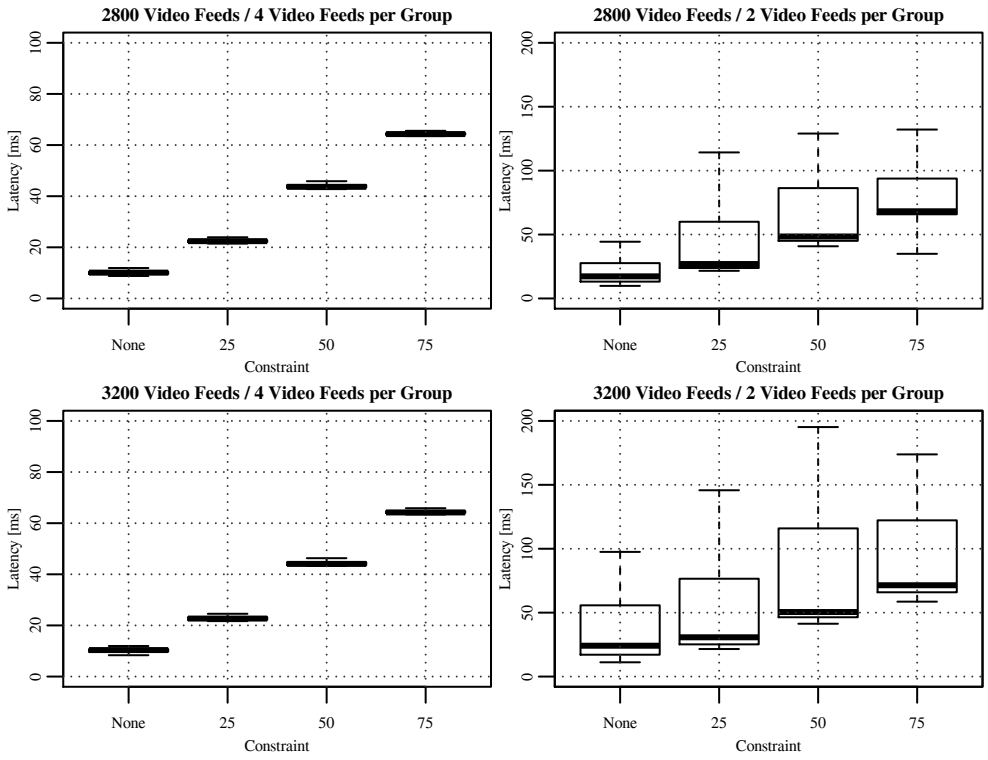


Figure 5.6: The figure shows the latency of the Livestream application under various workload and constraint configurations. Each configuration was run three times for three minutes. The boxplot of each configuration shows the *distribution* of the mean runtime sequence latency (computed from the global summary) over all adjustment intervals of all runs with that configuration.

5.2.3 Experiment 3: Load and Constraints (PrimeTest)

In the third experiment presented in this section we shall employ the *PrimeTest* application to explore the effects of the AOB technique for a range of load levels and constraints. In the *PrimeTest* application we can adjust the load in a much more fine grained manner by choosing a fixed send rate at the *Source* subtasks.

The results of the *PrimeTest* application are shown in Figure 5.7. Several aspects stand out in the results. The first is that configurations with immediate shipping consistently have the highest CPU utlization, which indicates a significant overhead for immediate shipping of small data items. The dominant pattern in the plot is that higher constraints lead to lower CPU utilization and shift the manifestation of bottlenecks to higher send rates. As indicated by the CPU utilization plot, bottlenecks manifest because the application becomes I/O -bound rather than CPU-bound. With immediate shipping, the highest send rate without bottlenecks is 40k data items per second. In comparison to that, the last send rates without bottlenecks are 50k, 55k, 55k and 60k data items per second for the configurations with a 25 ms, 50 ms, 75 ms and 100 ms constraint respectively. When looking at the rates where constraints are still fulfilled, a similar trend emerges. The higher the constraint is chosen, the higher is rate at which the constraint can still be enforced. For example, 25 ms constraint can be enforced at rates up to 45k data items per second, whereas the 75 ms constraint can be enforced at rates up to 55k data items per second.

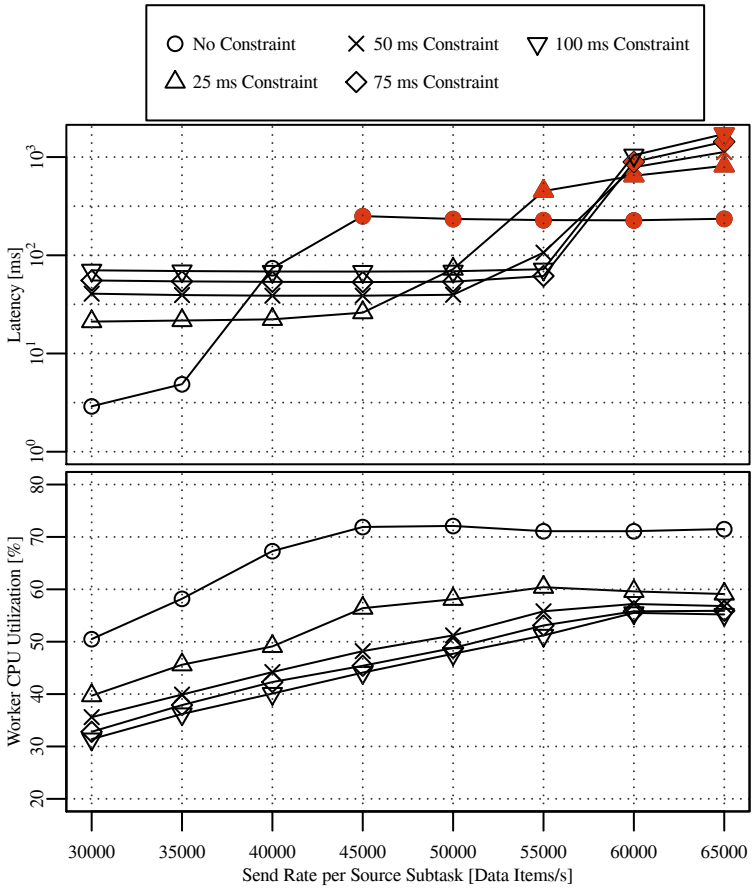


Figure 5.7: The figure shows the latency and mean worker CPU utilization of the *PrimeTest* application for various configurtions (send rate and constraint). The application was run with 50 *Source* and *Sink* subtasks as well as 200 *PrimeTester* subtasks, using the unelastic schedule shown in Figure 5.2, which resulted in 50 allocated workers. The application was configured with only one phase and with one step, in which data items are sent at a fixed rate. Each data point is the result of running the application three times for three minutes with the respective configuration, and computing the *mean* latency/CPU utilization over the adjustment intervals of those runs. Configurations with bottlenecks, where the *achieved* send rate was lower than the attempted rate are marked in *red*.

In summary, adaptive output batching allows to operate the I/O intensive *PrimeTest* application at higher rates with lower CPU utilization. Hence, in suitable appplications, this effect can can be exploited by the ES technique to

scale down while still fulfilling the constraint. Positive effects for CPU-bound applications that ship data items at high rates can be expected as well.

5.3 Evaluation of the ES Technique

This section provides an experimental evaluation of the ES technique based on the implementation inside the Nephele SPE described in Section 4.6.4. While the ES technique is the *focus* of experimentation in this section, all experiments also employ the AOB technique. The rationale behind this, is that the optimization techniques in this thesis are designed to be used in conjunction. Section 5.3.1 evaluates the ES technique based on the *PrimeTest* micro-benchmark. Section 5.3.2 evaluates the technique based on the *TwitterSentiments* application, that provides a heavily fluctation real-world workload.

5.3.1 Experiment 1: *PrimeTest*

In this experiment we compare the results of running the *PrimeTest* application in two configurations, each of which is scheduled with 32 Source and Sink subtasks. The configurations are as follows:

- **Elastic**: In this configuration, the application is attached with a 20 ms latency constraint. The Nephele SPE tries to enforce the constraint by applying the AOB and ES techniques. The *Prime Tester* task is elastic with a parallelism ranging from $p^{min} = 1$ to $p^{max} = 392$.

- **Unelastic with fixed buffers**: This configuration provides a point of comparison to judge the resource efficiency of the elastic configuration. The unelastic configuration is optimized towards maximum throughput and implements a simple batching strategy with fixed-size 16KB byte buffers on each channel. Each data item is serialized into a buffer, and buffers are only shipped when full. The parallelism of the *Prime Tester* task is fixed to 175 subtasks. This value has been manually determined to be as low as possible, while at the same time not leading to overload with backpressure at peak rates.

In both configurations, the *PrimeTest* application goes through the phases described in Section 5.1.2. The send rate of each *Source* subtask ranges from 10k to 80k data items per second and changes in steps of 10k. Note that running the application with lower constraints or immediate shipping improved the 95th latency percentile, but also resulted in configurations that were not able to sustain the peak send rate due to I/O bottlenecks.

