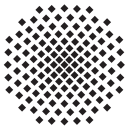


Universität Stuttgart

Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor's Thesis Nr. 184

Increasing the Bandwidth Efficiency of Content-based Routing in Software-defined Networks

Jonas Grunert

Course of Study:	Software Engineering
Examiner:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Supervisor:	Dr. rer. nat. M. Adnan Tariq
Commenced:	12 May, 2014
Completed:	11 November, 2014
CR-Classification:	C.2.1,C.2.4

Abstract

Content-based routing systems such as *publish/subscribe (pub/sub)* have become an important model for distributed systems with loosely coupled participants. Usually publish/subscribe systems are realized by overlay networks where dissemination and filtering of information is done in the application layer, which causes significant delay. The emergence of *software-defined networking (SDN)*, where switches with programmable TCAM memory allow dynamic configuration of networks, has opened new opportunities in realizing dynamic logic inside the network. Current publications have presented realizations of pub/sub systems based on SDN. In these systems the information filtering is not done in the application layer but directly inside the network by switches. This allows event filtering with low delay and *line-rate* performance.

However, SDN-based pub/sub systems are limited by the available resources. The TCAM memory of the switches, containing the forwarding rules, is very cost-intensive and hence the maximum number of rules and their complexity is limited. In order to provide bandwidth-efficient content-based routing it is necessary to use a large number of complex forwarding rules. Therefore the limitation of resources causes a drop of the routing quality and less bandwidth-efficient routing.

In this thesis, approaches to increase bandwidth-efficiency in the context of limited resources are proposed. To achieve efficient routing, the precision of *in-network filtering* must be high to avoid unnecessarily disseminated information, so-called *false positives*, which cause higher network utilization.
This thesis proposes and evaluates two approaches to increase the efficiency of in-network filtering: Selection of more important information to be used for filtering and improvement of the filtering itself. Several algorithms to rate the importance of information are proposed and evaluated. Furthermore, ways to combine the selection of information and the improved filtering are shown.
Our results show that the developed approaches can strongly reduce the number of false positives. The combination of best performing approaches can reduce the number of false positives by up to 75% and thereby increase the bandwidth efficiency significantly.

Kurzfassung

Content-based routing Systeme wie *Publish/Subscribe (pub/sub)* sind zu einem wichtigen Modell für verteilte Systeme mit lose gekoppelten Komponenten geworden. Üblicherweise werden Publish/Subscribe-Systeme mittels Overlay-Netzwerken realisiert, in denen die Verteilung und Filterung von Informationen auf Anwendungsebene durchgeführt wird, was eine signifikante Verzögerung verursacht. Das Aufkommen von *Software-Defined Networking (SDN)*, bei dem Switches mit programmierbarem TCAM-Speicher eine dynamische Konfiguration des Netzwerks ermöglichen, hat neue Möglichkeiten für die Realisierung dynamischer Logik im Netzwerk eröffnet. In aktuellen Veröffentlichungen wurden Implementierungen von SDN-basierten pub-/sub Systemen präsentiert. Bei diesen Systemen wird die Filterung von Informationen nicht auf Anwendungsebene sondern direkt im Netzwerk durch Switches durchgeführt.

Jedoch werden SDN-basierte Systeme durch die zur Verfügung stehenden Resourcen eingeschränkt. Der TCAM-Speicher der Switches, welcher die Weiterleitungs-Regeln enthält, ist sehr kostenintensiv und daher ist die maximale Anzahl von Regeln und deren Komplexität eingeschränkt. Um bandbreiteneffizientes content-based Routing zu ermöglichen ist es jedoch nötig viele komplexe Regeln zu verwenden. Daher verursachen die begrenzten Ressourcen einen Abfall der Routingqualität und damit eine geringere bandbreiteneffizient.

