FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Dependable Event Processing over High Volume Data Streams

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Andrea Floh
Matrikelnummer 0725144

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.-Prof. Dr. Schahram Dustdar
Mitwirkung: Univ.-Ass. Dr. Waldemar Hummer

Wien, 17.03.2014

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Dependable Event Processing over High Volume Data Streams

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering & Internet Computing

by

Andrea Floh
Registration Number 0725144

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.-Prof. Dr. Schahram Dustdar
Assistance: Univ.-Ass. Dr. Waldemar Hummer

Vienna, 17.03.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Andrea Floh
Reinpolz 11, 3962 Heinrichs

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

(Ort, Datum) (Unterschrift Verfasserin)

Abstract

The number of event processing systems is increasing more and more. In contrast to traditional systems, those event processing systems do not handle persistent data, which is mostly stored in databases, but instead they have to deal with events, which are received continuously over various communication channels and should be processed more or less immediately. Complex computations (e.g., mathematical calculations, or pattern detection) in terms of queries have to be performed for these events. The frequency of the arriving events is not necessarily steady. In fact, the system has do deal with ups and downs, which can influence the data volume heavilyload fluctuations, which can heavily influence the requirements concerning processing power. Especially in times of high data volume the processing systems have to deal with a huge number of events and should be able to manage these phases.

Different approaches of dealing with high volume data streams have been studied, but their applicability and efficiency may vary depending on the application scenario. General approaches or guidelines on how to treat such overload are not available.

This thesis covers strategies to handle phases of high data volume on event streams for a single event processing node. Currently, there are three established strategies for coping with loads that are too high, which have been used to treat overload situations caused by high data volume: load shedding, deferred execution and forwarding. These strategies are discussed and their applicability for different types of queries is evaluated. To that end, a taxonomy of queries in event processing systems is elaborated. The taxonomy covers different dimensions like the type of processing operation and the scope of the query.

Based on the features of the strategies and the different query types, the applicability of the strategies is analyzed in theory and an evaluation is performed to support the analysis. The strategies are implemented in a generic way and are integrated into the WS-Aggregation framework for the evaluation. This framework for distributed and event-based aggregation of web services data has been developed by the Distributed Systems Group at the Vienna University of Technology. Furthermore, the results of the evaluation are used to determine the strength of the influence of the different applicability criteria and to formulate problem statements for further research.

Kurzfassung

Ereignis-basierte Systeme (*event processing systems*) haben in letzter Zeit immer mehr an Bedeutung gewonnen und sind immer häufiger im Einsatz. Im Gegensatz zu traditionellen Systemen, welche persistierte Daten verarbeitet haben, die oftmals in eine Datenbank eingefügt und erst danach verarbeitet wurden, müssen Ereignis-basierte Systeme mit laufend ankommenden Daten umgehen können und diese meist auch umgehend verarbeiten können. Die Verarbeitung umfasst dabei komplexe Berechnungen, welche durchaus gewisse Ressourcen benötigen. Die Ankunftsrate der Ereignisse ist allerdings selten gleichbleibend. Vielmehr müssen die Systeme mit starken Schwankungen umgehen und sollten Spitzenfrequenzen verkraften, welche die Durchschnittsfrequenz um ein Vielfaches übersteigen. Speziell zu solchen Spitzenzeiten, wenn die Anzahl der ankommenden Ereignisse und deren Größe ein sehr hohes Datenvolumen ergeben, sollen diese Systeme trotzdem verlässlich arbeiten.

Verschiedene Ansätze für den Umgang mit solch hohen Datenvolumina wurden in der bisherigen Forschung vorgestellt, allerdings oft nur im Kontext von speziellen Anwendungsgebieten. Die Anwendbarkeit und Effizienz der Strategien ist aber je nach Einsatzzweck unterschiedlich. Generelle Richtlinien, welche Strategien in welchen Situationen verwendet werden sollen, sind derzeit nicht verfügbar.

Diese Arbeit behandelt Vorgehensweisen für den Umgang mit hoher Last auf einem einzelnen Verarbeitungsknoten. Aktuell sind hierfür vor allem drei Strategien populär: Lastabwurf (*load shedding*, verzögerte Verarbeitung (*deferred execution*) und Weiterleitung (*forwarding*). Im Rahmen der Arbeit werden diese Strategien vorgestellt und analysiert. Im Speziellen wird dabei untersucht, für welche Verarbeitungsszenarien diese Praktiken verwendet werden können. Hierfür wird auch eine Klassifizierung der Abfragen, die zur Verarbeitung verwendet werden, erstellt. Die Klassifizierung erfolgt dabei nach zwei Dimensionen: dem Operationstyp und dem Umfang der Abfrage.

Basierend auf den Eigenschaften der verschiedenen Abfragetypen wird die Einsetzbarkeit der Praktiken theoretisch analysiert und danach mit einer praktischen Evaluierung belegt. Die Strategien werden hierzu generisch implementiert und in das Framework WS-Aggregation integriert. Dieses Framework für die verteilte, ereignis-basierte Aggregation von Web-Service Daten wurde am Arbeitsbereich für Verteilte Systeme der Technischen Universität Wien entwickelt. Die Ergebnisse der Evaluierung werden im Speziellen auch genutzt um die verschiedenen Faktoren der Anwendbarkeit der Strategien zu gewichten und um festzustellen, in welchen Bereichen noch weitere Forschungsarbeit benötigt wird.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

Introduction

The motive of this chapter is to give a basic overview of the area of event processing and to explain the contributions that will be gained by this thesis. First an introduction to event processing is given. The focus lies on the context of the thesis as well as the explanation of several terms. It then leads on to the motivation and the reasoning for the importance of this thesis, followed by the concrete problem formulation with a reference to the current situation. Afterwards the concrete objectives of the work and its scope are defined, while other research areas are delineated. According to that, the methodological approach to fulfill the objectives is described. The chapter is closed up with a description of the structure of the remaining thesis.

1.1 Event processing

Event Processing (EP) is a computing paradigm that uses an event-driven approach [32]. In contrast to time-driven or request-driven applications, actions are not initiated because of a request by a client or a special point in time. Actions are triggered due to received events. Event-Based System (EBS) [96] are based on the event processing paradigm, which leads to a decoupling of the components in the system [89]. An *Event* is an occurrence within the system or the domain [47]. It represents a change in the current state of the system. The information on the occurrence is wrapped in a programming entity and is used for further processing. Based on the wrapped information, the receiver of the event can take the correct actions. Mostly, both the occurrence itself and the programming entity are called 'event'.

Events arise over time with different volumes and intervals. The traditional approach has been to store information and data of interest in a database in general. These persistent data sets have then been queried using the Database Management System (DBMS) and the results have been used for further processing or analysis. Nowadays saving those huge amounts of data and performing the queries later on is not a viable option. Events have to be processed immediately, sometimes even in real-time.

As a new approach, continuous queries are used for the processing. *Continuous Queries* [123] are applied once, but they are executed continuously as events are received over the data stream [13]. Thus events can also be processed in real time.

According to [89], event processing is divided into four different logical layers: in the *Event Generator* layer events are generated based on perceived or available information. The next layer is the *Event Channel*, which transports the generated events between the components. In the *Event Processing* layer events are received and analyzed. According to rules, further actions are taken: service invocations, invocations of business processes, notifications, publication of events and so forth. Finally, the *Downstream Event-driven Activity* layer includes all the activities that were triggered by the event processing. The physical structure of an event processing system does not necessarily need to reflect this logical division.

EBSs differ on the communication infrastructure that is used to transport the events between the components as well as on the specific purpose. The purpose of such an application can vary from monitoring to information dissemination or to realize ubiquitous systems.

As for the communication infrastructure, data streams are one of the possibilities that can be used. A *Data Stream* is an input source, where data, i.e. events, arrive on-line. The stream is defined as unbounded and the arriving data elements have an internal order. The order of the elements may not coincide with the order of the arrival, but the system itself cannot affect the order of arrival in a data stream or across several streams. Data arrives at a very high rate and after the processing of an element, it cannot be accessed easily anymore unless it is archived. [11, 29, 97]

Even though the size of events most often is relatively small, the high frequency of arriving events leads to a huge volume of data that has to be processed. Typical examples for event processing applications can be found in the financial sector. Stock markets or bank accounts are monitored and each transaction triggers an event. Even more events are generated in Radio-Frequency Identification (RFID) applications, when each product in a big warehouse is tagged with a RFID tag to trace all changes and movements. Another area with a huge amount of incoming events are Massive Multiplayer Online Games (MMOG). As popular games gain more and more users, the amount of events per second gets extremely high at peak times, for example after working hours. In Table 1.1 figures for these applications are shown. Whereas the events in banking and RFID applications show the sum for a whole day (including ups and downs), the number for MMOG games represents the amount of events per second at a peak.

Table 1.1: Event Data Rates [74]

application type	amount of events	time scope
banking applications	400 million events per day	average
RFID applications	4 trillion events per day	average
MMOG games	1 million events per second	peak

These input streams, having events arriving with a high frequency, also lead to the importance of dependability in EBSs. Laprie describes dependability in [76] as following: *Computer system dependability is the quality of the delivered service such that reliance can justifiably be placed on this service.* Therefore, whether an EBSs is dependable or not is determined by the

viewpoint. The EBS can either be seen from the perspective of an end user or from other systems that rely on the EBS. The EBS works dependable as long as the Quality of Service (QoS) criteria of the systems based on it are met. If an EBS is used by different systems, it can be dependable and undependable at the same time as their QoS criteria may be different. If a system is fault-tolerant, the EBS can still be rated as dependable if some errors happen during the event processing. Whereas for a system that is not fault-tolerant, the EBS is already seen as undependable. To ensure dependability it is not sufficient to meet the QoS criteria of another system on average, but the processing result must also comply with the criteria at peak times.

1.2 Motivation

Event processing is a computing paradigm that evolved based on existing technologies and procedures, and due to changing challenges over time. Currently one can see a trend for event-based systems. Companies and organizations are growing larger and the number of relevant events that occur in their systems increases evermore. To cope with the handling and monitoring of all those events, efficient computer systems are needed. Moreover, the computational processing enables a company to extract hidden information by using mining techniques.

One main reason why the event-based communication is getting more popular as an architectural style in business applications is, that it matches the real world context of the applications. Events represent state changes in the environment of the company and companies react to these events. As events can be perceived automatically in different ways, i.e. through sensor systems or notification services, an EBS can react to those events immediately and is therefore suited best.

According to [32], different types of current business pressures lead to the decision to use event processing as an architectural style for a business application, see Figure 1.1.

The system requirements *timeliness*, *agility* and *information availability* are a consequence of current pressures like increasing competition, globalization and so forth. Event processing is well-suited to meet those requirements. Timeliness can be enabled by real-time event processing. Agility is achieved as the system can provide various ways to treat occurred events. As event processing supports the decoupling of components, components can also be changed easily or additional components for event processing can be added. Information availability can be guaranteed as components that require notifications on certain events can subscribe for them and get informed as soon as events are published.

This common trend to use event-based systems makes this area a promising research area as there is still a lack of generic approaches. Event-based systems are used for a lot of different purposes [47,96]: monitoring, observation, information dissemination, enterprise application integration, ubiquitous systems, RFID systems and so forth. Currently several research prototypes and frameworks are developed, but few systems are used commercially in real business contexts. The prototypes usually support event processing for different purposes, but there is no general approach to handle temporary overload during peak times.

In such times of high volume the number of events that have to be processed per second can be in the range of thousands and up to one million [74, 130]. If an EBSs is designed in a way so that it can handle such amounts of events occurring during peak times in general, the

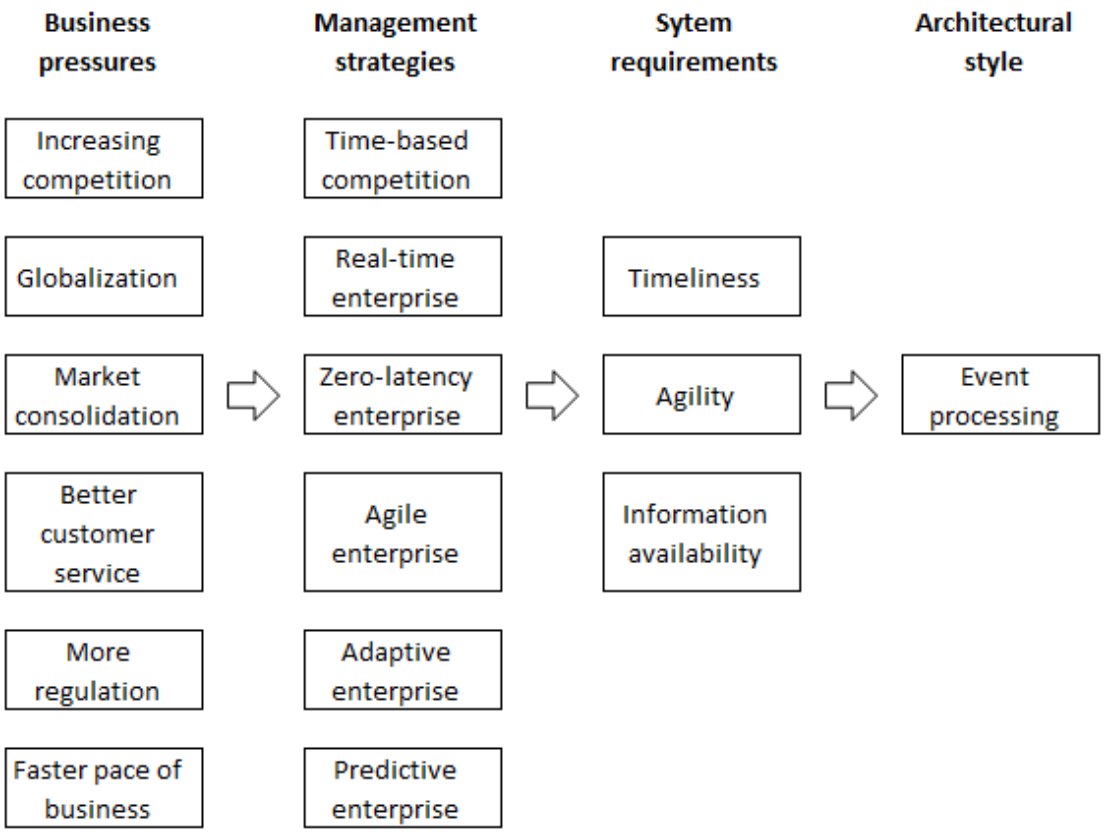


Figure 1.1: Business Pressures Leading to Event Processing. [32]

system will not utilize its resources during the average load times. As such peaks occur only sometimes, the system would not be busy most of the time, so oversizing the EBS would a waste of resources. On the other hand, if an EBS is capable to handle the amount of incoming events only on average, which means only if the events arrive evenly distributed over the time, the system cannot guarantee the reliable processing of the events at peak times. It will rather lead to processing errors or a crash of the system in case an overload.

To enable dependable event handling, the systems must be capable of dealing with periods of high volume data streams, which means a high frequency of incoming events and / or events having a big size. Strategies use different approaches for this purpose, such as providing more resources or adopting to a simpler handling of the events, which naturally means that the results of the EBS are temporarily less accurate. In either case, if the strategy itself or the adaption to using a strategy has an impact on the accuracy of a service, the dependability is threatened. If such a strategy is used, the impact it has on the accuracy of the result, must be determined in advance. Only in that case other systems or users can decide whether the system does satisfy their standards for dependability or not.

Currently, most of the used strategies are selected depending on the application and are specifically customized during development. Sometimes further intervention to handle critical situation is required during run-time. This means, an additional effort during the development of an EBS is required, because the load handling strategies have to be selected and customized for the requirements of the application.

To avoid that this effort in the development arises for every application, a generic approach should be devised. The desirable solution would be a generic approach that provides different load handling strategies for certain types of applications, which can be directly integrated. The automation of the identification of the best strategy would be another improvement, because then the strategies do not need to be analyzed during the development of the application. The strategy resulting in the best results would be picked automatically. The best case scenario would be to provide automated monitoring and analysis and that the strategy as well as the information on the quality of service is regularly adjusted. Thus, the treatment of high load could be outsourced and the development of event-based applications could again focus on the needs of the application itself and not on the general conditions of load handling.

1.3 Problem Formulation

In contrast to usual applications of the past, which processed persistent and mostly static data, nowadays more and more applications need to handle continuous data over streams [11]. In a Event Processing System (EPS) data represents events that occurred in the context of the system. Event Processing Agents (EPAs) are used for the processing of the received events as they perform continuous queries. As in ordinary systems, one EPA is not sufficient to handle all events or to calculate complex queries, so several EPAs are assembled in an Event Processing Network (EPN) and are used to process the events collectively. In this EPN, the processing nodes may work in parallel mode, or the result for a complex query is calculated by nodes, that form a directed acyclic graph and calculate partial results for the traversing events successively.

Since events do not occur homogeneously distributed over time, there are peaks that lead to periods of high load for single EPAs and therefore for the whole EPN. Event processing systems have to be capable of handling such peaks as well as the caused overload to enable dependable event processing. One single EPA may become the bottleneck of an EPN, if it cannot deal with the amount of received events.

Current EBSs are mostly built from scratch or use a simple framework as basis. Those frameworks usually do not provide sophisticated mechanisms to handle high load. So those systems have to handle overload by custom tailored solutions, which are usually adapted to the executed query type or even to the particular query. Therefore, the approaches to handle high load can hardly be reused in other systems.

The identified problem is, that at the present time providing an EBSs that can handle high load, causes overhead as no general approach is available. Moreover, the problem is not one-dimensional but multidimensional. The first dimension is the type of strategy that should be used, which is certainly related to the application and to the continuous queries. The next dimension is the scope of application and where the strategy should be applied. It can either be a strategy that comprises the whole EPN or one that is applied to a single EPA. In the latter case

a further decision has to be made, namely which EPA the strategy should be applied on. As the EPN is an acyclic graph formed by the EPAs, a strategy will be more or less efficient depending on where is is applied. And the third dimension is the adaptability. Poor adaptability is provided if the strategies have to be selected and applied during design-time or the deployment. Better adaptability is given when the strategies can be applied dynamically during run-time.

1.4 Aim of the Work

The field of research is rather wide, so this work will focus on a basic step towards a generally applicable solution for handling high load in EBSs. The results can then be used for further studies.

The goal of this work is to provide load handling strategies and to show how the strategies and their usage can be further improved. This includes presenting a concept on how to integrate load handling strategies, which can be applied dynamically during run-time, into an existing framework. To prove the concept, a sample of promising load handling approaches is implemented and integrated into a framework according to the concept. By implementing a simple EBS with the framework, the strategies can be applied to EPAs of the system. A comprehensive evaluation of the different strategies is used to gain insights on the applicability of the strategies.

A secondary goal of the thesis is a taxonomy of continuous queries. The taxonomy will uncover distinguishing characteristics of queries. Therefore, the taxonomy can be used to determine the different challenges that a generally applicable approach for handling high loads has to face. Based on the taxonomy, queries are selected that will be used for the comprehensive evaluation. The results of the evaluation are going to underline the analysis and assessment of the strategies.

The evaluation itself will provide information for the implementation of further strategies and decision criteria for algorithms, which determine suitable strategies for EBSs automatically.

Some potential research areas are not covered by this thesis. The thesis just covers strategies that are applied to one EPA, not to the whole EPN. Further on, it does not analyze on which EPA the strategy should be applied. The strategies will be applied to an EPA explicitly, so it does also not cover the monitoring of EPAs. As there are currently no published studies that indicate, which criteria are relevant to decide on a good load handling strategy, no automated decision algorithm will be implemented, but the evaluation will provide references for such an algorithm. As the strategies should be implemented as an extension to an existing framework, only a subset of known strategies are qualified for the thesis. The framework will not be revised completely, so the possible integration levels narrow the possible strategies. As already mentioned, the strategies are rated on their overall quality. This allows a basic assessment of the quality of service of the EBS during high load, but is not an accurate statement for dependability.

1.5 Methodological Approach

First a solid literature review is conducted. It includes the state of the art in EBSs and approaches to handle high load in those systems with a special focus on event stream processing. The review results in an overview of current approaches and leads to a selection of promising approaches.

The next step is to define a concept for the integration of these strategies into a framework for EBSs, which follows established designing concepts. The WS-Aggregation framework [60, 62] including an event-based query engine based on the Open-Source XQuery Engine 'MXQuery'[1] is used for the integration. In the conception phase the framework is analyzed to see how it can be extended properly without touching the MXQuery layer. In the concept, interfaces will be defined for the framework, that will enable the usage of several strategies. Afterwards the interface and the strategies are implemented and integrated into the framework. The strategies will be configurable so that they can be adapted for different load situations. Further on, the strategies are implemented in a generic way without any internal knowledge of the framework, so that they can be used in other frameworks too.

Following, the applications found in the review are analyzed to identify different types of queries in EBSs. Further on, the key features to distinguish query types are determined. The result of this phase is a query taxonomy including the declaration of the features to distinguish the queries and their usage in applications. Based on the gained information, sample queries for the different types are selected to be used for the evaluation. Additionally the key features of the query types are used to perform an analysis of the strategies and to provide a theoretical assessment.

To be able to compare and evaluate different strategies, an event processing environment is needed. A comprehensive experimentation environment is established using the extended WS-Aggregation framework by implementing a simple EBSs that can be used to simulate different load situations and which makes use of the implemented strategies.

For the evaluation each approach is used once with each sample query in the same event processing environment. The results of the executed evaluation scenarios are captured and processed. The theoretical analyses will be proved by the results of the evaluation. Using the gained insights, improvements and application scenarios for the different strategies are provided.

In the final phase, the results of the work will be considered in the context of the holistic problem formulation. It is judged, which conclusions can be drawn and how the results of the thesis can be used more extensively in further research.

1.6 Structure of the Work

After this short introduction the work continues with an exposition of the "State of the Art" in Chapter 2 and an overview of the "Related Work" in Chapter 3. The State of the Art covers the current state of EBSs with a special focus on Event Stream Processing (ESP) applications, established technologies and approaches in event processing over data streams. It also includes proved strategies to handle high load. Current research that is relevant for the topic is discussed in the Related Work. These two chapters will underline the motivation and the described problem (see Sections 1.2 and 1.3).

As the work is based on the WS-Aggregation framework, Chapter 4 "Background" will introduce the framework, its components and further used technologies.

[1] http://mxquery.org/ (Accessed: 2014-01-22)

After these theoretical arrangements, the practical part is described in Chapter 5 “Solution Design and Architecture” and subsequently the concrete solution is presented in Chapter 6 “Implementation”. In Chapter 7 the implemented strategies are evaluated and analyzed. Further on, opportunities for improving the strategies are revealed and the benefits of the improvements are outlined.

Finally, in Chapter 8 “Summary and Conclusion” the work is summarized and conclusions on the performance and the applicability of the developed solutions are drawn. The relevance for distributed event processing is discussed and the chapter is topped off with an outlook how further research can use the results of the thesis to provide a general approach for handling overload.

CHAPTER 2

State of the Art

This chapter summarizes the state of the art concerning ESP. At first, the chapter enlarges upon event based systems in general to provide a detailed understanding. Based on the different purposes of such systems, a basic classification including ESP is introduced and used to emphasize the features of ESP in comparison to other types of event based systems. The second part focuses on ESP. Starting with a more precise definition of ESP, the section continues with current applications (commercial and in research areas), which are used or have been proposed in the context of ESP. Further on, different event query languages, which are used in ESP systems, are shown. In addition, the capabilities of current systems to handle peaks of high load are discussed.

2.1 Event-based Systems

Event processing has been a main topic in research during the last view years. Moxey et al. [95] also reason that enterprises behave event-driven and therefore EP is now seen as the best way to support enterprises.

As already pointed out in the Introduction (see Section 1.1), event-based systems follow the event-driven approach. Events occur in the system or are received by it, they are processed and the results are delivered or they trigger further actions in the system. In addition to the classification of the four logical layers, further models and abstractions can be used to describe EBSs and their components.

Event-based systems can be seen from two different viewpoints: the logical and the physical one. The logical viewpoint is an abstraction of the event-based system. It describes the core value of the system independent from the technical realization. In [95] it is also called the conceptual model. The physical viewpoint manifests the implementation of the system, it is the technical realization of the logical perspective. There can be various physical realizations for a logical description of an event-based system. Depending on the context and purpose, one physical realisation may be suited better than another one.

The logical perspective of an event-based system consists of the following components: data sources and consumers, events, event processing agents and communication channels (event channels), see Figure 2.1.

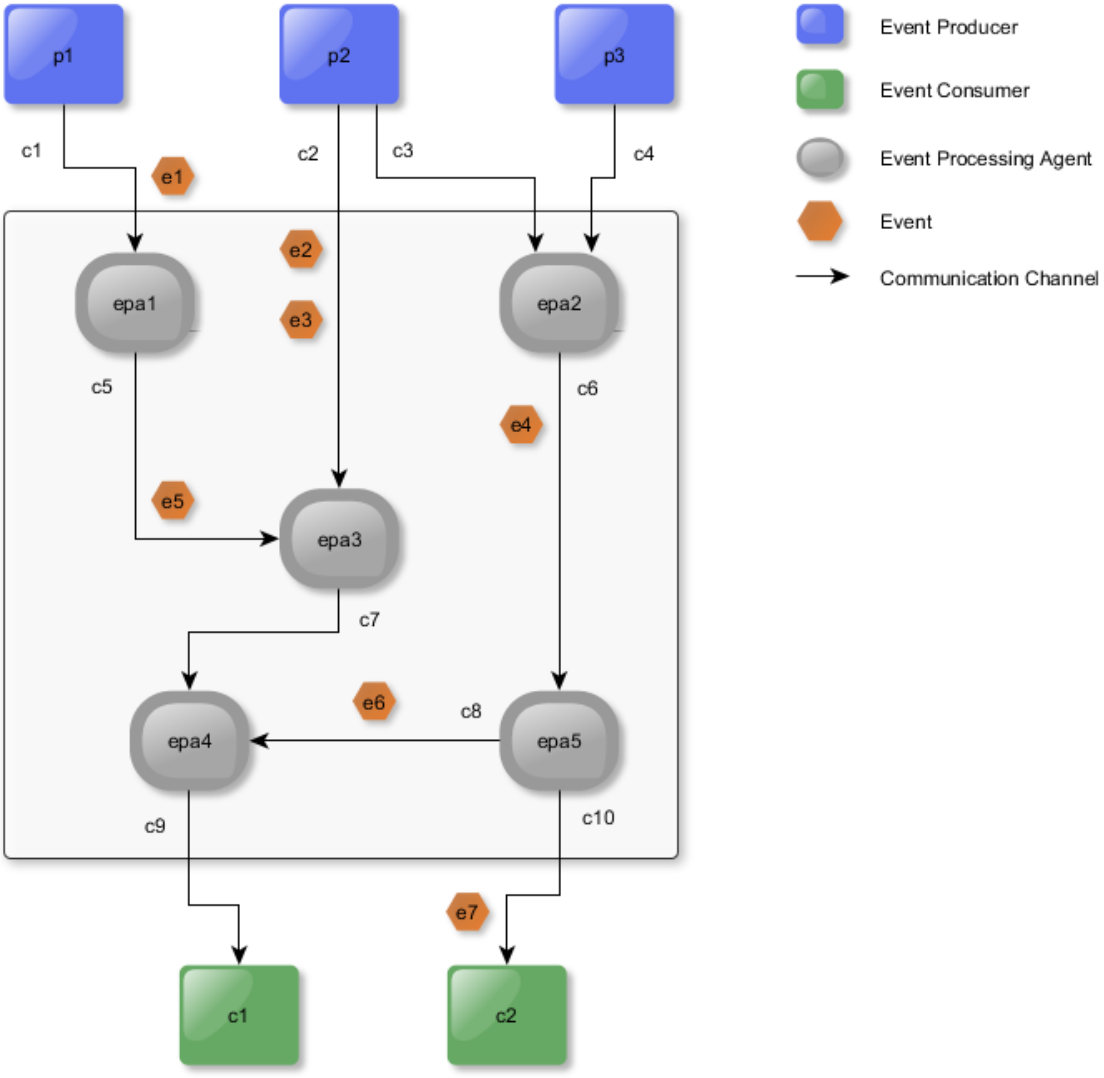


Figure 2.1: Logical Perspective of an Event-based System

Data Sources produce data, which is of relevance for the system, while *Data Consumers* consume data, which is of interest for them. Data is represented in form of events. Important is to note that data sources and data consumers do not need to know anything about each other. An *Event* is a data unit in the form of an object or a tuple. Events emitted by a data source are the input for the event-based systems, whereas the events consumed by a data consumer are the results of the event processing and therefore the output of the system. The processing of the events to transform them from input events to output events is done by the *EPAs*. Depending on

the complexity of the processing and the system, different amounts of EPAs can be appropriate. Each EPA processes the incoming events by performing i.e. aggregation or filtering and forwards the resulting events to next EPA or to a data consumer. Finally, *Communication Channels* are used to transport the events from the data sources along the EPAs to the data consumers. The overall event processing concept is determined by the *EPN*. The EPN is a directed acyclic graph describing the logical connections between the components, where EPAs and event consumers / producers are the vertices and the communication channels are the edges. This implies that the EPN defines the flow of events inside the event processing system. The overall processing of an event is the concatenation of the EPA processing logic along the path, which the event has passed in the EPN. Normally, the specification of a system defines the overall processing outcome that should be achieved by the system. Based on the specification a query, which has to be executed in the system, is constructed. A flexible system must be able to create an appropriate EPN based on the query. Moreover, as systems usually handle more than one query at a time, the systems have to optimize the utilization and sharing of resources between the queries. These are big challenges in the area of event-based systems.

Sharon and Etzion [113] define this conceptual model in a more formal way. An EPN is a graph $G = (V, E)$, where the vertices are composed of $V = C \bigcup P \bigcup A \bigcup EC$. C are the event consumers, P the event producers, A the event processing agents and EC are the event channels. The edges of the graph are described by ordered pairs of the nodes, which represent directed edges from the first node to the second one. Events always have to be transported via event channels, the ordered pairs must fulfill the following term: $E = \{(u, v) \mid (u \in (P \vee A) \rightarrow v \in EC) \wedge (u \in EC \rightarrow v \in (C \vee A))\}$ This states that a directed edge can only point from a producer or an agent to an event channel and an event channel can only point to an event consumer or an agent. So the *event causality* is ensured, which means that event producers only output events and consumers only receive events, which have been completely processed. As an event channel always needs a source and a target node, further restrictions are given: $\forall v \in EC, \exists e_{in}, e_{out} \in E : e_{in} = (u \in (P \vee A), v), e_{out} = (v, u \in (C \vee A))$ Additionally, each node in the EPN has to be connected to another one via event channels, so for each event producer there has to be an outgoing edge and for each consumer there has to be an ingoing edge: $\forall v \in P, \exists e_{out} \in E : e_{out} = (v, u \in EC)$ and $\forall v \in C, \exists e_{in} \in E : e_{in} = (u \in EC, v)$. In contrast to these endpoints, an event processing agent requires an ingoing and an outgoing edge: $\forall v \in A, \exists e_{in}, e_{out} \in E : e_{in} = (u, v), e_{out} = (v, w), with u, w \in EC$.

Sharon and Etzion [113] further state that a single event type is assigned to each event channel, whereas [95] explains that the conceptual model is completely abstract regarding the event and event types transferred via the communication channel. Further on, this formal model does not restrict the graph to be acyclic as circles can occur in the event processing.

The physical components of an event-based system are the nodes and their environment as well as the communication infrastructures. *Nodes* are the hosts of EPAs and are used to execute the actual event processing. The environment of the nodes is the *Processing Platform*, which can be a cloud environment, or a network of interconnected nodes that are handled by a registry. The *Communication Infrastructure* realizes the communication channels between the nodes. Examples of possible infrastructures are data streams or a messaging infrastructure.

The mapping from the logical to the physical perspective is done during the deployment of the event processing system. The EPAs are accordingly assigned to nodes and the logical communication channels are mapped to the physical channels.

Challenges of the mapping are to specify the amount of physical nodes and the assignment of the EPAs in order to maximize the utilization of the resources. Figure 2.2 shows a mapping of the logical EBS in Figure 2.1 to a physical deployment, where the five EPAs are mapped to three physical nodes.

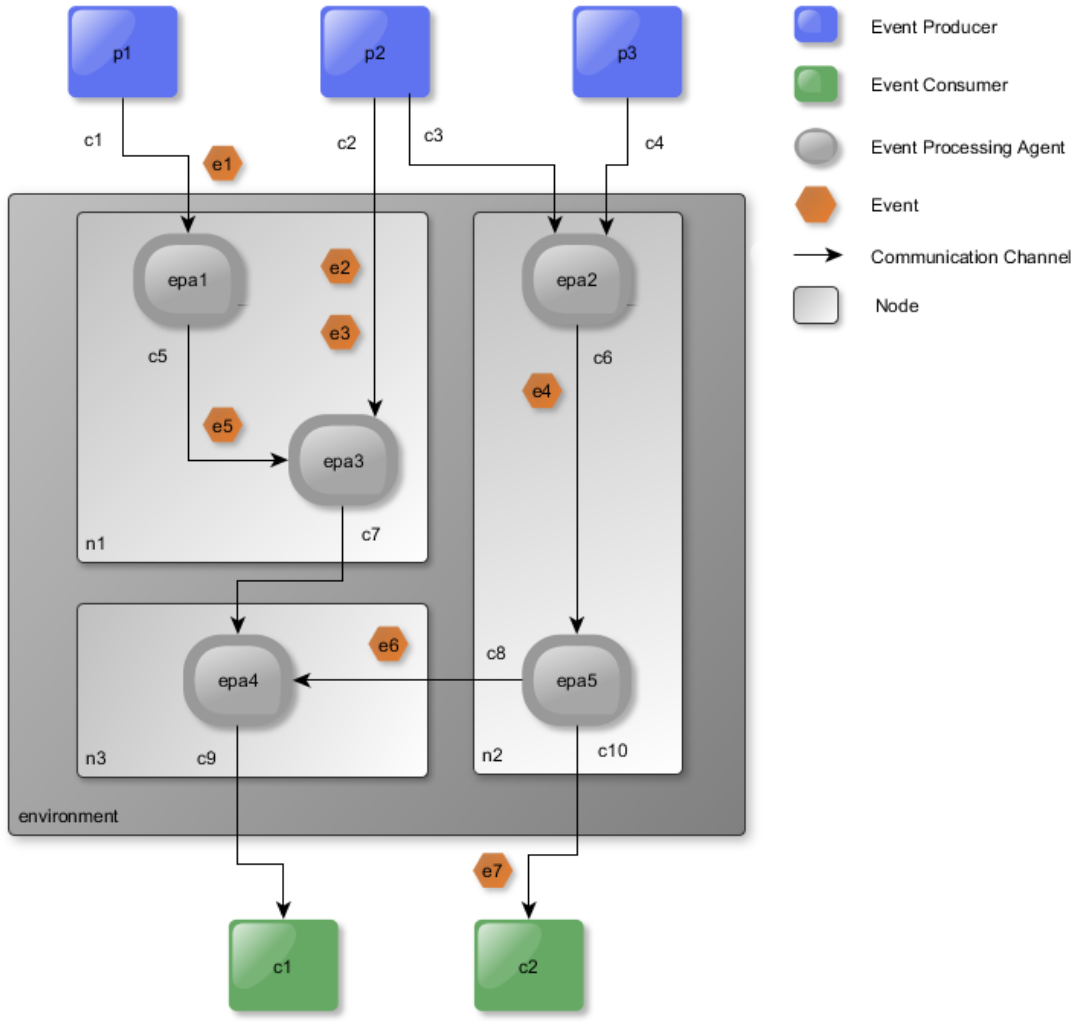


Figure 2.2: Physical Deployment of the Logical Perspective Shown in Figure 2.1

The design of an EBS depends on various aspects. For example, event-based systems can answer different purpose, so this has to be considered during the design of the logical and the physical realization of an EBS. There are a lot of non-disjoint fields of applications that are suitable for event processing. According to [47] and [96] the following fields can be distinguished:

Monitoring and Observation As monitoring is more and more becoming an automated process, event-based systems become increasingly useful in this area.

A popular area of monitoring applications is network monitoring, which includes the monitoring of network structures as well as the traffic across the network [13], where events represent a) data that has been sent or b) devices that have joined / left the network. But there are also other networks to observe like street networks with regard to traffic jams, or alternative forms of routes (such as for ships, trains or planes). Traffic monitoring often includes even more monitoring activities, as for example usage monitoring in street traffic is also relevant for road tolling systems [47].