Figure 5.8 shows the results of both running the application in both configurations. In the elastic configuration, the constraint could be enforced ca. 91% of all adjustment intervals. There is one significant constraint violation when the send rate doubles from 10k to 20k data items per second, while transitioning from the *Warm-Up* to the *Increment* phase. The reason for this is that during the *Warmup* phase, the *PrimeTester* parallelism dropped to 36 subtasks, which is a desired effect of the *Rebalance* algorithm, as it is designed to minimize resource consumption while still fulfilling the constraint. All other *Increase* phase steps resulted in less severe and much shorter constraint violations that were quickly addressed by scale-ups, because the relative increase in throughput decreases with each step. Since the ES technique is a *reactive* scaling strategy, constraint violations resulting from large changes in workload are to be expected. Most scale-ups are slightly larger than necessary, as witnessed by subsequent scale-downs that correct the degree of parallelism. The overscaling behavior is caused by (i) the queueing formula we use, which is an approximation, and most importantly (ii) by the *error coefficient* $e_{\mathbf{v}}$ (see Equation 4.5 in Section 4.6.2), which can become overly large when bursts of data items increase measured queue latency. However, we would argue that this behavior is useful, because it helps to resolve constraint violations quickly, albeit at the cost of temporary over-provisioning.

The 95th latency percentile is ca. 30 ms once scale-ups have resolved temporary queue build-up, and is naturally much more sensitive to changes in send rate than the mean latency. Since most of the latency is caused by (deliberate) output batching, 30 ms is within our expected range.

The unelastic but manually provisioned baseline configuration is optimized towards maximum throughput and tuned to withstand peak load with minimal resource consumption. In consequence, both mean latency and the 95th latency percentile are much higher and do not go lower than 348 ms and

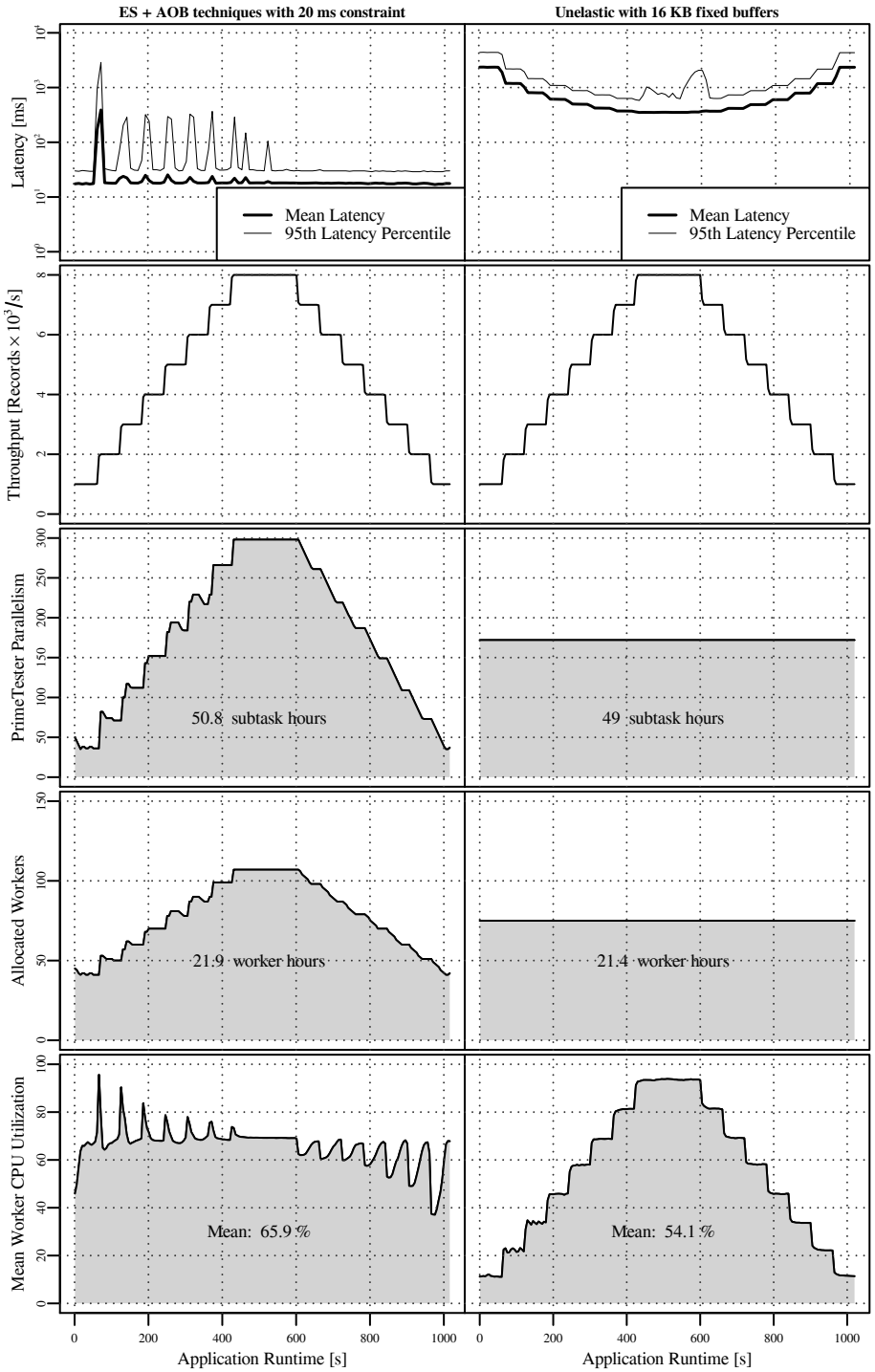


Figure 5.8: Result of running the *PrimeTester* application with and without elastic scaling.

	Constraint				
	20 ms	**30 ms**	**40 ms**	**50 ms**	**100 ms**
Intervals with fulfilled constraint	92.6%	92.2%	94.1%	94.1%	95.1%
95th latency percentile	30 ms	45 ms	56 ms	73 ms	128 ms
Subtask Hours (PrimeTester)	50.8	46.4	44.3	41.8	37.6
Worker Hours	21.9	23.8	20.2	19.7	18.6
Mean Worker CPU Utilization	65.9%	69.5%	70.2%	71.8%	73.9%

Table 5.1: Latency-resource trade-off for the elastic *PrimeTest* application.

564 ms respectively. We measure resource consumption in "subtask hours" and "worker hours" i.e. the amount of running subtasks and workers over time. Despite manual tuning, the amount of consumed subtask hours is almost equal to the elastic configuration. While the two configurations are at par in this respect, we should consider that choosing a higher latency constraint can be expected to yield lower resource consumption. To investigate the magnitude of this latency-resource trade off, the elastic configuration was executed again with the same workload but different constraints. The results are summarized in Table 5.1. The tableshow savings in worker hours of up to 13% for a 100 ms constraint, compared to the unelastic configuration, which still has more than three times the latency.

In summary, this experiment has shown that the *combination* of the ES and AOB techniques is effective at quickly reacting to fluctuations in load and minimizing the resource provisioning of stream processing applications. While the ES technique is reactive, and hence cannot anticipate constraint violations before they occur, it still achieved constraint fulfillment in over 92% of all adjustment intervals across a multitude of different configurations of the *PrimeTest* application. Moreover, this experiment has shown that in combination with adaptive output batching, the ES technique is equally or more resource efficient than manually optimized non-elastic configurations. While the queueing theoretic latency model in Section 4.6.2 succeeds at accurately estimating the effects of small scaling actions, it has the tendency to overscale when large fluctuations in load occur, which somewhat lowers its resource efficiency.

5.3.2 Experiment 2: *TwitterSentiments*

This section evaluates the effectiveness of the ES technique based on the *TwitterSentiments* application, that provides a realistic workload, and is driven by an accelerated, but otherwise time-accurate replay of recorded tweets. Please refer to Section 5.1.3 for a detailed description of the application

Figure 5.9 shows the result of running the *TwitterSentiments* application. Constraint (1) with $\ell = 215$ ms was fulfilled in 93% of all adjustment intervals. The 95th latency percentile stays close to the constraint throughout the runtime of the application, because most of the latency results from the windowed aggregation behavior of the UDFs of the *HotTopics* and *HotTopicsMerger* tasks. The parallelism of the *HotTopics* task is frequently adjusted to variations in the tweet rate. Due to two reasons both mean and 95th percentile latency on the constraint's sequence are relatively insensitive to tweet rate variations. First, the task latency caused by windowed aggregation dominates the constraint, which therefore leaves little room for output batching. Second, bursts in the tweet rates are fully absorbed by HotTopics tasks as they perform data-parallel pre-aggregation and produce fixed size lists with popular topics in fixed time intervals.