In dieser Arbeit werden Ansätze zur Verbesserung der Bandbreiteneffizienz, im Kontext beschränkter Ressourcen, vorgestellt. Um effizientes Routing zu erreichen muss die Präzision des *in-network filtering* hoch sein um unnötige Verteilung von Informationen, sog. *false positives*, welche die Netzwerkauslastung erhöhen, zu vermeiden.
Diese Arbeit präsentiert und evaluiert zwei grundlegende Ansätze zur Erhöhung der Effizient der Filterung: Auswahl wichtigerer Informationen die für das Filtern verwendet werden sollen und eine Verbesserung der Filterung selbst. Diverse Algorithmen zur Bewertung der Wichtigkeit von Informationen werden vorgestellt und evaluiert. Außerdem werden Wege gezeigt um die Auswahl wichtigerer Informationen und die verbesserte Filterung zu kombinieren.
Unsere Ergegnisse zeigen, dass die entwickelten Ansätze die Anzahl der *false positives* stark verringern können. Die Kombination der besten Ansätze kann die Anzahl der *false positives* um bis zu 75% verringern und dadurch die Bandbreiteneffizienz deutlich erhöhen.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

In the last decades, the amount of data exchanged via computer networks has grown significantly. There is a high demand for bandwidth efficient communication for loosely coupled applications. *Content-based networking* strategies allow efficient data exchange between loosely coupled applications. In content-based networks routing is not performed by sending data to a specified address, as done by traditional routing protocols but instead based on the information to be exchanged.
One paradigm of content-based networking is *publish/subscribe (pub/sub)* [BCM+99, CDNF01, GSAA04, FCMB06, PRGK09, TKKR09, JCL+10, BFPB10, TKK+11, TKKR12, TKR13]. In pub/sub systems *publishers* publish information, called *events*, into the network. All events consist a set of values describing the event. *Subscribers* express their interest in events in a certain range with a *subscription filter*. The function of the pub/sub system is to filter and forward events, published by publishers, to all subscribers interested in this type of event.
The majority of content-based publish/subscribe is realized by overlay networks where servers, called *brokers*, perform the forwarding of events [CRW01, Pie04, BBQV07, CJ11, TKKR12, TKR13]. Brokers filter incoming events and forward them to adjacent brokers and clients interested in this type of events. This type of system has two major drawbacks: Events are not directly disseminated end-to-end and the filtering of events is performed on the brokers by software in the application layer. Both causes a higher delay of events.

The emergence of *Software-defined networking (SDN)* [Com12] opens new possibilities for realizing dynamic logic inside the network. SDN networks consist of switches with programmable *TCAM memory*. Using the TCAM memory, SDN switches can match incoming packets and perform actions with this packet with low latency.
SDN networks are configured by installing forwarding rules, called flow rules, in the TCAM memory of the switches. Each flow rule consists of two parts: The matching rules and the actions to perform with the packet. Matching is done by comparing the header fields of incoming packets on the switch with the matching rules of the flow rules. If the header fields of a packet entering the switch matches a flow rule, the actions of the rule are performed with this packet. Possible actions are to send the packet to specified output ports and to manipulate packet header fields.
OpenFlow [Fou] is the de-facto standard for SDN networks. In OpenFlow a so-called controller can determine the network logic dynamically and configure the network by installing flow rules on the SDN switches. OpenFlow allows matching and manipulating of various packet header fields like target and destination IP, MAC and port.

The controller has a centralized view over the whole network and therefore knows the network topology, can react to topology changes, can install flow rules, and can measure metrics such as network utilization. This centralized view and the possibility to configure the network allows the controller to control and configure the network dynamically and realize complex logic inside the network.
As the whole network logic can be implemented by configuring SDN switches no additional components, such as brokers, are needed. All logic can be realized in the network. SDN switches using TCAM memory to match the packets can do the routing in line-rate performance which means packet filtering and forwarding with lowest latency and full network bandwidth.

The *PLEROMA* middleware system [TKBR14, KDTR12, KDT13] is a publish/subscribe system realizing content-based routing in software-defined networking. Instead of using brokers for filtering events the filtering is performed by SDN switches. As the whole pub/sub logic is realized by SDN flow rules matching packets all filtering is done in the network. This *in-network filtering* does not have the disadvantages of broker-based pub/sub systems mentioned above and offers line-rate performance. The event packets can be transmitted directly through the network without using an overlay network of brokers and the filtering can be done on the network-layer instead of the application layer.
Based on OpenFlow, PLEROMA configures the SDN network so that the network can perform all the event filtering. Publishers that disseminate events send event packets into the network with the target IP field set to an index representing the event, called *dz-expression (dz)* which is used for matching of flow rules on the SDN switches.