Another aspect of monitoring applications is security, see also [73]. An intrusion attempt into a system can be seen as an event or as the correlation of multiple events, and a reaction to handle the intrusion is required. Further on, failure or fraud detection is also based on the monitoring and analysis of a system. There is a subgroup of systems, which perform *Predictive Processing* [47]. Those systems do not only monitor the occurrence of events, for example security violations, but they predict the violations so that a real occurrence can be prevented.

If data is not just collected for statistical reason, the application has further purposes which can be out of some of the following areas. For example, if it is required to report information on an occurred security incident to another system or a person, the monitoring application is also a kind of an information dissemination system or interacts with such a system.

Information Dissemination These systems represent notification services, which have to notify one or several persons / systems when defined conditions arise. The observed data often has to be processed in real time and the amount of notification rules is not necessarily limited. Therefore, the communication flow can become complex.

Examples for this kind of systems are stock trading systems [2] or emergency systems. Furthermore, each application that sends personalized messages to users also has information dissemination characteristics.

Enterprise Application Integration Companies often use different applications to manage their work. Regardless of whether different applications are present because third-party, custom-build or legacy systems are used together or because applications were designed to be decoupled in advance: an event based mediator is needed to join the different business processes, see also [89]. The communication and coordination between different components is handled by a separate component that can take great advantage of the event driven approach.

Ubiquitous Systems Ubiquitous Systems are large scale systems that are integrated in the physical environment. Usually, they are implemented in a small area, but the geographical size of the systems is likely to increase in the future. The system continuously interacts with the environment and tries to adapt to the perceived events. Like other EBSs it must be able to detect changes and react to them. One main aspect of ubiquitous systems is the goal to

be as little intrusive as possible [125]. That means that as much complexity as possible is hidden: physically, so that it is not unpleasantly conspicuous, and also in regard to human tasks, which should be simple and at a minimum amount. Schilit et al. [110] explain that the right combination of automated adaptions and user friendly features is important, otherwise ubiquitous system can be a distraction for users.

Chen and Kotz [33] describe needs and design issues of ubiquitous systems. A possible use case for an ubiquitous system is a health monitoring system [100].

Radio-Frequency Identification Systems RFID tags placed on objects make it possible to identify, locate and track objects in an automated way [127]. Sensors record changes in the position of objects or subjects and thus trigger events that must be processed by an EBS. Examples are supply chain management [88, 116], product lifecycle management [115] or luggage systems at airports [136].

Again it is reasonable to combine this type of system with systems for different purposes, for example information dissemination, so that another system or a person can make use of the collected information.

Event-Driven Business Process Management Managing the processes of a company by modeling, orchestration and execution, but also monitoring the business activities, managing the business rules and performing mining to extract valuable business information is also performed event-driven. It is a combination of Complex Event Processing and Business Process Management [126].

Logistic providers, for example, are a good use case for a system of that kind. The processes of the provided services are known and can be realized in an event-driven way. But also in others sectors, i.e. financial services, this approach can be useful.

Massively Multiplayer Online Games MMOG have become an important issue in computer entertainment [45]. A player uses his own computer, but he is connected to all other players. As lots of players use the same resources, actions of the players have to be detected and distributed to players who are effected.

For example, according to a statistic published by Activision Blizzard in May 2013 [1], the game 'World of Warcraft' had 12 million active subscriptions in 2010, so at least some million players have probably played at the same time. Chen et al. [35] report 370.000 concurrent online users in the game 'Ragnarok Online' in 2006. Further on, it describes that the traffic produced by a single client is not that much, but because of the huge amount of players and their actions and movements, the amount of aggregated client traffic is enormous and this causes the *flash crowd effect*, so this high volume of events should not be unanticipated in MMOGs. The handling of such enormous amount of events is also discussed in [48].

[1] http://www.statista.com/statistics/208146/number-of-subscribers-of-world-of-warcraft/ (Accessed: 2014-01-22)

The common basis of all EBSs is a publish and subscribe architecture, see also [109]. Depending on the purpose of a system, it emphasizes certain features and specialties that are important. In systems designed for another purpose, these features may also occur, but different features are focused and enhanced. Hence there is no plain but only a rough classification for event-based systems.

Hummer et al. [58] present such a classification for EBSs that consists of five sub-areas, see Figure 2.3. These sub-areas also reflect different purposes and operational scenarios which result in the different focuses of the areas.

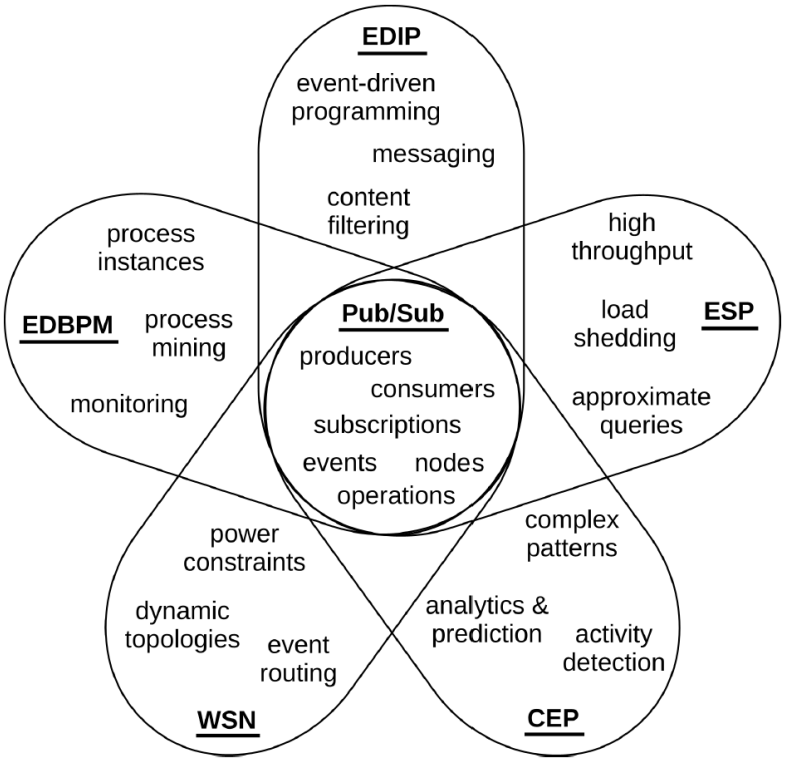


Figure 2.3: Classification of EBSs [58]

In the middle of the figure, the publish / subscriber paradigm is depicted as the basis of all EBSs. It defines the basic components of an EBS. The five sub-areas of the classification are shown in the outer part of the figure: event-driven interaction paradigms, event stream processing, complex event processing, wireless sensor networks and event-driven business process management.

Systems following the Event-Driven Interaction Paradigm (EDIP) are usually characterized by the information dissemination character based on an event-driven programming model. Event-Driven Business Process Management (EDBPM) covers EBSs that model the processes of a business and process events based on the monitoring and mining of the processes. Tasks of Wireless Sensor Networks (WSNs) are usually the efficient routing and aggregation of data in the network. Ubiquitous systems and observation, in particular for intrusion detection, also

belong to this class of EBSs. The same holds true for RFID systems. Complex Event Processing (CEP) is about analyzing and detecting patterns and coherence of events. So the actual processing of the events is more demanding than it is in other systems. It usually implies that multiple simple events are processed and the processing logic then generates new events (complex events), which encapsulate aggregated data and conclusions based on the analysis. ESP systems are systems managing a high throughput. Due to the huge amount of processed events, the processing results cannot and do not have to be exact all the time. The trade-off between accuracy and performance for example is done by using approximate queries.

These sub-areas provide a good overview on the different characteristics of EBSs, but the areas are not disjoint. For example, despite classifying an event-based system as an ESP system, when it utilizes data streams and manages a high throughput, it can also perform complex event processing. But even though the systems has characteristics of two different sub-areas, it may be rather considered as an ESP system, if it puts a stronger focus on the stream processing. Another reason, why a clear classification of EBSs is rather difficult, is that the terms are used with different meanings in the literature. While CEP is defined as a sub-area of EBS in this context, [47] states that CEP is a key principal of event-driven architecture and therefore also of EBS. In addition, the meaning of the terms ESP and CEP are not always used identically in literature [18].

But when following the distinction as introduced above, one can also identify to which category the system belongs by analyzing the terms used to describe it. In the five categories different terms are used to describe the same conceptual parts of the system. For example, an event is depending on the category called notification, tuple, datum or invocation, see [58].

As this thesis focuses on event-based systems, which have to cope with event data streams, the following section will take a closer look on event stream processing and its characteristics.

2.2 Event Stream Processing

The importance of *Event Stream Processing*, sometimes also referred as (Real-time) Stream Processing or Data Stream Management System (DSMS) [51], increased over the time especially because the amounts of data have been steadily increasing. According to [13], data was traditionally stored in databases and queried afterwards for processing operations. This is not appropriate for data that is changing very often and it is not required to retrieve the data multiple times for processing operations. More and more application contexts match these conditions: financial applications dealing with the stock market, network and traffic monitoring or sensor applications. Lots of data is added continuously, but it is not needed anymore after the information has been processed. ESP is a suitable approach for this emerging field of applications. *Data Streams* are adapted for the requirements as they support the continuous dispensing of information produced in the context of the applications. The transmitted information tuples represent events, which have occurred in the context and have to be handled by the application. This leads to the application of the event-driven approach as it meets all the requirements.

On the one hand event stream processing is related to EBSs in general and on the other hand it is related to stream processing. Even though the terms *Event Stream Processing* and *Event Processing* in general are used as an alias [47], there is a distinction between ESP and other

event processing systems. ESP differs from other event-based systems by the usage of data streams for the transmission of events. Data streams support a high frequency and throughput of events. As this is not required for all EBSs, other types do not need to use data streams as a communication infrastructure. As the processing of such huge amounts of data is rather expensive and time consuming, techniques to reduce the complexity are required, i.e. *Load Shedding*. Hummer et al. [58] illustrate the differences in the used terms in ESP in contrast to the common model for EBSs given in [95], see Table 2.1.

Table 2.1: Comparison of Terms (based on [58])

Conceptual Model	Event Stream Processing
event	tuple
producer	source
consumer	sink
event processing	operator
channel	stream
derived event	event pattern

Event stream processing is a sub-area of stream processing, as it only covers applications with event oriented data streams. Video streaming applications therefore do not belong to event processing, but belong to the stream processing applications. So the differentiation between *Stream Processing* and *Event Stream Processing* seems to be explicit. Applications that process normal data streams (i.e. a video stream) by analyzing the content and rising events based on the analyses do not count as ESP in general. The stream itself is not event oriented, but it is used as a basis of an event-based system. If the risen events are also transmitted using streams, the system can be considered as an event stream processing system. Otherwise, it is an event-based system but rather belongs to another sub-area.

Specific for ESP are the data streams, which have great impact on the design of the systems. Geisler [51] describes this characteristic in more detail. Data streams continuously provide event data and are certainly unbounded, which leads to one of the main challenges in event stream processing: While the resources of an ESP system are limited, the input is not. Therefore events are not stored, but results and statistics are created incremental. Typically only a small set of input events, a so called *Window*, is processed at a time. Input events are processed by *Continuous Queries*, which often have a SQL like syntax.

2.3 Event Stream Processing Systems and Research Prototypes

As there is a lot of research on the field of ESP, there are a lot of different implementations for event stream processing systems. Most of them are only used for research purpose and have not gained any relevance for the usage in business applications.

The following table shows an overview of event stream processing systems in alphabetical order. For mentioned query languages, see Section 2.4.

Table 2.2: Event Stream Processing Systems.

Name	Reference	Description
Aurora	[3, 28]	A database management system based on a DBMS-Active, Human-Passive (DHAP) model for streaming data. Provides a graph-based imperative query language.
Borealis	[2]	Extension of Aurora and Medusa. Advanced capabilities for more flexibility in form of revision of query results, query modification and scalable optimization.
Cayuga	[23, 43]	An event stream processing system, which provides pattern detection for generic purposes on high speed data streams. Queries are defined in the Cayuga Event Language (CEL).
COUGAR	[21]	A sensor database system that extends the traditional PREDATOR system by long running queries and the processing of sensor data in form of sequences.
Dryad	[65]	Dryad is an execution engine, which enables distributed, parallel execution. Small programs represent the vertices of the processing graph. Data is passed through channels along the edges between the vertices.
Esper	[1, 19]	A component to process large volumes of input data including event pattern matching and event series analysis. It is available for Java (Esper) and .NET (NEsper).
Medusa	[15, 36]	It is built upon Aurora, but it enables the collaboration of nodes in different administrative areas. Medusa is designed as an *agoric* system.
Nephele	[129]	A data processing framework for cloud environments. In contrast to other system, Nephele does not use queries to analyze and process the input data, instead it executes *Jobs* that have been provided as code in advance.
NEXT CEP	[111]	Implemented in Erlang, this event processing system focuses on optimizing processing by rewriting queries in a more efficient way.
NiagaraCQ	[34]	It is a distributed database system to query distributed Extensible Markup Language (XML) data sets.
Nile	[56]	Nile is a query processing engine that extends a DBMS by data stream processing.
OpenCQ	[79]	This is a distributed event-driven continual query system based on the distributed interoperable information mediation system DIOM [78].
Oracle Event Processing	[99]	Oracle (Complex) Event Processing is an event processing engine, which provides the ability to join data over streams and persistent data.

Table 2.2: Event Stream Processing Systems (Continued).

Name	Reference	Description
PIPES	[71]	PIPES can be used to build DSMS for data-driven query processing.
Storm	[37]	Stream processing in combination with queuing or database management systems can be realized with Storm. It is a free and open source system for distributed and fault-tolerant computing in real time.
STREAM	[4]	A data stream management system developed at Stanford University for continuous queries over data streams and persisted data sets.
Stream Mill	[124]	Stream Mill is also a data stream management system with a powerful query language that focuses on mining techniques.
InfoSphere Stream	[20]	This is the product name for the System S stream processing platform, a stream processing system that uses the SPADE processing engine and the equally called declarative query language. (Predecessors: SPC, SODA)
S4	[98]	Provides stream processing with the following characteristics: distributed, scale-able, continuous queries, partly fault-tolerant and for generic purposes.
TelegraphCQ	[72]	A continuous data-flow processing system based on PostgreSql using shared continuous queries to reduce load and the aiming to support interactions of continuous and historical queries.
Tribeca	[117]	Runs pre-compiled queries on a single data stream source with user-defined operators.

2.4 Event Query Languages

According to [51], *Query Languages* are used to define continuous queries, sometimes also called *Standing Queries*. Query languages can support a declarative or an imperative syntax. Imperative query languages provide operators, which are arranged as a graph to specify the data-flow. Declarative languages are widely used and mostly based on SQL.

For example, assume an event stream with tuples containing stock market data in the form $[timestamp, stockid, sector, value]$. A query specifying the calculation of the hourly average value of stocks for the IT sector using an imperative language is depicted in Figure 2.4. The same query using a declarative language is shown in Listing 2.1.

```
1 select avg(value) as avgValue from StockUpdate(sector='IT').win:time(60 min)
```

Listing 2.1: Example Query using the EQL.

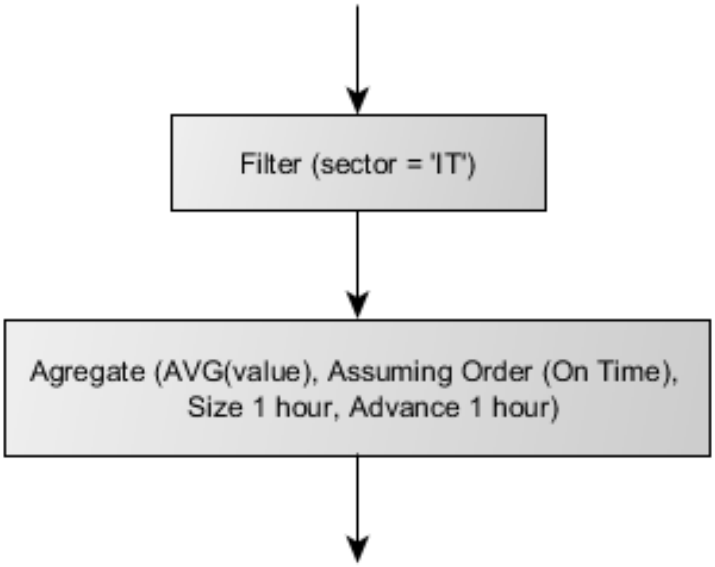


Figure 2.4: Example Query using Aurora's SQuAl.

An important feature of event query languages is the *Window Operator*. The window specifies which part of the stream is evaluated and therefore limits the amount of input events that have to be stored. The window operator is needed to enable a continuous query using a *Blocking Query Operator* (see [11]), as they require all data to be available. Without the bonds to the window, the query would not be able to return. Further on, the window operator enables the system to drop events, once they are not referenced by a window of a query anymore.

Some examples for event query languages are listed in Table 2.3.

Table 2.3: Event Query Languages.

Name	Reference	Description
CEL	[43]	Cayuga Event Language based on the Cayuga algebra [42].
CEDR	[16, 17]	Complex Event Detection and Response. It is a declarative query language based on three aspects: event pattern expression, lifetime declaration, selection expression to create new complex events.
Chimera	[87]	Query language implemented in Chimera for defining ECA rules with an extension for composite events.

Table 2.3: Event Query Languages (Continued).

Name	Reference	Description
CQL	[6]	Continuous Query Language. Declarative query language based on SQL supporting windows. It is used in the STREAM prototype [4].
ESL	[82]	Expressive Stream Language. Declarative language like SQL, supporting user defined aggregates and frequent item sets.
EQL	[122]	Esper Query Language. Declarative language providing event pattern matching and event stream queries.
Oracle CQL	[84]	The Oracle Continuous Query Language is a declarative language based on SQL supporting data streams.
SAMOS	[50]	Event language of the Swiss Active Mechanism-Based Object-Oriented Database System.
Snoop	[30]	Model independent event specification language. In addition to traditional database events it supports temporal, explicit and composite events.
SPADE	[20]	A declarative query language that supports typical operators of the relational algebra but also user defined operators.
SQuAl	[3]	Aurora Stream Query Algebra. An imperative query language providing nine different operators.
XML-QL	[34]	Declarative high-level query language for XML data. Used in NiagaraCQ.
XQuery	[22]	XQuery is a functional programming and a query language for XML data sources. Using query engines for data streams, it can also be used as event query language in ESP, i.e. with the MXQuery Engine [22].

2.5 Handling of High Load

Stream processing systems regularly have to deal with peaks of incoming data. Peaks can arise at certain hours of the day, when a lot of people do the same thing or because of a trigger incident that leads many people to act (i.e. critical changes in the stock market). ESP systems somehow have to deal with situations, where many times over the average data rate has to be processed within a short amount of time. Almost every handling is better than a crash of the system. Nevertheless, strategies provide different result qualities as they normally increase the number of events that can be processed, but decrease the accuracy of the results. As the degree of accuracy loss depends not only on the strategy but also on the query that is currently executed, it is hard to guarantee a certain degree of accuracy for a provided load handling strategy. Anyway, more and more ESP systems try to provide a better handling of peaks of high load.

For example, Aurora uses *Load Shedding* [3]. A central scheduler takes input data from the storage and handles it to a box for processing. At the same time the QoS data is monitored by a separate component. The values response time, dropped tuples and produced values are generally considered QoS data in Aurora. For each of those values, graphs can be adjusted to specify how they are taken into account in the QoS calculation. But also value-based QoS information is supported. In case the performance gets too low because of an overload, the monitor activates a load shedder component. This initiates a load shedding process, which is repeated until the QoS constraints are satisfied again. The load shedding is either done by dropping tuples at strategic points in the calculation or by filtering certain tuples. Less important tuples are identified and removed by a filter box as early as possible. This is also called semantic load shedding.

Borealis [2] as a predecessor of Aurora also supports the dynamic modification of queries. This cannot only be used to change the semantic of a query in case of updates, but also to replace a query by a *cheaper query* - one that does not require as many resources but also produces less accurate results. Further on, different strategies for optimization are applied, i.e. to overcome throughput or latency problems by using distribution and scaling.

Both, Aurora and Borealis, put much effort into the placement of load shedding operators, which has a great impact on the benefit of the operators.

Query rewriting is also used in NEXT CEP [111]. The system does not rewrite queries in case of high load, but optimizes and distributes the queries in advance so that the system can scale properly.

Niagara [34] does not provide special features to handle peaks of high load, but it tries to prevent overload by scalability in the dimension of supported queries, which is realized by grouping queries so that different queries share resources like computation time and memory.

TelegraphCQ [72] tries to handle high load with scalability by using parallelization and adaptive load-balancing using the Flux operator [112], which repartitions stateful query processing operators during run-time.

Nephele [114] is the first system that utilizes dynamic allocation and de-allocation of resources in a cloud for optimizing data processing during run-time.

CHAPTER 3

Related Work

This chapter gives an overview of the related work and research. As this thesis aims to define a strategy for *dependable* event processing, research on dependability is discussed first. Since dependability also deals with faults, the chapter continues with a brief summary of faults in EBSs. Afterwards, research approaches for load handling that are implemented in other EBS are presented. In the last part, further approaches in event processing, which may also be useful in relation to the handling of high loads, are discussed.

3.1 Dependability

Dependability is a general and broad concept, which involves several aspects such as availability, security and of course faults. Therefore it is hard to define and evaluate the dependability of a system.

Avizienis et al. [9] define basic terms and concepts for dependable computing: it says that the original definition of the term dependability is 'the ability to deliver service that can justifiably be trusted'. That means that another system depending on the service, or a person who uses the service, can be sure that the output is correct. That requires the existence of a specification for the service, otherwise it cannot be decided whether dependability is ensured or not. Another definition for dependability, that is less strict, is 'the ability to avoid service failures that are more frequent and more severe than is acceptable [9]'. This definition grants the possibility of deviations of the system from the specification. So according to the second definition it is acceptable if the system runs in a degraded mode, as long as the result is still acceptable. Whether the system is still dependable or not depends on the system or the human that consumes the output.

Figure 3.1 shows the attributes that influence the dependability: availability, reliability, safety, integrity and maintainability. Further on the figure explains the relation of dependability to security as they have some attributes in common.

Availability is used to define the accessibility of a system as well as its ability to provide correct service. This attribute can be seen as the most important one as dependability cannot be

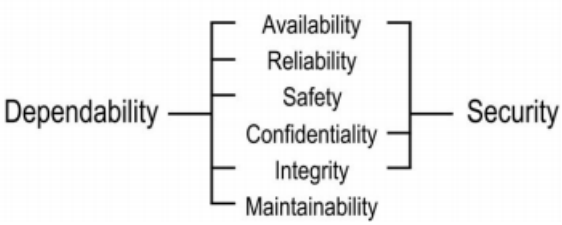


Figure 3.1: Dependability Attributes. [9]

provided without the system being available. The remaining attributes may have strong influences on the dependability or not. It depends on the system's definition and purpose. *Reliability* means that a system is not just available for some time, but it is continuously ready for operation. *Safety* presupposes that the system works properly and does not cause any problems for users or dependent systems. *Integrity* signifies that the system behaves appropriate over time. Finally *Maintainability* is relevant to dependability as a system can only continue being dependable, if it is properly maintained and if no problems occur during the maintenance. In Figure 3.1 another attribute namely *Confidentiality* is included. Confidentiality is also of concern for dependability, but it is rather designated to security.

Independent from the respective definition, dependability is always at risk, if faults occur. Faults may change the state of a service and can influences the attributes of dependability. Furthermore one can say that there is some kind of interrelationship between dependability and faults. Whether a fault threatens the dependability of a system or not, depends on the fault tolerance of a system.

3.2 Faults in Event-Based Systems

Laprie et al. [75] treat common terms and concepts regarding faults. For a beginning the three terms fault, error and failure have to be distinguished.

A *Failure* is any misbehavior of a system preventing the output from meeting the requirements or expectations. An *Error* is the trigger of a failure. The error is a change in the system state that leads from a valid to an invalid state, temporarily or permanently. A *Fault* is original cause of a failure that enables the system to produce a failure. A fault can be a weak implementation of a feature according to the specification or the specification itself, if it does not correlate with the presumed behavior.

A fault may exist without ever causing a failure. As long as no error occurs because of a fault, it is called *dormant*. Once an error occurs, the fault is *active*. For example, if a feature of a component is not implemented correctly, the component holds a fault. As long as the feature is not used by a user or another component, no error will occur and the fault is dormant. The error occurs for the first time, if the feature is requested. The fault becomes active and produces an error. The error can either be *latent* or *detected*, if there is a mechanism for detecting errors. An error can cause damage in two different ways: On one side it can propagate by creating new errors in the component, because if the incorrect implementation leads to an error state,

other features may be affected. On the other side it may lead to a failure. An error results in a failure if the error passes the borders of the component and gets observable to users or to other components, which rely on the erroneous system. If a dependent system has to deal with a failure of another system, the failure represents a fault for this system, which again can cause an error.

Faults can emerge from different reasons and are therefore classified from different viewpoints. Avizienis et al. [9] have identified eight basic viewpoints that lead to 16 elementary fault classes. Relevant combinations of these classes were considered in the resulting taxonomy. It includes 31 combinations, which can be assigned to three groups: development faults, physical fault and interaction faults. Physical faults are overlapping with the other groups, whereas development and interaction faults are disjoint.

Further research has been done on fault taxonomies for specific domains: [27] presents a fault taxonomy for Service-Oriented Architecture, [57] introduces a multi-perspective fault taxonomy for grid faults that is based on seven basic aspects. The fault taxonomy for web service composition in [31] is based on [9] and it is therefore easy to compare the matrix representations of these taxonomies.

Another fault taxonomy based on [9] is the unified taxonomy for EBSs in [58]. The authors have used 12 out of 16 basic fault classes for the specialized taxonomy. The fault classes belonging to the basic viewpoints *Objective* and *Intent* have been omitted, as the taxonomy does not focus on security. Further, two fault classes deduced from the viewpoint *Level in Solution Stack* have been added. According to this viewpoint faults are partitioned into *Platform Faults*, which originate in the underlying event processing system, and *Business Logic Faults*, that occur in applications using the platform. An overview of the viewpoints and fault classes can be seen in Figure 3.2.

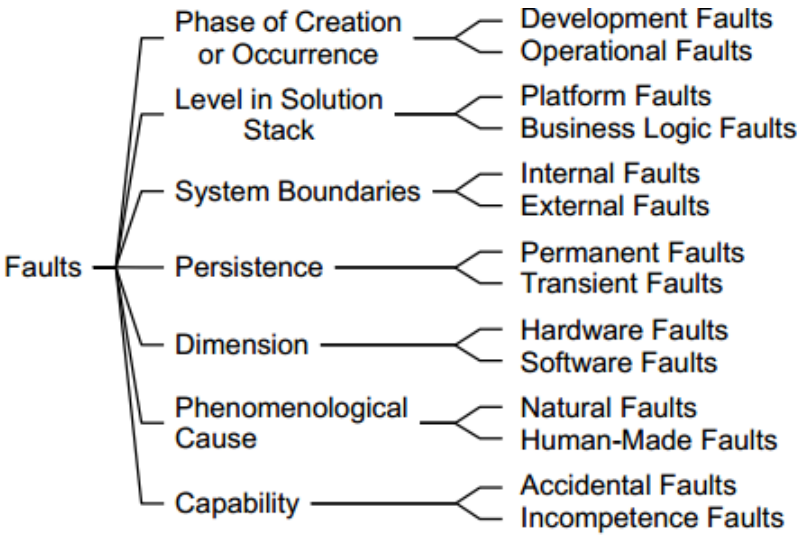


Figure 3.2: Basic Fault Classes for EBSs. (Taken from [58], Based on [9].)

Different faults for the combinations of the fault classes are given in the paper. For EBSs in general *Overloaded Channel* is a possible fault. It is classified as follows: operational - the

fault is not created during the development, it occurs during the operational time; platform - it does not depend on the business logic, because it can happen with any business logic on top of the platform; internal - the fault is not caused by any external circumstances; incompetence - the fault occurs because the limitations are not considered properly. The cause of the fault is the combination of the input, the system state and the software asset.

More specific faults for Event Stream Processing described in this taxonomy are: *Buffer Overflow*, *Race Conditions*, *State Mismatch*, *Bad Query Substitutions*, *Cyclic Processing Logic* or *Node Failure*. The classification of those faults and the overloaded channel fault can be seen in Figure 3.3. Further faults are not mentioned here, as they are out of scope for this thesis.

	Phase of Creation or Occurrence		Level in Solution Stack		System Boundaries		Persis-tence		Dimension		Phenome-nological Cause		Capability	
Overloaded Channel		X	X		X			X		X		X		X
Buffer Overflow		X	X			X		X		X		X		X
Node Failure		X	X		X		X		X		X		X	
Race Conditions	X		X		X			X		X		X		X
State Mismatch		X		X		X		X		X		X	X	
Bad Query Substitution	X		X		X		X			X		X		X
Cyclic Processing Logic		X		X	X		X			X		X		X
	Development	Operational	Platform	Business Logic	Internal	External	Permanent	Transient	Hardware	Software	Natural	Human-made	Accidential	Incompetence

Figure 3.3: Classification of Faults in ESP (Based on [58]).

The buffer overflow fault is very relevant to this thesis as this fault occurs if an EPA cannot allocate the required memory for incoming events. This can happen especially during periods of high load.

3.3 Approaches to Handle High Load

The research done on strategies to handle high load is increasing. As there are so many different topics and developments in this field, there are many customized approaches for special topics or scenarios, but it is hard to find a general approach. The following sections discuss approaches present in state of the art literature that can be of use to handle high load.

3.3.1 Approaches Dedicated for Burst Handling

Many approaches have been devised to deal with high loads in certain cases. Load shedding, a technique dropping input events to reduce load, has been used in different applications and have therefore been adapted to the given circumstances. For example, load shedding approaches often consider just special types of queries, like in [12]. They present a load shedding approach not

only addressing load shedding on a single node to prevent overload, but trying to optimize the placement of load shedders in the whole EPN to maximize throughput and minimize inaccuracy. But on the other hand they have some limitations: the approach is just working for query trees, so the query cannot perform join operations on data streams. The algorithms are designed for sliding window aggregate queries, which can also include some filtering, but no other operators. As the approach is focused on aggregation queries, it uses this restriction to improve the quality by adapting the calculated average values based on the past values for each dropped event. Another approach for window aggregate queries is proposed in [120].

Query rewriting, which can be seen as a more general approach than load shedding, has already been used in the context relations schemata like databases or also XML schemata (i.e. in [108, 133]). This approach is also used to increase the performance in EBSs, for example in *Borealis* and *NEXT CEP*.

The Borealis system [2], see Section 2.3, addresses the needs of adapting queries. On the one hand this includes the adaptation of operator parameters, if the definition of events of interest changes, but on the other hand also the change of operators, if the query itself should be changed. These changes are handled by automated modifications. The local optimization further includes a semantic-based load shedder, that drops events of low-priority in phases of high load. Borealis reuses the boxes-and-arrows model of Aurora, see Section 2.3, where the boxes are the query operators and the arrows are the data flows. In comparison to Aurora, Borealis does not only support data input lines for the boxes, but it also supports special control lines, which use operator parameters or functions to change the box behavior and therefore the operator itself. In the research prototype *NEXT CEP* [111] on the other hand, query rewriting is performed before the deployment. It uses its own high-level event pattern language. Queries stated in this language are analyzed before the deployment and then transformed into a core language that consists of six operations according to the query rewriting algorithms. These algorithms select an efficient deployment plan for new operators by reusing already deployed operators and transforming queries to more efficient but still equivalent patterns.

Typically, overload can be handled by providing more resources: parallelization or replication, which means in case of high-load one or more EPAs are added and the load is distributed among these components. An approach addressing not only this form of parallelization, but the whole construction of and the distribution in an EPN to maximize the throughput is described in [74]. First, the structure of the EPN dependency graph is analyzed. The EPN is formed by creating an EPA for each event processing action. Then parallelization is done on two dimensions: horizontal parallelization is realized by creating several strata. A stratum contains EPAs of different, independent sub-graphs, that can be executed in parallel. Vertical parallelization happens within each stratum by providing multiple, homogeneous EPAs that handle the same types of input events. The distribution of input events to the different nodes of a stratum is done by a proxy. To support stateful query operations, events cannot be distributed randomly to the available nodes, as they do not share a common state. The distribution decision is done by the proxy using a hash function, that analyses the incoming events based on its context values and determines, which EPA is responsible for the event. The proxy therefore has to decide semantically and the hash function depends on the specific query operation. Further adaptations for dynamic load distribution during run-time are done by vertical parallelization, as further EPAs

can be added to a stratum or superfluous nodes can be removed. This may cause additional overhead, for example the overhead of substituting the current distribution hash function of the proxy or the migration of the current state to another node.

3.3.2 Other Approaches Worth Considering

Although many approaches have not been explicitly designed to handle high load, they are still related to the topic, for example increasing availability or providing better throughput in general - not only in phases of high load.

One possibility to ease situations of high load or to hide temporary failures, buffer overflows for example, is *deferred execution* or *delayed processing*. This mechanism can hardly be applied in real-time applications. An approach taking advantage of delays has also been implemented in *Borealis* [14]. It is called *Delay, Process and Control (DPC)* and delivers best-effort results by observing a maximum delay limit. In case of problems, the time frame gained by the permitted delay is used to search for a replication that can process the input or to remove the problem.

Most event-based systems act rather static, which means that the EPN can hardly be modified during run-time. In comparison to that, [83] presents an approach based on the query processing mechanism *Eddy* [10], which enables an adaptive processing of events. The approach is called *Continuously Adaptive Continuous Queries (CACQ)* and enables modifications in the order of the query operators. Hence the route of an event is dynamically chosen. Combined with cross-query sharing the approach increases the performance and could also be used to provide better handling of high load.

Brito [24] shows an approach for the parallelization of stateful event stream processing components. It uses optimistic parallelization to improve the performance of components. For stateful computations the correct processing order is important. In case of parallelization the shared data has to be locked, so that just the actual processor of the next event can manipulate the state. Optimistic parallelization enables the parallel processing of several events, even if it is not their turn yet. The execution uses Software Transactional Memory (STM), which leads to the transactional processing of events. Speculative event processing is done in a transaction. If it is the turn of such a speculatively processed event, the transaction can be committed, if no conflicts arise. Otherwise the event has to be processed again. The approach uses two kinds of conflict predicates, a Boolean predicator and a predicate-based predicator, to decide whether an event should be speculatively processed or not. The predicates are gained through static analysis, run-time analysis, automated knowledge acquisition during run-time or information provided by the user. Depending on the quality of the predicators, the optimistic parallelization can lead to a performance increase or not. But even if the predicators are poor, a correct processing is guaranteed.

In [104], improvements on performance are gained by a rather different approach, namely Thread Level Speculation (TLS). TLS is an automatic program parallelization, which is based on speculative method level parallelization. Normally, a thread executes a method and after returning it goes on with the code after the call statement. With TLS, the method is executed, but by using a predictive return value, another thread already starts with the calculations following the method call. After returning, the process is joined. If no errors or discrepancies have occurred, the thread can just continue. Otherwise the speculative calculations have to be discarded

and the process has to continue as usual after the method call. Before the execution of a program, preparation has to be done. This includes a static analysis, attribute parsing and especially the insertion of fork and join points. The execution can be single- or multi-threaded. By using multi-threaded execution a program can be sped up by the times of two.

High availability is not only provided by distributing load and improving the capabilities of the EPAs to avoid overload on the nodes, but also by ensuring availability of the nodes itself. Hwang et al. [64] present approaches to provide high availability by assigning an equally prepared secondary node to each primary node. Different recovery types can be supported: gap, rollback or precise recovery. Gap recovery is the fastest recovery type, as events arriving between the crash of the primary node and the takeover of the secondary node are ignored. Therefore, information may also be lost. Rollback recovery results in a higher latency, as the secondary node has to update his state to the state of the primary node and then completes to process the incoming events instead of the primary node. No information is lost and the output is equal to the output without a failure as input events a preserved till the processing is finished. In case of non-deterministic queries the output may not be exactly the same as without a failure. Precise recovery results in exactly the same output as without a failure, but it has the greatest overhead and latency.

Brito et al. [25] also propose an approach for fault tolerance in the form of active replication with low overhead. A speculation mechanism was implemented using STM, enabling nodes to process events without knowing the final ordering and forwarding speculative output events. This can reduce the latency significantly. Further on, multi-threaded processing of events is also realized by STM as final results are just produced after the commit of a transaction and transactions are always performed in the correct order, which is required for stateful query operations.