Constraint (2) with $\ell = 30$ ms was fulfilled during 96% of all adjustment intervals, however there are significant latency spikes caused by tweet bursts. Outside of such bursts, the 95th latency percentile stays close to 25 ms. The tendency of the ES technique to slightly over-provision with respect to what would be necessary to fulfill the constraint, can be observed in this benchmark as well. Here, the reason is the heavily varying tweet rate, that often changes faster than the scaling strategy can adapt. Because scale-ups are fast and scale-downs are slow, the application stays slightly over-provisioned. This is further evidenced by a fairly low mean CPU utilization of 55.7% on workers. Again, choosing a higher latency constraint would trade off utilization and resource consumption against latency.

The tweet rate varies heavily between day and night, peaking with 6734 tweets per second at around 2400 s. Most notable about the peak is that its tweets affected very few topics. In such a situation the *Filter* subtasks become less selective, which results in a significant load spike for the *Sentiment* task. The resulting violation of Constraint (2) was mitigated by a significant *Sentiment* scale-up with ca. 28 new subtasks. Other tasks were not scaled

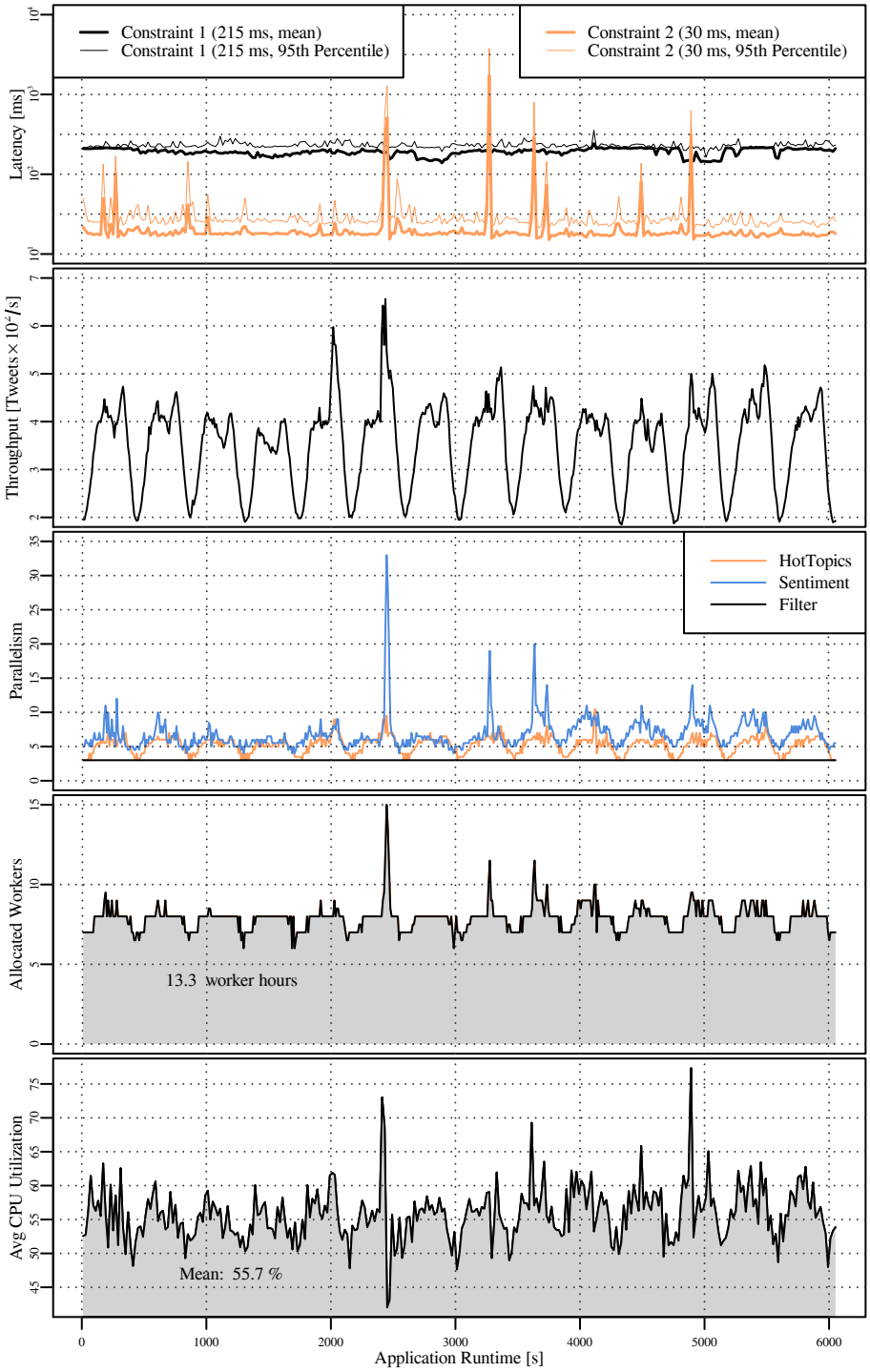


Figure 5.9: Result of running the *TwitterSentiments* application with the AOB and ES techniques.

up to the same extent, because their relative change in load was not as high.

In summary, this experiment has shown that the combination of the ES and AOB techniques successfully enforces latency constraints even for real-world applications with a bursty workload. Compared to the *PrimeTest* evaluation in the previous section, load bursts however seem to reduce resource efficiency, due to the inertia of the scaling mechanism. One way of mitigating this effect is to resort to latency constraints with more "slack" time, that can absorb bursts to some degree. However, it depends on the latency requirements of the application, whether this is a viable option or not.

5.4 Evaluation of the DTC Technique

This section provides an experimental evaluation of the DTC technique as described in Section 4.7. While the DTC technique is the *focus* of experimentation in this section, all experiments also employ the AOB technique. Much like in the previous section, the rationale behind this, is that the optimization techniques in this thesis are designed to be used in conjunction. Section 5.4.1 we begin by an initial experiment based on the Livestream application, that demonstrates the overall effect of the DTC technique. Section 5.4.2 explores the effects of the DTC technique for a range of Livestream-based workload configurations.

5.4.1 Experiment 1: Livestream

The first experiment we take a look at the Livestream application running on the Nephele SPE and observe the combined effects of the AOB and DTC techniques. For this experiment, cluster resources were manually provisioned to be sufficient.

Figure 5.10 describes the configuration parameters in more detail and shows the results of the experiment. As visible from the plot, the dynamic task chaining algorithm chose to chain the *Decoder*, *Merger*, *Overlay* and *Encoder* tasks, because the sum of their respective maximum subtask utilizations was lower than the maximum allowed chain utilization ρ_{max}.[7] After an initial

[7]See Algorithm 10 in Section 4.7.4. The rationale for choosing ρ_{max} is similar to that of the maximum utilization threshold in elastic scaling. Choosing $\rho_{max} = 0.9$ is a relatively safe trade-off that reduces the risk of bottlenecks, due to minor workload fluctuations.

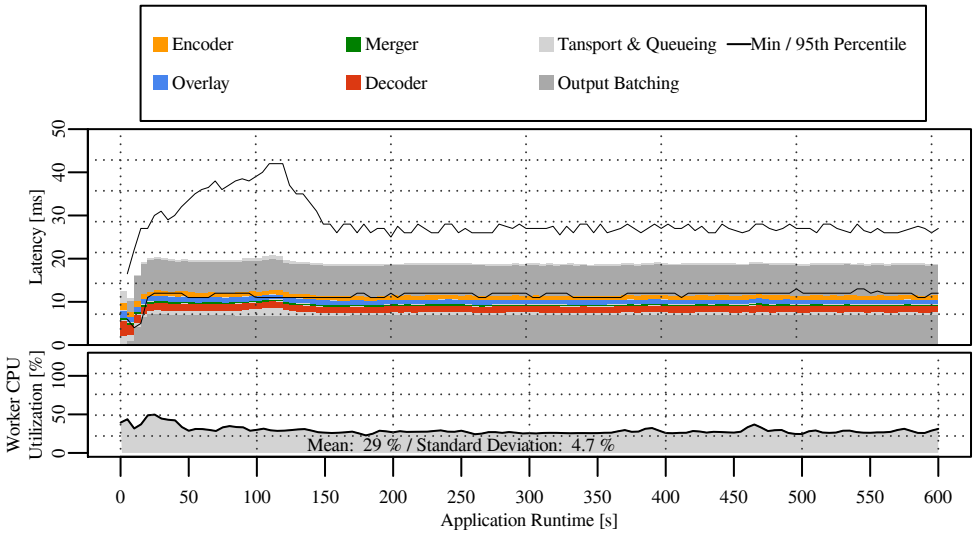


Figure 5.10: Latency and worker CPU utilization of Livestream application with 3200 video feeds, group size 4 and a 50 ms constraint. Latency is plotted as one bar per adjustment interval, based on the respective mean values from global summaries. The sink subtasks additionally track per-data item latency and compute the minimum and 95-th latency percentile, which are included in the plot. The Nephele SPE was configured to apply both the AOB and DTC techniques. The maximum allowed chain utilization of the DTC technique was set to $\rho_{max} = 0.9$.

calibration phase, the mean latency stabilized between 18-19 ms. This is significantly lower than the 50 ms constraint, because the delays caused by serialization, output batching and queueing are eliminated in the chained part of the data flow. Due to the generally low latency, the constraint was fullfilled all the time.