However, SDN-based pub/sub systems cause new types of problems: The resources on SDN switches are limited, the TCAM memory is very cost-intensive and therefore only a limited number of flow rules can be installed. Furthermore only certain fields in the packet headers can be matched which means that there is only a limited amount of information that can be encoded in the packet header fields and matched by the switches. In case of *PLEROMA* this limits the maximum length of the dz-expression to the number of available bits in the target IP field.
As all filtering and routing decisions are based on flow rules matching the event dz-expression, the precision of the dz is crucial for the filtering of events. For a high filtering precision many flow rules and long dz-expressions are needed. Therefore the limited number of flow rules and the limited number of available bits in the target IP field leads to a lower filtering precision. A lower filtering precision leads to a higher number of unnecessarily disseminated events, so-called *false positives*. This causes a higher network utilization which can lead in the worst case to network overload and system failure.

The contribution of this thesis is to develop and evaluate ways to increase the bandwidth efficiency of content-based routing in SDN for publish/subscribe systems like PLEROMA. The contributions in detail are:

- Development of algorithms, improving the bandwidth efficiency by selecting event value *dimensions* to generate more informative *dz-expressions* which allow better filtering.

Dimensions are definitions of event value ranges. In a system with n dimensions, all events consist of n values and the range of each value is defined by the corresponding dimension.

- Development of an approach to improve the in-network filtering by generating better *dz-expressions*. The dz generation is improved by improving the partition of event spaces. The event space is an n dimensional space for systems with n dimensions and represents the value range of all dimensions.
- Evaluation of all proposed algorithms, including the combination of dimension selection and improved partitioning.

In Chapter 2 the *PLEROMA* middleware system and the system model this thesis is based on is presented. The chapter presents the architecture, the system model and problems of PLEROMA.

In Chapter 3 various algorithms are proposed which improve the generation of dz-expressions by selecting dimensions to use for dz generation.

In Chapter 4 a way to improve the dz generation itself will be presented. The proposed algorithm improves the partitioning of the event space during spatial indexing which is used to generate the dz-expressions.

In Chapter 5 all dimension selection algorithms and the improved partitioning will be evaluated. The algorithms will be evaluated under different circumstances to find out strengths and weaknesses.

Finally Chapter 6 summarizes the thesis and gives an outlook towards future work.

Chapter 2

PLEROMA Middleware

The work presented in this thesis is based on the system model of the *PLEROMA* middleware system, a SDN-based publish/subscribe system, presented in [TKBR14]. This chapter will give an overview over the concepts of *PLEROMA*, the implementation of it and will point out potential problems of the concepts. The Chapters 3 and 4 will present approaches to reduce these problems.

2.1 Middleware Overview

2.1.1 Architecture

Figure 2.1 shows an overview of the architecture of the *PLEROMA* middleware system. The main component of the system is the controller. It observes the network and configures the SDN network by installing and deleting flow rules on the switches.
The hosts are connected to the SDN network through which the events are disseminated. Hosts can be publishers or subscribers. Event filtering and forwarding between publishers and subscribers is performed by the SDN switches according to the installed flow rules.
Control messages from hosts, like advertisements or subscriptions, can be sent from hosts to the controller directly by using a fixed IP address IP_{fix}. All packets with this destination address are sent directly to the controller by any switch in the network. The address IP_{fix} is therefore reserved and may not be used for any other purpose. Network changes, such as advertisements or a subscriptions, are recognized and handled by the controller. The handling of network changes will be explained in detail in Section 2.2.

PLEROMA uses the OpenFlow standard which allows matching of packets header fields like target and destination IP address, MAC and port. For every flow rule the matching rules and a set of actions to perform can be defined, such as manipulating packet header fields, redirecting a packet to a specific outgoing port or to multiple ports. The whole network logic of PLEROMA is implemented by configuring SDN switches with flow rules determined by the PLEROMA controller.