CHAPTER 4

Background

This chapter explains the relevant background of the solution, which is presented subsequently. As the implementation of strategies to handle high loads cannot be evaluated without a suitable EBS, the strategies have to be integrated into such a system. The framework *WS-Aggregation* [60] was chosen as the basis for the solution and implementation. First, as this framework uses the Vienna RuntimeEnvironment for Service-oriented Computing (VRESCo), Section 4.1 deals with this topic. Following the framework itself is introduced in Section 4.2. Both VRESCo and WS-Aggregation were developed as Research Prototypes at the Distributed Systems Group (DSG) of the Vienna University of Technology [1]. The Web Services Aggregation Query Language (WAQL), described in Section 4.3 is the used query language in WS-Aggregation. Finally, Section 4.4 introduces Aspect Oriented Programming (AOP), which will be used to implement a measurement for the evaluation of the solution.

4.1 Vienna Runtime Environment for Service-oriented Computing

The Service Oriented Architecture (SOA) paradigm is an approach to enable decoupling of services and to prevent service binding at design time. As service registries in the context of SOA have not become prevalent [92], VRESCo [61, 90, 91] aims to address the challenges of Service Oriented Computing (SOC) and to provide a sophisticated run-time environment for services, including a registry for those services. The challenges of SOC that have been observed are *service meta-data*, *service querying*, *quality of service*, *dynamic binding and invocation*, *service versioning* and *event processing*. The architecture of VRESCo is depicted in Figure 4.1.

The VRESCo Runtime Environment can be accessed directly via Simple Object Access Protocol (SOAP) calls or using the Client Library, which is based on Daios [77]. Services and their meta-data can be published to the environment using the *Publishing/Metadata Service*. The information is stored in the *Registry Database*. VRESCo does not only store the name and endpoint of a service. Its data model contains a *service meta model* and a *service model*. The meta

[1] DSG, Institute of Information Systems (http://www.infosys.tuwien.ac.at/index.html (Accessed: 2014-01-22))

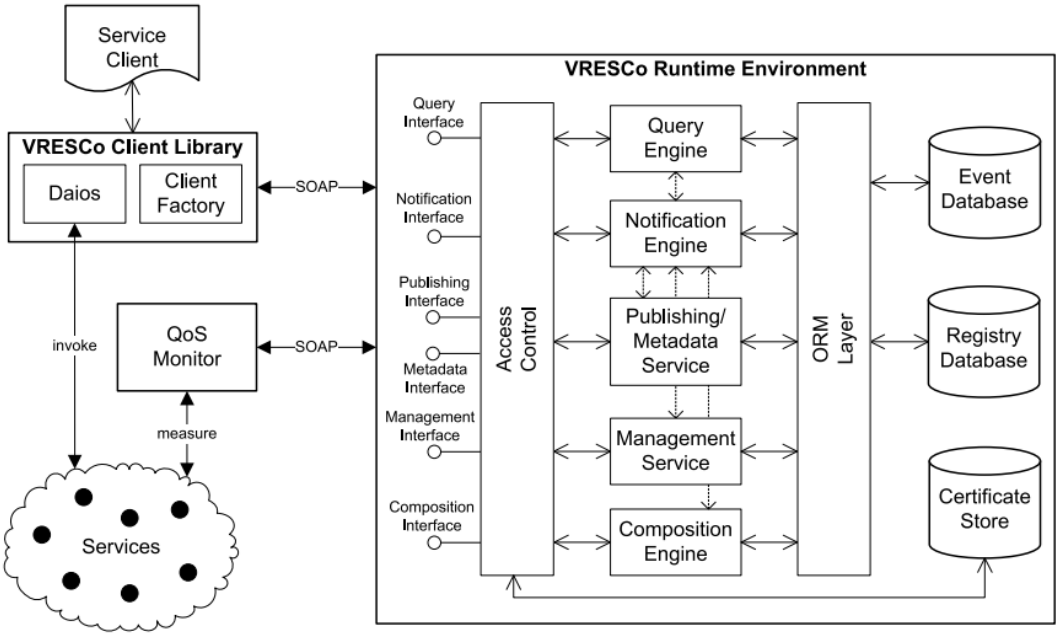


Figure 4.1: VresCo Runtime Environment. [90]

model classifies services into different *Categories*, which represent the purpose of a service. Each category is described by *Features* and *Concepts*, which represent the functions a service offers. For each concrete service that provides features of a category an entry in the service model is created. It includes QoS data (*static data* like features or costs and *calculated data* for example response time or latency) and information on the revision (to be able to provide provenance information), and it requires a *Mapping Function*. This enables the VRESCo Runtime Environment to act as a mediator for services that fulfill the same purpose but have different syntactical interfaces.

The lookup of services utilizes the *Query engine*, which processes queries written in the VRESCo Query Language (VQL), a declarative query language. The queries are translated into SQL and executed on the Registry Database. Using the VQL, services with special features and properties can be discovered based on the abstract features specified in the service meta model. On invocations the abstract features have to be mapped to the concrete operations. The VRESCo Mapping Framework (VMF) uses the mapping functions to transform the feature input into the concrete service parameters and proceeds vice versa with the output.

The *Notification Engine* is used to inform external and internal services of changes and updates. External services may be informed of changes in QoS and the binding or invocation of services. Internal updates concern users, services, versions or meta-data. The Notification Engine is based on Esper [122] and defines listeners for these events to be able to propagate the events to subscribers. Internal services can also provide their own listener for Esper. The *Management Service* is used to handle user data for the access control as well as to manage QoS

data. The *QoS Monitor* reports updated data via this service. Further on the *Composition Engine* can be used to compose services according to specified constraint.

In comparison to other registry approaches VRESCo provides does well. It does not provide versioning of meta-data like ebXML [2] and Mule [3], but supports all other relevant features and manages the challenges of SOC.

4.2 WS-Aggregation: Distributed and Event-Based Aggregation of Web Services Data

Due to a lack of generic frameworks the WS-Aggregation framework was designed to be loosely coupled and to be able to aggregate heterogeneous data from different sources on the web.

The framework basically consists of three different components: the *Registry*, the *Gateway* and several *Aggregators*, see Figure 4.2. Further on *Target Services* are used to aggregate data, but they do not need to be part of the framework. As already mentioned, the VRESCo Runtime Environment is used as registry.

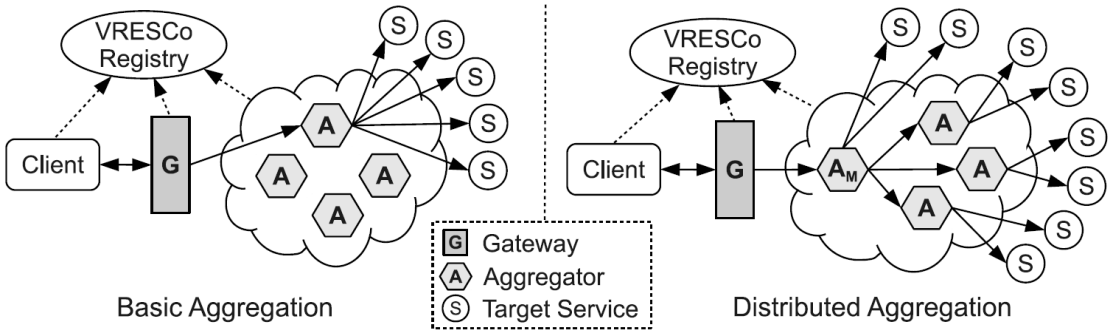


Figure 4.2: WS-Aggregation Framework. [60]

The aim of WS-Aggregation is to provide a single-site interface for clients to state their aggregation request while the rest of the system is hidden. This is realized by a gateway. A client sends its aggregation request to the gateway and retrieves the result. It does neither know how many aggregators are needed to process the request nor how the data is aggregated using different data service nodes. To obtain the address of the gateway the client has to use the registry.

In WS-Aggregation the registry stores the gateway, the aggregator nodes and the target services. For that reason they are published to the registry. Clients use the registry to look up the gateway, from then on the client only communicates with the gateway.

The gateway also utilizes the registry to obtain information on the deployed aggregators. Different features can be used for describing and looking up nodes. According to the given topology one or several aggregator nodes are chosen for each request. Normally the selection

[2] http://docs.oasis-open.org/ebxml-bp/2.0.4/HTML/ebxmlbp-v2.0.4-Spec-cs-en.htm (Accessed: 2014-01-22)

[3] http://www.mulesoft.org/ (Accessed: 2014-01-22)

of the aggregator nodes is based on performance criteria to distribute load. But the decision can also be made location based or in connection with special characteristics and features of an aggregator node (e.g. specialized nodes for data conversion).

The aggregator nodes that belong to the established topology of a request are responsible for the aggregation of the data and the completion of the request. In Figure 4.2 one can see the differentiation of *basic aggregation*, whereby the aggregation request is processed by a single aggregator node and *distributed aggregation*, where several aggregator nodes are arranged in a tree structure with a root aggregator node that is called master aggregator (A_m). Whether basic or distributed aggregation is used, is determined by the topology. Three different types of topologies are supported: *basic*, *predefined* and *ad-hoc*.

A basic topology consists of a single node that takes care of the whole request. Distributed aggregation is realized by a predefined topology. The topology is created on given characteristics of the desired tree (branching factor, height). The leaves of the tree are responsible for the data collection and then for passing on the intermediate result to their parent, as do all other nodes. The root aggregator composes the final result. An ad-hoc topology is instantiated as a basic topology, therefore only one aggregator node is assigned to the request, but this node is enabled to delegate parts of the request to other aggregator nodes. So the basic aggregation can switch to distributed aggregation.

For all three topologies, the master aggregator node or respectively the single node passes the final result on to the gateway, which returns it to the client.

WS-Aggregation was developed to address the following requirements:

- *multi-site query, single-site view*: A client uses just a single site to perform its aggregation requests. The aggregation request uses multiple sites (sources) for retrieving the information for the query and for calculating the results.
- *heterogeneity*: The different data providers do not need to be of the same type. Different sources like SOAP Web services, RESTful Web services or web documents can be used.
- *variable input data*: The input of data services may differ and an input can also be used for different requests.
- *self-adaption*: Data providers and aggregators are not defined at deployment time. The nodes and their number can change during run-time and the system needs to adapt to these changes.
- *performance*: Scalability in different dimensions is required for good performance. The dimensions include the number of data providers and aggregation nodes, the amount of the transferred data and the number of parallel aggregation requests handled by the aggregation nodes.

The WS-Aggregation framework can be used to implement EBSs that handle web service event streams. It tries to enhance performance by distributing queries to several aggregator nodes, executing requests in parallel as well as optimizing queries. The framework automatically monitors the available resources. Aggregator nodes provide meta-data on their workload, so the amount of aggregators can be increased if there are too few free resources left, or decreased if the workload can be handled by less aggregators.

At the moment the framework does not provide features to react to peaks of high load concerning a certain event stream or on the level of queries. Therefore it can be used to implement

and test further strategies. New strategies can be evaluated by using WS-Aggregation as an event-based system and the results can also be compared to the performance without applying load handling strategies.

4.3 Web Services Aggregation Query Language

The Web Services Aggregation Query Language (WAQL) is based on XQuery, see Section 2.4, and was introduced in [59] and [60]. The query language was designed to state aggregation queries in a more convenient way, as complex requests get rather complicated using XQuery. WAQL does not only provide the possibility to use different data sources in a query, but also supports different types of data sources and different result types by using several data converters. CSV files, JSON data, non-XML-compliant Hypertext Markup Language (HTML) pages and BibTex files can be transformed to XML. Converted CSV data can either be displayed as HTML tables or be again converted back to CSV. As a WAQL input has to be transformed to a XML input, which can be sent to a target service, WAQL inputs represent rules to generate such a request.

One element of a WAQL input are request templates, which can be used to create multiple requests having a similar structure. The requests differ from each other based on the values the request is using. These values are provided in the form of lists in the request template. The syntax for a value lists is: $\$(list)$, where the list is provided as XQuery Expression. By building the Cartesian product of the lists, all possible combinations for the request structure are created. For each combination a request is performed. If not all combinations are desired, lists can also be correlated by using a numerical identifier after the dollar sign. If two or more lists of equal length have the same identifier, the values are just combined in pairs based on the ordering, see Listing 4.1 for an example.

Given a web service that provides a method to retrieve the stock values of companies based on some filtering attributes specifying a date, a WAQL query can be used to simplify the generation of aggregation requests.

```
1 <getStockValue>
2   <attribute>
3     $1('name', 'industry', 'country')
4   </attribute> <value>
5     $1('companyA', 'IT', 'AT')
6   <value> <date>
7     $('3/31/2012', '6/30/2012', '9/30/2012', '12/31/2012')
8   </date>
9 </getStockValue>
```

Listing 4.1: Exemplary WAQL Query.

The request for the method *getStockUpdate* contains three lists, see lines 3, 6 and 9. Because of the numeric identifier 1, the first two lists are linked. Those values are used in pairs. Given the three correlated pairs *(name, companyA)*, *(industry, IT)* and *(country, AT)* and the four values for *date*, twelve requests are generated. The resulting XQuery requests retrieving the quarterly stock value for the company named 'companyA', for companies in the branch 'IT' and for companies from the country with the code 'AT'.

Another element of WAQL queries are data dependencies, indicated by the syntax *${selector}*, where the *selector* is an XPath selector processed on the input. If no information is provided, the location is automatically evaluated at run-time. As for request templates, a numerical identifier after the dollar sign can be used to identify the input source of the data. The selector is then just applied on that input source. For example, *$3{//change}* parses the node named *change* from result event of input source *3*. With data dependencies, a WAQL query with multiple sub-requests can use events resulting from one sub-request as input for another sub-request. WS-Aggregation optimizes this feature by analyzing the sub-requests and performing independent sub-requests in parallel, while dependent sub-requests are performed in the correct order.

As WS-Aggregation uses a third-party XQuery engine to process the aggregation requests, the WAQL queries have to be transformed into valid XQuery statements. This is done by the Preprocessor of the WAQL Engine (WAQL-PP)[4]. The preprocessor detects and resolves data dependencies in the query. After this step, the query is transformed to a XQuery FLOWR expression, which can be finally be processed by an XQuery engine.

4.4 Aspect-oriented Programming

AOP has already been proposed as a programming paradigm in 1997 [68] and is categorized as a Post-object programming (POP) mechanism, see [105]. The purpose of AOP is to gain a better separation of concerns. Separation of concerns [63] aims for a higher abstraction and decoupling of components to enable a better understanding and maintainability. Object-oriented programming, which is the current dominant programming paradigm [46] enables separation of concerns in one dimension. Normally, a separation of core functionalities (features, business rules) is done. Strongly coherent properties and operations are encapsulated in objects. But many concerns cannot be classified functional, they are rather properties of a system and, moreover, they are crosscutting across functional components [39]. The concerns result from *aspectual requirements* [46]. Such crosscutting may be synchronization, real-time or global constraints, authentication or logging. (For further example see [39, 46, 105].)

Normally, these crosscutting concerns are tangled into the code of the components or the components call certain subroutines, which results in duplicated code in several objects. With AOP, concerns are implemented as so called aspects. According to the description of the relationship between aspects and components, the aspects are woven into the code of the components, see Figure 4.3.

According to [49], aspects are quantified assertions for the behavior of a program. The aspect defines conditions and actions, which are executed when the conditions arise. Three types of AOP can be distinguished: static black-box, static clean-box and dynamic AOP. *Black-box AOP* treats the components as a black box and the quantification is done on the public interfaces of the components. Therefore, black-box AOP works without the source code of the components and may be rather reusable and maintainable as it does not depend on the actual implementation of the components. In *White-box AOP* the quantification is performed over the parsed structure.

[4] http://www.antforge.org/waqlpp (Accessed: 2014-01-22)

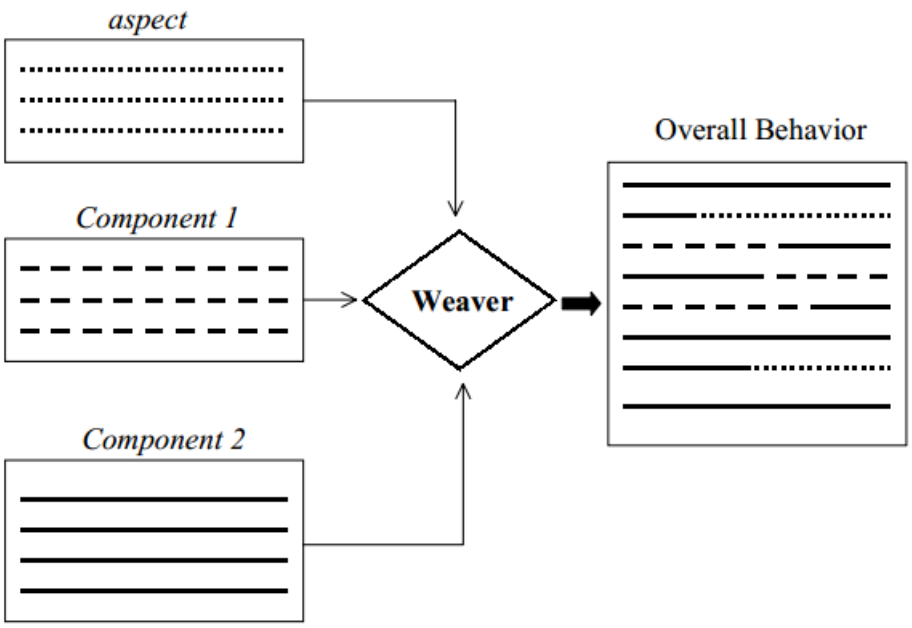


Figure 4.3: Weaving of Aspects. [39]

It is not restricted to the public interface and so it provides more possibilities. On the other hand white-box AOP is harder to implement and builds dependencies on the actual implementation. *Dynamic AOP* quantifies over run-time happenings, for example exceptions, certain calls in a temporal scope or historical patterns.

Another distinction can be made on the level of weaving [39]. The level defines when the aspects and the code of the components are combined. *Pre-compile weaving* mingles the aspects into the components and then the woven code is compiled. *Compile-time weaving* performs the combination during the compilation - the byte code therefore contains the intermingled code. *Run-time weaving* makes use of reflective architectures, where the weaving takes place during the execution. This type of weaving is more complicated, but enables to update and modify aspects during the execution.

For the evaluation of the solution, events have to be traced inside the EBS. As the routing and processing of events is spread over many classes, AOP in the form of AspectJ[5] is going to be used to measure the event processing. AspectJ provides the possibility to expand Java programs by AOP. It therefore provides the following constructs [67]: aspects, join points, point cuts and advices. An *Aspect* encapsulates modular units of crosscutting implementation. Aspects can contain normal Java code and the remaining constructs. *Join Points* represent points in the dynamic execution. *PointCuts* refer to collections of join points and are used to represent the conditions that have to arise for the execution of an action. Join points can match methods and constructors, but also fields or exceptions. Finally, *Advices* are assigned to point cuts and

[5]http://eclipse.org/aspectj/ (Accessed: 2014-01-22)

include the actions that should be performed if the condition specified by the point cut arises. The execution of the advice can either be around, before or after the execution of the join point. In AspectJ, which implements clear-box AOP, weaving can take place at compile-time, post-compile time or at load-time. In each case static weaving is performed and the resulting class files - and therefore the behavior - is the same in all cases.

CHAPTER 5

Solution Design

In this chapter the design of the solution is described. Crucial for the solution are the strategies for handling the high loads and the taxonomy of queries, as the strategies should be applicable for all different types of queries. Therefore this chapter starts with the results of the literature review concerning query types are presented as the taxonomy explains basic terms that are relevant for burst handling. Next follows a discussion of possible strategies as solutions for burst handling and the presentation of promising strategies that are implemented and evaluated in the next steps. Following the integration into the framework WS-Aggregation is discussed.

5.1 Taxonomy of Queries

Several papers describe theoretical basics of event processing or concrete applications and prototypes that offer certain features for event processing. An excerpt of the literature that was used to form the taxonomy is given in the Appendix, see Section A.1.

According to the collected data, two different dimensions to classify queries can be identified: scope and operation.

Scope states on how many and on which events the query is executed, see Figure 5.1. If the query processes each event separately (and is hence not required to keep the state of previous events), it is *Stateless*. *Stateful* queries process a group of events together. This set of events is called *Window*. Stateful queries differ from each other depending on further attributes. One of these attributes is the *Unit*, which is used to define the windows. *Logical* means that the size of the window is given by a time specification. Therefore, those windows are called *time-based*. The second option is *Physical*. In that case the size of a window is measured in events and is also called *count-based* or *tuple-based*. Another distinctive attribute for windows is how the borders of the window are defined. In case of a *Fixed window*, the borders are set by absolute values (either positions or time stamps) and do not change. Therefore, the set of input events that are processed is fixed and the events do not change. A *Growing window* starts at the very beginning and evaluates at every unit (time or tuple). *Landmark windows* start or end at a fixed

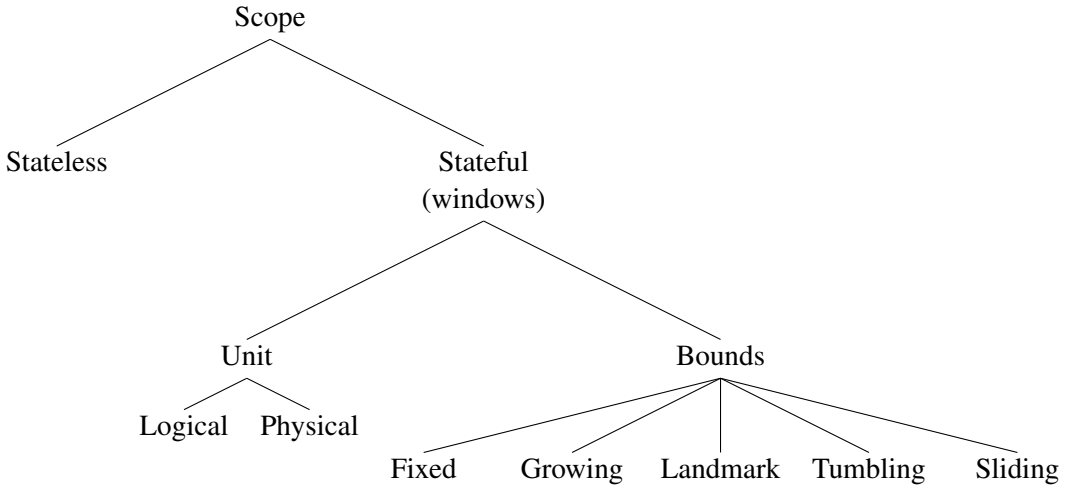


Figure 5.1: Query Taxonomy, Dimension Scope.

point. So either the end is now, which means that the size of the window is increasing over time and more and more elements have to be processed or the end is defined (most likely in the future) and with each unit a new window is opened. Both of the two remaining bound types, Tumbling and Sliding window can be time-based or count-based, but the size of the windows can be varying. The start and the end bound are expressed by conditions. Both bounds move on with the progress of time. The distinction between these window types is based on whether those windows are overlapping or not. If there is no intersection of the windows, the query has a *Tumbling window*. So a new tumbling window starts after the preceding window ends. If windows are intersecting, they are called *Sliding windows*. Because of the intersection, an event may belong to multiple windows. As a consequence, one event is processed multiple times in different windows and influences several output events. For both, tumbling and sliding windows, an offset can be given by the start condition, so that the next window does not start right after the preceding window or respectively the last input event. Cugola and Margara [40] do not describe this in terms of an offset, but define tumbling windows as a variant of a sliding window, which has an offset greater or equal than the window size. *Pane windows* are sliding windows with an offset smaller than the window size. So they are still intersecting.

Mokbel et al. [93] also use the term *Predicate-based* to refer to a generalized window definition. The input events for a window are not selected based on the unit, but on a predicate that has to be fulfilled. A similar concept is proposed by [5], called *Partitioned* windows, and defines a tuple-based window on a subset of the inputs events. So events are only considered for a window, if they meet certain preconditions. Further concepts are *Historical* and *Suffix* windows.

While [7] defines windows ending with the current time (meaning with the last input) as suffix windows, and windows that end before now as historical, [93] uses the term historical for windows that make use of earlier events.

The differences of the scopes in processing a set of input events are depicted in Figure 5.2, where all queries are assumed to be count-based. The arrows along the time dimension define

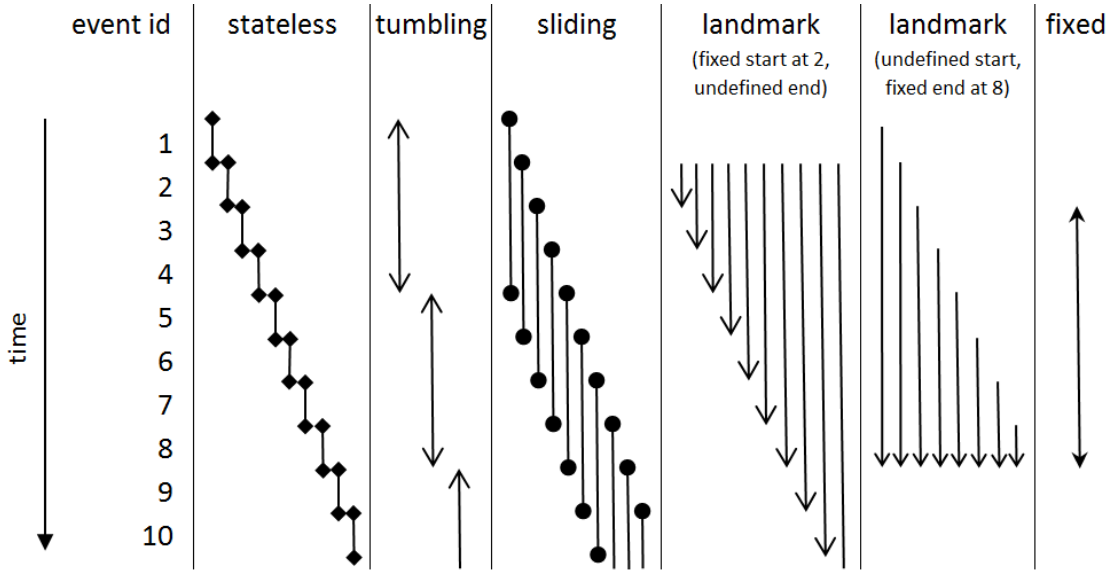


Figure 5.2: Query Scopes.

on which events the query is evaluated. Having ten input events, the first column shows that a stateless query is evaluated ten times - for each event once. Assuming a window size of 4 events and an offset of 0, a stateful query with a tumbling window is evaluated twice, as two windows of size 4 are already complete (window $\#1$: events 1 to 4, window $\#2$: events 5 to 8). A third window is started (window $\#3$: events 9 and 10), but the query is not evaluated yet, as two events are still missing. For the same assumption, a query is evaluated more often with a sliding window than with a tumbling window, as sliding windows support intersections of input events. Starting at the fourth input element, each further event completes another window. Therefore such a query is evaluated 7 times for ten input events (for each window completed by the events $\#4$ to $\#10$). Four more windows stay incomplete and have to wait for further input events. The lower-bound ($start = 2$) landmark query in the fourth column is evaluated 9 times (beginning with the second input event), while the window size is increasing from size 1 to 9. Another evolved window is going to be processed for each further input event. In contrast, for the upper-bound ($end = 8$) landmark window in the next column a new window is started with each incoming event and all of them are closed at position 8. Therefore such a query would be evaluated 8 times. The fixed query is evaluated only once when the specified window is completed.

Stateless and stateful queries with a tumbling or a fixed window consider each incoming event only once. So each input event just influences one output event. Queries with sliding windows consider each input event multiple times. On average an event is processed once on each position in the window. Only the first and last elements that do not form a complete window are processed less often (events $\#1$ to $\#3$ and $\#8$ to $\#10$ in the example). With landmark windows, the sooner an event arrives, the more often it is processed.

The second dimension, *Operation*, describes how the query actually processes the events. Most of the literature shows that queries usually combine different operations to accomplish a purpose. Further on different terms are used for similar operations, so Figure 5.3 illustrates the common query operations.

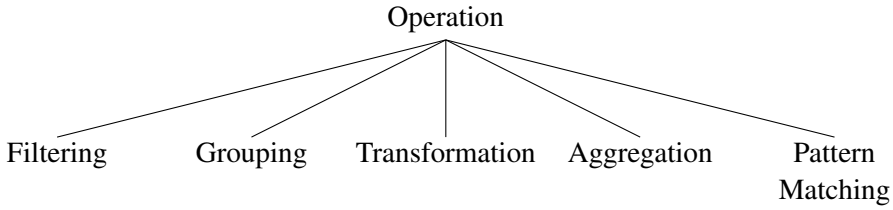


Figure 5.3: Query Taxonomy, Dimension Operation.

Filtering means the selection of special events depending on predefined properties. This operation is often important to reduce the amount of data and therefore, it is often used as a first step in event processing. Filtering is often referred to as *Selection*, since some events are selected and others are not. *Grouping* [128] is a possibility to group input events based on defined properties. The further processing steps differ depending on the group. A special form of grouping is the *group by* clause in aggregation queries. The *group by* operator also performs a type of grouping.

Another query operation is called *Transformation*, which is used to transform events regarding their structure or content. Above all, transformation is important to transform simple events to complex events, which is a main part of complex event processing systems. The terms *Elaboration* [40,121] (through projection or renaming) and *Mapping* [81] (a given input to a predefined output format) are also used to describe this operation. Further forms of transformation can also be *Sorting* and *Joining*. *Aggregation* operations can perform data aggregation within one or several events. Typical operations in aggregations are the calculation of the maximum, the minimum or the average value. These values can either be calculated overall, or for different groups in combination with the *group by* operator. But aggregation does not necessarily mean calculating statistical values. Input events can also be summarized by other rules. The last operation is *Pattern Matching*. This term is often connected to CEP. It means that input events are matched against a specified pattern and an output is triggered, if the pattern is matched. Basically, this operation can be seen as a mix of filtering, transformation and aggregation and is thus often used in combination with these terms. The pattern defines how the events have to be filtered. Using aggregation and transformation, the information of the matched events is used to create the output event.

Even though the terms for classifying queries are not used consistently throughout the literature, common features can be identified. The foundation for the taxonomy is the distinction of the dimension: the scope defines which events are evaluated by the query together and the operation describes what is done with the events. As a last point, different query languages may not support all features, but rather a subset of them. Existing event based systems usually support only one query language and not several, so this should be taken into consideration when deciding on the system.

5.2 Approaches to Handle High Loads

Different approaches to cope with high loads can be found in literature. Most of them have been proposed and implemented in research prototypes of ESP systems. As this thesis tries to compare strategies, which can be applied on a single node, only strategies that meet this requirement are proposed. Nevertheless, the presentation of the strategies also approaches the benefits and problems that arise in the context of using them in the whole EPN.

Resources for these strategies are listed in the Appendix, see Section A.2. The following section presents the outcome of the literature review and combines the findings of multiple references.

As a reminder, the underlying problem of high loads: ESP systems cannot be designed to cope with any amount of data, as the highest amount of data cannot be predicted and this design would be a waste of resources. Therefore, systems are designed to be capable of the average data load and some tolerable peaks. But streams tend to be bursty [69] and situations with a load magnitudes higher than the average load can occur. Another detail of such scenarios is that such peaks are caused by a certain occurrence in the context of the system, which also leads to extraordinary values in the incoming events compared to the values during normal processing. So one can assume that events are more important during such peaks. The following strategies should be used to manage peaks of high load and to keep on processing the events with as much accuracy as possible.

5.2.1 Management of Strategies

Strategies for handling load bursts are typically managed in two parts: On the one side, there must be a controlling component and on the other side, a monitoring component is required to maintain an information base for controlling decisions.

The management of strategies can either be realized centralized or decentralized. If the management is done decentralized, each node can maintain itself, but therefore no overall optimization for the system can be done. Additionally, the monitoring and controlling also represents load for a node, so it does not improve the situation in case of overload. In contrast, if the management is done centralized, this component represents a single point of failure. In return, the monitoring and controlling does not waste that much resources on nodes, which have to process events, but still it increases the network traffic to get information from the nodes. Nevertheless, the component cannot use too complex algorithms for monitoring and controlling, as in case of an overload decisions have to be made quickly. A good combination of centralized and decentralized management approaches might be a good solution.

Monitoring of the System

Monitoring the system is an essential activity in order to reasonably apply load handling strategies. A system has to know whether there is an overload at any node or not. If there is an overload, more detailed information on the reason is desirable. An overload can have different causes. For example a tuple input rate that is too high or an inefficient EPN. Detailed knowledge on the cause may improve the choice of the right strategy.

A monitoring component is responsible to track the QoS data concerning the system. This includes the throughput and the workload of all the nodes in the EPN. More detailed monitoring on individual streams and queries is desirable. Statistics and characteristics of input streams are in particular of interest, as these values can be used to estimate intermediate results. A synopsis maintained by a sample or compressed wavelets can be used for this.

Controlling of the Strategies

Every time the *load equation* [12] ($processingrate \geq inputrate$) is not fulfilled, the controlling component has to prepare a strategy to reduce the load on the affected node. If the load equation is not fulfilled for a longer time, the memory is used up and no further memory is available for incoming events. Before this state is reached, the strategy has to be applied. It is the responsibility of the controlling component to provide appropriate configuration values for the strategy. Further on, the success of the strategy has to be supervised. If the strategy could not firm the system, it has to be modified. Otherwise, the controlling component has to determine, whether the system can currently only work with the applied strategy, or if the system is back in a normal state and the strategy can be removed. If a strategy is removed too early, the system is overloaded again and a new strategy has to be applied. This would cause superfluous overhead.

So the controlling component is a very complex component and has to be well configured and it needs detailed and up-to-date information from the monitoring component to be able to provide optimal overload handling.

5.2.2 Load Shedding Strategy

Load Shedding is one of the simplest strategies that can be performed, when a system suffers from too much load. In short, load shedding just means to drop a certain portion of incoming events (tuples), so that the remaining processing effort is decreased and therefore, the throughput is increased. But logically, this dropping leads also to a decreased quality of the results.

Load shedding has not been invented in the context of ESP. It has already been used for a long time in other areas, like multimedia or networking applications, where it was rather called *package discarding*. In ESP, common descriptions for load shedding are also 'gracefully degrade performance' or 'graceful degradation'.

The four main questions when using load shedding are:

1. How should the tuples be shed?
2. When should tuples be shed?
3. Where should the shedding be done?
4. How many tuples should be shed?

Types of Load Shedding

Different types of load shedding can be performed. Distinctive features are the algorithm to identify the tuples to be dropped, the level on which the tuples are dropped and the realization of the dropping.

Varieties. The simplest way is *naive load shedding*, which is also called *random load shedding* or *sampling*. The naive approach has one parameter, the sampling rate. This parameter p ($0 < p < 1$) defines the probability of a tuple to be processed, whereas $(1 - p)$ is the probability of dropping the tuple without any further processing. So independent of the content of the event, each one has the same chance to get processed or dropped. By configuring the sampling rate p the amount of shed events can be controlled. This is relevant or the fourth question. Another approach is *semantic load shedding*. In this case only events, which are claimed to be less important, are shed. The importance can be evaluated based on different criteria: the actual content or a utility function. Content-based filtering means that the strategy can be configured with criteria data that identifies more or respectively less important tuples. In the other case, a utility function to calculate the utility of each event is established for the system. Tuples with a higher utility will be processed, whereas tuples with a low utility are dropped. A certain threshold has to be set to define which tuples will be dropped.

Realization. The actual load shedding can be implemented in different ways. One way is to use *dynamic query rewriting* to modify the query and filter tuples, which are dropped. The same filtering mechanism can also be implemented by adding *filter operators* into the EPN. Special *drop operators* can be used instead of those filter operators as well. In either case, for load shedding to work efficiently, the actual effort of shedding a tuple (and deciding whether a tuple is shed or not) should be minimal in comparison to the processing of the tuple.

Management of Load Shedding

The management deals with the question, when load shedding should be performed. This question has already been discussed in Section 5.2.1. Also the remaining two questions have to be handled by the management component for overload strategies, but as there are peculiarities for load shedding, they are discussed in the following sections.

Placement of a Load Shedder

The placement of a load shedder is important as it has impact on the efficiency of the strategy and answers the question, where load should be shed. As tuples are dropped by a load shedder, no further processing is done for the tuple. Therefore, one could assume that every processing step of the tuple that has happened before is redundant. This leads to the statement that optimal load shedding is always performed at the beginning of an EPN, otherwise it is not optimal as resources might have been wasted.

Nevertheless, it is not suitable to always put load shedders at the beginning of an EPN, especially if load shedding should not be applied for all queries in the same way. Figure 5.4 illustrates how the placement of a load shedder affects the impacts of the strategy.