Note that the parameters of the experiment as well as the structure of the plot are intentionally identical to Figure 5.5 in Section 5.2.1, that shows results of the exact same workload, however in an AOB-only configuration. In comparison with the AOB-only configuration, it can be observed that the mean worker CPU utilization is about 33% lower, when combining the AOB and DTC techniques.

In summary, this experiment has shown that the DTC technique leads to a significant reduction in latency for the Livestream application. Due to reduced data item shipping overhead, a 33% reduction in CPU utilization

could be achieved for the Livestream application.

5.4.2 Experiment 2: Load and Constraints

Analogous to Section 5.2.2, in this section we vary the load and constraints of the Livestream application in order to gain a better understanding of the effects of combining the AOB and DTC techniques under various conditions.

The results shown in Figure 5.11 compare the effects of immediate shipping with DTC, against the combined use of the AOB and DTC techniques. All configurations with constraints we able to meet their constraint in 100% of all adjustment intervals - an effect that has already been observed in the previous experiment.

Note that the workload parameters of the Livestream application as well as the structure of the plot are intentionally identical to Figure 5.6 in Sec-tion 5.2.2, which shows results for AOB-only configurations. As previously stated, choosing a group size of two instead of four, increases the load on the *Overlay* and *Encoder* tasks in the Livestream application. In these high-load scenarios, all configurations in Figure 5.6 resulted in a rather unstable latency. In contrast to that, all configurations in Figure 5.11 show a very *steady* latency. This highlights one aspect of the DTC technique. The sched-ule of the Livestream application as shown in Section 5.1.2 places 10 sub-tasks in each worker, but the underlying compute nodes in this experiment are equipped with only four cores. Oversubscribing the available CPU cores on a worker in this manner, causes the subtask threads to thrash under high load, which generally degrades performance. As the DTC technique reduces the number of threads, this also reduces thrashing, resulting in a steadier latency for the application.

In summary, this experiment has shown that dynamic task chaining is a useful not only towards reducing, but more importantly towards stabilizing latency, because it also reduces thread thrashing in schedules that oversubscribe CPU cores with subtasks.

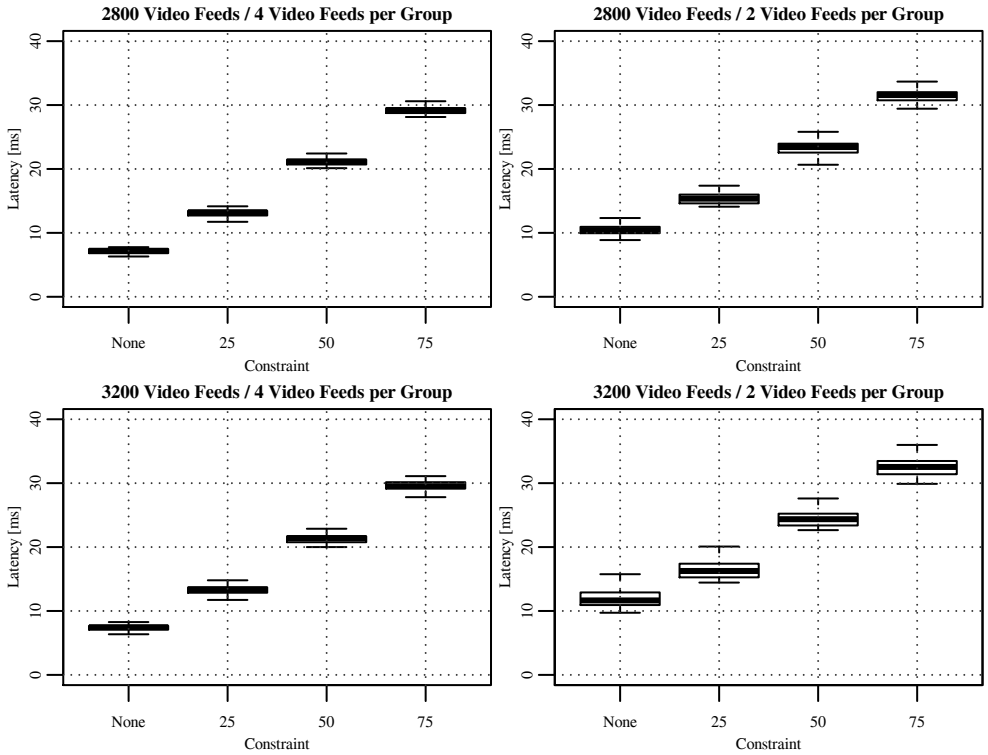


Figure 5.11: The figure shows the latency of the Livestream application under various workload and constraint configurations when concurrently applying the AOB and DTC techniques. Since the DTC technique is also applicable without constraints, configurations without constraint ("None") also employ the DTC technique, but rely on immediate shipping. Each configuration was run three times for three minutes. The boxplot of each configuration shows the *distribution* of the mean runtime sequence latency (computed from the global summary) over all adjustment intervals of all runs with that configuration.

5.5 Conclusions

In this chapter we have experimentally evaluated the techniques for latency constraint enforcement described in Chapter 4 with a range of different workloads.

The experiments have shown that using the AOB technique by itself to enforce latency constraints can be expected to give mixed results, depending on the nature of the application. For applications that process small data items at high rates, the AOB technique provides several benefits as it (i) reduces CPU utilization due to shipping overhead, and most importantly (ii) lowers the risk of I/O bottlenecks, that degrade latency and inhibit scaling. However, for applications where shipping overhead is negligible, adaptive output batching seems to offer little benefit.

In combination with adaptive output batching, the ES technique has been shown to enforce latency constraints most of the time, even in a real-world stream processing workload with a heavily fluctuating and bursty workload. In experiments, it successfully mitigated latency constraint violations within few adjustment intervals. However, especially during bursts, the estimation quality of the latency model degrades, causing the ES technique to temporarily overprovision resources. Despite that, in experiments with the *PrimeTest* application, the ES technique had a resource footprint comparable or lower to that of a configuration that was manually optimized to withstand peak-load with minimal resources.

The DTC technique complements the above techniques and leads to a lower and steadier latency in applications that are *chainable*. While some of this effect can be accredited to the reduced shipping overhead, the DTC technique is especially effective at mitigating the effects of thrashing in situations where CPU cores are oversubscribed with subtask threads.

In conclusion, the fact that all three techniques optimize separate aspects of stream processing makes them well-suited to be used in conjunction and enables them to enforce latency constraints for a wide range of application workloads. This constitutes an improvement over the current state of the art in stream processing, where engine configuration settings and often also the resource provisioning need to be manually tuned to obtain the latency required by an application, which is a cumbersome and non-trival process.

Chapter 6: Related Work

Contents

Section 3.3 has already provided extensive background on distributed stream processing engines for UDF-heavy data flows. This chapter focuses on related work with respect to the use of batching, chaining, and elastic scaling techniques.

6.1 Batching Strategies

Batching is a popular technique to amortize the often substantial overhead of certain operations in stream/event processing over many data items. Choosing appropiate batch sizes is generally a trade-off between increased latency (batches need to fill up) and overhead reduction.

In [27] Carney et al. propose *train processing* to reduce the overall data item processing costs in the centralized Aurora [1] DSMS in a multi-query scenario. The Aurora system explicitly schedules when which query operator shall process a data item. Thus, processing data items in *trains* (batches) reduces the number of scheduling decisions that need to be made. Train scheduling trades off train size against the impact of increased latency on a user-defined Quality of Service (QoS) function, that maps latency to a QoS value.

In [126] Welsh et al. describe SEDA, a staged, event-driven architecture for internet services. Each stage of a SEDA application runs on a single node and has an input queue from where *batches of events* are dispatched to a pool of worker threads. A *batch controller* within each stage adapts the number of events per batch to the current workload. In SEDA, batching amortizes the overhead of data item dispatching and improves data and instruction cache locality (see also [71]). Since batching increases both throughput and

latency, the batching controller adjusts the batch size to be as small as possible without choking throughput. Beginning with the largest batch size, it observes the throughput of the stage and reduces its batch size until throughput begins to degrade. This control-theoretic approach to batching strives to find an optimal operating point.