For SDN switches routing packets through the network, the packet header fields of event packets need to be configured so that the packets can be matched by the switches. When manipulating header fields it has to be taken care of not conflicting with other network services.

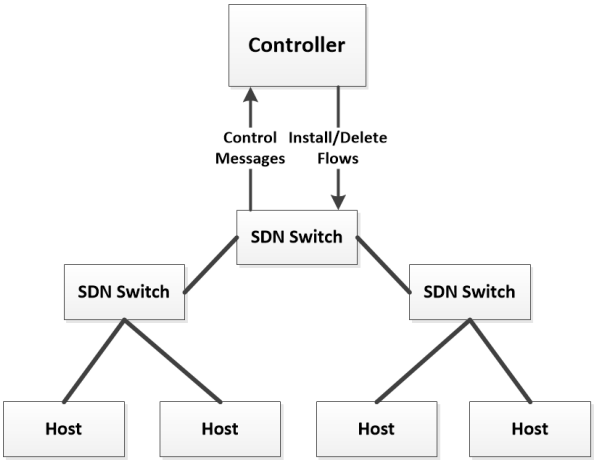


Figure 2.1: PLEROMA middleware architecture

Therefore *PLEROMA* uses IP-Multicast addresses to set the target IP fields for packet configuration and matching. Hosts send packets and set the target IP field to a constant multicast prefix followed by an index representing the event information, called *dz-expression*.

2.1.2 Content-based model

PLEROMA follows the content-based subscription model. This means, that every event is composed of dimensions value pairs. The range of all dimensions forms a multi-dimensional space, called event space Ω.
Every event is represented by a point in Ω, every advertisement and subscription, for events in a certain range, is represented by a multi-dimensional space. Based on the principle of *spatial indexing* [KDTR12], the event space is divided into regular subspaces to approximate events, advertisements and subscriptions. The subspaces are identified by binary strings called *dz-expressions (dz)*. These binary strings are later uses to identify and match event network packets with installed flow rules.

Every bit, from left to right, identifies a subspace partition of the event space Ω. As illustrated in figure 2.2 the space is divided along alternating dimensions. Every additional bit specifies a more fine granular subspace, every bit halves the size of the subspace.

Figure 2.3 shows an example of a subspace that can be represented by two bits. The subspace from (50,0) to (100,50) includes the both subspaces with the *dz* 100 and 101. These two subspaces together form the subspace 10, from (50,0) to (100,50).

If necessary, a subspace can be represented by multiple *dz*. Figure 2.4 shows an example where a subspace from (50,0) to (75,100) can not be represented by only one *dz*. In this case two *dz* 100 and 110 are used to represent the space. They don't have a common parent-subspace so it is necessary to use both *dz* to describe the space precisely.

All in all, the properties of Dz-expressions can be summarized by four major properties:

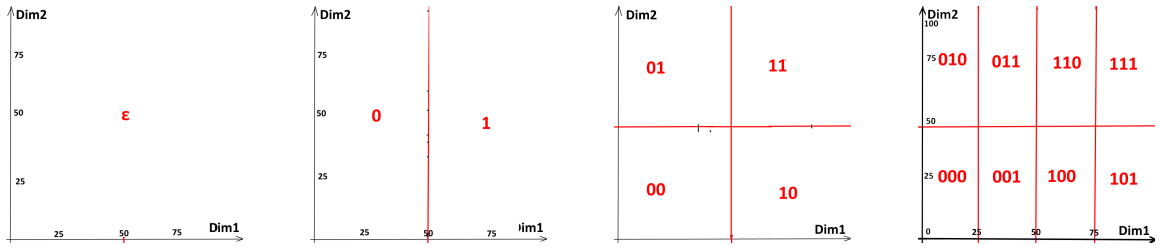


Figure 2.2: Illustration of separation of two-dimensional event space into subspaces, represented by *dz*