If the load shedder $s1$ is used, the shedding takes place at the very beginning. Therefore, no processing resources have been used for tuples and no resources have been wasted. But the load shedder affects two queries: the ones that are consumed by $c1$ and $c2$. But in some cases, this might not be a desirable handling of overload. If the query consumed by $c1$ has a much higher priority or the reason for the overload is just the event processing on $epa4$, then it might be a

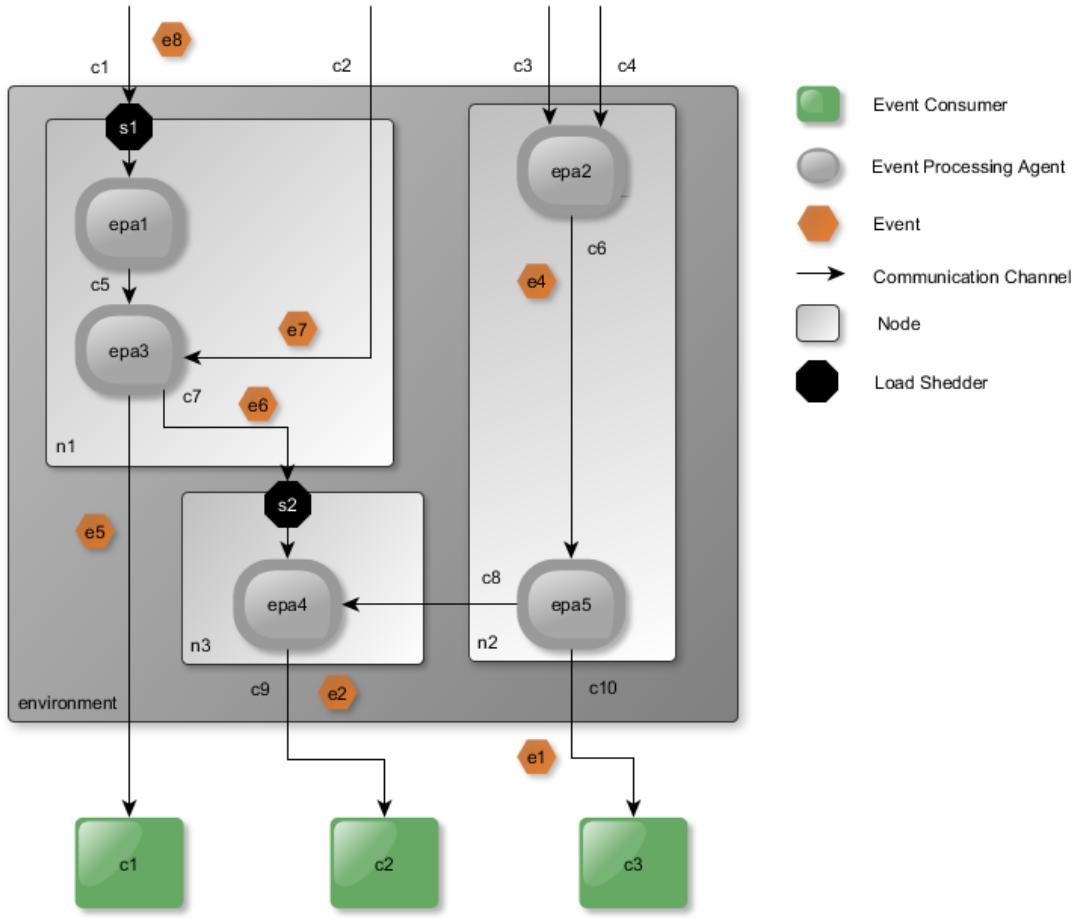


Figure 5.4: Placement of a Load Shedder.

better solution to use the load shedder $s2$. The optimal placement of a load shedder strongly depends on the level of the overload handling as well as the priority of streams or queries.

Level of Load Shedding

The load shedding approach can be applied on different levels. The level characterizes in more detail where the load will be shed. On the one hand tuples can be shed either right after they have arrived, on the other hand they can also be shed right before the actual processing. Another possibility to influence the level is whether the load shedding is performed for all queries, which are currently executed, or only for some queries. So the queries could have priority values and the higher the priority the smaller is the probability that events of the query will be shed. The

same distinction can be made for streams. Performing the load shedding just in general without any regard to streams or queries is a simple approach, but a great benefit of a finer load shedding is that it is suited better for typical situations. For example, when a node is overloaded, the cause is probably not a burst on all input streams, but rather on one input stream, while the others have a normal data rate. Then it might be a good idea, to perform load shedding only on this input stream. The same situation can occur on query level. In an overload situation it is likely, that one query is too complex and takes too much processing time and all other queries suffer under this load. Then it is advisable to perform load shedding only for the complex query, so the workload is decreased but the results of the simple queries are still correct.

Configuration of Load Shedding

The configuration of the applied load shedders deals with the question, how many tuples should be shed. This question can be represented as an optimization problem. The goal is to shed as many tuples as needed so that the load in the system becomes manageable again. But naturally there are some constraints that limit the amount of tuples that can be shed. These constraints represent the accuracy of the results. Depending on the query type, the accuracy of the query results is degraded by a load shedder. Depending on the QoS agreements or the fault tolerance of upstream activities, a certain accuracy has to be maintained. If the system does not provide any QoS contracts, the management component can just use load shedding with a high drop-out rate. Otherwise the component has to find a proper solution for the optimization problem.

5.2.3 Deferred Execution Strategy

Deferred execution is an alternative strategy to handle high load, where the processing load is temporarily reduced and postponed to a later point in time. Normally, deferment is rather used during normal processing to favor more important events. Different approaches are used for this prioritization: either less important events are deferred and the execution is postponed or the events are reordered in advance.

Deferment can also be used to reduce the current overload by persisting events that cannot be processed with the available resources at the moment. Those persisted events are processed after the peak of high load when further system resources are available. But the strategy can only be used efficiently if the operation of persisting the events is not too expensive. Further on, if the claim to the accuracy of the results is very high, the strategy cannot work as efficient as with less strict requirements.

First, if the ordering of results is very important, then the strategy cannot just persist the current overload of input events and proceed with the processing of new incoming events. Instead all incoming events have to be persisted and just the oldest events have to be loaded each time when processing resources are available. This leads to a lot more write and read operations. In contrast, if the output order is not that important, a bunch of events can be persisted in case of an overload but no events have to be loaded for the ongoing processing, so more resources can be used for the actual processing during the peak of high load.

Secondly, results of stateful queries are affected in a different way than the results of stateless queries. For stateless queries, the correct results are produced but the ordering might be different.

In stateful queries, the persisting of events leads to different events per window and therefore the results of the deferred execution differ from the results of a normal execution, except when the additional write and read operations to maintain the correct order are accepted like explained in the last paragraph.

In the following subsections, the deferred execution strategy is explained in more detail using the same structure as for load shedding. The management of the strategy is discussed separately, as this topic has already been discussed in Section 5.2.1.

Types of Deferred Execution

Some distinctive features of deferred execution types have already been addressed in the introduction. A closed differentiation is provided here.

Varieties. The variations of deferred execution arise from the approach of event persisting. The order of events can either be preserved or not. The difference is illustrated with the aid of a minimalist example in Figure 5.5.

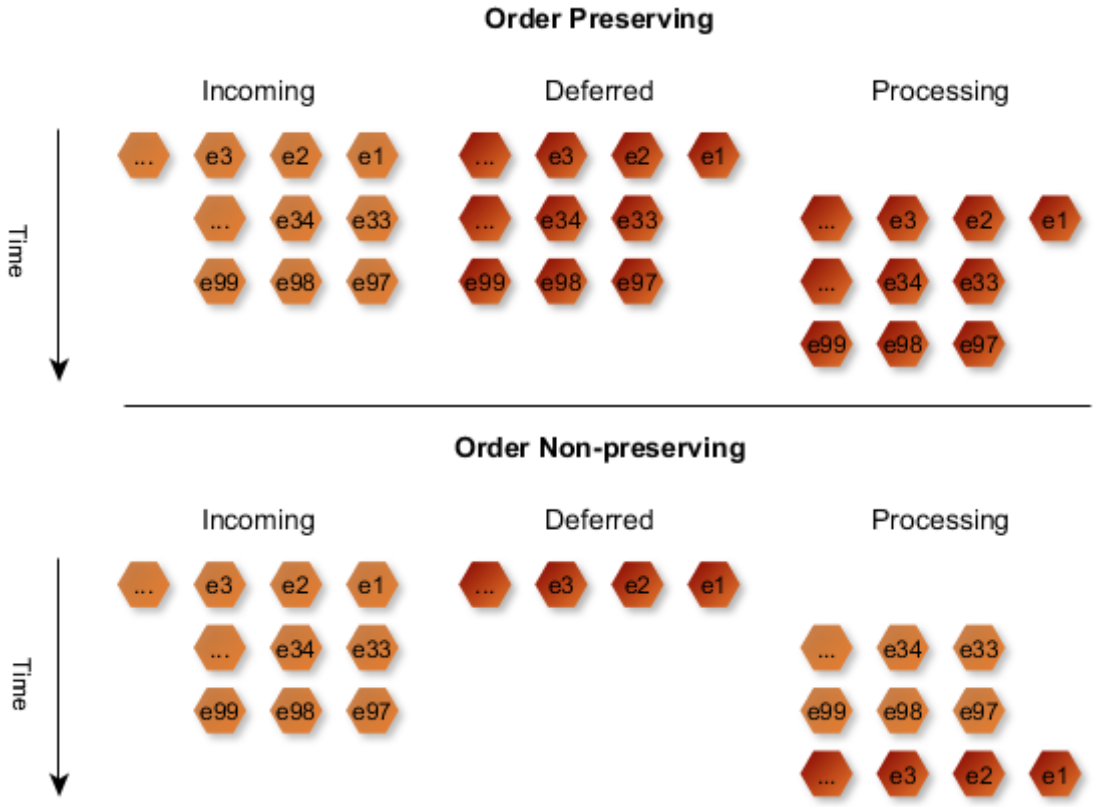


Figure 5.5: Variations of Deferred Execution.

The first part (*Order Preserving*) shows a deferred execution that preserves the order of incoming events. In the second part (*Order Non-preserving*), the order is not preserved as in-

coming events are only preserved when the overload occurs, but if events that arrive shortly afterwards can be processed, they are processed immediately without deferment. All tuples, which are deferred, are depicted in red, so one can see, that in the first approach, all incoming tuples are deferred and therefore it takes three write and read operations to process the tuples. In comparison, the second approach only needs one write and load operation. So even in this small example, the second approach is three times more efficient than the first one. Even though the second approach may need to defer some additional events if the peak lasts for a longer time, the amount of write and read operations still remains very small in comparison to the first approach. In case of stateless queries the results do not differ for the two approaches but the order of the output events changes with the second approach. For stateful queries, the processed results may not be correct anymore, if the second approach is used. The windows that are processed are not filled in the same order as without the deferment, so the calculated results are different. The errors can be minimized, if the set of deferred tuples is of the same size as the window, but for sliding windows the errors cannot be removed entirely.

Realization. Different realizations are possible according to the system that uses the deferred execution strategy. Basically, different realizations differ from each other in the way of how events are persisted and later reloaded. Possible realizations are storing and retrieving using databases or files. The actual realization should be chosen based on performance criteria. No available operators can be used to realize the deferment of events, therefore this strategy has to be integrated in the system in a different way, or a new operator has to be developed.

Placement of Deferred Execution

In contrast to the load shedding strategy, the placement of the deferred execution is not as critical or complex. As the processing of any event will be completed sooner or later, no previous processing steps are unnecessary. Therefore, the deferment can be performed at the exact node that is affected by the overload. So the strategy is also appropriate for dealing with high loads on a single node.

Level of Deferred Execution

The possibilities on which level the deferment of events can be performed are equal to the different levels of a load shedding strategy. Depending on the characteristics of the overload cause, general deferment or a deferment on stream or query level can be appropriate.

Configuration of Deferred Execution

One parameter that is relevant for this strategy is the amount of tuples that should be persisted at once. Persisting each event individually produces a lot of overhead. But if too many events are persisted in one operation, it takes more time and in the meanwhile new events that cause more overload can arrive, which is why a good trade-off is desired.

The same applies to the loading of the persisted events. When the peak of high load is over, the events have to be loaded and processed. Depending on the workload, more or less events can be read at the same time without causing an overload again.

5.2.4 Forwarding Strategy

The forwarding strategy uses additional resources (like additional EPAs or nodes) to cope with the high load and therefore belongs to the *scale up techniques*. Strategies that use distribution and parallelization approaches also belong to this category. In case of an overload, the ESP system has to provide additional resources for the query processing. These can either be newly added components or reused components, that are not busy at the moment. The new component is integrated into the current processing workflow and undertakes a part of the processing effort.

In comparison to other scale up techniques, which are more present in the literature, forwarding does not modify the whole EPN and redistribute the operators, it just shares the load of one EPN by outsourcing a part of the processing effort. Other components in the system are not affected by this strategy.

Forwarding produces additional effort, as the original node has to establish a connection to the used nodes, to configure the query processing on these nodes and also the forwarding itself and the returning of the results takes time. This effort has to be traded off against the gained performance to see whether a forwarding strategy makes sense.

In the following subsections, the forwarding strategy is explained in more detail using the same structure as for the strategies before. Again, remarks to the management of the strategy can be found in Section 5.2.1.

Types of Forwarding

Concrete forwarding strategies can vary in the following properties.

Varieties. One distinctive feature is the provision of the additional resources in form of processing nodes. A system can either use existing nodes, which are not busy, or it can add new nodes in case of an overload. Ideally, existing nodes are used as long as there are still nodes that can deal with further event processing tasks, and new nodes are started only if the overload can not be forwarded to an existing node anymore.

Another property of a forwarding strategy is whether the forwarding is stateless or stateful. In case of stateless queries, it does not make any difference and stateful forwarding would not make sense. In case of stateful queries, stateless forwarding cannot preserve the order of the events and therefore leads to different results than the normal processing. As a simple example, assume a stateful query with a sliding window of size 2 and eight incoming events. In variant A, the events are processed without forwarding. The input order of the events is preserved and the results are computed correctly, see second column in Table 5.1.

Variant B uses stateless forwarding with two nodes. So the processing of the incoming events is shared by three nodes and each event is distributed among the nodes using a Round-robin balancing. The third column of Table 5.1 shows which windows are calculated on which EPA. The results of the processing reveal qualitative and quantitative errors. Caused by the forwarding, windows are processed on three different nodes. So on each EPA a separate window is created and the first result is processed, when the first window is filled with input events. Not until the fourth incoming event, a window is completed and the first result is computed. In comparison to normal processing without forwarding, the fourth input event results in the third output event. So we can see, that because of the three windows instead of one, not all combi-

nations of input events are processed, which leads to less results and therefore to a quantitative error. This quantitative error increases with the amount of nodes used for forwarding and the window size.

Table 5.1: Stateless Forwarding Strategy with a Sliding Window

	Variant A without forwarding	Variant B forwarding (2 nodes, 1 event)	Variant C forwarding (2 nodes, 2 events)
$epa1$	$(1,2)$, $(2,3)$, $(3,4)$, $(4,5)$, $(5,6)$, $(6,7)$, $(7,8)$, $(8,?)$	$(1,4)$, $(4,7)$, $(7,?)$	$(1,2)$, $(2,7)$, $(7,8)$, $(8,?)$
$epa2$		$(2,5)$, $(5,8)$ $(8,?)$	$(3,4)$, $(4,?)$
$epa3$		$(3,6)$, $(6,?)$	$(5,6)$, $(6,?)$

Further on, the stateless forwarding leads to incorrect combinations of input events in the processed windows. Consecutive events are never processed by the same node, so none of the processed windows is equal to a processed window in Variant A. Variant C is a special case. It also uses two additional nodes for forwarding, but two consecutive events are processed by the same node. So the Round-robin is performed for sets of incoming events with the size of 2 (equal to the window size). In this case, the strategy also produces results with a quantitative error, as, like before, three separate windows are started on the nodes. But as the window size and the amount of collectively forwarded events are equal, some of the processed windows are all equal to windows, which have been processed in variant A. So there the qualitative error is smaller than for variant B.

Table 5.2: Stateless Forwarding Strategy with a Tumbling Window

	Variant A without forwarding	Variant B forwarding (2 nodes, 1 event)	Variant C forwarding (2 nodes, 2 events)
$epa1$	$(1,2)$, $(3,4)$, $(5,6)$, $(7,8)$	$(1,4)$, $(7,?)$	$(1,2)$, $(7,8)$
$epa2$		$(2,5)$, $(8,?)$	$(3,4)$
$epa3$		$(3,6)$	$(5,6)$

In comparison, assume a tumbling window instead of the sliding window. The results for the different variants are shown in Table 5.2. Variant A obviously processes less windows compared to the sliding window example, as the windows cannot overlap. For a tumbling window, Variant B produces almost the correct amount of results. The quantitative errors depend on the amount of nodes and the window size, but it is smaller in comparison to the sliding window example. As in the sliding window example, the qualitative errors are based on incorrect combinations of input events in the processed windows. Variant C on the other hand produces the correct results.

So if the forwarding is used for a bunch of events with the same size as the window size, stateless forwarding is an efficient strategy for a tumbling window query.

Stateful forwarding requires a shared memory for the different EPAs. Only one window is used for the incoming events but the windows are processed by different EPAs. The shared memory adds a lot of complexity to the event processing. Even though the input order is preserved with stateful forwarding, additional coordination is required for the coordination of the results to be able to guarantee the correct ordering of the results.

Realization. The concrete realization strongly depends on the environment of the ESP. The environment predetermines how nodes can be added or reused. For the communication between the original EPA and the EPAs used for forwarding the normal communication infrastructure can be used or additional channels can be added.

In addition, for this strategy the node has to provide an input channel for the result events of the nodes used for forwarding. These events have to be received and forwarded like normal output events.

Placement of Forwarding

As all events are going to be processed with the forwarding strategy, it can be used on every node. If stateless forwarding is applied on a stateful query, it may produce better results to place the forwarding strategy near the end of the EPN. Otherwise the processing errors are introduced in the beginning and may increase during the further processing steps.

Level of the Forwarding

As for the other strategies, forwarding can also be applied on different levels. But for this strategy it has to be considered that different levels lead to a different amount of additional effort. This effort adds up by the following parts: in each case, an effort has to be made to request the additional EPAs and to establish a connection to them. Further on, the queries, which have to be processed for the forwarded events, have to be configured on the used nodes. This effort depends on the level on which the strategy is applied. If the forwarding strategy is applied for the whole node, all queries have to be configured on the used nodes. If the strategy is applied on the query level, only the affected query has to be configured. A middle way is the event stream level, as only the queries that process events of this stream have to be configured. Finally, the additional effort is also increased by the actual forwarding. This effort depends on the level, but also on the load. If most of the load is caused by one query, then the difference between forwarding on the query level or for the whole node is not that big. So it is more influenced by the load as by the level.

Configuration of Forwarding

The most important parameters to configure the forward strategy is the amount of nodes that should be used for the forwarding. In addition, a flag can be used to indicate whether the original node should continue to contribute in the processing or a parameter to specify how many of the events should be processed by the original node can be used. If the node performs no further

processing but only forwards the events and receives the results, it is called *pure forwarding*. Further on, if a set of consecutive events should be forwarded, the size of this set should be configurable. If the strategy is applied on the query level, it is also possible to use a flag that indicates whether the size of this set should be retrieved from the window size of the query.

5.3 Integration into the WS-Aggregation Framework

The goal of the thesis is the integration of the approaches presented in Section 5.2 into the WS-Aggregation framework. The thesis focuses on the integration and evaluation of the strategies for the different types of queries defined in 5.1. This section discusses the design issues of the implementation and integration into the framework based on the features that have been used to describe the strategies: management, placement, level, configuration and type.

As already stated, the thesis covers only the application of load handling strategies on one node. An automated *management* of the strategies for monitoring and controlling is therefore not provided. The strategies are being applied manually to ensure a controlled situation in the evaluation. As the strategies are evaluated only on one node, the decision on the *placement* of the strategy is obsolete. The implementations of the strategies for the WS-Aggregation framework should provide the possibilities for a flexible *configuration*, as explained in Section 5.2. The strategies are only applied with predefined configuration values in the scope of this thesis, but they should be variable for the further usage in the framework, so that a monitoring and management component can apply strategies with adjusted configuration values.

The integration of the strategies should not be realized on a deep layer of the framework, but rather on a higher one. So the strategies should not be integrated on the level of the query processing engine. This enables three options for the *level*, on which the strategies can be applied: on the level of the node, of an event stream or of a query. The node level would be the most general solution as all events are treated equally. This would not be very adaptive, therefore only the other two levels are qualified for the implementation. The stream level enables the prioritization of different streams, whereas the query level enables to prioritize queries. So both levels offer more or less equal opportunities as the integration into the WS-Aggregation on the stream level seems to be the reasonable choice based on a costs-benefits analysis.

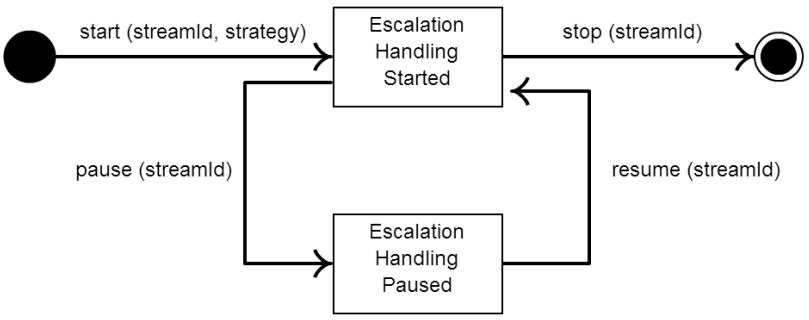


Figure 5.6: Escalation Handling State Diagram.

The term *Escalation* is introduced to describe an emergency situation, when the load is too high and the correct processing of events cannot be guaranteed anymore. Therefore, an escalation must be handled by a proper load strategy. The process of escalation handling is depicted in Figure 5.6.

When an escalation occurs, the escalation handling can be started. As the handling is performed on the level of event streams, the id of the stream has to be specified along with the strategy that should be used for the handling. A running escalation handling can either be stopped, if it is assumed that no further handling will be required soon, or paused, if the escalation handling should just be discontinued for a short time. After pausing an escalation handling it can be resumed, which switches it back to the running state. No explicit possibility to change the current escalation handling is provided, but the escalation handling for a certain stream can be modified by starting a new escalation with modified parameters or a different strategy for the same stream. The old handling will be stopped correctly and the new strategy will take over.

The framework is extended by an aggregator node that provides further capabilities for burst handling. The original aggregator node of the WS-Aggregation framework is described in [60]. The *Aggregation Interface*, the *Metadata Interface* and the *Management Interface* can be used to request the node. Internally, the tasks of the aggregator node are performed by the following components: *Request Distribution Engine*, *Target Service Invoker*, *Multicast Engine*, *Configurator*, *Performance Monitor*, *WAQL Engine* and *Registry Proxy.*

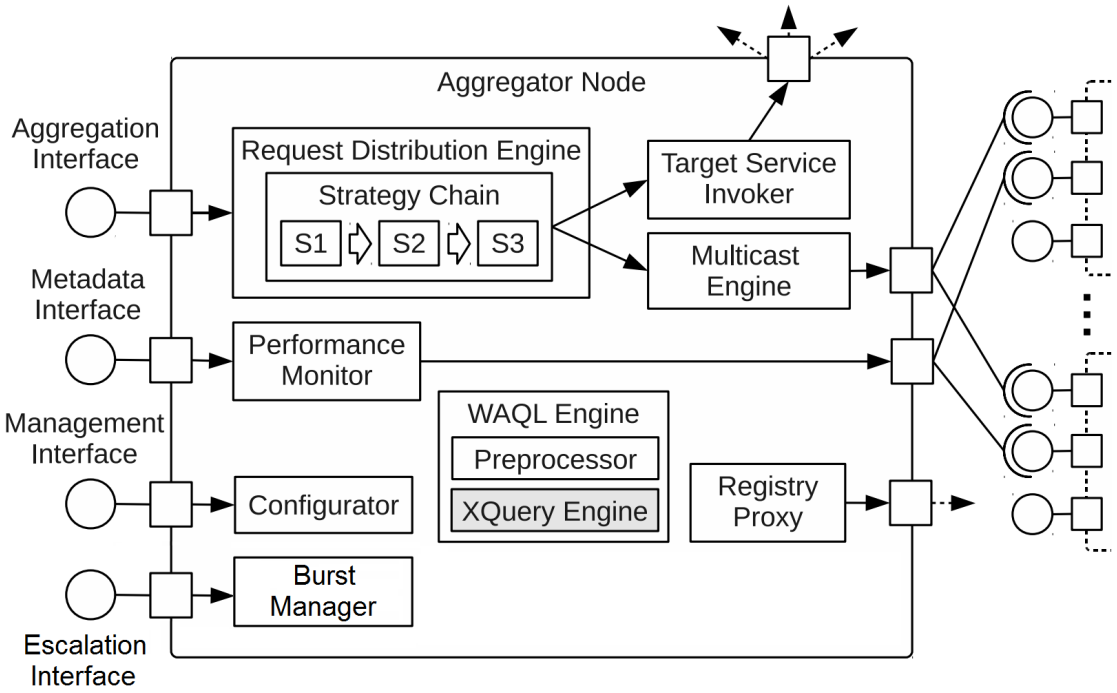


Figure 5.7: Burst Capable Aggregator Node (Based on [60]).

The integration of the burst handling is depicted in Figure 5.7 The capabilities are integrated in a component called *Burst Manager*, which is controlled by the *Escalation Interface*.

The burst capable aggregator node extends the original aggregator node and therefore provides the usual capabilities as well as the features of the burst manager. The burst manager checks for incoming events whether the event has to be handled by a burst handling strategy or not. When no escalation handling is configured, the burst capable aggregator performs like a normal aggregator. If a strategy for burst handling is applied, the strategy decides how the event is further processed.

Caused by the integration into the framework on this level, a stateful implementation is not reasonable, so the strategies are implemented in a stateless manner. At first a simple implementation of the strategies is provided and implemented. Later on, enhancements for the strategies are investigated in the evaluation and the summary.

CHAPTER 6

Implementation

This section describes the implementation of the burst handling strategies and the integration into the WS-Aggregation framework. As a first step, the integration of the strategies and the burst manager into the WS-Aggregation framework is explained. Next, the escalation interface definition is presented and finally, the implementation of the strategies and their variations are explained.

6.1 Integration of the Escalation Concept into WS-Aggregation

The concept of the integration has been described in Section 5.3. As explained, the original aggregator node is extended by a burst manager component. A basic class diagram describing the associations and most important methods for this integration is shown in Figure 6.1.

The diagram contains two main interfaces, one for the escalation and one for the burst manager. By using the interfaces *Escalation* and *BurstManager*, the burst handling strategies can be integrated into a framework. The escalation interface defines the basic attributes for managing and controlling the escalation handling. The burst manager interface is the interface between the normal event processing and the burst handling. By these methods, the strategies can request data from the system or provide inputs and results.

For the integration into WS-Aggregation, an implementation of the burst manager interface is provided, see *BurstManagerWSAggr*. To integrate the escalation interface, a new aggregator node, the *BurstCapableAggregatorNode* is implemented, which extends the original *AggregatorNode* and implements the *Escalation* interface. The burst capable aggregator node overrides some methods of the original node to integrate the *BurstManager* component.

The individual components are explained in the following subsection.

6.1.1 Escalation Interface Definition

The *Escalation Interface* defines the operations to manage the burst handling. The three operations map the operations used in the state diagram, see Figure 5.6. The method *handleOver-*

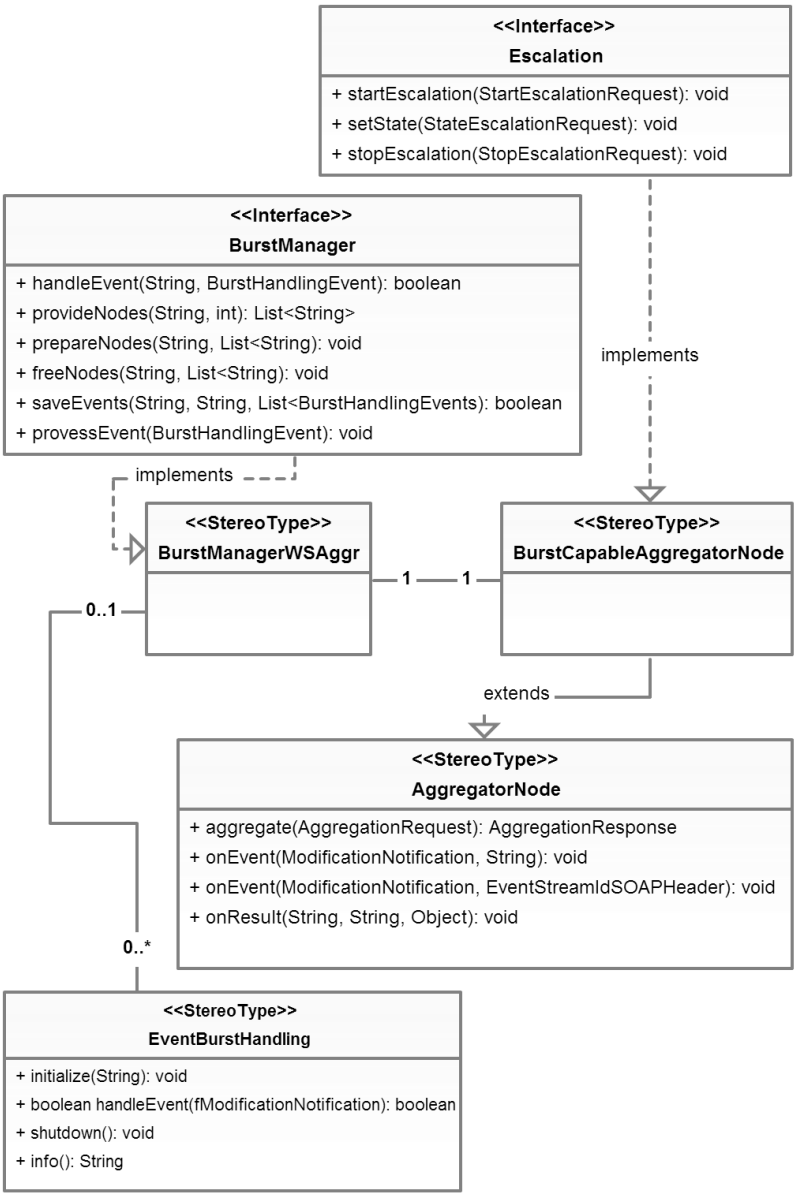


Figure 6.1: Class Diagram.

loadedQuery is used to start the escalation handling. The names of the other methods are self-explaining. The parameters are wrapped in request objects, which contain the parameters for the respective operation. The *StartEscalationRequest* requires an event stream id and the strategy that should be applied. The *StateEscalationRequest* contains the event stream id to identify the escalation handling and a Boolean to identify the state. The *StopEscalationRequest* only

requires the event stream id to identify the escalation that has to be stopped. The interface is shown in Listing 6.1.

```
1  public interface Escalation {
2
3          public void startEscalation(StartEscalationRequest request)
4                  throws EventBurstHandlingException;
5
6          public void setState(StateEscalationRequest request);
7
8          public void stopEscalation(StopEscalationRequest request);
9
10 }
```

Listing 6.1: Escalation Interface.

6.1.2 Burst Manager

The *Burst Manager* component provides an interface and an implementation for the WS-Aggregation Framework.

Burst Manager Interface

The interface of the burst manager defines methods that are required for the burst handling strategies to communicate with the framework. The interface is shown in Listing 6.2.

```
1  public interface BurstManager {
2          public boolean handleEvent(final String eventStreamId,
3                  final BurstHandlingEvent event)
4                  throws EventBurstHandlingException;
5
6          public List<String> provideNodes(final String eventStreamId,
7                  final int amount) throws EventBurstHandlingException;
8
9          public void prepareNodes(final String eventStreamId, List<String>
10                 nodeIdentifiers) throws EventBurstHandlingException;
11
12         public boolean forwardEvent(final String eventStreamId,
13                 final String nodeIdentifier,
14                 final BurstHandlingEvent event);
15
16         public void freeNodes(final String eventStreamId,
17                 List<String> nodeIdentifiers);
18
19         public boolean saveEvents(final String strategyIdentifier,
20                 final String eventStreamId,
21                 final List<BurstHandlingEvent> eventsToSave);
22
23         public List<BurstHandlingEvent> loadEvents(final String strategyIdentifier,
24                 final String eventStreamId)
25                 throws EventBurstHandlingException;
26
27         public void processEvent(final BurstHandlingEvent event)
28                 throws EventBurstHandlingException;
29 }
```

Listing 6.2: Burst Manager Interface.

In general, the burst manager needs to examine each event and decide whether an escalation handling has to be applied or not. So the method *handleEvent* is called for each incoming event and the event is either forwarded to the escalation handling or further processed by the current node.

To establish a forwarding strategy, further nodes are required. The methods *provideNodes* and *prepareNodes* are used to set up nodes, which are used for forwarding. The method *forwardEvent* actually forwards an event to another node and *freeNodes* is used during the shutdown to finalize the event processing of the forwarded events on the used nodes. Depending on the implementation of the forwarding method, the burst manager also has to take care of a way to get the results of the forwarded events.

For the deferred execution, the burst manager has to provide methods to persist and retrieve events, namely *saveEvents* and *loadEvents*. Further on, as deferred events have to be passed on for processing, the method *processEvent* is needed to pass the event to the processing node.

For load shedding strategies, no special methods are needed for interaction, as events are simply not further processed.

Burst Manager Implementation

The implementation of the burst manager represents a mediator between the existing framework and the implemented strategies. It mainly maps the present event stream ids to the escalation handling.

Deferred events are persisted to the in-memory HSQL database, which is also used by the WS-Aggregation framework itself. A persistence entity is used, which contains the relevant information: the event data, a time stamp, the event stream id and an identifier of the escalation handling. These values are then used to retrieve the deferred events for the escalation handling in the correct order.

The forwarding of events utilizes the normal communication infrastructure of the WS-Aggregation framework, but the events get assigned a special event stream id. Therefore, results of forwarded events arrive at the original aggregator node as incoming event. The special event stream ids are used to identify results, so that results ca be moved on to the result handling.

6.1.3 Burst Capable Aggregator Node

The new *Burst Capable Aggregator Node* extends the existing aggregator node and implements the escalation interface. The methods of the escalation interface are implemented and control the actual burst handling for the aggregator node.

The main point of the extension is the integration of the *Burst Manager* component, which particularly affects the *onEvent* method of the aggregator. Each time an event arrives, the burst manager is used to determine whether or not the event has to be processed. Only in case the burst manager confirms that the event has to be processed, the aggregator node continues with the event processing as usual.

In general, the burst capable aggregator node could as well just implemented the second interface, the *BurstManager*, but for a clearer separation the burst manager was implemented

separately. The implementation for the WS-Aggregation is the *BurstHandlingWSAggr*. It implements the *BurstManager* interface and additionally manages the actual burst handling. That means that the burst manager implementation manages which burst handling strategy is currently applied for which event stream. Therefore, although the burst capable aggregator node (implementing the escalation interfaces) gets the requests concerning the control of strategies, it forwards it to the burst manager, which is in charge of keeping track of the currently applied burst handling strategies.

The realization of the escalation handling for the burst capable aggregator node by implementing the two interfaces *Escalation* and *BurstManager* is illustrated in the sequence diagram in Figure 6.2.

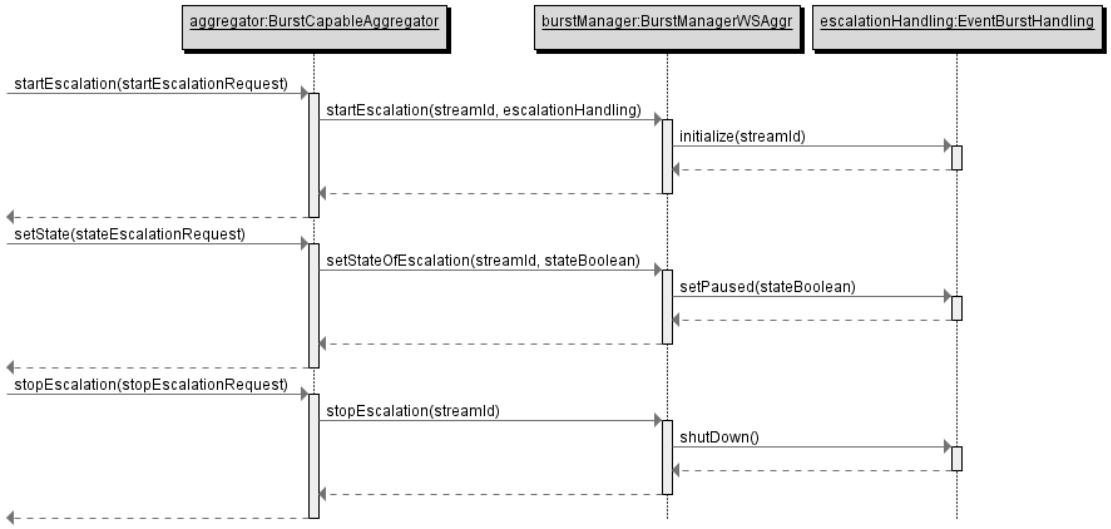


Figure 6.2: Sequence Diagram of the Implemented Escalation Handling.