Spark Streaming [132] and Comet [48] perform batched stream processing (BSP) and occupy a middle ground in the design space between batch and stream processing. In the BSP model arriving stream data is first put into an input data set. In Comet, this input data set resides in a distributed file system, whereas in Spark Streaming the input data set is reliably stored in the memory of cluster nodes. Once every time interval this input data set is then processed with a mini-batch job. Conceptually, each input data set can be seen as a coarse batch of input data. The BSP model enables the reuse of batch programming models for incremental data processing, e.g. LINQ [87] and the Spark programming model [132, 131]. Moreover, as noted in [48], the BSP model prevents the overhead of multiple I/O intensive passes over a large data set, when all that needs to be processed is a time-constrained window of that data. In contrast to pure stream processing, the latency of batched stream processing ranges from multiple seconds (Spark Streaming) to minutes (Comet), as each triggered batch job needs to be scheduled on a cluster. Recently, in [33] Das et al. have extended Spark Streaming to adaptively minimize the latency of the BSP model by adjusting batch sizes to the current data arrival rate, achieving a lower but still multi-second latency.

In the field of embedded systems, *synchronous data flows* [72] are a common model for modeling stream computation. As the input and output rates of each processing node in a synchronous data flow is known in advance, this model suits itself to static optimizations. *Execution scaling* described in [106] is a static compiler optimization that trades off instruction and data cache locality by batching the execution of computations nodes in the data flow graph.

All of the aforementioned batching approaches focus on *processing* data items in batches, which requires the ability to exert control over when a data item is processed. In contrast to that, the AOB technique presented in this thesis focuses on *shipping* data items in batches, that are as large as possible without violating application-defined latency constraints. The threads that

process those data items are scheduled by the operating system and not the engine. Therefore, it depends on the thread scheduling policy of the operating system whether this leads to batched processing or not. The Nephele SPE developed as part of this thesis is Java-based, and since current Java virtual machines use preemptive scheduling, a batch of data items is not guaranteed to be processed without preemption.

6.2 Chaining

Chaining, also often called *fusion*, is a common optimization technique that avoids the overhead of data item shipping by sacrificing pipeline parallelism. Commonly, chained operators are executed within the same thread, which is safe as long they do not require more than the capacity of a single processor.

The authors of [40] describe SPADE, a declarative frontend for the System S distributed stream-processing engine [8]. The compiler of SPADE chains SPADE operators into processing elements (PEs) that are distributed over a cluster. Based on profiling runs, it searches for an operator-to-PE mapping that minimizes the number of PEs without overwhelming the processors on cluster nodes. The authors of COLA [64] extend the default SPADE optimization strategy to also balance the load across PEs.

Another compile-time application of chaining can be found in [44], that describes how to compile StreaMIT [116], a portable language for streaming applications, for a range of different architectures such as embedded devices, consumer desktops and servers. Based on input/output rates known prior to compilation, StreaMIT chains adjacent operators, to fit the number of operators to the number of processors of the underlying hardware platform. This also enables further compiler optimizations, e.g. scalar replacement and register allocation, that improve data access locality within the memory hierarchy.

In [27] Carney et al. propose *superbox scheduling*, that reduces the scheduling overhead in the centralized Aurora [1] DSMS when executing multiple queries. Superboxes are equivalent to queries and scheduled as a whole. Hence the operators within each superbox are executed in a chained manner.

A well-known example of chaining in the field of parallel data processing are

the chained map functions of Apache Hadoop. Before starting a Hadoop job, a user can specify a series of map functions to be chained. Hadoop will then execute these functions in a single process, and invoke the user code of a map function in the chain directly on the output of the previous map function. If the chained mappers are stateless (as typically expected from map functions in Hadoop), it is even safe to pass data items from one mapper to the other by reference.

All of the above systems perform chaining statically, i.e. chains are constructed at compile-time or prior to execution. In constrast to that, the DTC technique presented in this thesis adapts the chaining decision at runtime, based on workload statistics. Dynamic chaining is also used by Flextream [55], a compilation framework for streaming applications, that targets multi-core platforms. It is highlighted by its capability to dynamically recompile the streaming application to better match the changing availablity of system resources (cores and memory) on the underlying platform. As part of the recompilation process, Flextream groups the actors in the stream graph on cores in a manner that balances load as evenly as possible. While load balancing is also one of the goals of the DTC technique, its primary goal is to reduce latency by removing intermediate queues, rather than react to a changing resource availability.

6.3 Elastic Scaling

The idea of elastic scaling, often also called *auto-scaling*, has been extensively studied in the context of today's IaaS and PaaS cloud platforms. Their ability to rapidly and automatically (de)provision resources, provides the opportunity to build services that adapt their resource allocation to the current workload. A typical cloud-based web application consists of multiple tiers. At each tier a load balancer distributes incoming requests to a pool of servers. A plethora of auto-scaling techniques has been described for this scenario over the course of the past decade. According to [82], the auto-scaling problem can be split into two subproblems, i.e (1) the prediction of future load and (2) making resource provisioning decisions to match that load. In the following, we will discuss some of the *decision making* strategies found in the literature, as they address a problem that of the ES technique in Section 4.6. The decision-making techniques found in literature are commonly based on

(i) thresholds, (ii) reinforcement learning, (iii) control theory, (iv) queueing theory, or a combination of these techniques.

A threshold-based technique tracks whether a performance metric (e.g. CPU utilization, request rate, response time) crosses a user-defined threshold (e.g. CPU utilization larger than 90%) for a certain amount of time, and *reactively* triggers a scale-up or scale-down. It is also common to specify an inactivity time ("cool-down") after each scaling action, in order to deal with inertia. Treshold-based techniques are common in todays commercial cloud platforms[1][2] due to their apparent simplicity and ease-of-use, but have also been explored in research [39, 47]. However, as noted in [39], choosing appropiate thresholds and cool-down periods is tricky and must be done individually for each application in order to avoid oscillations.

Reinforcement learning [113] is a model-free, predictive approach to automated decision making, that learns through the interaction between an agent (i.e. the auto-scaler) and the environment (i.e. the cloud-based web application). At each point in time, the environment has a *state* that the agent can observe. Based on that state, the agent makes a decision that yields a new environment state and a reward, e.g. a reduction in response time. Decision are made basde on a table that maps each observed state to the action that has previously produced the highest reward. To discover the best actions, the agent has to occasionally execute an action that has not been tried yet and observe its reward. The issues of vanilla reinforcement learning for auto-scaling are (a) a bad-initial performance due to required training time and (b) the size of the mapping table that grows exponentially with the number of oberved state variables. In [115] Tesauro et al. address the first issue by a combination of reinforcement learning with a queueing-theoretic model, where the latter is only used during an offline training phase at the beginning. The authors of [102] propose a scheme that reduces the training time by adapting several state-to-decision mappings after each decision. Both publications [115, 102] approach the second issue by using a more space efficient neural network to replace the lookup table.

Control theory [68] studies how a dynamic system can be controlled to obtain a desired behavior and has been applied to the problem of auto-scaling

[1] Amazon Autoscaling, `http://aws.amazon.com/autoscaling/`
[2] Rightscale, `http://www.rightscale.com/`

cloud-based services. Most of the published schemes are *reactive* feedback controllers that observe a performance metric, e.g. response time, and issue actions to correct deviations of the metric from a target value. A simple and common type of feedback controllers are *integral controllers*, that adjust the actuator value, e.g. the number of servers in a tier, proportionally to the deviation of the performance metric from a target value. Lim et al. propose *proportional thresholding* [76, 77] that combines threshold-based auto-scaling with an integral controller. As in plain threshold-based auto-scaling, a change in the actuator value only occurs when the performance metric is outside a target range. Proportional thresholding narrows the desired target range as the number of allocated resources increases. This allows for more precise control, when scaling decisions are most needed, e.g. during a flash crowd. Several papers [97, 6, 22] describe more complex controllers that do not assume a linear relationship between actuator value and performance metric.

In queueing systems, requests arrive at an input queue and are processed by one or more servers. Classical queuing theory [45] models the steady-state behavior of queueing systems and allows to predict important performance aspects such as queue lengths and queue waiting time. The most important characteristics of a queueing system are usually expressed with the Kendall notation in the form X/Y/Z, where X describes the probability distribution of request inter-arrival times, Y describes the probability distribution of request distribution times (also called the *service time* distribution), and Z is the number of servers that process incoming requests. For simple cases, such as exponentially distributed inter-arrival and service times (M/M/Z in Kendall notation), queuing theory gives precise formulae, whereas if such distributions are not known (denoted by "G"), the available queueing formulae are an approximation. The authors of [115, 125] model a single-tier web application with n servers as n queueing systems, each of which receives $1/n$-th of an exponentially distributed request load. Given functions that map request response time to a (possibly negative) economic value, both papers use the queueing model to optimize the number of allocated servers towards an economic optimum. In [115] the queuing-based optimization is however only used during the offline training-phase of a reinforcement learning approach. The authors of [121] describe a scheme for meeting response time targets by elastically scaling multi-tier web-applications. After breaking down the end-to-end response time into per-tier response times, they model each tier as n

G/G/1 servers and determine for each tier the necessary number of servers to achieve the previously determined per-tier response time. This approach bears some resemblance to the ES technique in described in Section 4.6 of this thesis. The main differences are that (1) in [121] the per-tier target response times are assumed to be given, whereas the ES technique chooses the per-tier response times implicitly in a way that reduces the overall required resource provisioning, and (2) the strategy in [121] provisions for expected peak load, while the ES technique provisions only for the currently measured load.