Further details of the management of the strategies and the actual handling of events during high load are explained in the next section. Similar diagrams for illustration purposes are provided in Appendix B.

6.2 Implementation of the Strategies

The burst handling strategies extend the common base class *EventBurstHandling*. It defines the essential methods for the burst handling. Three different strategies have been implemented: a forward, a deferred execution and a load shedding strategy. For the latter one, three different types have been implemented.

The hierarchy of the implemented strategies is depicted in Figure 6.3. The properties shown in the classes represent the configuration possibilities for the strategies. The implementation of the strategies is explained in the following subsections.

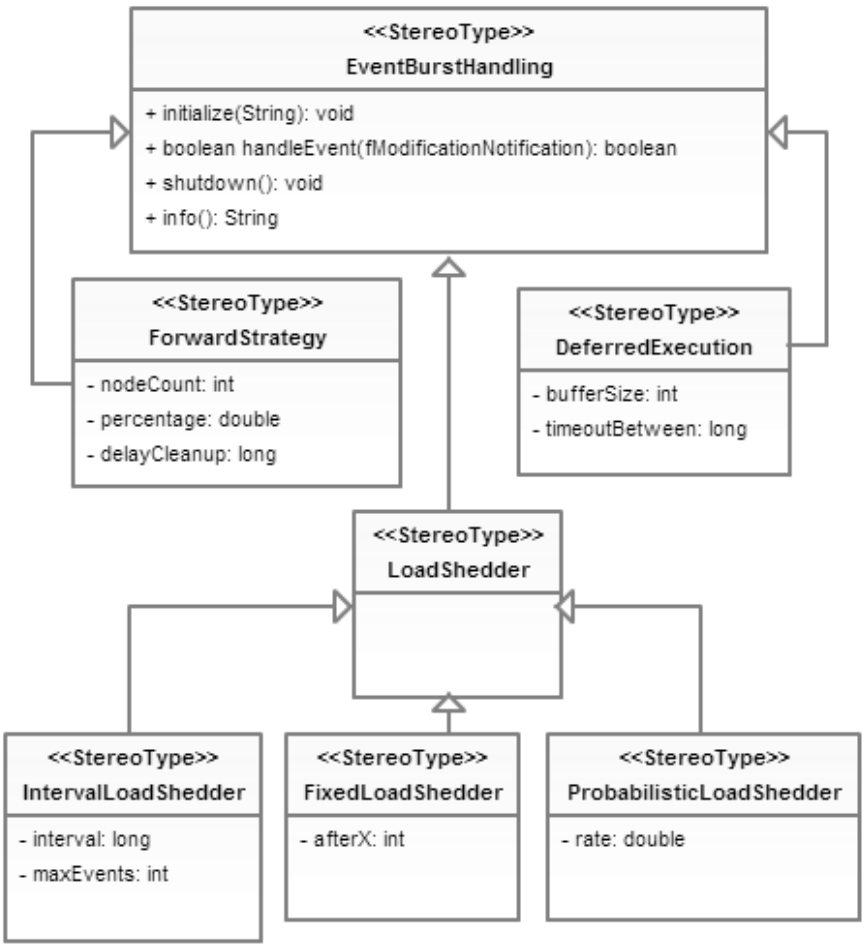


Figure 6.3: Event Burst Handling Hierarchy.

6.2.1 Load Shedding Implementation

The implementation of the load shedding strategies is quite simple. In the initialization phase only some parameters have to be set according to the configuration. When a strategy receives an event, it decides based on the parameters, whether the event should be further processed or not and returns the result. No special actions have to be performed in the shutdown phase.

How the shedding decision is made depends on the concrete implementation of the shedder, see the subsequent sections. The sequence diagram for the event processing during an escalation with a load shedder is shown in Figure B.1 in the Appendix B.

Interval Load Shedder

The interval load shedder can be configured with the following parameters:

interval This parameter defines the duration of one interval in milliseconds.

maxEvents The second parameter states the amount of events that can be processed during one interval.

As long as the maximum number of events for the current interval is not reached, the events have to be processed. Afterwards all events are dropped until the next interval starts.

Fixed Load Shedder

The fixed load shedder allows a certain amount of events until one event has to be dropped. It therefore provides the following configuration parameters:

afterX Defines the amount of events that can be processed before one event has to be shed.

A counter is increased for each event that is received and processed. When the configured amount of events has been processed, one event is shed and the counter is reset.

Probabilistic Load Shedder

The probabilistic load shedder is the most common implementation of a load shedder. It also provides only one configuration parameter:

rate This parameter defines the probability of the event to be shed and therefore has to fulfill the restriction $0 < rate < 1$.

For each received event a random number in the range $[0, 1]$ is generated. If the generated number is smaller than the *rate* parameter, the event is shed. Otherwise the event has to be processed by the node.

6.2.2 Forwarding Implementation

The forwarding strategy uses other nodes to forward events to in order to reduce the load of the current node. The following parameters are provided to configure the strategy:

nodeCount Defines the amount of nodes that should be used for the forwarding.

percentage This percentage value defines the amount of events that should be processed locally. If it is set to 0.0, the current node does not process any incoming events but only forwards them. For any other amount, the current node processes that amount of the incoming events itself and forwards the rest of it.

delayCleanup A delay with the duration of this parameter is used to wait for further result events before the strategy is stopped.

When the strategy is initialized, the burst manager is used to retrieve available nodes for the forwarding. Then the burst manager has to configure all queries that are relevant for the event stream id handled by the escalation on the used nodes. The initialization, when the escalation handling is started, is illustrated in Figure B.2 in the Appendix B.

When an event is received from the burst manager, the responsible node is determined. If the current node is responsible, the event is not forwarded and the burst manager is informed that the event still has to be processed. If a forwarding node is responsible, the burst manager is forced to forward the event and no further local event processing is done. The handling of the results of the forwarded events is not part of the strategy as it depends on the actual forwarding mechanism. The event handling is depicted in Figure B.3 in the Appendix B.

When the escalation handling is stopped, no more events are forwarded. Depending on the parameter *delayCleanup*, the strategy delays the clean up call to the burst manager so that results of forwarded events can be returned to the original aggregator node. The sequence diagram in Figure B.4 in the Appendix B illustrates the shutdown.

6.2.3 Deferred Execution Implementation

The deferred execution strategy is used to store incoming events temporarily in a database. The following parameters are provided to configure the strategy:

bufferSize If the buffer size is equal to zero, no buffering is used. This means that every incoming event is immediately stored in the database. If the buffer size is greater than zero, a buffer with the given size is initialized and the events are persisted when the buffer is full.

timeoutBetween This timeout is used to define how often deferred events should be loaded and processed.

If a buffer should be used, it is initialized during the initialization phase. When an event is received, it is either saved in the buffer or immediately persisted. No events are processed by the current node immediately, so the strategy always returns *false* to the aggregator node. After the amount of milliseconds specified in *timeoutBetween*, events are loaded and handled to the current node for processing. The deferred escalation handling is shown in the sequence diagrams in the Figures B.5 and B.6 in the Appendix B.

During the shutdown phase, events are not deferred anymore but they have to be processed by the current node. In addition, the remaining persisted events are loaded and passed to the node for processing. The *timeoutBetween* parameter is again used to slow down the forwarding so that the current node is not overloaded by the deferred events. The shutdown phase of the deferred strategy is illustrated in Figure B.7 in the Appendix B.

CHAPTER 7

Evaluation

This chapter documents the evaluation approach and its outcomes. The evaluation is divided into two parts: first the evaluation of the strategies applied to stateless queries and secondly, the evaluation in case of stateful queries. This separation is done, as stateless queries are much simpler to handle than stateful queries in case of load bursts and the outcome cannot be directly compared.

At first, the used scenarios for the evaluation are generally described in Section 7.1. Then, in Section 7.2, the implementation of the evaluation and the used scenarios are explained. Afterwards the actual evaluation is performed for the two types of queries, see Section 7.3 for the evaluation of stateless queries and Section7.4 for the stateful queries. First, the strategies are analyzed and assessed regarding the application scenario and the findings are then verified by the execution of the evaluation scenarios. Before the overall summary and the conclusions are documented in Chapter 8, the results are summarized in Section 7.3.3 for stateless evaluation and in Section 7.4.3 for the stateful evaluation.

7.1 Evaluation Scenario

In this section the evaluation scenario and its characteristics for the different query types will be described. The choice of queries is influenced by the features of WS-Aggregation and its query language WAQL as well as by their usefulness in real world applications.

For the evaluation a simple scenario has been chosen. The scenario is an ESP system that processes updates on the stock market. The updates of several stocks can be bundled in one update event representing an input event for the system. The system processes the events and reports the outputs to a specified receiver.

As the evaluation tries to provide results for a comparison of the qualities of the strategies for different query types, the scenario has been adapted for the query types presented in the taxonomy in Section 5.1. In addition, a complex scenario including all query operation types has been defined as in real-world applications a query usually performs more than one operation type.

Regarding the scope, stateless and physical stateful queries with sliding windows are used. Fixed windows will not provide substantial insights and landmark windows have not shown to be prominent in the literature review. Tumbling windows are a special kind of sliding windows, so they are not evaluated separately. As the evaluation uses only one query and one node to test the dependability during high load, there are no further processing steps and thus grouping is not important, so it will not be used as an exemplary query. This also applies for pattern matching. As this operation rather belongs to complex event processing than to stream-based event processing, the example queries for the scenarios will only cover filtering, transformation and aggregation.

The same queries could be used to analyze the strategies for the different scopes. This approach is not used in the thesis as stateful queries enable more complex queries than stateless ones. These more complex queries also have to be taken into account for a good analysis, therefore different queries are used in the evaluation. For an overall evaluation of the strategies, a combined query that uses all three operation types is evaluated additionally.

Nevertheless, all queries are based on the same general scenario and are going to process events of the same input format. So first the general XML schema for the input events of the queries is defined. The queries themselves consist of two parts: the first part defines the scope and declares type and size of the window. In case of stateless queries the window size is set to one. The operation part is the second part and expresses how the events of a window are processed. Since the query definition for the different scopes is the same for all operations, the scope definitions follow the input schema. The different operations are introduced and explained during the analysis of the strategies. As some of the operations do not maintain the format of the input events, the XML schemata for deviating output formats are going to be defined along with the query they belong to.

7.1.1 XML Schema for Input Events

The same input format of events is used for all queries. The XML schema is shown in Listing 7.1. Events that match this schema represent stock updates. An update is compounded of an id and several records, which contain the relevant data of a stock change. Apart from the current price and the relative change, the name, the industry, the currency and the time are included in the data. One stock update contains at least one record and does not further restrict the input. So it can also contain multiple records for one stock (identified by its name). A payload attribute has been added to the schema to be able to vary the size of the input events.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3  elementFormDefault="qualified">
4    <xs:element name="stockupdate">
5      <xs:complexType>
6        <xs:sequence>
7          <xs:element name="id" type="xs:string" maxOccurs="1" minOccurs="1"/>
8          <xs:element name="record" maxOccurs="unbounded">
9            <xs:complexType>
10             <xs:sequence>
11               <xs:element name="industry" type="xs:string"/>
12               <xs:element name="name" type="xs:string"/>
13               <xs:element name="change" type="xs:decimal"/>
```

```
14                  <xs:element name="price" type="xs:decimal"/>
15                  <xs:element name="currency" type="xs:string"/>
16                  <xs:element name="time" type="xs:dateTime"/>
17                  <xs:element name="payload" type="xs:string"/>
18                </xs:sequence>
19              </xs:complexType>
20            </xs:element>
21          </xs:sequence>
22        </xs:complexType>
23      </xs:element>
24  </xs:schema>
```

Listing 7.1: XML Schema for Input Events.

In Listing 7.2 a simple exemplary XML input event that conforms to the presented input schema is shown. The stock update contains two records of different companies.

```
1  <stockupdate>
2      <id>1-123</id>
3          <record>
4                  <industry>IT</industry>
5                  <name>Company ABC</name>
6                  <change>2.9</change>
7                  <price>122.4</price>
8                  <currency>USD</currency>
9                  <time>2012-01-01T12:12:12</time>
10                 <payload>some additional data</payload>
11         </record>
12         <record>
13                 <industry>Banking</industry>
14                 <name>Company XYZ</name>
15                 <change>10.8</change>
16                 <price>79.9</price>
17                 <currency>EUR</currency>
18                 <time>2012-01-01T12:12:12</time>
19                 <payload>some additional data</payload>
20         </record>
21 </stockupdate>
```

Listing 7.2: Exemplary XML Input Event.

7.1.2 Scope Definitions

In this thesis, two scopes are considered for the evaluation: stateless and sliding window queries. The stateful scope is considered to be count-based. Time-based queries are not used here.

For stateless processing, every incoming event is treated separately. As in WS-Aggregation a query cannot be evaluated without a window definition, the stateless processing is realized by a tumbling window with a size of 1, see Listing 7.3. The *start* and the *end* clauses define the size of a window. Since both clauses just evaluate to $true$, a window contains just one event.

```
1  for tumbling window $w in $input
2  start at $spos when true()
3  end at $epos when true()
4  [...]
```

Listing 7.3: Stateless Scope.

The definition of the stateful scope looks similar. Again, the *start* clause opens a window at each new event by simply evaluating to $true$, whereas the *end* clause now only closes a window if a certain offset between the start and the end position is reached. The sliding window definition is shown in Listing 7.4. For the evaluation sliding windows with an offset of one are used, so a new window is started for each incoming event. This listing uses an exemplary window size of 5000.

```
1 for sliding window $w in $input
2 start $s at $spos when true()
3 end $e at $epos when ($epos - $spos) >= 4999
4 [...]
```

Listing 7.4: Stateful Scope (Sliding Window).

7.1.3 Queries

The three query types are shortly explained in the context of the evaluation scenario. The actual queries are introduced during the evaluation.

Filtering is used to get a relevant subset of the given events. For example, when monitoring a stock market, certain events may be interesting and should therefore be reported. The output format of a filtering query is usually the same as the input format. Filtering just omits some of the events, while the rest is passed on unchanged. Filtering certain elements of input events and returning modified output events may also be passed for filtering, but as the structure of the element changes, such queries are classified as transformation queries in this thesis. Thus the schema for the results of the filtering query is the same as for the input events, but the emitted events may contain less records.

A *Transformation* query can have the same output format as the input format, if it just transforms values of events. Like in the stock market example, the currency may be transformed. Often the transformation of the event structure is desired too. For example, if the resulting event should be used as input for another processing step, which requires a different format. Especially if an application collects data from different sources and wants to process them together, it often needs to convert the inputs into a uniform format. This can be done by transformation queries. Both types of transformation are considered in the evaluation.

An *Aggregation* query is used to condense the information of multiple single events, for example the average value of stock changes. Analogous to descriptive statistics, key figures are used to describe a huge amount of data. Without further restrictions all data is aggregated to one result. Using the *group by* operator, a restriction is added and the aggregation is done for different sets of events. Even pattern matching can be seen as an aggregation operation, as multiple records are condensed to the desired information (pattern occurred or not occurred). In any case, the output schema of an aggregation query can differ from the input schema, so it may also be classified as transformation query.

The *Combined* query uses filtering, transformation and aggregation at once. The output schema of the results is likely to be different from the input schema.

7.2 Implementation of the Evaluation

One goal of the evaluation is to provide a very simple set up with a basic environment. This environment is realized as a cloud with the cloud operating system OpenStack[1].

As the target of interest is a single burst capable aggregator node, which is stressed by too much data, this aggregator is deployed on a separate node. Other required components, for example. The registry and the gateway, are deployed on other nodes. As one node does not necessarily generate enough load to really stress the aggregator node, several data service nodes are used.

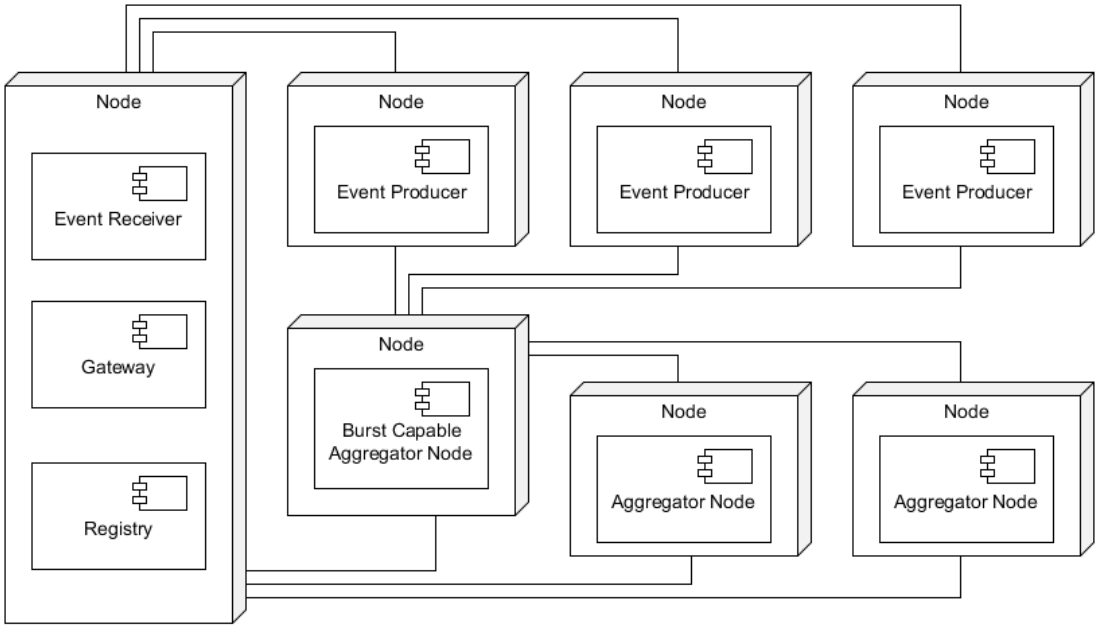


Figure 7.1: Deployment Diagram for the Evaluation.

If the evaluated strategy needs further aggregator nodes, these nodes are provided with the same setup as the burst capable aggregator node. The setup is depicted in Figure 7.1.

All nodes used in the cloud have the same system features, see Table7.1:

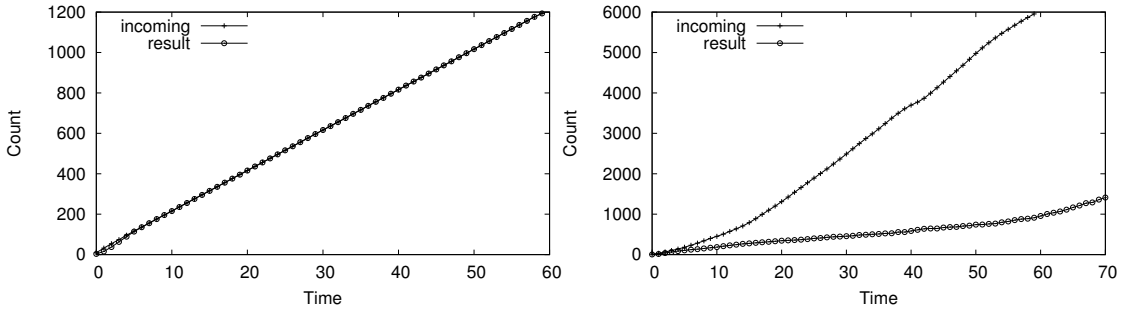
Table 7.1: Node Capabilities

Property	Value
Flavor	$m1.small$
RAM	$1GB$
VCPUs	$1VCPU$
Disk	$40GB$

[1] http://www.openstack.org/ (Accessed: 2014-01-22)

First the registry and the gateway have to be deployed successfully. Then all other nodes can be started and register themselves in the registry. The two additional aggregator nodes are only needed, if the forwarding strategy is applied. A controller application states the aggregation request using the gateway, provides the query used for the evaluation and registers the event producers as inputs and the event receiver as the consumer of the request. If a burst handling strategy is applied, the controller then starts the escalation on the burst capable aggregator node. When the event producers start to send events, they send it directly to the burst capable aggregator. The events are handled by the aggregator itself or forwarded to other aggregator nodes. In either case, the results are sent from the burst capable aggregator to the event receiver. The simulated overload is caused by a combination of a high event frequency and big sized events resulting in a high volume of the incoming stream

As these strategies are only applicable if the overload can be treated by reducing the processing time of an event, the input data is chosen in a way that the processing of an event requires some effort. If the overload is only caused by the input / output operations whereas the effort for event processing is relatively low, those strategies are of no use. As an example, see Figure 7.2. With small event sizes and a moderate input frequency, the node manages to respond immediately (Figure 7.2a). When a higher input frequency is used, the node cannot manage the events anymore, but as the main effort is caused by receiving and saving the event and finally sending the result (Figure 7.2b) the overload cannot be handled with the presented burst handling strategies.



(a) 20 Events / Second

(b) 100 Events / Second.

Figure 7.2: Stateless Filtering Query with One Data Service Node.

The evaluation will use similar input data and configurations per query for each strategy, so that the strategies can be compared. Conclusive, the results are interpreted to define features, which are relevant for choosing the correct burst handling strategy.

The recording of the evaluation scenario is realized through ApectJ (see Section 4.4. The recording should not be too complex as it uses resources of the burst capable aggregator node, so it is designed rather simple. More complex recording can be performed as well, but it influences the outcome.

See Listing 7.5 for the pointcuts that have been defined to monitor the event processing.

```
1           /**
2            * Starting evaluation. (Get test run info.)
3            */
4           @Pointcut("execution(public void at.ac.tuwien.infosys.highfreq.
5           bursthandling.StartBurstCapableAggregator.main(..))")
6           public void startTestRun() {
7           }
8
9           /**
10           * BurstCapableAggregator received event for processing.
11           */
12          @Pointcut("execution(public void at.ac.tuwien.infosys.aggr.node.
13          BurstCapableAggregatorNode.onEvent(
14            at.ac.tuwien.infosys.aggr.monitor.ModificationNotification,
15            at.ac.tuwien.infosys.aggr.monitor.ModificationNotification.
16              EventStreamIdSOAPHeader))")
17          public void startProcessing() {
18          }
19
20          /**
21           * Aggregator finished event processing.
22           */
23          @Pointcut("execution(public void at.ac.tuwien.infosys.aggr.events.query.
24          EventingQueryCoordinator.addResultFromInputAndNotifyClient(
25            String, String, org.w3c.dom.Element))")
26          public void endProcessing() {
27          }
```

Listing 7.5: Monitoring Pointcuts.

The first pointcut is not used for the actual monitoring, but it extracts information at start-up, which is used for saving the monitored information. The other two pointcuts are used to observe the amount of incoming events and the amount of events that have been processed and forwarded to the receiver. The values are saved each second. Further on, each second runs a timer task, which queries the current overall CPU load and overall memory consumption. Those values are also saved for the evaluation. Both, the CPU and memory load, are queried using the Unix *top* command. The values are saved using the *Generic Test Result* provided by the DSG, which can also be used to create diagrams via *gnuplot*.

In general, the data service nodes produce and send data for 60 seconds. Afterwards they stop. The recording of the evaluation lasts for 70 seconds. This enables an estimation of whether or not the node is able to recover after the end of the high load. To be able to gain more insights on the results, the event receiver, which does not use any resources of the burst capable aggregator, logs all received results.

7.3 Evaluation of Stateless Queries

In this section the strategies *load shedding*, *deferring* and *forwarding* are evaluated when applied to stateless queries.

7.3.1 Analysis

Based on the theoretical background provided on the strategies, the following assumptions can be stated:

Table 7.2: Effects of Strategies on Stateless Queries

	Stateless Query
Forwarding	• complete results • incorrect result order likely • possible delay • forwarding overhead
Deferring	• complete results • possible incorrect result order (depends on the implementation) • intentional delay • persisting overhead
Shedding	• incomplete results • correct result order

The forwarding strategy enables a node to provide all results based on the input, but based on the distributed processing, the order of the results is likely to be incorrect. Moreover, because of the forwarding overhead, a delay of the results is added. In case of an overload, forwarding can only be applied successfully, if the processing time is essentially higher than the time needed to forward the event and return the result. Otherwise, the forwarding strategy needs more resources to forward the events than to process it and the node stays overloaded despite the burst handling. In case of forwarding, it must also be considered that the amount of incoming events is even increasing, as results from the forwarding nodes are received like normal input events.

In contrast, the persisting overhead does not need to be small in comparison to the processing effort, but the node must be able to persist the events without being too busy to receive further incoming events. Depending on the capabilities of the node, events can still be processed in the meantime. If the implementation supports order preservation, the persisting overhead is greater, but the burst capable aggregator will always provide the complete results in the correct order. As the present solution supports order preservation, no incorrect result order will affect the evaluation results. After the overload situation, the node will stay busy until all deferred events have been processed.

Shedding intentionally drops events and therefore cannot provide complete results. However, the delivered results are in the correct order. Events get processed faster and without delay. After the overload, the node is immediately back in the normal state.

Which strategy is suited best for a query mainly depends on the receiver of the results: depending on whether missing results, incorrect order or missing results are tolerable, the best suited strategy can be chosen.

7.3.2 Scenario Execution

The queries are analyzed one after another for the evaluation scenario. The filtering query is used first to demonstrate the procedure in detail. The evaluation is continued similarly for the other queries, but the results are presented in a more compact way.

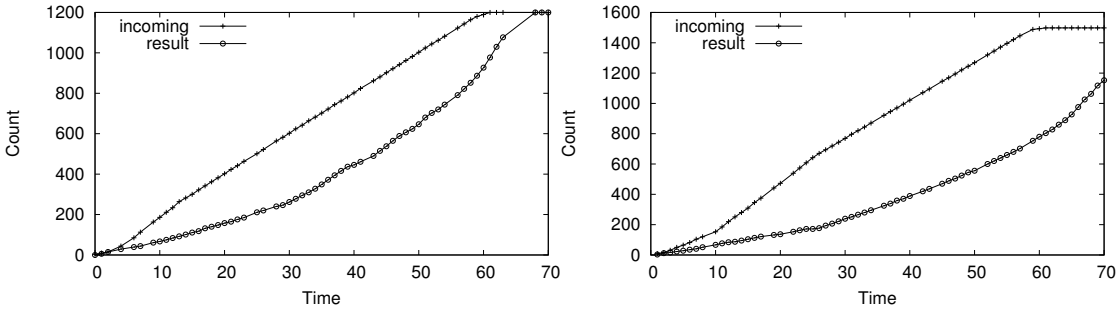
Stateless Filtering Query

The exemplary query filters events of the presented input type and reports those records that are from the specific industry IT and have a positive change of at least 5.0%. All other records are considered as unimportant and are not used in the result. The stateless query for filtering is shown in Listing 7.6.

```
1  for tumbling window $w in $input
2    start at $spos when true()
3    end at $epos when true()
4    return <stockresult>
5           <id>{$w/data/id}</id>
6           {for $tmp in $w/data/record
7             return if ($tmp/industry="IT" and $tmp/change >= 5.0)
8               then $tmp
9               else ()
10          }
11   </stockresult>
```

Listing 7.6: Filtering Query (Stateless).

This stateless query processes each input event once and returns an event containing the filtered stock update.



(a) 20 Events / Second

(b) 25 Events / Second.

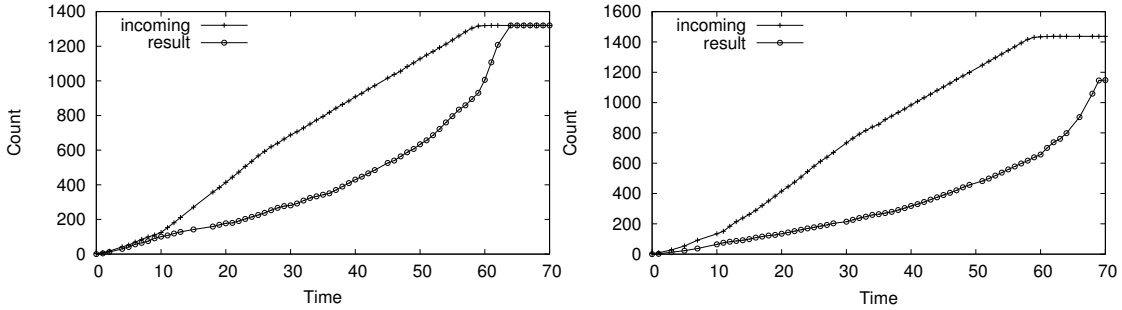
Figure 7.3: Stateless Filtering Query with One Data Service Node.

By executing the query without a burst handling strategy and different amounts of input events, the burst capable aggregator node is tested for an overload. Figure 7.3 shows that the

burst capable aggregator can barely cope with an input frequency of 20 events per second, and therefore not with 25 events per second.

With 20 events per second the node manages to process all events within the 10 seconds after the high load. With 25 events per second, this is not possible anymore. The node starts to recover, but after 70 seconds still more than 300 results are missing. To explore the border between dependable event processing and an

overload situation more closely, event rates in between have been simulated. In Figure 7.4 the result for $22.\overline{33}$ events per second can be seen on the left side, on the right side the result for 24 events per second is shown. Around 22 events per second can still be handled quite well. 24 events per second also lead to an overload, but at the end less results are missing in comparison to an input of 25 events per second.



(a) $22.\overline{33}$ Events / Second

(b) 24 Events / Second.

Figure 7.4: Stateless Filtering Query with Several Data Service Nodes.

The CPU load tends to stick to 100% during the event processing. If no further events arrive and no overload is there, it immediately decreases to a minimum. If there is an overload, the node still processes events and the CPU load stays at a high rate. The system has a base memory utilization of around 40% in the beginning. The memory utilization increases steadily during the scenario execution. In the most demanding scenarios it settles down at around 75%.

Based on these facts, an input rate of 25 events per second is used for the strategies. Two data service nodes produce events for one minute to reach a combined frequency of 60 events per second. This results in 1500 events during the producing time.

Load Shedding The results of the evaluation scenario with probabilistic load shedders using a different sampling rate can be seen in Figure 7.5. The *p-values* in the caption indicate the used sampling rate. *None* always refers to the evaluation execution without an applied burst handling strategy.

By simple calculation one could assume, that as the burst manager is capable of 1200 events, the sampling rate should be around 20%. Using a sample rate of 10% the aggregator manages as many results as without the strategy. But on average the aggregator has still more than 200 events left, which have to be processed.

Using a sample rate of 20%, the aggregator manages a higher throughput. On average 100 more events than without having the strategy can be processed and no further events wait for

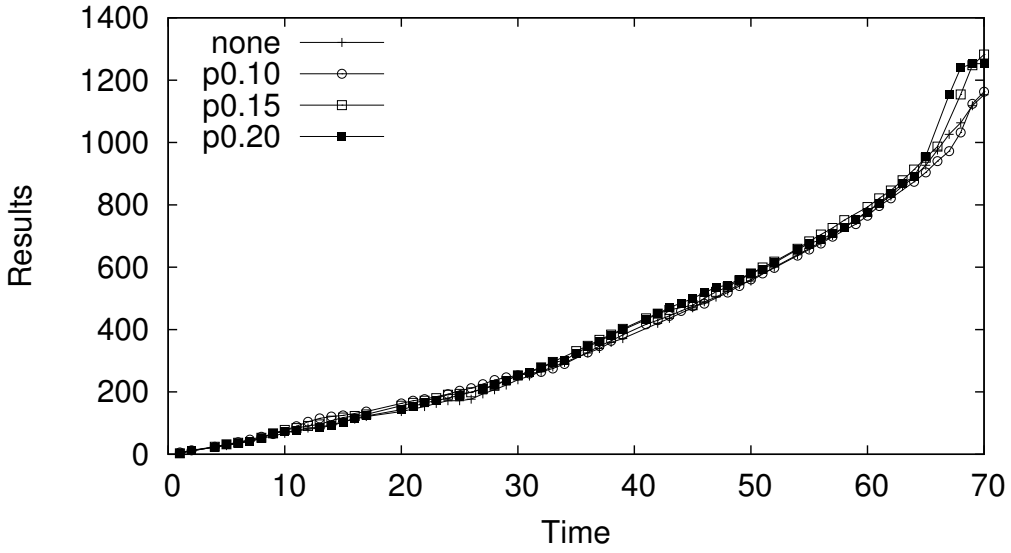


Figure 7.5: Probabilistic Load Shedding for a Stateless Filter Query (25 Events / Second).

processing. In the depicted result for a sampling rate of 20%, one can see that at the end the amount of results does not really increase anymore. So the aggregator does not need the 10 additional seconds to recover.

By only shedding 15% of the incoming events, slightly more results can be achieved, but 25 events still wait to be processed after the overload. So if the overload does not take too long, a sampling rate of 15% would be more appropriate, as the aggregator works in a stable mode and less events are shed.

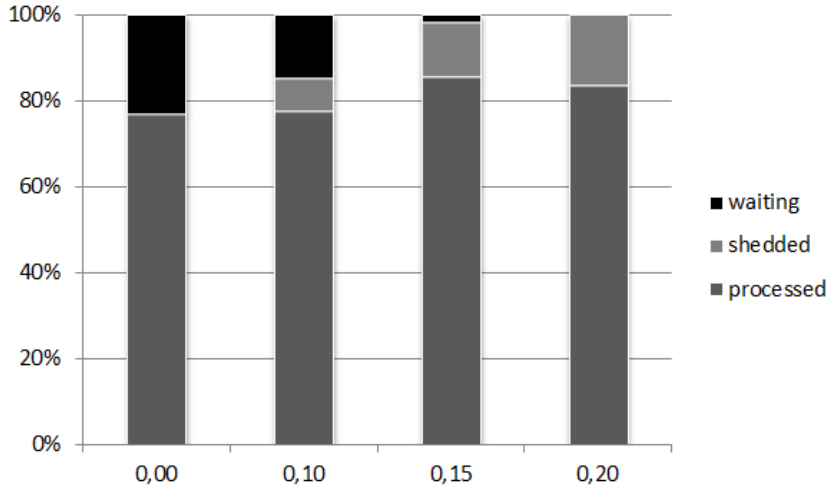


Figure 7.6: Benefits of Different Sampling Rates.

If more than 20% are used as sampling rate, the amount of processed results decreases. This leads to no additional benefits. The resources of the burst capable aggregate should not be utilized at a maximum anymore. Figure 7.6 shows the relative results of the different sampling rates. Important for load shedding is not only the maximum amount of processed events, but also that no more waiting events are present.

Deferred Execution The results of the evaluation scenario with the deferred execution strategy can be seen in Figure 7.7. In the caption, the *b-values* represent the used buffer size, the *t-values* define the timeout length.

With regard to the incoming rate of the events, suitable sizes for the buffer and the timeout, which is the pause between retrieving and processing deferred events, can be calculated. The results show that the choice of these parameters has a great impact on the results.

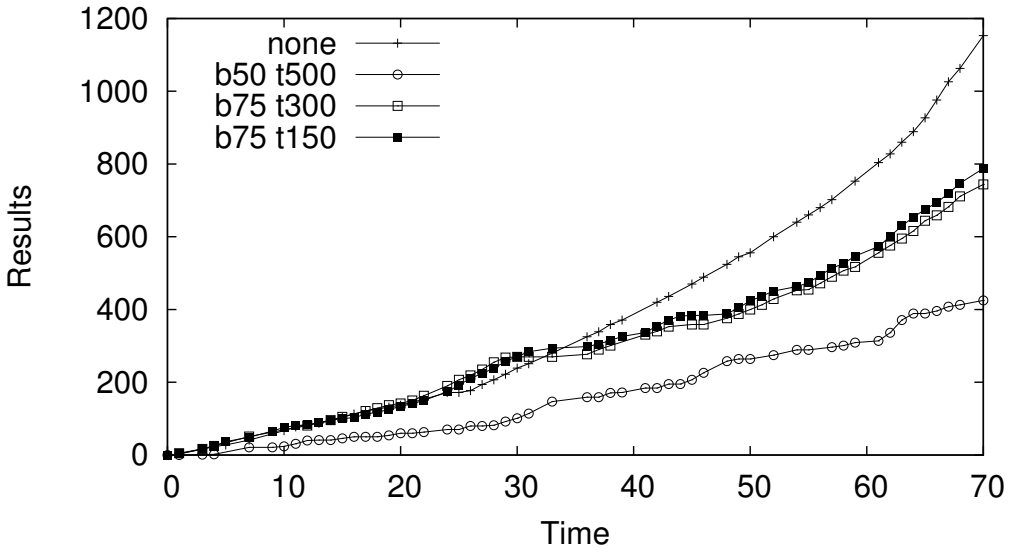


Figure 7.7: Deferred Execution for a Stateless Filter Query (25 Events / Second).

As in comparison to load shedding, all events have to be processed and an additional overhead of persisting and retrieving the events is introduced, this strategy cannot produce as many output events, as load shedding or event processing without burst handling can. As a trade-off it can keep the aggregator in a stable mode and no results are lost.

The results show, that different buffer sizes and timeouts can fit better or worse to the current situation. Moreover one can see, that events are not necessarily processed all the time, but rather after each timeout.

Nevertheless, the additional effort leaves its marks in the result counter. It is at least possible to process more than the half of the incoming events in time. In the first 30 seconds, the node even performs better with the deferred strategy than without it, but then the output events get more and more delayed. With deferred execution, the CPU load is similar or slightly decreases, but the memory utilization increases to 80%.

By using this strategy, results are delayed, but therefore no results are lost. The delay could be minimized, by persisting only some events while others are processed immediately. But this approach was not taken into account as it leads to an incorrect order of the results. Further improvement might be gained by a different persistence approach, i.e. by a more performant database or by a less strict consistency model, in example *weak consistency* [53].

Forwarding The results of the evaluation scenario with the forwarding strategy can be seen in Figure 7.8. Again, the agenda of the figure describes the configuration values for the strategy. The *n-values* name the amount of nodes used for the forwarding, the *p-values* indicate how many of the incoming events have already been processed by the original node (percentage). The rest of the events has been forwarded to the other nodes.

Based on the capability to process 1200 nodes in a minute, one could assume that forwarding 20% of the events and processing 80% locally should be sufficient to handle the overload. Unfortunately, this is not the case, as the forwarding introduces additional effort. The results of the forwarded events increase the incoming rate of events. Those results have to be identified and handled separately.

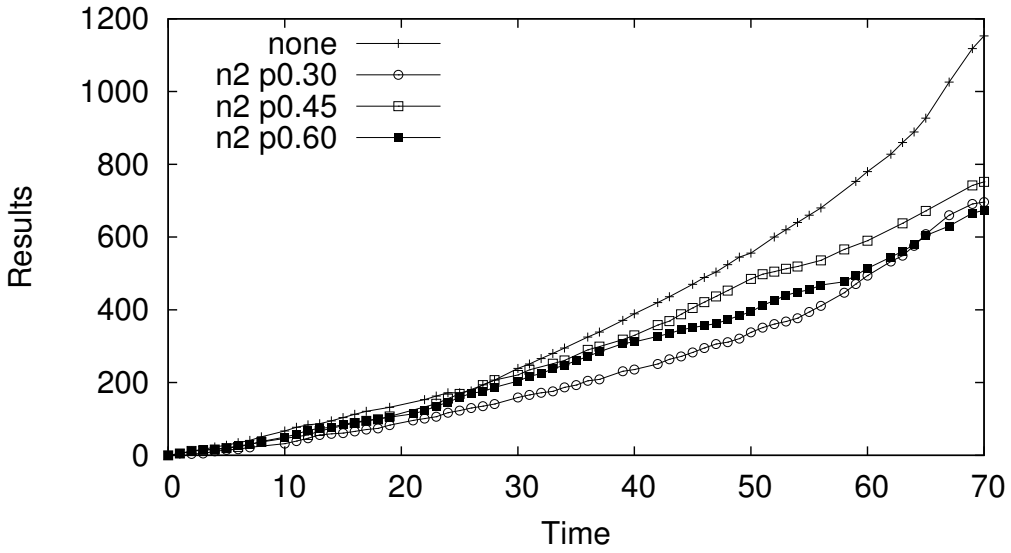


Figure 7.8: Forwarding for a Stateless Filter Query (25 Events / Second).

The results show, that about half of the events can be processed, when 55% are forwarded to other nodes. In the case of more or less events being forwarded, the amount of results starts to decrease. If fewer events are forwarded, the node gets overloaded again due to incoming events. If more events are forwarded, more effort on sending and receiving the events is needed, which also leads to a decrease in performance. As the remaining events are not dropped and still have to be executed, this strategy is not able to cope with the overload.

Based on the results it can be stated, that the forwarding strategy can only be applied successfully, if the processing effort of an event is significantly higher than the forwarding effort.

Compared to simulations with simpler events and event calculation, the strategy already performs better. But the complexity of the processing in this scenario is still not sufficient to achieve a real good performance.

Comparison The comparison of the strategies for the filtering query can be seen in Figure 7.9. Load shedding can produce as many results as the overloaded node, but in contrast to the overload, no more events are required to be processed. So possible results are lost, but the node is not overloaded anymore and it is up to date. The deferred execution leads to less output, but all missing events are still persisted and going to be processed. So no events have been lost, but the node is not up to date anymore. Nearly half of the events still have to be processed. The forwarding strategy manages even less result events. The overhead of the additional communication is too high in comparison to the event processing time, so it cannot perform optimally.

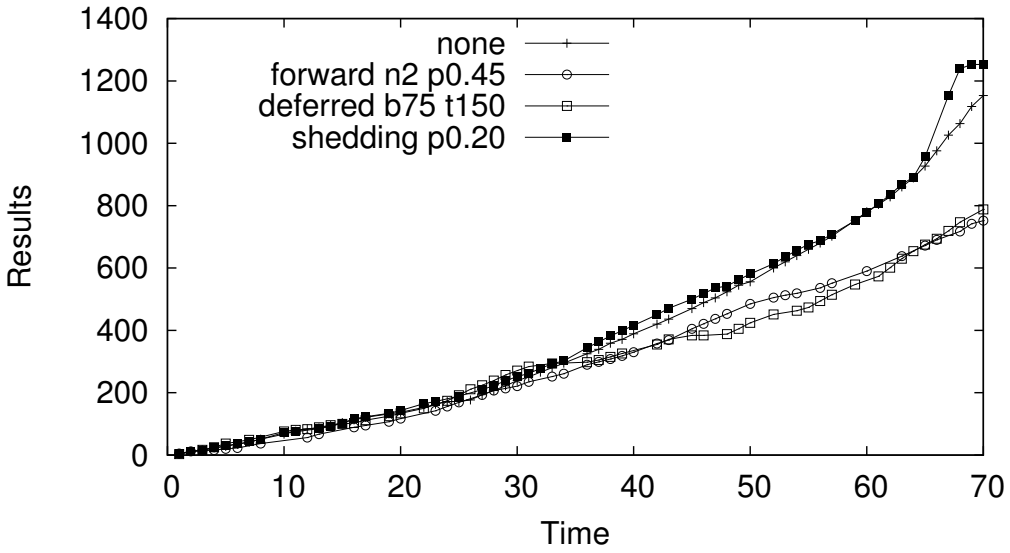


Figure 7.9: Comparison of the Strategies for a Stateless Filter Query (25 Events / Second).

Figure 7.10 shows the efficiency of the strategies represented in relative values.

Only the load shedding approach manages as many output events as processing without burst handling. The second displayed category *other* represents different event states. In the cases of no strategy and forward strategy, it represents the overload. For the deferred strategies it represents the deferred events. They are not an overload now, but they still have to be processed. For the load shedding strategy events in the category *other* represent shed events.

Stateless Transformation Query

In this scenario two different transformations for the stock updates are implemented by the query. First the currency will be transformed (converted) to USD ($) instead of EUR (€)[2]. Further

[2] Based on the exchange ratio on 2013-08-12.

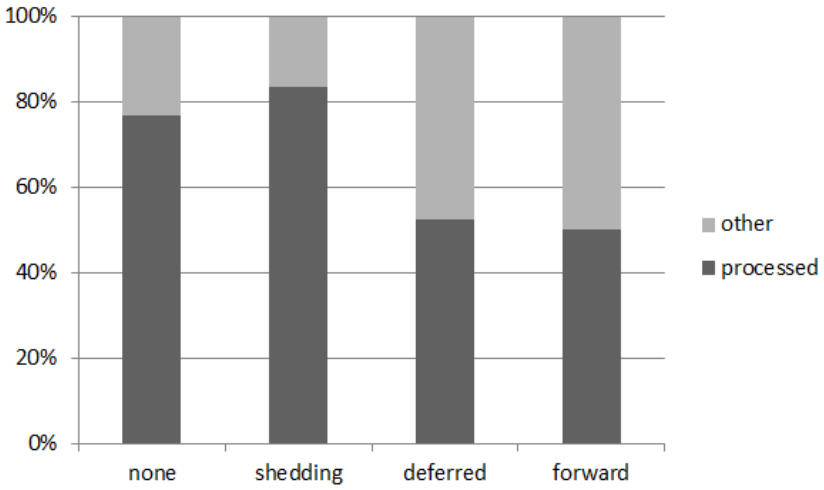


Figure 7.10: Relative Comparison of the Strategies for a Stateless Filter Query (25 Events / Second).

on, the structure of the element will be changed. The XML schema of the output events is shown in Listing 7.7. Instead of the elements *name* and *change*, the output contains an element *companyState*. This element is a concatenation of the replaced elements where the *change* is put between parenthesis. The change of structure is realized by introducing a new element and using the XQuery string function *concat*, which concatenates a sequence of strings. The stateless transformation query is shown in Listing 7.8.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="stockupdate">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="record" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="industry" type="xs:string"/>
              <xs:element name="companyState" type="xs:string"/>
              <xs:element name="price" type="xs:decimal"/>
              <xs:element name="currency" type="xs:string"/>
              <xs:element name="time" type="xs:dateTime"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 7.7: XML Schema for Output Events of the Stateless Transformation Query.

```
1  for tumbling window $w in $input
2  start at $spos when true()
3  end at $epos when true()
4  return <stockresult>
5    <id>{$w/data/id}</id>{
6    for $tmp in $w/data/record
7    return
8      <record>
9        {$tmp/industry}
10       <companyState>{fn:concat($tmp/name, ' (', $tmp/change, ')')}</companyState>
11       {if ($tmp/currency = "EUR")
12         then <price>{$tmp/price * 1.33100}</price>
13         else(<price>{$tmp/price/text()}</price>) }
14       {if ($tmp/currency = "EUR")
15         then <currency>USD</currency>
16         else(<currency>{$tmp/currency/text()}</currency>) }
17       {$tmp/time}
18     </record>
19 }</stockresult>
```

Listing 7.8: Transformation Query (Stateless).

For the transformation of the currency, the value is multiplied with the exchange ratio and the old value is replaced with the new one.

The capability of the aggregator node is the same as for the transformation query. But an overload has a greater impact on the transformation query than on the filtering query. So even though the node manages 1200 events in the observation period, it can manage less events than the filtering query, if 1500 events have to be processed.

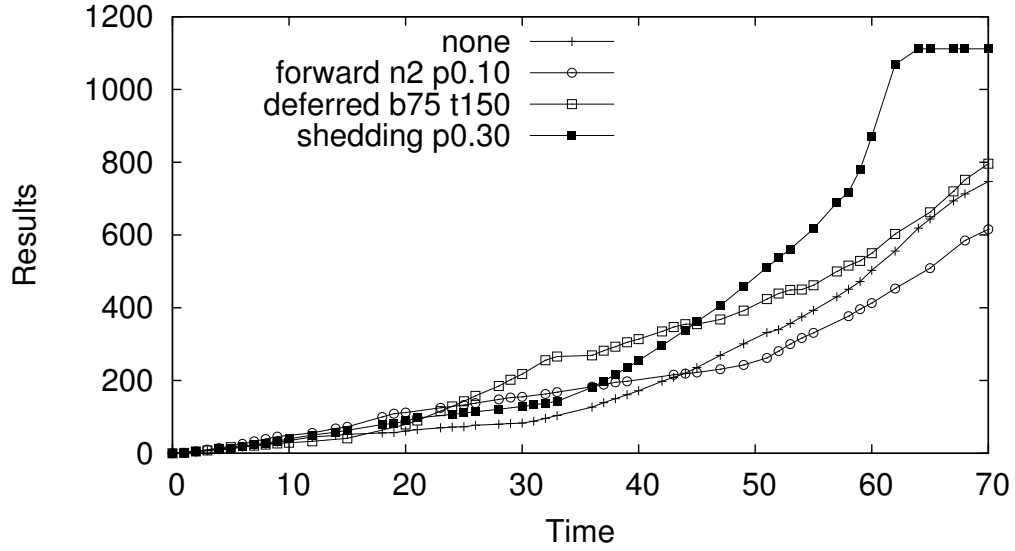


Figure 7.11: Comparison of the Strategies for a Stateless Transformation Query (25 Events / Second).

The results of the strategies can be seen in Figure 7.11. Compared to the burst situation, the strategies perform better, especially the load shedder.

Figure 7.12 shows the efficiency of the strategies in relative values. In this case the shedding strategy as well as the deferred execution outperform the overload situation. The forward strategy cannot be applied successfully again. If the actual event processing was more complex and required more effort in comparison to the sending and receiving of the events, the forward strategy would be suited better.

In comparison to the filtering query, the transformation query usually has bigger result events which also have to be sent, as information cannot be dropped. As the structural change of the query does not make a big difference, the output events are more or less of the same size as the input events except for the payload.

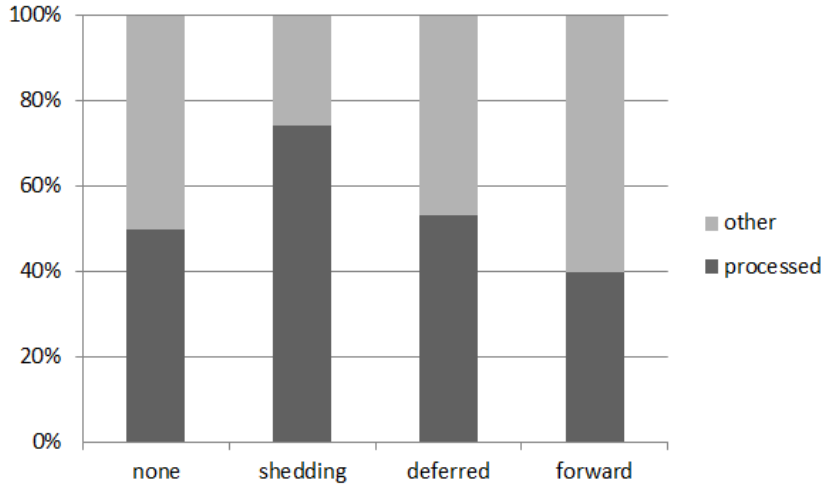


Figure 7.12: Relative Comparison of the Strategies for a Stateless Transformation Query (25 Events / Second).

Stateless Aggregation Query

In case of the stock updates, the output for this scenario should contain aggregated values for the current price and the relative change per industry. It is not absolutely required that the input events of an industry use the same currency. If input events contain different currencies per industry, a record for each currency is created. Further on, the output event does no longer contain the element *time*, nor does it contain the element *name*, as the values do not belong to a certain industry. The XML output schema is the same as for the input events, excluding the elements *name* and *time*, see Listing 7.9.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3  elementFormDefault="qualified">
4    <xs:element name="stockupdate">
5      <xs:complexType>
6        <xs:sequence>
7          <xs:element name="id" type="xs:string"/>
8          <xs:element name="record" maxOccurs="unbounded">
```

```
 9             <xs:complexType>
10               <xs:sequence>
11                 <xs:element name="industry" type="xs:string"/>
12                 <xs:element name="change" type="xs:string"/>
13                 <xs:element name="price" type="xs:decimal"/>
14                 <xs:element name="currency" type="xs:string"/>
15               </xs:sequence>
16             </xs:complexType>
17           </xs:element>
18         </xs:sequence>
19       </xs:complexType>
20     </xs:element>
21   </xs:schema>
```

Listing 7.9: XML Schema for Output Events of the Stateless Aggregation Query.

As one can see, omitting these elements is already some form of transformation as well, so this query can be classified as some kind of transformation query too. To ensure reasonableness of the query, this transformation is accepted for the exemplary aggregation query.

The stateless aggregation query is shown in Listing 7.10.

```
1  for tumbling window $w in $input
2  start at $spos when true()
3  end at $epos when true()
4  return <stockresult>
5    <id>{$w/data/id}</id>{
6    for $tmp in $w/data/record
7    let $industry := $tmp/industry
8    let $currency := $tmp/currency
9    group by $industry, $currency
10   order by $industry, $currency
11   return
12     <record>
13       {$industry}
14       <change>{avg($tmp/change)}</change>
15       <price>{avg($tmp/price)}</price>
16       {$currency}
17     </record>
18 }</stockresult>
```

Listing 7.10: Aggregation Query (Stateless).

In contrast to the strategies so far, the size of the output event is definitely decreased as only a summary of the input is returned without the details and all the payload. Therefore, the performance of the aggregator is better and it manages up to 1500 events. An overload occurs when 1800 events are put in, therefore the evaluation was performed at an input rate of 30 events per second. The result of the strategies can be seen in Figure 7.13.

Again, only the load shedder can really cope with the overload. The forwarding strategy performs better now that the result events, which have to be returned to the original aggregator as well as to the receiver, are much smaller. The deferred execution can still store all the input, but the overhead cannot be compensated.

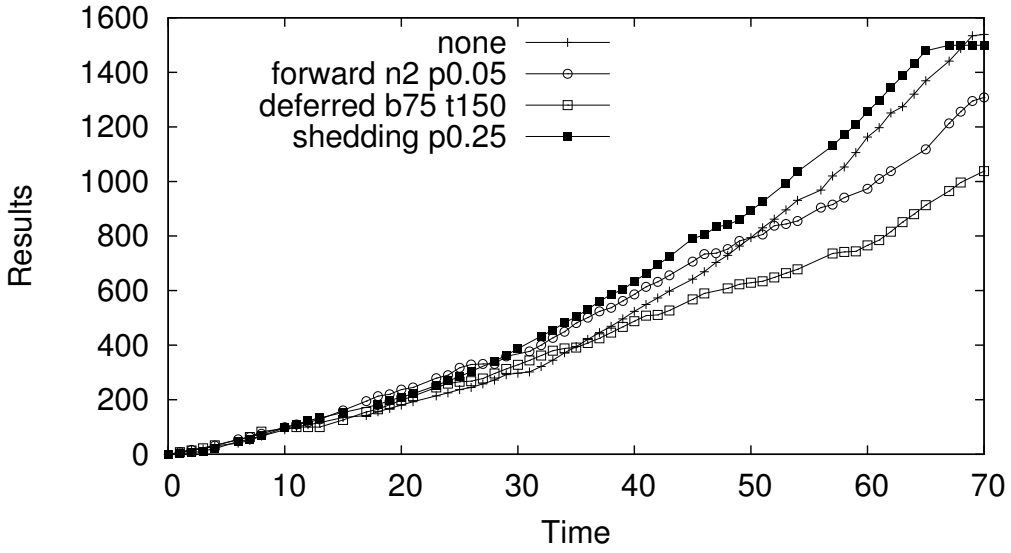


Figure 7.13: Comparison of the Strategies for a Stateless Aggregation Query (30 Events / Second).

Figure 7.14 shows the efficiency of the strategies in relative values. It does not reveal any special insights.

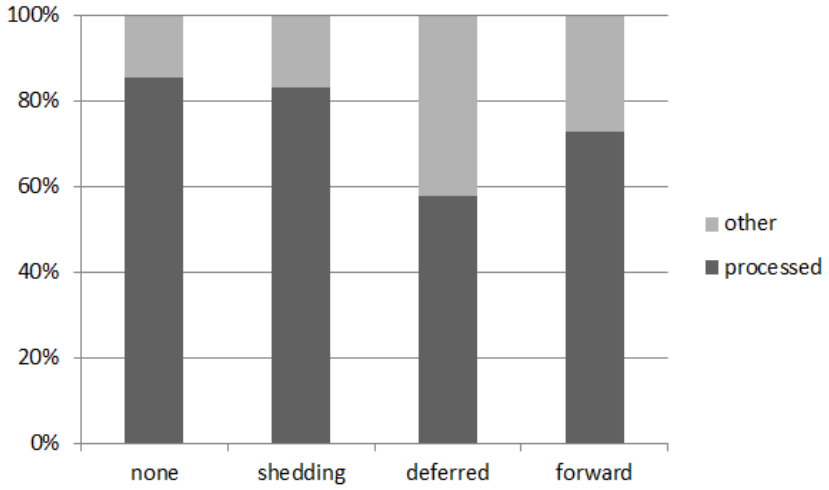


Figure 7.14: Relative Comparison of the Strategies for a Stateless Aggregation Query (30 Events / Second).

Combined Query

As a more complex scenario, the calculation of the covariance of two variables and respectively the correlation coefficient is chosen. It combines filtering, aggregation and transformation operations in one query. Out of all incoming events the covariance of the stock changes of two specified companies ($companyA$ and $companyB$) has to be calculated. Only the values of those companies are used for the calculation, therefore the rest of the reports has to be filtered. As several records are used to calculate the variance and the covariance, an aggregation is performed. Finally, as the output of the query holds information on the variance and the covariance, the format of the output is completely different from the input format, so a transformation is done as well.

Moreover, this calculation is representative for queries in real-word applications, as these key figures are often used to monitor and analyze share prices and their changes.

Regardless of the scope, the resulting events have to conform to the XML Schema in Listing 7.11. The output consists of the variances of the stock changes for both companies as well as the covariance and correlation coefficient of the stock changes.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3  elementFormDefault="qualified">
4    <xs:element name="result">
5      <xs:complexType>
6        <xs:sequence>
7          <xs:element name="variance">
8            <xs:complexType>
9              <xs:sequence>
10               <xs:element name="companyA" type="xs:decimal"/>
11               <xs:element name="companyB" type="xs:decimal"/>
12             </xs:sequence>
13           </xs:complexType>
14         </xs:element>
15         <xs:element name="covariance" type="xs:decimal"/>
16         <xs:element name="correlation" type="xs:decimal"/>
17       </xs:sequence>
18     </xs:complexType>
19   </xs:element>
20 </xs:schema>
```

Listing 7.11: XML Schema for Output Events of the Combined Query.

In Listing 7.12 the stateless combined query is introduced. The calculation of the variance, covariance and correlation coefficient follows the definitions in [8], see Equations 7.1, 7.2 and 7.3.

$$var(x) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{7.1}$$

$$cov(x,y) = \frac{1}{n}\sum_{i=1}^{n}(x_i \cdot y_i) - \bar{x} \cdot \bar{y} \tag{7.2}$$

$$r(x,y) = \frac{cov(x,y)}{\sqrt{var(x)} \cdot \sqrt{var(y)}} \tag{7.3}$$

For calculating the correlation coefficient r, the standard deviation, which is the square root of the variance, is needed. As XQuery does not provide a method to calculate the square root [26] two helper methods are declared. These helper functions calculate an approximation of the square root using the Newton-Raphson method [132].

```
declare function local:step(
  $n as xs:integer, $xn as xs:double, $x as xs:double
) as xs:double{
  let $next := $xn - (($xn * $xn - $x) div (2 * $xn))
  return if ($n <= 0) then $next else local:step (($n - 1), $next, $x)
};

declare function local:sqrt(
  $x as xs:double
) as xs:double {
  local:step(20, $x * 0.5, $x)
};
(for tumbling window $w in $input
start at $spos when true()
end at $epos when true()
  let $avgA := avg($w/data/record[name="CompanyA"]/change)
  let $avgB := avg($w/data/record[name="CompanyB"]/change)
  let $countA := count($w/data/record[name="CompanyA"]/change)
  let $countB := count($w/data/record[name="CompanyB"]/change)
  let $count := if ($countA > $countB) then $countA else $countB
  let $changeA := for $tmpA in $w/data/record[name="CompanyA"]/change
    return $tmpA
  let $changeB := for $tmpB in $w/data/record[name="CompanyB"]/change
        return $tmpB
  let $sumP := sum(for $i in (1 to $count)
        return $changeA[position()=$i] * $changeB[position()=$i])
  let $sumA := sum(for $tmp in $changeA
        return (($tmp - $avgA)*($tmp - $avgA)))
  let $sumB := sum(for $tmp in $changeB
        return (($tmp - $avgB)*($tmp - $avgB)))
  let $varA := $sumA div $count
  let $varB := $sumB div $count
  let $covar := ($sumP div $count) - ($avgA*$avgB)
  return <result>
    <id>{$w/data/id}</id>
        <variance>
      <companyA>{($varA)}</companyA>
      <companyB>{($varB)}</companyB>
    </variance>
    <covariance>{($covar)}</covariance>
    <correlation>
      {$covar div (local:sqrt($varA) * local:sqrt($varB))}
    </correlation>
  </result>
)
```

Listing 7.12: Combined Query (Stateless).

As only the relative changes of the stocks are used for the calculation, the values do not depend on the currency. Hence, no transformation is needed. The query does not require an equal amount of records for both companies. If there are more records for one company than for the other, just the first pairs are used for the calculation. All remaining records do not affect the calculated values.

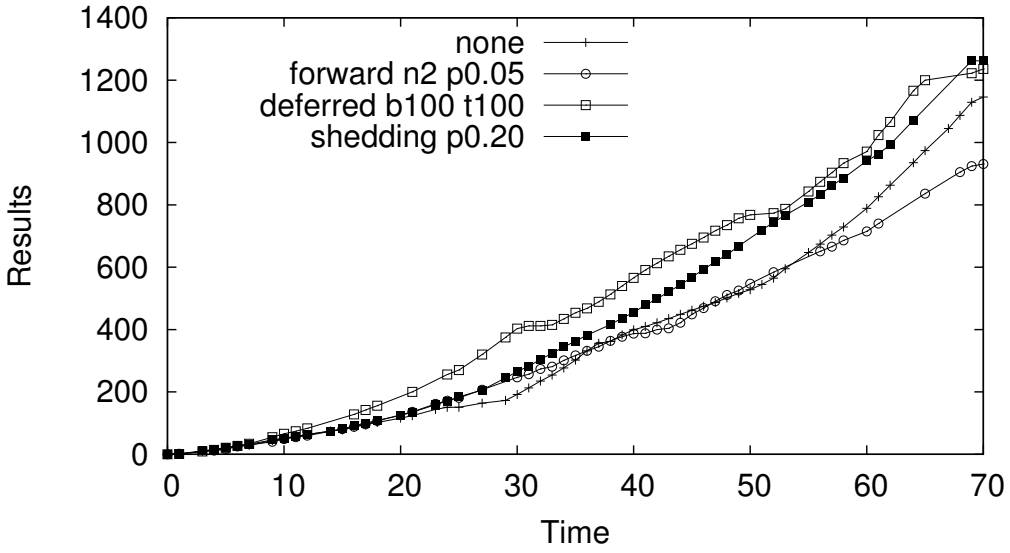


Figure 7.15: Comparison of the Strategies for a Stateless Combination Query (25 Events / Second).

For this query, again 1500 events are enough to stress the aggregator node. Even though the output events are of the same size as for the aggregation query, the processing is more demanding. The result of the strategies can be seen in Figure 7.15. The load shedder and the deferred execution outperform the overloaded node. The forwarding strategy still lags behind. Figure 7.16 shows the efficiency of the strategies in relative values. The load shedder and the deferred execution produce more results and improve the overload situation.

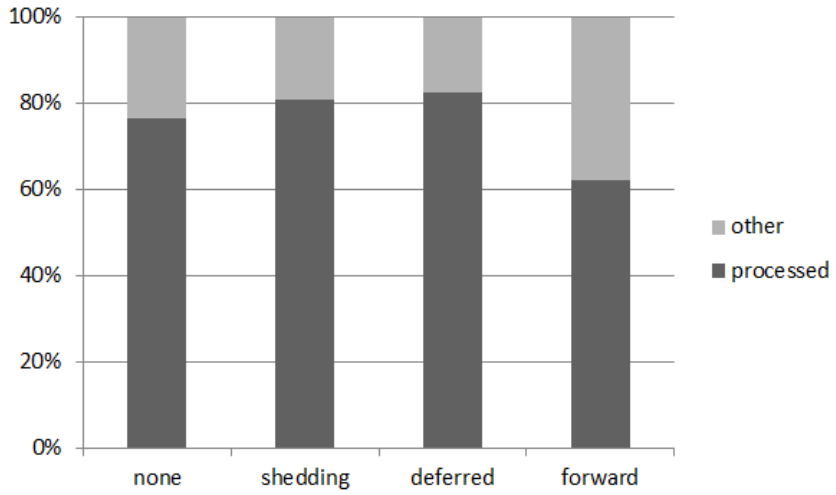


Figure 7.16: Relative Comparison of the Strategies for a Stateless Combination Query (25 Events / Second).

7.3.3 Summary for Stateless Queries

The applied evaluation scenario supports the analysis of Section 7.3.1, but provides deeper insights. It shows, that additional features have a great influence on the applicability of the strategies. And how the features are differently weighed.

Additional features which have to be minded are the ratio of effort caused by sending / receiving events to the effort of the real event processing. It has a huge impact on whether a strategy can cope with the overload or not. One point that distinguishes the queries in particular is the format of the output event. If the output is smaller than the input, the query seems to be more performant.

The forwarding strategy provides indeed an incorrect result order, but this is the smallest issue. For stateless queries, which are rather simple, the overhead of forwarding the input and receiving the results is too high to improve the overload situation. The resulting delay is too high and the node is still overloaded. The implementation of the deferred execution enables a correct result order, but the persisting overhead is also rather high in comparison to the event processing, so the intentional delay gets quite big. Especially if the high load continues for a longer time, the delay is going to increase too much. The only strategy that seems to be applicable for stateless queries in general is the load shedder. If the correct sampling rate is chosen, at least as many events as the overloaded node can handle are handled correctly. The others are shed and lead to missing results.

If missing results are tolerable, the load shedder is recommendable. Otherwise the deferred execution should be chosen, but a high delay has to be expected.

To assess the strategies based on their memory utilization, a closer investigation of the memory would be needed. The values of the *top* command show the rises in the memory consumptions, but as the Java Virtual Machine (JVM) does not immediately free the memory that is not used anymore, the benefits of the forwarding and the deferred execution cannot be seen in the traced evaluation outcomes.

Another outcome of the evaluation scenario is that, based on all these factors, it is hard to choose the correct parameters for the burst handling strategies. To automate this part of the burst handling, a sophisticated monitoring of the system and good algorithms to determine the best parameters have to be provided.

7.4 Evaluation of Stateful Queries

In this section application of the strategies *load shedding*, *deferring* and *forwarding* on stateful queries is evaluated.

7.4.1 Analysis

Based on the theoretical background, provided on the strategies, the following assumptions can be stated:

Table 7.3: Effects of Strategies on Stateful Queries

	Stateful Query
Forwarding	• incomplete results • incorrect result order likely • incorrect result values because of different window combinations • possible delay • forwarding overhead
Deferring	• complete results • possible incorrect result order (depends on implementation) • possible incorrect result values because of different window combinations (depends on implementation) • intentional delay • persisting overhead
Shedding	• incomplete results • incorrect result values because of different window combinations

As for stateless queries, additional nodes are provided to share the processing load when using the forward strategy. For stateful queries, the forwarding strategy implicates additional problems for the quality of the results. Additional to the incorrect result order, the results deliver incorrect values and are incomplete as the events are distributed and the processed windows are not filled with the same events. This could be prevented by using a shared memory for the events or by always forwarding all events of a window to the responsible node. Especially for sliding windows this would imply that an event has to be forwarded to multiple nodes, as it is evaluated in different windows. This would lead to an enormous overhead. In the present solution, incorrect results are accepted due to different window combinations.

Concerning deferred execution and the present implementation there are no differences for stateless and stateful queries. If order preserving is not supported by the strategy, incorrect result values would be implied, as different events are evaluated in a window.

The results of the load shedding strategy are again incomplete, but in addition the strategy provides incorrect result values, as a dropped input event affects several evaluated windows.

7.4.2 Scenario Execution

As for the stateless queries, the three strategies are applied on the different types of queries and finally on the combined query. Again, first the amount of events the node can take is determined

and then the strategies are used in a situation of overload. For stateful queries the overload is not only caused by the amount of input events and their size, but also by the window size of the query.

Stateful Filtering Query

The stateful filtering query used for the evaluation filters the fifteen best stock changes. As for the stateless query, the input and output formats of the events are the same. The filtering query is shown in Listing 7.13.

```
1  for sliding window $w in $input
2  start $s at $spos when  true()
3  end $e at $epos when ($epos - $spos) >= %windowSize
4    let $ordered :=
5      for $k in distinct-values($w/data/record/change/xs:double(.))
6        order by $k descending
7        return $k
8          return
9          <stockresult>
10           {for $tmp in $w/data/record[index-of($ordered, xs:double(change)) le 15]
11            order by $tmp/change/xs:double(.)
12            return $tmp}
13         </stockresult>
```

Listing 7.13: Filtering Query (Stateful).

Depending on size of the incoming events, the output event of the query can be a lot smaller than an input event, as less records are contained in the output. As the fifteen best distinct values are used to determine the best stock records, at least fifteen records should be contained in the output. If records have the same change value, more records may be outputted.

While 900 events can be managed by the node, an overload occurs for this query when 1200 events per minute are received. The evaluation scenario therefore has been performed with 1200 input events and a window size of 50. The results of the different strategies are shown in Figure 7.17.

As for the stateless queries, load shedding can provide as many events as the node is able to handle, but manages the rest of the overhead by dropping the events. The other two strategies cannot provide as many events in the specified evaluation time, but distribute or defer the load, so the event processing takes more time.

The quality of the results is heavily influenced by the applied strategy. In case of load shedding, the quality depends on the events that are shed. If low values are shed, the result is not influenced at all, but if one of the highest values is shed, the result entry is missing in the result of every window the event would have belonged to. So for stateful queries, a semantic load shedder would guarantee much better results than a normal probabilistic load shedder.

The deferred execution only manages to provide about a quarter of the result events in time, but it does not influence the quality of the results. It takes more time, but all results are correct.

The forward strategy manages more result events, but again the quality suffers because of the strategy. As the window size is 50, only 1050 instead of 1150 results can be calculated, as each of the nodes has a separate buffer for the window. As this leads to different input event combinations in the windows, the results are likely to be erroneous. But as no event is

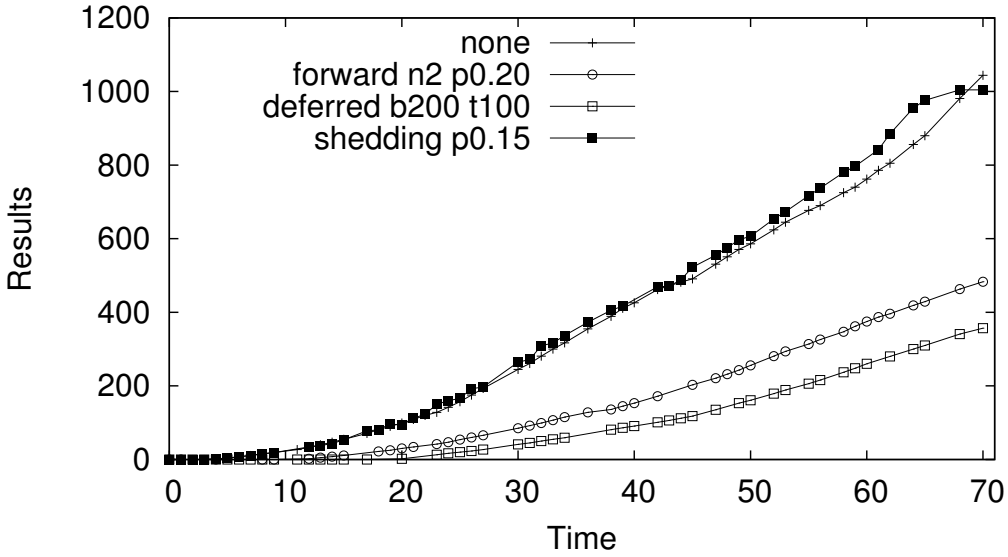


Figure 7.17: Comparison of the Strategies for a Stateful Filtering Query (20 Events / Second).

dropped, each one of the fifteen highest changes of the correct results, will also be included in the erroneous output events. It may occur scarcer, though, and records not occurring in the correct results are also reported (as so-called unrepresented events).

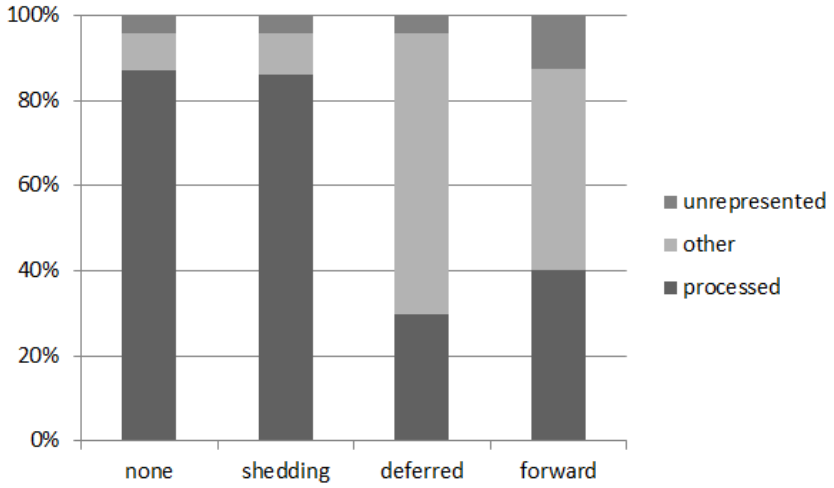


Figure 7.18: Relative Comparison of the Strategies for a Stateful Filtering Query (20 Events / Second).

Figure 7.18 shows the relative comparison between the strategies. A common characteristic of the results for the stateful evaluation is that the so called *unrepresented events* are always the same except for the forwarding strategies. In general, those events represent the input events,

which do not result in an output event as the window is not yet filled. As in case of the forwarding strategy multiple windows are filled in parallel, more output events get lost.

Stateful Transformation Query

The scenario for the stateful transformation is the calculation of the normalized stock change per window. For that aim, the best stock change of the window is used as reference and all records are normalized. The positive changes will therefore be in the range of zero to one. The negative changes may also be smaller than -1. The output format is quite similar to the input format, as only the original change value is replaced by the normalized change value. Additionally, the payload attribute is omitted in the output. The transformation query is shown in Listing 7.14.