Section 3.3.5 of this thesis has already described several policies for elastic scaling in the area of stream processing engines that execute *UDF-heavy data flows*. In the closely related field of engines for *stream-relational continuous queries*, the StreamCloud [46] engine also exploits cloud elasticity and can dynamically change the parallelism of query operators at runtime. Its elastic reconfiguration protocol takes special care of stateful operators with sliding-window semantics (e.g. a windowed aggregates). The goal of StreamCloud's auto-scaling policy is to avoid bottlenecks by evenly balancing CPU load between operators. The authors of FUGU [50] optimize operator placement towards a desired *host* CPU utilization, by migrating operators at runtime. Migration decisions are made using reinforcement learning. In [49] FUGU is extended to reduce the latency spikes caused by migration of stateful operators, towards fulfilling an end-to-end latency constraint. In this thesis, we have not looked at the problem of operator/subtask placement, as this decision is delegated to a scheduler.

Chapter 7: Conclusion

Distributed stream processing plays an increasingly important role in domains, where large volumes of data need to be analyzed with low latency, such as real-time recommendation of website content [110, 120], social media analytics [118] and network security management [31]. Today's distributed stream processing engines execute UDF-heavy data flows on large, resource managed clusters and provide the technical foundation for such applications. Although application-specific requirements on *latency* are usually implicitly clear, today's engines are optimized towards providing the lowest latency possible, which (1) disregards any "slack" time that the application's latency requirements may leave, and (2) means that the cluster resource provisioning has to be manually determined, in order to meet these requirements with minimal resource provisioning.

This thesis proposes a novel approach to automatically and efficiently enforce application-specific latency requirements in stream processing applications running on resource-managed clusters. The contributions of this thesis pertain to (i) the specification of *latency constraints* in order to capture such requirements, and (ii) *optimization techniques* that enforce latency constraints in a resource-efficient manner, despite variations in workload characteristics, e.g. data arrival rates.

The first contribution is a formal model to specify latency constraints in UDF-heavy data flows. This enables application developers to intuitively express application-specific latency requirements for critical parts of the data flow. The underlying notion of *latency* is based on the *mean* delays caused by data transfer and UDF behavior. It can be measured with low overhead and provides vital input to the optimization techniques.

The main novelty of the thesis consists of three techniques – Adaptive Output Batching (AOB), Dynamic Task Chaining (DTC) and Elastic Scaling (ES) – that collaboratively enforce latency constraints in a resource-efficient and workload-adaptive manner. They are tied into a feedback cycle that begins with the collection of workload statistics about the execution elements of a

UDF-heavy data flow. Based on this data, they determine and exploit the "slack time" the constraints leave, to optimize the execution towards constraint fulfillment. While each technique optimizes a separate aspect of the execution, they are carefully designed to *complement* each other. Specifically, the AOB and DTC techniques create the operating conditions for the ES technique to be successful, and vice versa.

The AOB and DTC techniques reduce the substantial overhead of data shipping present in many application workloads, however they depend on sufficiently provisioned cluster resources. The AOB technique batches emitted data items as much as the constraints allow, which trades off latency against shipping overhead. The evaluation conducted in this thesis has shown that this not only maximizes the amount of useful work the application can perform, but also mitigates the risk of I/O bottlenecks. The latter property is especially important, since I/O bottlenecks quickly lead to constraint violations and inhibit the effects of elastic scaling when they are most needed.

The DTC technique reduces the communication overhead between subtasks running on the same worker. This is achieved by ad-hoc adaptation of the subtask-to-thread mapping, so that the load of the subtasks is distributed fairly over as few threads as possible, without introducing computational bottlenecks. This advances the state-of-the art in stream processing, as the existing engines perform chaining prior to execution, which assumes that either the input/output rates are known prior to execution, or can be reliably obtained in profiling runs [40, 60]. The experimental evaluation shows, that the DTC technique leads to an overall reduction in latency and *jitter*, i.e. variations in latency. Hence, the effects of the DTC technique can mitigate situations with increased latency, e.g. during load bursts, which reduces constraint violations.

The ES technique is a novel decision policy for elastic scaling in distributed stream processing engines, that uses a queueing theoretic model of the application to automatically find the minimum parallelism that still fulfills application-defined latency constraints. In combination with a resource-managed cluster, the engine scheduler can adapt the resource allocation to the current application workload, by deallocating unused workers after a scale-down, and by allocating new workers in response to a scale-up. Compared to a static resource allocation, this improves resource efficiency during

stream processing, especially for applications with large workload variations. Unlike existing elastic scaling policies for stream processing engines that are designed to prevent overload, the ES technique is the first policy of its kind to explicitly model the effects of load on queueing latency.

Although the thesis addresses various research aspects of distributed stream processing with latency constraints, there are several interesting directions for further investigation in this area:

- **Efficient Management of Tail Latency**: As indicated in Section 4.2, latency constraints as defined in this thesis do not specify a hard upper latency bound for every single data item, but an upper bound on *mean* latency. This is at odds with the traditional notion of a *real-time* system [108], where one would optimize for a certain latency percentile, rather than mean latency. The challenge lies within the following observation. Let us assume a sequence of n connected subtasks and channels, each of which has a 95th latency percentile of t. The probability that a data item, which passes through the sequence, incurs a latency lower or equal to $n \times t$ is 0.95^n, given that processing and shipping times are not correlated. Hence, to enforce the p-th latency percentile for the sequence, one has to enforce the q-th percentile for all of its channels and subtasks, where $q > p$. It is an interesting open research question how one can achieve this goal without significant resource over-provisioning.

- **Optimized Subtask Placement**: As indicated in Section 4.3, the optimization techniques presented in this thesis rely on the engine's scheduler to map subtasks to workers. While there is signficant prior work on optimizing such placement decisions [26, 100, 130, 127, 64], it is an interesting research question how a latency-aware operator placement strategy can be designed, that migrates subtasks to increase resource efficiency. First publications have already picked up this idea [49].

- **Proactive Elastic Scaling**: In many stream processing applications, the data arrival rate varies with a certain regularity or pattern. The ES technique proposed in this thesis is however purely reactive, i.e. it cannot anticipate recurring workload patterns. A proactive approach to elastic scaling could reduce the number latency constraint violations due to foreseeable changes in the workload. This shares some

resemblance to the challenge of proactive auto-scaling of cloud-based applications, which has been extensively studied [23, 62, 28, 42, 107]. It is an interesting research question whether and how these methods can be adapted for elastic scaling of distributed stream processing applications.

Appendix A: Workload Statistics

The following table lists and describes the workload statistics required from Statistics Reporters for each constrained subtask and channel.

Statistic	Description
$l_v \in \mathbb{R}^+$	*Mean subtask latency* of subtask v, which is the mean of the subtask latencies over all the data items passed to v during the last adjustment interval, via any of its ingoing channels.
$l_e \in \mathbb{R}^+$	*Mean channel latency* of channel e, which is the mean of the channel latencies over all the data items sent through e during the last adjustment interval.
$obl_e \in \mathbb{R}^+$	*Mean output batch latency* of channel e's, which is defined as the mean delay data items incurred due to output batching during the last adjustment interval. (see Section 4.5).
$\overline{S_v}$, $Var(S_v) \in \mathbb{R}^+$	Mean and variance of subtask v's *service time*. Service time is the queueing theoretic notion of how long a subtask is busy with a data item. Service time is sampled randomly over all data items read from the subtask's input queue, regardless of the channel or stream they have originated from. For subtasks with ingoing channels from only a single stream, service time is equivalent to read-ready subtask latency. S_v is a random variable with an unknown probability distribution.
$\overline{A_v}$, $Var(A_v) \in \mathbb{R}^+$	Mean and variance of the data item *inter-arrival time* at subtask v's input queue. The inter-arrival time is sampled over all data items arriving at the queue, regardless of the channel or stream they have originated from. A_v is a random variable with an unknown probability distribution.

$w_v \in \mathbb{R}^+$	*Mean queue waiting time* of subtask v, which is defined as the mean delay data items have incurred at the input queue of subtask v during the last adjustment interval. Mean queue waiting time is sampled over all data items arriving at the input queue of v, regardless of the channel or stream they have originated from.
Derived from above workload statistics.	
$c_X = \frac{\sqrt{Var(X)}}{\overline{X}}$	Coefficient of variation of $X \in \{S_v, A_v\}$.
$\lambda_v = \frac{1}{\overline{A_v}}$	Data item *arrival rate* at subtask v.
$\mu_v = \frac{1}{\overline{S_v}}$	*Service rate* of subtask v, i.e. its maximum processing rate.
$\rho_v = \lambda_v \overline{S_v}$	*Utilization* of subtask v.