```
1  for sliding window $w in $input
2  start $s at $spos when  true()
3  end $e at $epos when ($epos - $spos) >= %windowSize
4  let $maxChange := max($w/data/record/change)
5  return <stockresult>
6  {
7    for $tmp in $w/data/record
8    return
9      <record>
10       {$tmp/industry}
11       {$tmp/name}
12       <change>{$tmp/change div $maxChange}</change>
13       {$tmp/currency}
14       {$tmp/time}
15     </record>
16 }</stockresult>
```

Listing 7.14: Transformation Query (Stateful).

An input of 900 events per minute combined with a window size of 50 is already sufficient to cause an overload for this query. The results for the evaluation with these parameters can be seen in Figure 7.19.

The quantitative results are quite similar to the filtering query, but especially the deferred execution performs better. The relative values for the results can be seen in Figure 7.20.

The quality of the results remains the same for the deferred execution, but strongly depends on the range of the input data for the load shedding and the forwarding strategy. All records of a result depend on highest stock change in the current window. For load shedding, if this element is shed, all records of the result are incorrect. If the range of the input data is quite homogeneous, the error might not be that significant. If the range of the input data is huge and inhomogeneous, the error can also be huge. The same applies to forwarding. If the highest value is forwarded to a different node, it is not present on the other nodes as a reference for the normalization, so it is only used in some windows, but not in all of the correct ones.

Not all transformation queries are necessarily dependent on one special input event, so results can be qualitatively better for different queries.

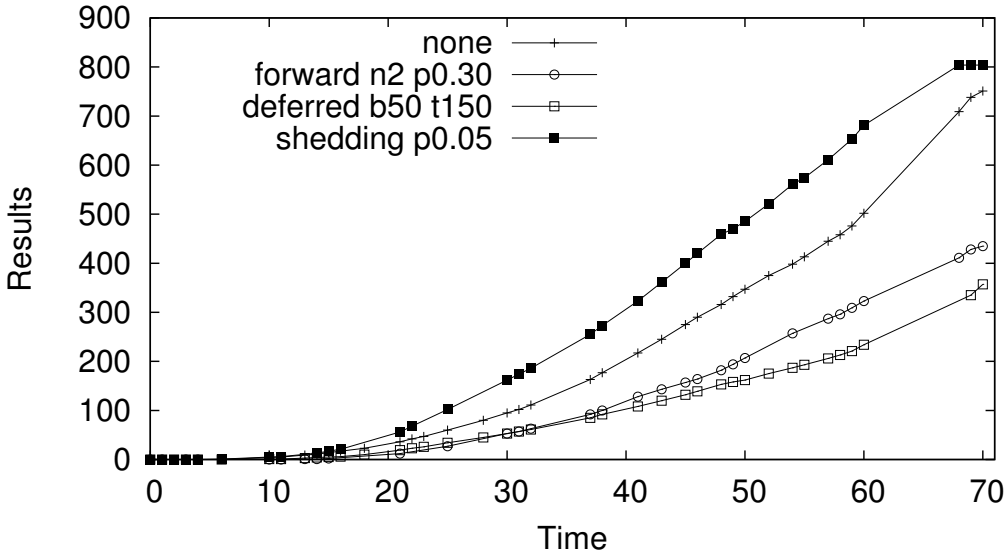


Figure 7.19: Comparison of the Strategies for a Stateful Transformation Query (15 Events / Second).

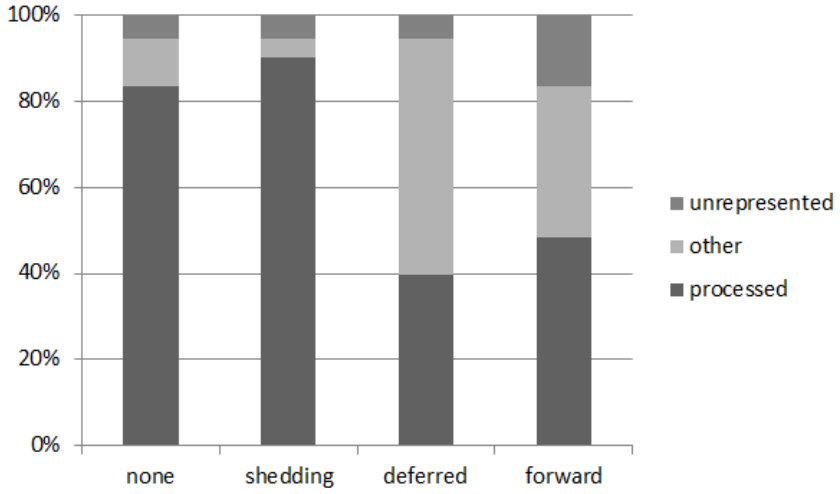


Figure 7.20: Relative Comparison of the Strategies for a Stateful Transformation Query (15 Events / Second).

Stateful Aggregation Query

The stateful aggregation query is the same as the stateless query except for the window definition. The stateless window definition is replaced by the sliding window as it was used for the stateful queries before. See Section 7.3.2 for the query and the output schema.

With the aggregation query defining a window size of 100, the node still manages 1500 input events per minute. An overload occurs when the window size is set to 200, hence this value is used for the evaluation.

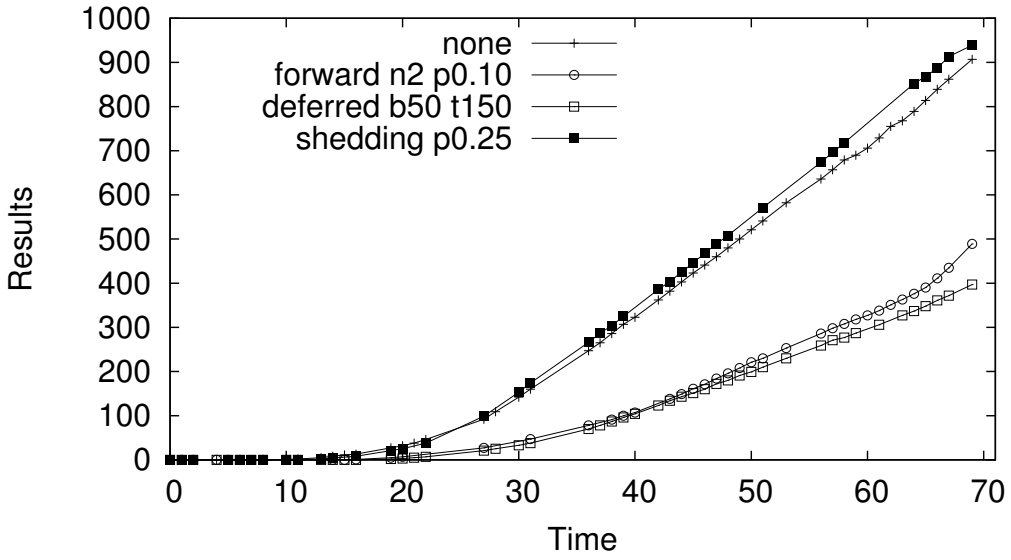


Figure 7.21: Comparison of the Strategies for a Stateful Aggregation Query (25 Events / Second).

The results of the stateful aggregation query can be seen in Figure 7.21. The corresponding relative results are shown in Figure 7.22.

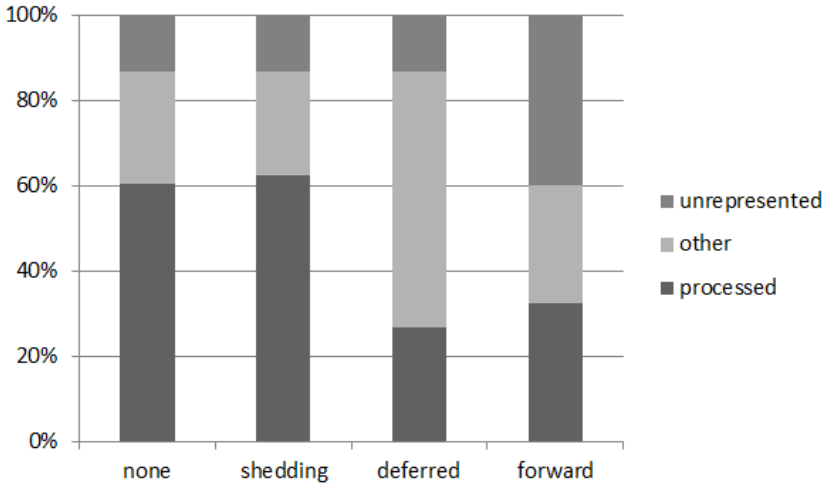


Figure 7.22: Relative Comparison of the Strategies for a Stateful Aggregation Query (25 Events / Second).

The relative results show, that the forwarding strategy with the current implementation is not applicable for high window sizes as more and more result events are unrepresented. The same applies for using more forwarding nodes for one overloaded node.

The present aggregation query uses the average operation for the processing. Therefore, the qualitative errors are not so high. Only when the input values have a wide range and the shedding or forwarding hits the input events one-sided, the error of the results might be significant for the load shedding and the forwarding strategy. Especially with an increasing window size, the relative errors for the aggregation operation are reduced. But the errors can be worse for different queries. If different aggregation operations are used, the errors can be more significant. Above all, the error grows evermore over the time for the sum operation.

Combined Query

The stateful combined query is also equal to the stateless scenario query except for the window definition. See Section 7.3.2 for the query and the output schema. An overload of the event processing node is caused with an input of 1200 events per minute and the combined query having a window size of 100.

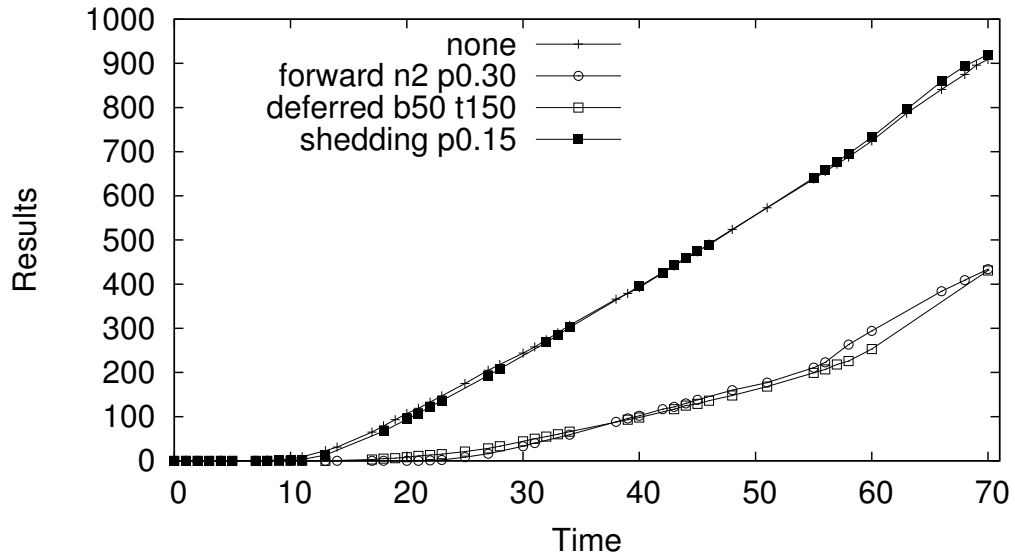


Figure 7.23: Comparison of the Strategies for a Stateful Combination Query (20 Events / Second).

The results for the stateful combination query are shown in Figures 7.23 and 7.24. The numbers do not significantly differ from the previous stateful evaluations results. But qualitatively this query subsumes all the risks of the previous types. Even though the aggregation operation of the query mitigates the error, the filtering, transformation and the usage of the sum and the count operation lead to qualitative errors in the results. The size of the errors cannot be generalized, as it depends not only on the processed query but also on the input data.

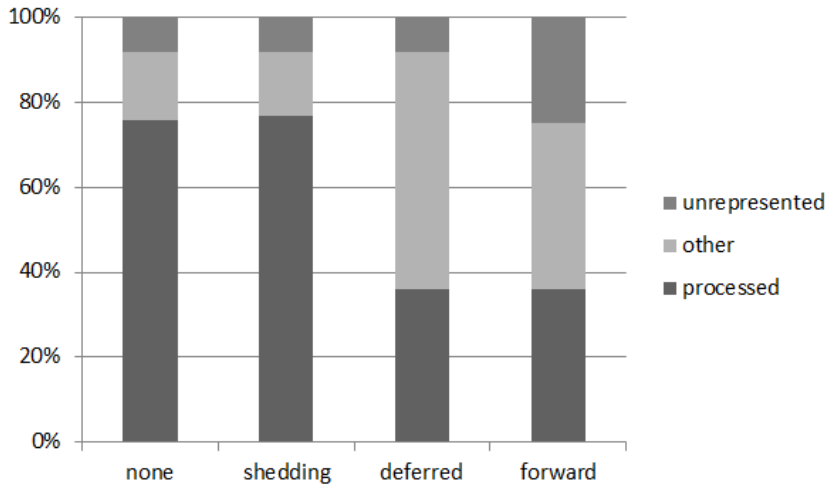


Figure 7.24: Relative Comparison of the Strategies for a Stateful Combined Query (25 Events / Second).

7.4.3 Summary for Stateful Queries

The quantitative results of the evaluation scenarios support the statements of the analysis in Section 7.4.1. Again, the results show that some strategies react differently to different scenario parameters. For example, the load shedding strategy can more or less be applied easily for all scenarios, whereas the forward strategy is strongly affected by the window size. So in addition to the features, which have already been identified in the stateless analysis, the window size is also of importance in the stateful context. As for the stateless queries, the load shedding approach is easy to use for all stateful queries too. It provides results fast, manages the overload, but may result in significant qualitative errors for stateful queries if no semantic load shedding is used. The deferred execution and the forwarding approach cannot provide as many results in the same time. Especially the deferred execution provides results rather slowly because of the persisting overhead. In return, it is the only strategy that provides correct results at all the time. A different implementation would be able to provide more results in the same amount of time, but then qualitative errors have to be accepted too. The present implementation of the forwarding strategy is not suitable to handle overload in combination with stateful queries. Especially the restriction for big window sizes is disqualifying. To be applicable for stateful queries, a forwarding strategy would have to work with a shared memory, so that the different nodes share the same window of input events.

The traced memory values are again not significant for providing a closer assessment of the strategies. Nevertheless, the evaluation results show that the memory consumption increases less in comparison to the stateless evaluation scenarios. Without a burst handling strategy, the memory utilization increases by around 5%. The explanation for this is, that the incoming events for the stateless evaluation are bigger than the events for the stateful evaluation as they contain more records. Therefore, the size of the input events has a great impact on the memory utilization.

CHAPTER 8

Summary

This chapters summarizes the findings of the thesis and draws conclusions regarding the problem formulation. Additionally, an outlook for future work is presented.

At present, more and more ESP systems try to provide features to handle overload. These approaches are often specialized for certain use cases and are not generally applicable. In this thesis, approaches that are used in state-of-the-art applications have been examined. More precisely, *Load Shedding*, *Deferred Execution* and *Forwarding* have been studied closely. For a well-founded evaluation of the strategies, the different characteristics of the environment have to be taken into account. As the scope of the thesis was limited to the application of the strategies on a single node and not on any node in an EPN, the placement of the strategy did not influence the evaluation. Strategies may perform better or worse, depending on the query that is processed on the node. Therefore, a basic taxonomy of queries has been defined. The taxonomy features two dimensions: the scope and the operation. The *Scope* of a query can either be stateless or stateful, whereby stateful can further be divided into different window types. The thesis mainly focuses on *Sliding Windows*. *Tumbling Windows* have not been treated separately as the results of sliding windows can also be used for assessing tumbling windows. The different types of the dimension *Operation*, which have been investigated, are *Transformation*, *Filtering* and *Aggregation*.

The concept of the integration on the burst handling strategies is based on two interfaces. The *Burst Manager* interface represents the actual integration, the decoupling of the event burst handling and the ESP system. The *Event Burst Handling* interface represents the controlling of the burst handling strategies. Therefore, the burst handling can be integrated in different systems and additional burst handling strategies can be added. The strategies implemented in Java have been integrated into the WS-Aggregation framework. The strategies have been applied on a rather high level of the application, in concrete terms, the event stream level. The strategies have been designed in a configurable and simple way, so they admit of improvement, if a more specific implementation is needed. The interfaces have been used to extend the framework by creating a *Burst Capable Aggregator Node* and the implementation of the *Burst Manager*. The latter one

functions as mediator between the framework and the strategies by hiding the implementation details of the framework from the strategies.

The overload simulations have been evaluated by stressing a deployed EPA while running a query of the mentioned types with a huge amount of data. Additionally, the events had a rather big size to enforce an overload. Based on the analyses and the evaluation of the implemented strategies, the applicability of the strategies for the different query types has been explored. The next section presents the conclusions that can be drawn based on the evaluation.

8.1 Conclusion

To subsume the results of the evaluation, the applicability of the different strategies strongly depends on the requirements of the event consumer. If timeliness is important, the only suitable strategy for the scenario queries considered in this thesis is load shedding. In return, missing or incorrect results have to be taken into account for both, stateless and stateful queries. If timeliness is not that important, the forwarding strategy can be used. If all input and output events have to be forwarded explicitly, the overhead in the evaluated scenarios is quite big, so the results can have a significant delay. Further on, the order of the results may be incorrect. For stateful queries, a forwarding strategy without a shared memory is not suitable, as too many events are not processed due to the window buffers being handled separately, resulting in less windows being completed. If the quality of the results including the ordering and the correctness is the most important criterion, the only applicable strategy for the evaluation scenarios is the deferred execution. In the present implementation it preserves the order of the input events and therefore guarantees correct results in the right order for both stateless and stateful queries. Depending on the used solution, persisting the events can be very expensive. So if timeliness is important in addition to correct results, a fast persistence approach has to be used.

In general, the evaluation shows that the load shedding approach performs equally well for all different query types. Also the configuration parameter for the shedder is easy to determine. The deferred execution and the forwarding approach perform much better for stateless queries than for stateful queries, which is caused by the input events being required to remain in the memory for a longer period of time. Further on the results show, that the deferred execution performs worse used on filtering queries in comparison to other queries.

The evaluation results of the memory utilization have not been significant and could not provide further insights for the assessment. The memory consumptions have to be traced in more detail and not just on the system level.

Finally, one more observation based on the evaluation results can be drawn. The results show that the practical effect of the strategies depends on the correct configuration for the particular scenario. While the sampling rate for the load shedding approach can be calculated quite easily, the tuning of the other approaches is more complex, which could be an obstacle for the automation.

8.2 Future Work

Within this thesis the applicability of known burst handling strategies has been analyzed for different kinds of queries in order to get a common understanding of the following questions:

- Which strategy is suited for which query type?
- If a strategy can only be applied to a query limitedly, which drawbacks does the strategy bring along?

Nevertheless, there are further research topics, which are out of the scope of this thesis but deserve to be treated by future work. The main topic that needs to be investigated more closely is the *Management of Burst Handling Strategies*, which has already been briefly discussed in Section 5.2.1. A management component should be able to recognize overload situations and to apply the best suited strategy. Therefore, two different fields regarding to the management have to be studied:

Monitoring & Analyzing: An ESP system has to provide a component that monitors the EPAs of the deployed EPN. It has to analyze the load information of the nodes and to spot indications for an overload before the actual overload happens. Such indicators should be studied in future work in order to provide indicators for detecting overload situations in advance.

Controlling & Decision Making: If an overloaded node is detected, the management component has to decide on the suitable strategy and where the strategy should be applied in the EPN. Further work could provide algorithms to automate these decisions. This thesis provides findings that can be used as a basis for the decision making. But to be able to make the decisions, the management component has to analyze which queries are processed on the affected node and which input event streams are related to these queries. Based on that information, a strategy for one of the event streams (or several strategies for some event streams) can be chosen. Further research should provide algorithms to choose the placement and clarify whether the choice of the strategy is independent of the placement in the EPN or not.

Further topics in the area of burst handling in ESP are still unexplored and may also be content of future work. For example, on which level the burst management is integrated. The presented strategies may provide better results in case of a deeper integration into the ESP system. For example, the forwarding strategy may be improved a lot by using a shared memory for all the processing nodes. This can only be achieved by integrating the strategy on the level of the event processing engine. Future work could analyze how great the impact of a deeper integration level is, and if it compensates the greater implementation effort and the loss of flexibility. Another form of the integration level could also be not to perform the burst handling on event stream level, but for example on the query level. In that case a different analysis would be required in the management component as it would not be about identifying high volume input streams but queries that use a high amount of processing resources. In return, the burst handling has a different impact on the outcome and may allow more precise results.

Besides, the extension of the strategies can also be part of further research. Strategies can be designed more configurable to enable the management component to react in a better way by adapting the strategies according to the findings of the query analysis.

Another topic for further research is the question, which QoS statements can be guaranteed by an ESP system using such load handling strategies, and how. As the qualitative errors caused

by the strategies depend on many factors, especially on the meaning of the processed query and the input data that is rather unknown in advance, this topic is quite complex.

Finally, as shown in the evaluation an overload eventuates when several factors occur together. Such factors are the size of the input events, the input frequency, the query that is currently processed and the window size in case of stateful queries. Different combinations of these factors cause an overload, so the potential relationships of those factors have to be studied to be able to recognize an overload in advance. An analysis of these coherences is needed as a basis for an operational monitoring component.

APPENDIX A

Literature Review

A.1 Query Taxonomy

The following tables (A.1, A.2) show an excerpt of the literature, which the query taxonomy is based on. The first column refers to the reference in the Bibliography (see Page 124). The second column gives a brief summary on the topics treated by the reference. Further explanations on these topics and terms have been given in Section 5.1.

Table A.1: Covered Literature for the Taxonomy Dimension Scope

	SCOPE
[121, 138]	operators: stateless, stateful
[62]	operators: stateless, stateful (changing streams) window bounds: tumbling, sliding
[40]	window types: logical (time-based), physical (count-based) window bounds: tumbling, sliding, fixed, landmark; single items
[52, 54, 55]	sliding window
[7]	window types: time-based, tuple-based, historical, suffix
[93]	window types: time-based, tuple-based, historical, predicate-based
[4]	window types: time-based, tuple-based, partitioned window bounds: sliding
[103]	window types: time-based, tuple-based window bounds: tumbling, sliding, landmark

Table A.2: Covered Literature for the Taxonomy Dimension Operation

	OPERATION
[7, 93]	aggregation
[80]	filtering: important to reduce size, aggregation: especially combined with transformation (simple events to complex events)
[130]	transformation: simple events to complex events, pattern matching: correlation of events
[128]	filtering, grouping, aggregation, transformation
[40]	select (filtering), elaboration (transformation by projection or renaming), aggregation
[44]	filtering, aggregation, grouping, event patterns
[81]	filtering (using event patterns), mapping (aggregation and transformation, also with event patterns)
[55]	filtering, correlation for event patterns, transformation, aggregation
[121]	selection, mapping, joining, aggregation, sorting

A.2 Load Handling Strategies

The following tables (A.3, A.4, A.5) show an excerpt of the literature, which has been used to define the approaches to handle overload situations, see Section 5.2. The first column refers to the reference in the Bibliography (see Page 124). The second column gives a brief summary on the topics treated by the reference.

Table A.3: Covered Literature Concerning the Load Shedding Strategy

	LOAD SHEDDING
[38, 131]	load shedding (respectively packet discarding) in other environments (multimedia, networking)
[12]	for sliding window aggregation queries, placement of shedders, load shedding as an optimization problem, load equation
[3]	load shedding based on QoS data, semantic load shedding
[118]	random and semantic (based on a utility function) load shedding, drop operators, placement
[119]	placement, optimization problem, offline load shedding planning, load monitoring, centralized versus distributed approaches
[107]	shadow query plan (result estimation of shed tuples to cope with special data during burst situations), using query rewriting, synopsis data structures
[41]	sliding window join queries, semantic tuple dropping
[94]	synopsis compression, sampling - eliminating tuples probabilistically, load shedding - dropping chunks of tuples

Table A.3: Covered Literature Concerning the Load Shedding Strategy (Continued)

	LOAD SHEDDING
[28]	scheduling and load shedding (random, semantic)

Table A.4: Covered Literature Concerning the Forwarding Strategy

	FORWARDING
[74]	horizontal partitioning considering semantic dependencies using stratification
[24]	optimistic parallelization for stateful operators
[137]	performance ratio metric to measure the performance of a query, dynamic load balancing
[86]	intra-operator parallelism
[66]	parallel query execution with *window split* and *window distribute* (partition - compute - combine)
[112]	parallel processing using adaptive partitioning with load balancing
[36]	dynamic query reconfiguration
[65]	partial aggregation, horizontal partitioning

Table A.5: Covered Literature Concerning the Deferred Execution Strategy

	DEFERRED EXECUTION
[2, 3]	deferred execution of events, which have no great impact on QoS data
[134, 135]	deferred execution in semantic contexts
[101]	deferred choice operator
[70]	transactional conditions: deferred coupled / decoupled
[102]	deferred processing in long term event processing
[139]	deferred operation in an escalation phase to find false positives
[85]	transactional deferred processing
[106]	no explicit deferment, but reordering to satisfy user preferences
[10]	no explicit deferment, but reordering for optimization using the eddy operator

APPENDIX B

Sequence Diagrams

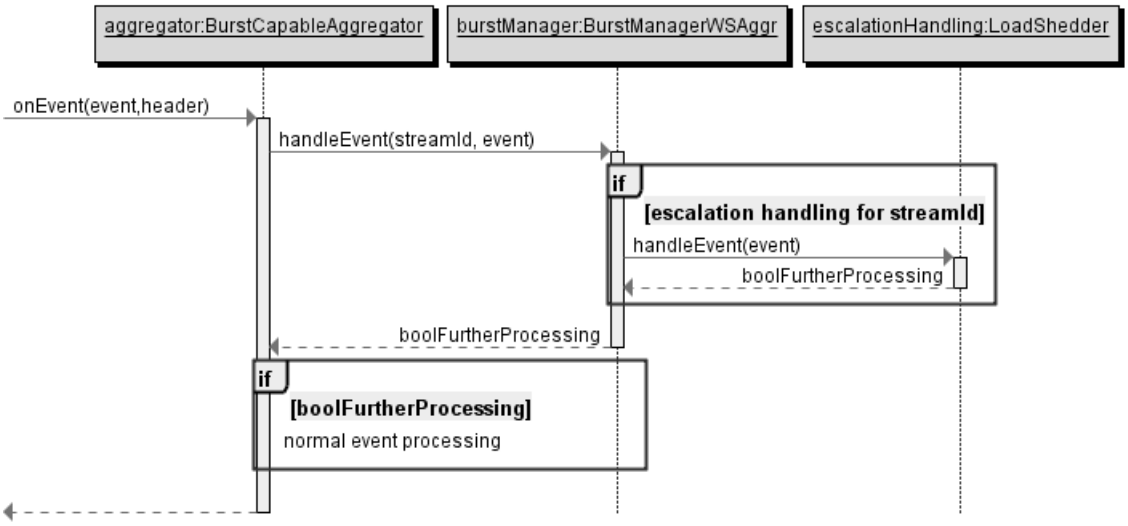


Figure B.1: Sequence Diagram of the Escalation Handling with Load Shedding.

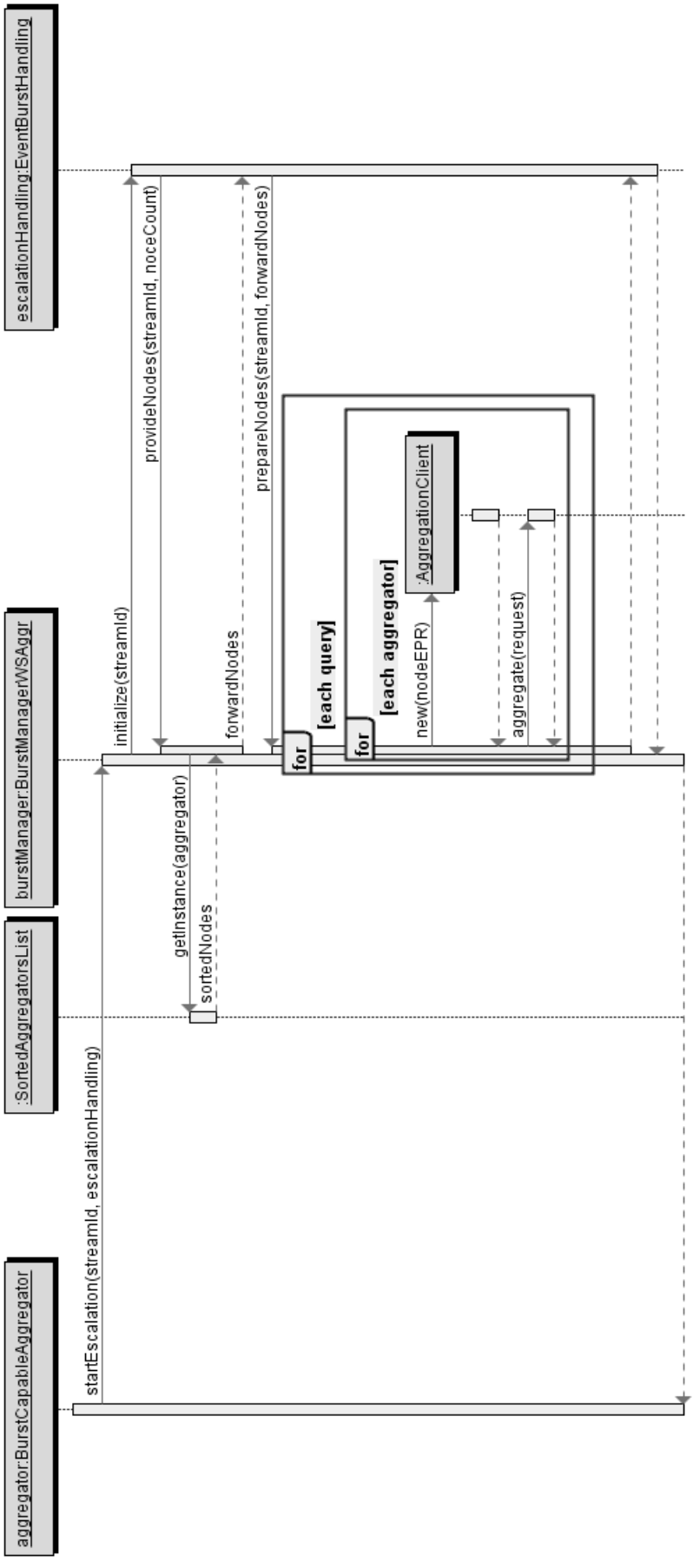


Figure B.2: Sequence Diagram of the Initialization of the Forwarding Strategy.

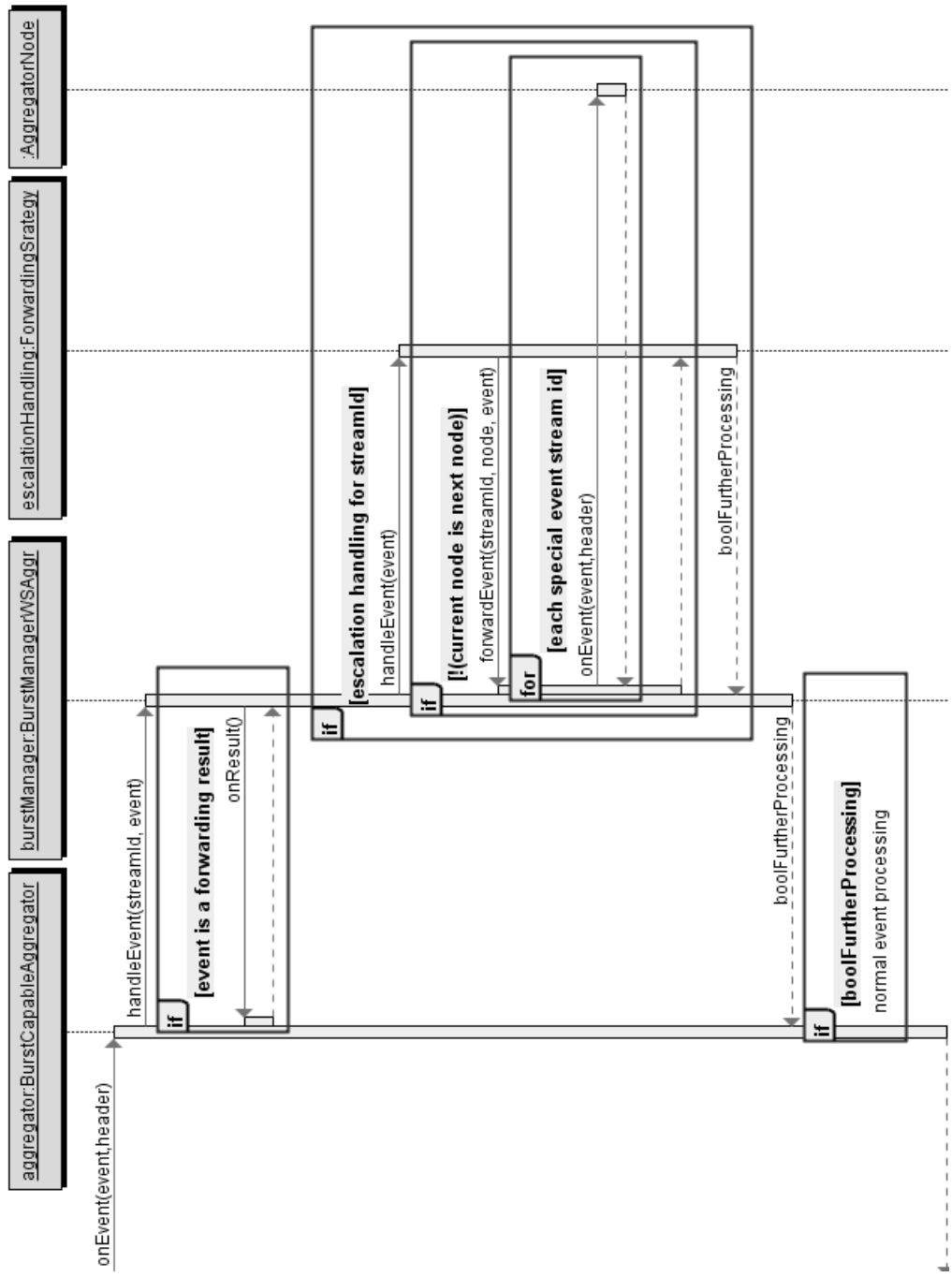


Figure B.3: Sequence Diagram of the Escalation Handling with a Forwarding Strategy.

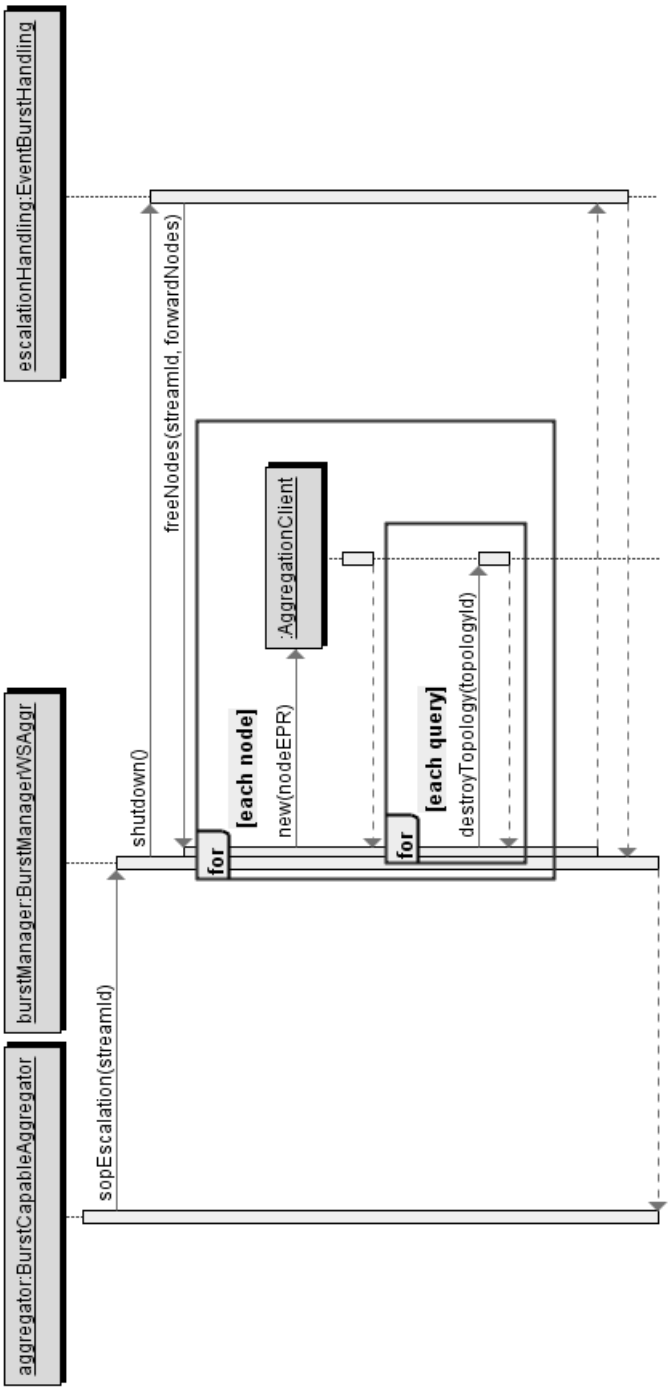


Figure B.4: Sequence Diagram of the Shutdown of the Forwarding Strategy.

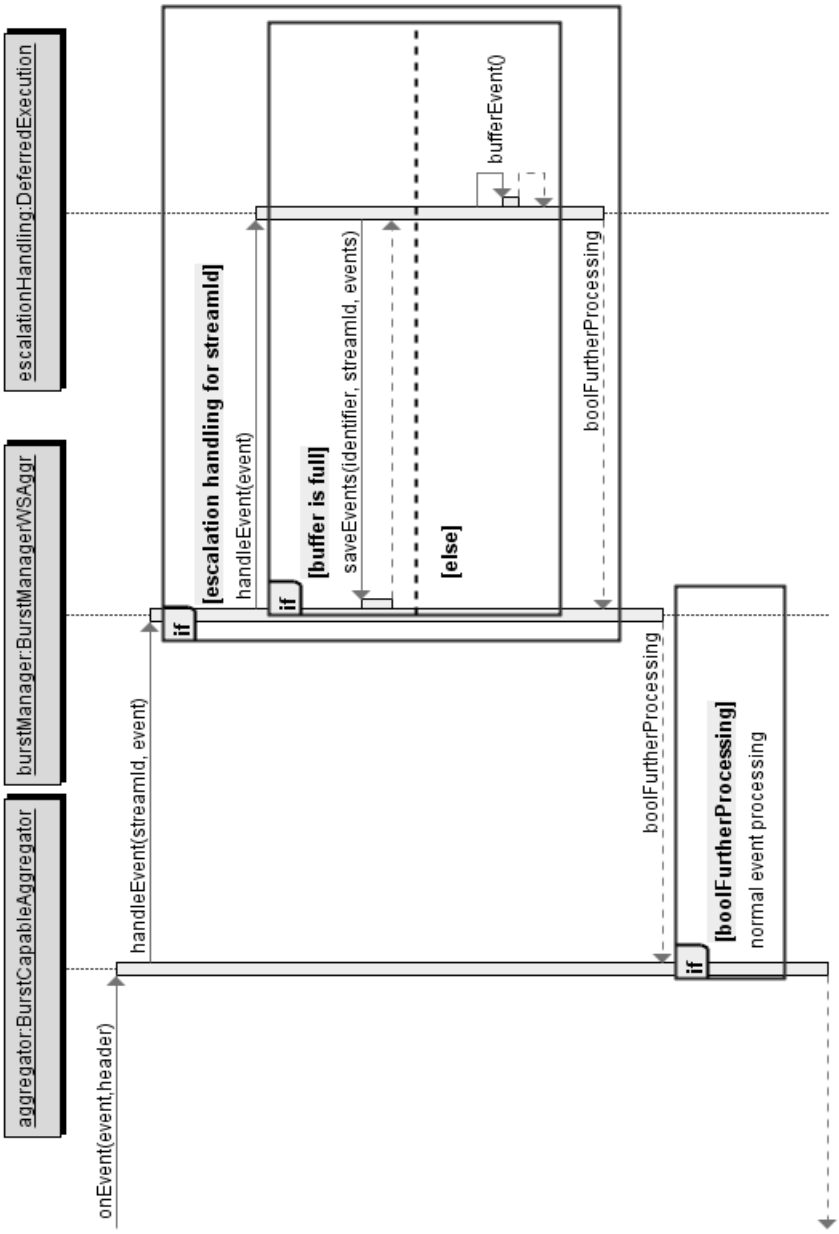


Figure B.5: Sequence Diagram of the Escalation Handling with Deferred Execution (Deferment).

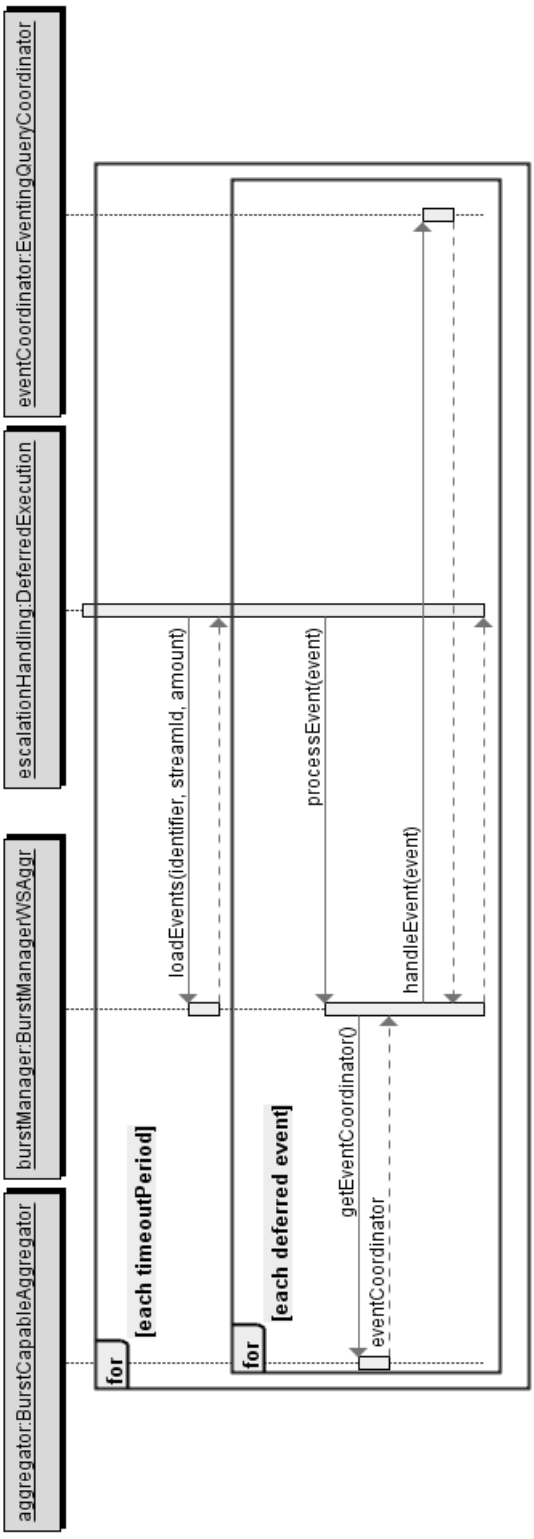


Figure B.6: Sequence Diagram of the Escalation Handling with Deferred Execution (Processing).

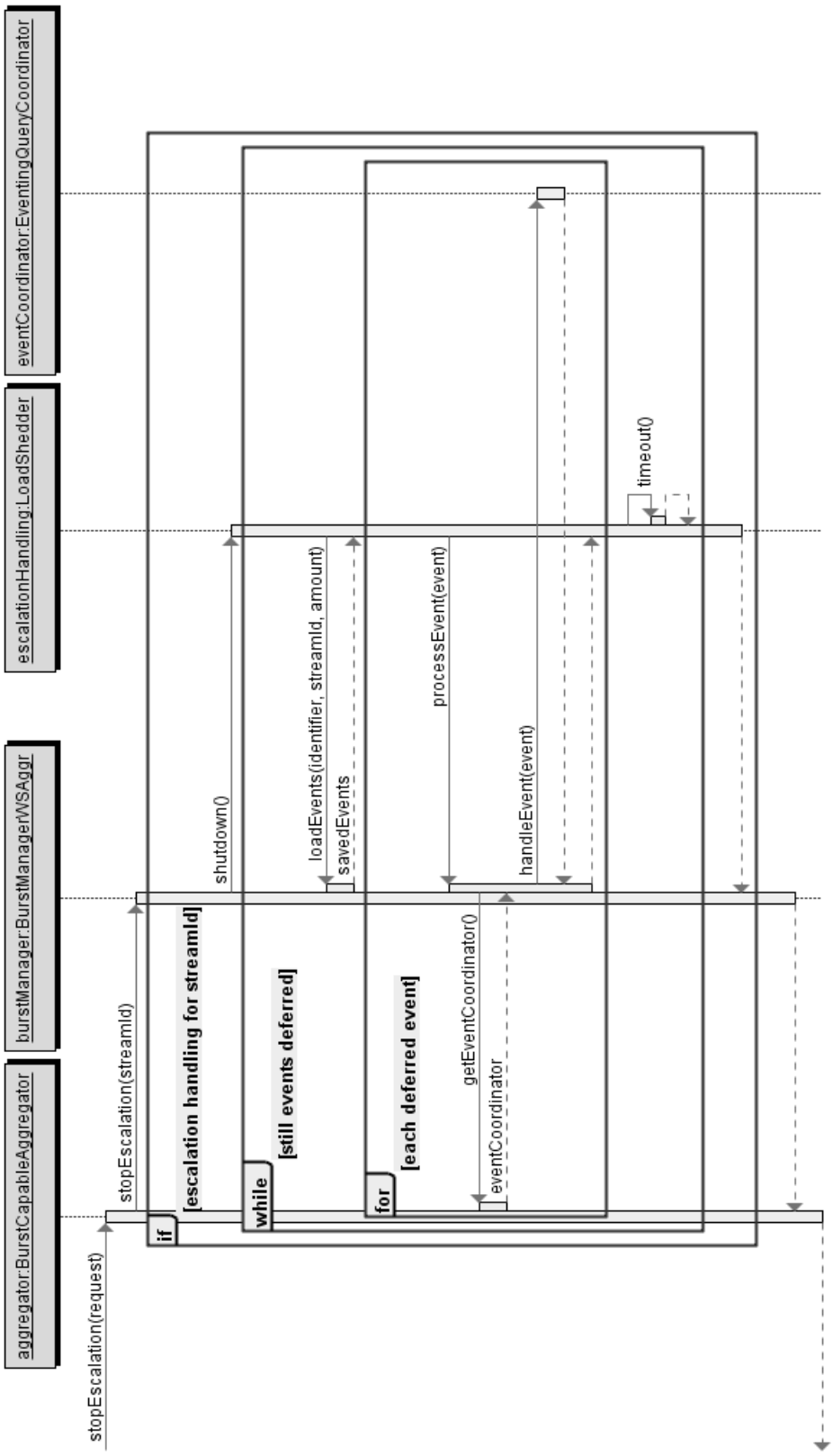


Figure B.7: Sequence Diagram of the Shutdown of the Deferred Execution.

Bibliography

[1] Espertech event stream intelligence: Esper & nesper. `http://esper.codehaus.org/`, Last visited on 2012-10-01.

[2] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *In CIDR*, pages 277–289, 2005.

[3] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal - The International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[4] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. Stream: The stanford data stream management system. *Book chapter*, 2004.

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

[6] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal - The International Journal on Very Large Data Bases*, 15(2):121–142, 2006.

[7] Arvind Arasu and Jennifer Widom. Resource sharing in continuous sliding-window aggregates. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renee J. Miller, Joses A. Blakeley, and K. Bernhard Schiefer, editors, *VLDB*, pages 336–347. Morgan Kaufmann, 2004.

[8] Tilo Arens, Frank Hettlich, Christian Karpfinger, Ulrich Kockelkorn, Klaus Lichtenegger, and Hellmuth Stachel. *Mathematik*. Spektrum Akademischer Verlag, 1. aufl. 2008 edition, 2 2008.

[9] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004.

[10] Ron Avnur and Joseph M Hellerstein. Eddies: Continuously adaptive query processing. In *ACM sigmod record*, volume 29, pages 261–272. ACM, 2000.

[11] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

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

[13] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, September 2001.

[14] Magdalena Balazinska, Hari Balakrishnan, Samuel R Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Transactions on Database Systems (TODS)*, 33(1):3, 2008.

[15] Magdalena Balazinska, Hari Balakrishnan, Jon Salz, and Mike Stonebraker. The medusa distributed stream-processing system. `http://nms.csail.mit.edu/projects/medusa/overview.pdf/`, Last visited on 2013-12-07.

[16] Roger S Barga and Hillary Caituiro-Monge. Event correlation and pattern detection in cedr. In *Current Trends in Database Technology–EDBT 2006*, pages 919–930. Springer, 2006.

[17] Roger S Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. Consistent streaming through time: A vision for event stream processing. *arXiv preprint cs/0612115*, 2006.

[18] Tim Bass. Mythbusters: Event stream processing versus complex event processing. In *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*, DEBS '07, pages 1–1, 2007.

[19] Thomas Bernhardt and Alexandre Vasseur. Esper: Event stream processing and correlation. `http://www.onjava.com/pub/a/onjava/2007/03/07/esper-event-stream-processing-and-correlation.html`, 2007. Last visited on 2013-12-07.

[20] Alain Biem, Eric Bouillet, Hanhua Feng, Anand Ranganathan, Anton Riabov, Olivier Verscheure, Haris Koutsopoulos, and Carlos Moran. Ibm infosphere streams for scalable, real-time, intelligent transportation services. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1093–1104. ACM, 2010.

[21] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Mobile Data Management*, volume 1987, pages 3–14. Springer, 2001.

[22] Irina Botan, Donald Kossmann, Peter M Fischer, Tim Kraska, Dana Florescu, and Rokas Tamosevicius. Extending xquery with window functions. In *Proceedings of the 33rd international conference on Very large data bases*, pages 75–86. VLDB Endowment, 2007.

[23] 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.

[24] Andrey Brito. Optimistic parallelization support for event stream processing systems. In *Proceedings of the 5th Middleware doctoral symposium*, MDS '08, pages 7–12, New York, NY, USA, 2008. ACM.

[25] Andrey Brito, Christof Fetzer, and Pascal Felber. Multithreading-enabled active replication for event stream processing operators. In *Reliable Distributed Systems, 2009. SRDS'09. 28th IEEE International Symposium on*, pages 22–31. IEEE, 2009.

[26] Michael Brundage. *XQuery: The XML Query Language*. Pearson Higher Education, 2004.

[27] Stefan Bruning, Stephan Weissleder, and Miroslaw Malek. A fault taxonomy for service-oriented architecture. In *Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium*, HASE '07, pages 367–368, Washington, DC, USA, 2007. IEEE Computer Society.

[28] Don Carney, Uur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 215–226. VLDB Endowment, 2002.

[29] Sharma Chakravarthy. *Stream data processing: a quality of service perspective: modeling, scheduling, load shedding, and complex event processing*, volume 36. Springer, 2009.

[30] Sharma Chakravarthy and Deepak Mishra. Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering*, 14(1):1–26, 1994.

[31] K. S. Chan, Judith Bishop, Johan Steyn, Luciano Baresi, and Sam Guinea. A fault taxonomy for web service composition. In Elisabetta Nitto and Matei Ripeanu, editors, *Service-Oriented Computing - ICSOC 2007 Workshops*, pages 363–375. Springer-Verlag, Berlin, Heidelberg, 2009.

[32] K. Mani Chandy and W. Roy Schulte. *Event Processing - Designing IT Systems for Agile Companies.* McGraw-Hill, 2010.

[33] Guanling Chen and David Kotz. Context aggregation and dissemination in ubiquitous computing systems. In *WMCSA*, pages 105–. IEEE Computer Society, 2002.

[34] Jianjun Chen, David J DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. *ACM SIGMOD Record*, 29(2):379–390, 2000.

[35] Kuan-Ta Chen, Polly Huang, and Chin-Laung Lei. Game traffic analysis: An MMORPG perspective. *Computer Networks*, 50(16):3002–3023, Nov 2006.

[36] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stanley B Zdonik. Scalable distributed stream processing. In *CIDR*, volume 3, pages 257–268, 2003.

[37] Storm Community. Storm concepts. `https://github.com/nathanmarz/storm/wiki/Concepts`, 2012. Last visited on 2013-12-28.

[38] Charles L Compton and David L Tennenhouse. Collaborative load shedding for media-based applications. In *Multimedia Computing and Systems, 1994., Proceedings of the International Conference on*, pages 496–501. IEEE, 1994.

[39] Constantinos A Constantinides, Atef Bader, Tzilla H Elrad, Paniti Netinant, and Mohamed E Fayad. Designing an aspect-oriented framework in an object-oriented environment. *ACM Computing Surveys (CSUR)*, 32(1es):41, 2000.

[40] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[41] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 40–51. ACM, 2003.

[42] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards expressive publish/subscribe systems. In *Advances in Database Technology-EDBT 2006*, pages 627–644. Springer, 2006.

[43] Alan J Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, Walker M White, et al. Cayuga: A general purpose event monitoring system. In *CIDR*, volume 7, pages 412–422, 2007.

[44] Liang Dong, Dong Wang, and Huanye Sheng. Design of rfid middleware based on complex event processing. In *Cybernetics and Intelligent Systems, 2006 IEEE Conference on*, pages 1–6, 2006.

[45] Abdennour El Rhalibi and Madjid Merabti. Agents-based modeling for a peer-to-peer mmog architecture. *Comput. Entertain.*, 3(2):3–3, April 2005.

[46] Tzilla Elrad, Robert E Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.

[47] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[48] Stefano Ferretti and Marco Roccetti. Fast delivery of game events with an optimistic synchronization mechanism in massive multiplayer online games. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, ACE '05, pages 405–412, New York, NY, USA, 2005. ACM.

[49] Robert E Filman and Daniel P Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced separation of Concerns, OOPSLA*, volume 2000, 2000.

[50] Stella Gatziu and Klaus R. Dittrich. Samos: An active object-oriented database system. *IEEE Data Eng. Bull.*, 15(1-4):23–26, 1992.

[51] Sandra Geisler. Data Stream Management Systems. In Phokion G. Kolaitis, Maurizio Lenzerini, and Nicole Schweikardt, editors, *Data Exchange, Integration, and Streams*, volume 5 of *Dagstuhl Follow-Ups*, pages 275–304. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2013.

[52] Lukasz Golab and M Tamer Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '03, pages 500–511. VLDB Endowment, 2003.

[53] Richard A. Golding. A weak-consistency architecture for distributed information services. *Computing Systems*, 5(4):379–405, 1992.

[54] The STREAM Group. Stream: The stanford stream data manager. Technical Report 2003-21, Stanford InfoLab, 2003.

[55] Daniel Gyllstrom, Eugene Wu, Hee-Jin Chae, Yanlei Diao, Patrick Stahlberg, and Gordon Anderson. Sase: Complex event processing over streams. *Conference on Innovative Data Systems Research (CIDR)*, pages 407–411, 2007.

[56] Moustafa A Hammad, Mohamed F Mokbel, Mohamed H Ali, Walid G Aref, Ann Christine Catlin, Ahmed K Elmagarmid, Mohamed Eltabakh, Mohamed G Elfeky, Thanaa M Ghanem, Robert Gwadera, et al. Nile: A query processing engine for data streams. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, page 851. IEEE, 2004.

[57] Juergen Hofer and Thomas Fahringer. A multi-perspective taxonomy for systematic classification of grid faults. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, PDP '08, pages 126–130, Washington, DC, USA, 2008. IEEE Computer Society.

[58] Waldemar Hummer, Christian Inzinger, Philipp Leitner, Benjamin Satzger, and Schahram Dustdar. Deriving a unified fault taxonomy for event-based systems. In *Proceedings of*

the 6th ACM International Conference on Distributed Event-Based Systems, DEBS '12, pages 167–178, New York, NY, USA, 2012. ACM.

[59] Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. A Step-By-Step Debugging Technique To Facilitate Mashup Development and Maintenance. In *4th International Workshop on Web APIs and Services Mashups (Mashups'10), co-located with ECOWS 2010*, 2010.

[60] Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. Ws-aggregation: Distributed aggregation of web services data. In *26th ACM SIGAPP Symposium On Applied Computing (SAC'11), SOAP Track*, 2011.

[61] Waldemar Hummer, Philipp Leitner, Anton Michlmayr, Florian Rosenberg, and Schahram Dustdar. *VRESCo - Vienna Runtime Environment for Service-oriented Computing*, pages 299–324. Service Engineering. European Research Results. Springer, 2010.

[62] Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Distributed continuous data aggregation over web service event streams. Technical Report (TUV-1841-2011-04), Vienna University of Technology, 2011.

[63] Walter L Hürsch and Cristina Videira Lopes. Separation of concerns. Technical report by the College of Computer Science, Northeastern University, 1995.

[64] J-H Hwang, Magdalena Balazinska, Alex Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 779–790. IEEE, 2005.

[65] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[66] Milena Ivanova and Tore Risch. Customizable parallel execution of scientific stream queries. In *Proceedings of the 31st international conference on Very large data bases*, pages 157–168. VLDB Endowment, 2005.

[67] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *ECOOP 2001 Object-Oriented Programming*, pages 327–354. Springer, 2001.

[68] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming, in proceedings of the european conference on object-oriented programming (ecoop), finland. *Springer-Verlag LNCS*, 1241:16, 1997.

[69] Jon Kleinberg. Bursty and hierarchical structure in streams. *Data Mining and Knowledge Discovery*, 7(4):373–397, 2003.

[70] Arne Koschel and Peter C Lockemann. Distributed events in active database systems: Letting the genie out of the bottle. *Data & Knowledge Engineering*, 25(1):11–28, 1998.

[71] Jürgen Krämer and Bernhard Seeger. Pipes: a public infrastructure for processing and exploring streams. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 925–926. ACM, 2004.

[72] Sailesh Krishnamurthy, Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Samuel Madden, Frederick Reiss, and Mehul A. Shah. Telegraphcq: An architectural status report. *IEEE Data Eng. Bull.*, 26(1):11–18, 2003.

[73] Christopher Kruegel, Thomas Toth, and Clemens Kerer. Decentralized event correlation for intrusion detection. In *Proceedings of the 4th International Conference Seoul on Information Security and Cryptology*, ICISC '01, pages 114–131. Springer-Verlag, 2002.

[74] Geetika T. Lakshmanan, Yuri G. Rabinovich, and Opher Etzion. A stratified approach for supporting high throughput event processing applications. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 5:1–5:12, New York, NY, USA, 2009. ACM.

[75] J.C. C. Laprie, A. Avizienis, and H. Kopetz, editors. *Dependability: Basic Concepts and Terminology*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.

[76] Jean-Claude Laprie. Dependable computing and fault-tolerance. *Digest of Papers FTCS-15*, pages 2–11, 1985.

[77] Philipp Leitner, Florian Rosenberg, and Schahram Dustdar. Daios: Efficient dynamic web service invocation. *IEEE Internet Computing*, 13(3):72–80, 2009.

[78] Ling Liu and Calton Pu. A dynamic query scheduling framework for distributed and evolving information systems. In *Distributed Computing Systems, 1997., Proceedings of the 17th International Conference on*, pages 474–481. IEEE, 1997.

[79] Ling Liu, Calton Pu, and Wei Tang. Continual queries for internet scale event-driven information delivery. *Knowledge and Data Engineering, IEEE Transactions on*, 11(4):610–628, 1999.

[80] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[81] David C. Luckham and Brian Frasca. Complex event processing in distributed systems. Technical report, Stanford University, 1998.

[82] Chang Luo, Hetal Thakkar, Haixun Wang, and Carlo Zaniolo. A native extension of sql for mining data streams. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 873–875. ACM, 2005.

[83] Samuel Madden, Mehul Shah, Joseph M Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 49–60. ACM, 2002.

[84] Oracle Product Management and Development Teams. Oracle complex event processing: Lightweight modular application event stream processing in the real world. `http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/oracle-37.pdf`, 2009. Last visited on 2013-12-28.

[85] Dennis McCarthy and Umeshwar Dayal. The architecture of an active database management system. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, SIGMOD '89, pages 215–224, New York, NY, USA, 1989. ACM.

[86] Manish Mehta and David J DeWitt. Managing intra-operator parallelism in parallel database systems. In *VLDB*, volume 95, pages 382–394, 1995.

[87] Rosa Meo, Giuseppe Psaila, and Stefano Ceri. Composite events in chimera. In *Advances in Database Technology - EDBT 96*, pages 56–76. Springer, 1996.

[88] Katina Michael and Luke McCathie. The pros and cons of rfid in supply chain management. In *Proceedings of the International Conference on Mobile Business*, ICMB '05, pages 623–629, Washington, DC, USA, 2005. IEEE Computer Society.

[89] B.M. Michelson. Event-driven architecture overview. *Patricia Seybold Group, Feb*, 2006.

[90] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. End-to-end support for qos-aware service selection, binding, and mediation in vresco. *IEEE Trans. Serv. Comput.*, 3(3):193–205, July 2010.

[91] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. Selective service provenance in the vresco runtime. *Int. J. Web Service Res.*, 7(2):65–86, 2010.

[92] Anton Michlmayr, Florian Rosenberg, Christian Platzer, Martin Treiber, and Schahram Dustdar. Towards recovering the broken soa triangle: a software engineering perspective. In *2nd international workshop on Service oriented software engineering: in conjunction with the 6th ESEC/FSE joint meeting*, IW-SOSWE '07, pages 22–28, New York, NY, USA, 2007. ACM.

[93] Mohamed F. Mokbel, Xiaopeng Xiong, Moustafa A. Hammad, and Walid G. Aref. Continuous query processing of spatio-temporal data streams in place. *Geoinformatica*, 9(4):343–365, December 2005.

[94] 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 *Proceedings of CIDR Conference*, pages 1–16, 2002.

[95] Catherine Moxey, Mike Edwards, Opher Etzion, Mamdouh Ibrahim, Sreekanth Iyer, Hubert Lalanne, Mweene Monze, Marc Peters, Yuri Rabinovich, Guy Sharon, et al. A conceptual model for event processing systems. *IBM Redguide publication*, 2010.

[96] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[97] S Muthukrishnan. Data streams: Algorithms and applications. *Theoretical Computer Science*, 1(2):117–236, 2005.

[98] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[99] Oracle. Oracle event processing. `http://www.oracle.com/us/products/middleware/soa/overview/complex-event-processing-ds-066411.pdf`, 2007. Last visited on 2013-12-28.

[100] Chris Otto, Aleksandar Milenković, Corey Sanders, and Emil Jovanov. System architecture of a wireless body area sensor network for ubiquitous health monitoring. *J. Mob. Multimed.*, 1(4):307–326, January 2005.

[101] Adrian Paschke. A homogenous reaction rule language for complex event processing. In *In Proc. 2nd International Workshop on Event Drive Architecture and Event Processing Systems (EDA-PS*, 2007.

[102] Adrian Paschke and Paul Vincent. A reference architecture for event processing. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, page 25. ACM, 2009.

[103] Kostas Patroumpas and Timos Sellis. Window specification over data streams. In *Current Trends in Database Technology–EDBT 2006*, pages 445–464. Springer, 2006.

[104] Christopher J. F. Pickett and Clark Verbrugge. Software thread level speculation for the java language and virtual machine environment. In *Proceedings of the 18th international conference on Languages and Compilers for Parallel Computing*, LCPC'05, pages 304–318, Berlin, Heidelberg, 2006. Springer-Verlag.

[105] Adam Przybyłek. Post object-oriented paradigms in software development: a comparative analysis. In *Proceedings of the International Multiconference on ISSN*, volume 1896, page 7094, 1896.

[106] Vijayshankar Raman, Bhaskaran Raman, and Joseph M Hellerstein. Online dynamic reordering for interactive data processing. In *VLDB*, volume 99, pages 709–720, 1999.

[107] Frederick Reiss and Joseph M Hellerstein. Data triage: An adaptive architecture for load shedding in telegraphcq. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 155–156. IEEE, 2005.

[108] Shariq Rizvi, Alberto Mendelzon, Sundararajarao Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 551–562. ACM, 2004.

[109] Jonas Rommelspacher. Ereignisgetriebene architekturen. *Wirtschaftsinformatik*, 50(4):314–317, 2008.

[110] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*, WMCSA '94, pages 85–90, Washington, DC, USA, 1994. IEEE Computer Society.

[111] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 4:1–4:12, New York, NY, USA, 2009.

[112] Mehul A Shah, Joseph M Hellerstein, Sirish Chandrasekaran, and Michael J Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 25–36. IEEE, 2003.

[113] Guy Sharon and Opher Etzion. Event-processing network model and implementation. *IBM Systems Journal*, 47(2):321–334, 2008.

[114] G SIREESHA and L BHARATHI. Exploiting dynamic resource allocation for efficient parallel data processing in the cloud. *International Journal of Engineering*, 1(6), 2012.

[115] Stevan Stankovski, Gordana Ostojić, and Milovan Lazarević. Rfid technology in product lifecycle management. *Engineering the Future, InTech*, pages 281–296, 2010.

[116] Martin Strassner. *RFID im Supply Chain Management*. Gabler Edition Wissenschaft. Dt. Univ.-Verl., 1. aufl. edition, 2005.

[117] Mark Sullivan. Tribeca: A stream database manager for network traffic analysis. In *VLDB*, volume 96, page 594, 1996.

[118] 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.

[119] Nesime Tatbul, Uur Çetintemel, and Stan Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd international conference on Very large data bases*, pages 159–170. VLDB Endowment, 2007.

[120] Nesime Tatbul and Stan Zdonik. Window-aware load shedding for aggregation queries over data streams. In *VLDB*, volume 6, pages 799–810, 2006.

[121] Junichi Tatemura, Arsany Sawires, Oliver Po, Songting Chen, K. Selcuk Candan, Diviyakant Agrawal, and Maria Goveas. Mashup feeds: continuous queries over web services. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1128–1130, New York, NY, USA, 2007.

[122] Esper Team and EsperTech Inc. Esper reference version 4.10.0. `http://esper.codehaus.org/esper-4.10.0/doc/reference/en-US/pdf/esper_reference.pdf/`, Last visited on 2013-12-07.

[123] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, SIGMOD '92, pages 321–330, New York, NY, USA, 1992. ACM.

[124] Hetal Thakkar, Barzan Mozafari, and Carlo Zaniolo. Designing an inductive data stream management system: the stream mill experience. In *Proceedings of the 2nd international workshop on Scalable stream processing system*, pages 79–88. ACM, 2008.

[125] Jean-Yves Tigli, Stephane Lavirotte, Gaetan Rey, Vincent Hourdin, Daniel Cheung-Foo-Wo, Eric Callegari, and Michel Riveill. Wcomp middleware for ubiquitous computing: Aspects and composite event-based web services. *ANNALS OF TELECOMMUNICATIONS (AOT)*, 64(3):197–214, 2009.

[126] Rainer von Ammon, Christoph Emmersberger, Torsten Greiner, Florian Springer, and Christian Wolff. Event-driven business process management. In *Fast Abstract, Second International Conference on Distributed Event-Based Systems, DEBS 2008, Rom, Juli 2008*, 2008.

[127] Fusheng Wang, Shaorong Liu, Peiya Liu, and Yijian Bai. Bridging physical and virtual-worlds: Complex event processing for rfid data streams. In *10th International Conference on Extending Database Technology (EDBT'2006)*, 2006.

[128] Weixin Wang, Jongwoo Sung, and Daeyoung Kim. Complex event processing in epc sensor network middleware for both rfid and wsn. In *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, ISORC '08, pages 165–169, Washington, DC, USA, 2008. IEEE Computer Society.

[129] Daniel Warneke and Odej Kao. Nephele: efficient parallel data processing in the cloud. In *Proceedings of the 2nd workshop on many-task computing on grids and supercomputers*, page 8. ACM, 2009.

[130] 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*, SIGMOD '06, pages 407–418, New York, NY, USA, 2006. ACM.

[131] Cui-Qing Yang and Alapati VS Reddy. A taxonomy for congestion control algorithms in packet switching networks. *Network, IEEE*, 9(4):34–45, 1995.

[132] Tjalling J. Ypma. Historical development of the newton-raphson method. *SIAM Rev.*, 37(4):531–551, December 1995.

[133] Cong Yu and Lucian Popa. Constraint-based xml query rewriting for data integration. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 371–382. ACM, 2004.

[134] Chuanzhen Zang and Yushun Fan. Complex event processing in enterprise information systems based on rfid. *Enterprise Information Systems*, 1(1):3–23, 2007.

[135] Chuanzhen Zang, Yushun Fan, and Renjing Liu. Architecture, implementation and application of complex event processing in enterprise information systems based on rfid. *Information Systems Frontiers*, 10(5):543–553, 2008.

[136] Ting Zhang, Yuanxin Ouyang, and Yang He. Traceable air baggage handling system based on rfid tags in the airport. *J. Theor. Appl. Electron. Commer. Res.*, 3(1):106–115, April 2008.

[137] Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Ji Wu. Efficient dynamic operator placement in a locally distributed continuous query system. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, pages 54–71. Springer, 2006.

[138] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. Dynamic plan migration for continuous queries over data streams. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 431–442, New York, NY, USA, 2004.

[139] Michael Zoumboulakis and George Roussos. Escalation: Complex event detection in wireless sensor networks. In *Smart Sensing and Context*, pages 270–285. Springer, 2007.

List of Abbreviations

AOP Aspect Oriented Programming.

CEP Complex Event Processing.

DBMS Database Management System.

DHAP DBMS-Active, Human-Passive.

DSG Distributed Systems Group.

DSMS Data Stream Management System.

EBS Event-Based System.

EDBPM Event-Driven Business Process Management.

EDIP Event-Driven Interaction Paradigm.

EP Event Processing.

EPA Event Processing Agent.

EPN Event Processing Network.

EPS Event Processing System.

EQL Esper Query Language.

ESP Event Stream Processing.

HTML Hypertext Markup Language.

JVM Java Virtual Machine.

MMOG Massive Multiplayer Online Games.

POP Post-object programming.

QoS Quality of Service.

RFID Radio-Frequency Identification.

SOA Service Oriented Architecture.

SOAP Simple Object Access Protocol.

SOC Service Oriented Computing.

SQuAl Stream Query Algebra.

STM Software Transactional Memory.

TLS Thread Level Speculation.

VMF VRESCo Mapping Framework.

VQL VRESCo Query Language.

VRESCo Vienna RuntimeEnvironment for Service-oriented Computing.

WAQL Web Services Aggregation Query Language.

WSN Wireless Sensor Network.

XML Extensible Markup Language.