Appendix B: Cluster Testbed

All experiments were conducted on a compute cluster with 130 available commodity servers as well as a master node. Each server is equipped with an Intel Xeon E3-1230 V2 3.3 GHz (four real CPU cores plus hyper-threading activated) and 16 GB RAM. The servers were connected via regular Gigabit Ethernet links to a central HP ProCurve 1800-24G switch.

On the software side, all nodes run Linux (kernel version 3.3.8) as well as and Java 1.7.0_13. To maintain clock synchronization among the workers, each server launched a Network Time Protocol (NTP) daemon. During the entire experiments, the measured clock skew was below 2ms among the servers. Moreover, Apache Hadoop YARN[1] 2.6.0 is installed on the cluster.

[1] `http://hadoop.apache.org/`

Bibliography

[1] D.J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2): 120–139, 2003.

[2] D.J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.H. Hwang, W. Lindner, A.S. Maskey, A. Rasin, E. Ryvkina, et al. The design of the Borealis stream processing engine. In *Second Biennial Conference on Innovative Data Systems Research*, CIDR '05, pages 277–289, 2005.

[3] Adaptive Computing, Inc. TORQUE Resource Manager - Adaptive Computing. `http://www.adaptivecomputing.com/products/open-source/torque/`, 2015.

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

[5] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review*, 38(4):63–74, 2008.

[6] Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *Network Operations and Management Symposium*, NOMS'12, pages 204–212. IEEE, 2012.

[7] Amazon Web Services, Inc. Amazon Kinesis. `http://aws.amazon.com/de/kinesis/`, January 2015.

[8] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani.

Spc: A distributed, scalable platform for data mining. In *Proceedings of the 4th international Workshop on Data mining standards, services and platforms*, pages 27–37. ACM, 2006.

[9] Henrique Andrade, Buğra Gedik, and Deepak Turaga. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, 2014.

[10] Apache Software Foundation. Apache Flink: Home. `http://flink.apache.org/`, January 2015.

[11] Apache Software Foundation. Welcome to Apache Hadoop! `http://hadoop.apache.org/`, January 2015.

[12] Apache Software Foundation. Samza. `http://samza.apache.org/`, January 2015.

[13] Apache Software Foundation. Storm, distributed and fault-tolerant realtime computation. `http://storm.apache.org/`, January 2015.

[14] Arvind Arasu, Shivnath Babu, and Jennifer Widom. CQL: A language for continuous queries over streams and relations. In *Database Programming Languages*, volume 2921 of *Lecture Notes in Computer Science*, pages 1–19. Springer Berlin Heidelberg, 2004.

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

[16] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[17] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *Proceedings. 20th International Conference on Data Engineering*, pages 350–361. IEEE, March 2004.

[18] Roger S Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. Consistent streaming through time: A vision for event stream processing. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR'07)*, pages 363–374, 2006.

[19] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3), 2013.

[20] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: A programming model and execution framework for web-scale analytical processing. In *Proc. of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 119–130. ACM, 2010. ISBN 978-1-4503-0036-0.

[21] Dominic Battré, Matthias Hovestadt, Björn Lohrmann, Alexander Stanik, and Daniel Warneke. Detecting bottlenecks in parallel DAG-based data flow programs. In *Proc. of the 2010 IEEE Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '10, pages 1–10. IEEE, 2010.

[22] Peter Bodık, Rean Griffith, Charles Sutton, Armando Fox, Michael Jordan, and David Patterson. Statistical machine learning makes automatic control practical for internet datacenters. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'12. USENIX Association, 2009.

[23] Peter Bodık, Rean Griffith, Charles Sutton, Armando Fox, Michael Jordan, and David Patterson. Statistical machine learning makes automatic control practical for internet datacenters. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, pages 12–12, 2009.

[24] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proc. of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1151–1162. IEEE, 2011. ISBN 978-1-4244-8959-6.

[25] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker

White. Cayuga: a high-performance event processing engine. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1100–1102. ACM, 2007.

[26] Michael Cammert, Christoph Heinz, Jürgen Krämer, Bernhard Seeger, Sonny Vaupel, and Udo Wolske. Flexible multi-threaded scheduling for continuous queries over data streams. In *IEEE 23rd International Conference on Data Engineering Workshop*, pages 624–633. IEEE, 2007.

[27] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases*, pages 838–849. VLDB Endowment, 2003.

[28] Eddy Caron, Frédéric Desprez, and Adrian Muresan. Pattern matching based forecast of non-periodic repetitive behavior for cloud clients. *Journal of Grid Computing*, 9(1):49–64, 2011.

[29] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736. ACM, 2013.

[30] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[31] Cisco Systems, Inc. Services for Security - Cisco. `http://www.cisco.com/c/en/us/products/security/service-listing.html`, January 2015.

[32] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce Online. In *Proc. of the 7th USENIX conference on Networked systems design and implementation*, NSDI '10, pages 21–21. USENIX Association, 2010.

[33] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. Adaptive stream processing using dynamic batch sizing. In *Proc. of the ACM Symposium on Cloud Computing*. ACM, 2014.

[34] Thomas H Davenport and DJ Patil. Data scientist. *Harvard Business Review*, 90:70–76, 2012.

[35] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[36] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144. ACM, 2014.

[37] Vasant Dhar. Data science and prediction. *Commun. ACM*, 56(12): 64–73, December 2013.

[38] Klaus R Dittrich, Stella Gatziu, and Andreas Geppert. The active database management system manifesto: A rulebase of adbms features. In *Rules in Database Systems*, pages 1–17. Springer, 1995.

[39] Xavier Dutreilh, Nicolas Rivierre, Aurlien Moreau, Jacques Malenfant, and Isis Truck. From data center resource allocation to control theory and back. In *2010 IEEE 3rd International Conference on Cloud Computing (CLOUD'10)*, pages 410–417. IEEE, 2010.

[40] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1123–1134. ACM, 2008.

[41] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463, 2014.

[42] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Manage-*

ment (CNSM), 2010 International Conference on, pages 9–16. IEEE, 2010.

[43] Google, Inc. Google Cloud Dataflow. `https://cloud.google.com/dataflow/`, January 2015.

[44] Michael I Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S Meli, Andrew A Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, et al. A stream compiler for communication-exposed architectures. In *ACM SIGPLAN Notices*, volume 37, pages 291–303. ACM, 2002.

[45] Donald Gross. *Fundamentals of queueing theory*. John Wiley & Sons, 2008.

[46] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, 2012.

[47] Rui Han, Li Guo, Moustafa M Ghanem, and Yike Guo. Lightweight resource scaling for cloud applications. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'12),*, pages 644–651. IEEE, 2012.

[48] Bingsheng He, Mao Yang, Zhenyu Guo, Rishan Chen, Bing Su, Wei Lin, and Lidong Zhou. Comet: batched stream processing for data intensive distributed computing. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 63–74. ACM, 2010.

[49] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proc. of the 8th ACM International Conference on Distributed Event-Based Systems*. ACM, 2014.

[50] Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. Auto-scaling techniques for elastic data stream processing. In *2014 IEEE 30th International Conference on Data Engineering Workshops*. IEEE, 2014.

[51] Joseph L Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M Tilbury. *Feedback control of computing systems*. John Wiley & Sons, 2004.

[52] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. USENIX Association, 2011.

[53] Martin Hirzel, Henrique Andrade, Bugra Gedik, Vibhore Kumar, Giuliano Losa, MMH Nasgaard, R Soule, and KL Wu. SPL stream processing language specification. Technial Report RC24, 2009.

[54] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46:1–46:34, March 2014.

[55] Amir H Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*, pages 214–223. IEEE, 2009.

[56] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[57] Jeong-Hyon Hwang, Magdalena Balazinska, Alex Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 779–790. IEEE, 2005.

[58] IBM, Inc. IBM Load Leveler: User's Guide, September 1993.

[59] IBM, Inc. Managing big data for smart grids and smart meters. `http://public.dhe.ibm.com/common/ssi/ecm/en/imw14628usen/IMW14628USEN.PDF`, January 2015.

[60] IBM, Inc. Ibm infosphere streams. `http://www.ibm.com/software/products/en/infosphere-streams`, February 2015.

[61] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, March 2007.

[62] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Computer Systems*, 28(1):155–162, 2012.

[63] Raj Jain and Imrich Chlamtac. The p2 algorithm for dynamic calculation of quantiles and histograms without storing observations. *Communications of the ACM*, 28(10):1076–1085, 1985.

[64] Rohit Khandekar, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Joel Wolf, Kun-Lung Wu, Henrique Andrade, and Buğra Gedik. Cola: Optimizing stream processing applications via graph partitioning. In *Middleware 2009*, pages 308–327. Springer, 2009.

[65] J. F. C. Kingman. The single server queue in heavy traffic. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 57, pages 902–904. Cambridge University Press, 1961.

[66] Brent K Kingsbury. The network queueing system. Technical report, Sterling Software, Inc., 1986.

[67] Donald E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison Wesley, 3rd edition, 1997.

[68] BJ Kuo and F Golnaraghi. *Automatic control systems*. Wiley, 2009.

[69] Wang Lam, Lu Liu, STS Prasad, Anand Rajaraman, Zoheb Vacheri, and AnHai Doan. Muppet: Mapreduce-style processing of fast data. *Proceedings of the VLDB Endowment*, 5(12):1814–1825, 2012.

[70] Wang Lam, Lu Liu, Sts Prasad, Anand Rajaraman, Zoheb Vacheri, and AnHai Doan. Muppet: Mapreduce-style processing of fast data. *Proc. VLDB Endow.*, 5(12):1814–1825, August 2012. ISSN 2150-8097.

[71] James R Larus and Michael Parkes. Using cohort scheduling to enhance server performance. In *ACM SIGPLAN Notices*, volume 36, pages 182–187. ACM, 2001.

[72] Edward Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[73] Jure Leskovec, Anand Rajaraman, and Jeffrey D Ullman. *Mining of massive datasets*. Cambridge University Press, 2011.

[74] Daniel Lezcano, Serge Hallyn, and Stéphane Graber. Linux Containers. `https://linuxcontainers.org/`, 2015.

[75] David A Lifka. The ANL/IBM SP scheduling system. In *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 295–303. Springer Berlin Heidelberg, 1995.

[76] Harold C Lim, Shivnath Babu, Jeffrey S Chase, and Sujay S Parekh. Automated control in cloud computing: challenges and opportunities. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 13–18. ACM, 2009.

[77] Harold C Lim, Shivnath Babu, and Jeffrey S Chase. Automated control for elastic storage. In *Proceedings of the 7th international conference on Autonomic computing*, pages 1–10. ACM, 2010.

[78] Michael J Litzkow, Miron Livny, and Matt W Mutka. Condor - A hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, pages 104–111. IEEE, 1988.

[79] Lockheed Martin, Inc. Lockheed Martin Introduces Open Source Software Platform for Simpler Real-Time Analytics Processing and Analysis. `http://www.lockheedmartin.com/us/news/press-releases/2015/january/isgs-streamflow-open-source-182015.html`, January 2015.

[80] Björn Lohrmann, Daniel Warneke, and Odej Kao. Massively-parallel stream processing under QoS constraints with Nephele. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 271–282, New York, NY, USA, 2012. ACM.

[81] Björn Lohrmann, Peter Janacik, and Odej Kao. Elastic stream processing with latency guarantees. In *The 35th International Conference on*

Distributed Computing Systems (ICDCS 2015), Under review. IEEE, 2015.

[82] Tania Lorido-Botrán, José Miguel-Alonso, and Jose Antonio Lozano. Auto-scaling techniques for elastic applications in cloud environments. *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09*, 12:2012, 2012.

[83] D Luckham. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. Technical report, Stanford University, 1996.

[84] David Luckham. *The power of events*, volume 204. Addison-Wesley Reading, 2002.

[85] Ming Mao and Marty Humphrey. A performance study on the vm startup time in the cloud. In *2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, pages 423–430. IEEE, 2012.

[86] Alessandro Margara and Gianpaolo Cugola. Processing flows of information: From data stream to complex event processing. In *Proceedings of the 5th ACM International Conference on Distributed Event-based System*, DEBS '11, pages 359–360, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0423-8. doi: 10.1145/2002259.2002307. URL `http://doi.acm.org/10.1145/2002259.2002307`.

[87] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the. net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706. ACM, 2006.

[88] Maged Michael, Jose E. Moreira, Doron Shiloach, and Robert W. Wisniewski. Scale-up x Scale-out: A case study using Nutch/Lucene. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, 2007.

[89] Microsoft, Inc. Microsoft Azure Cloud Services - Stream Analytics. `http://azure.microsoft.com/en-us/services/stream-analytics/`, January 2015.

[90] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *First Biennial Conference on Innovative Data Systems Research*, CIDR '03, pages 245–256, 2003.

[91] D.G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: A universal execution engine for distributed data-flow computing. In *Proc. of the 8th USENIX conference on Networked systems design and implementation*, NSDI '11, pages 9–9. USENIX Association, 2011.

[92] Z. Nabi, R. Wagle, and E. Bouillet. The best of two worlds: Integrating ibm infosphere streams with apache yarn. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 47–51, Oct 2014.

[93] John Nagle. Congestion control in IP/TCP internetworks. `http://tools.ietf.org/html/rfc896`, 1984.

[94] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 170–177. IEEE, 2010.

[95] Jakob Nielsen. *Usability engineering*. Elsevier, 1994. ISBN 978-0125184069.

[96] Oracle, Inc. Utilities and Big Data: A Seismic Shift is Beginning. `http://www.oracle.com/us/industries/utilities/utilities-big-data-wp-2012876.pdf`, January 2015.

[97] Pradeep Padala, Kai-Yuan Hou, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 13–26. ACM, 2009.

[98] Bo Pang and Lillian Lee. Opinion mining and sentiment analysis. *Foundations and trends in information retrieval*, 2(1-2):1–135, 2008.

[99] Gregory F Pfister. *In search of clusters*. Prentice-Hall, Inc., 1998.

[100] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, pages 49–49. IEEE, 2006.

[101] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 1–14. ACM, 2013.

[102] Jia Rao, Xiangping Bu, Cheng-Zhong Xu, Leyi Wang, and George Yin. Vconf: a reinforcement learning approach to virtual machines auto-configuration. In *Proceedings of the 6th international conference on Autonomic computing*, pages 137–146. ACM, 2009.

[103] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 7:1–7:13. ACM, 2012.

[104] Kai-Uwe Sattler and Felix Beier. Towards elastic stream processing: Patterns and infrastructure. In *First International Workshop on Big Dynamic Distributed Data (BD3)*, pages 49–54, 2013.

[105] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 351–364. ACM, 2013.

[106] Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. Cache aware optimization of stream programs. *ACM SIGPLAN Notices*, 40(7):115–126, 2005.

[107] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 5. ACM, 2011.

[108] Kang G. Shin and Parameswaran Ramanathan. Real-time computing: A new discipline of computer science and engineering. *Proceedings of the IEEE*, 82(1):6–24, 1994.

[109] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

[110] Spotify, GmbH. How Spotify Scales Apache Storm. `https://labs.spotify.com/2015/01/05/how-spotify-scales-apache-storm/`, January 2015.

[111] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *Proceedings of the 23rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 263–274. ACM, 2004.

[112] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34 (4):42–47, 2005.

[113] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[114] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases*, volume 29, pages 309–320. VLDB Endowment, 2003.

[115] Gerald Tesauro, Nicholas K Jong, Rajarshi Das, and Mohamed N Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *IEEE International Conference on Autonomic Computing (ICAC'06)*, pages 65–73. IEEE, 2006.

[116] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002.

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

[118] Twitter, Inc. A Storm is coming: more details and plans for release. `https://blog.twitter.com/2011/storm-coming-more-details-and-plans-release`, August 2011.

[119] Twitter, Inc. New Tweets per second record, and how! `https://blog.twitter.com/2013/new-tweets-per-second-record-and-how`, August 2013.

[120] UBM, LLC. Yahoo Talks Apache Storm: Real-Time Appeal. `http://www.informationweek.com/big-data/big-data-analytics/yahoo-talks-apache-storm-real-time-appeal/d/d-id/1316840`, October 2014.

[121] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 3(1):1, 2008.

[122] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 5:1–5:16. ACM, 2013.

[123] Ovidiu Vermesan and Peter Friess, editors. *Internet of Things Applications - From Research and Innovation to Market Deployment*. The River Publishers, 2014.

[124] Stratis D Viglas and Jeffrey F Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 37–48. ACM, 2002.

[125] Daniel Villela, Prashant Pradhan, and Dan Rubenstein. Provisioning servers in the application tier for e-commerce systems. *ACM Transactions on Internet Technology (TOIT)*, 7(1):7, 2007.

[126] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. *ACM SIGOPS Operating Systems Review*, 35(5):230–243, 2001.

[127] Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer. Soda: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Middleware 2008*, pages 306–325. Springer, 2008.

[128] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418. ACM, 2006.

[129] Yingjun Wu and Kian-Lee Tan. Chronostream: Elastic stateful stream computation in the cloud. In *2015 IEEE 31st International Conference on Date Engineering (forthcoming)*, 2015.

[130] Ying Xing, Jeong-Hyon Hwang, Uğur Çetintemel, and Stan Zdonik. Providing resiliency to load variations in distributed stream processing. In *Proceedings of the 32nd international conference on Very large data bases*, pages 775–786. VLDB Endowment, 2006.

[131] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[132] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

[133] Xiaolan J Zhang, Henrique Andrade, Buğra Gedik, Richard King, John Morar, Senthil Nathan, Yoonho Park, Raju Pavuluri, Edward Pring,

Randall Schnier, et al. Implementing a high-volume, low-latency market data processing system on commodity hardware using ibm middleware. In *Proceedings of the 2nd Workshop on High Performance Computational Finance*, page 7. ACM, 2009.