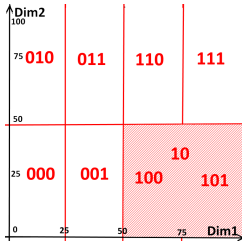


Figure 2.3: Indexing of a subspace from (50,0) to (100,50) with one *dz* 10

1. A shorter *dz* has a larger subspace. Longer *dzs* are more fine granular.
2. The corresponding subspace of dz_i is covered by the subspace of dz_j if dz_j is a prefix of dz_i. The additional bits of dz_i determine the more fine granular subspace of the subspace dz_j.
3. Two subspaces dz_i and dz_j are overlapping if dz_i covers dz_j or vice versa. The overlap is the subspace of the longer *dz*.
4. In case of overlapping subspaces, where no space covers the other completely, the non overlapping part must be identified by multiple subspaces.

Events are always represented by exactly one *dz* which specifies the location of a point in the event space, representing the event. Advertisements and Subscriptions can be represented by multiple *dz*, denoted as *DZ* to approximate the space of the advertisement/subscription.

An event e is sent in a packet with a dz_e, that represents the dimension values of e, as

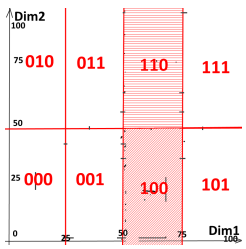


Figure 2.4: Indexing of a subspace from (50,0) to (75,100) with two *dz* 100 and 110

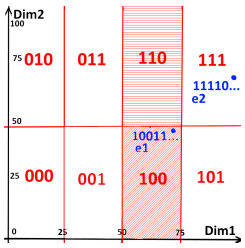


Figure 2.5: Indexing of a subscription and two events

index. The in-network filtering is based on the principle that a subscription s must receive e if e is inside the subspace of s. The set of dz DZ_s represents the subspace of s. If dz_e is inside a subspace dz_s out of DZ_s, e is matched by the subscription s.

Figure 2.5 illustrates this. The subscription $s1$ is represented by the two dz 100 and 110. The event $e1$, represented by dz_{e1}=10011..., is matched by the subscription $s1$ because it is in the subspace 100, as 100 is a prefix of dz_{e1}.
In contrast, $e2$, represented by dz_{e2}=11110... is not matched by $s1$ because neither 100 nor 110 are a prefix of dz_{e2}.

*PLEROMA*s event delivery is based on this type of *event filtering*. The controller installs *flow rules* matching packet header fields on the above explained indexes. Publishers emit events with their corresponding dz encoded in the destination IP address. Flow rules on the switches in the network match on dz for the following subscribers. An event is delivered to a subscriber if there is a path of matching flows towards this subscriber.
As overlapping subscriptions can share common event delivery paths, thanks to the subspace relationship of *dz-expressions*. This means that an event packet can be sent through the network only once for multiple subscribers along the common path which reduces the network traffic.

2.2 Content-based routing

This section explains the network reconfiguration, based on the above explained *spatial indexing*. The controller reacts on changes in the network, calculates routing paths and updates the switches flow tables.

2.2.1 Maintenance of Spanning Trees

Upon every advertisement/subscription and unadvertisement/unsubscription the *controller* has to reconfigure the SDN network to implement the publish/subscribe functionalities. The goal of the in-network filtering is to achieve bandwidth and latency efficient delivery of events to all interested subscribers. Besides efficient in-network filtering one more constraint is efficient network updating including calculation and installing/modifying flow rules.

The easiest solution, with the lowest latency, for the network configuration would be to install shortest paths between all publishers and subscribers. Though this strategy causes a high number of flow rules to install, upon every network change a huge number of flows must be installed/modified. Besides it is difficult to use common paths to reduce the network traffic as it would be possible using spatial indexing.

Therefore *PLEROMA* uses dynamic sets of spanning trees connecting publishers and subscribers. The creation and update of spanning trees is publisher driven, spanning trees are updated upon advertisements and unadvertisements. Spanning trees represent routes on which publishers can forward events.
Flow updating is subscriber driven. For all subscribers, flows along the spanning trees are installed to deliver the event packets to the interested subscribers. Finally the events are forwarded along spanning trees along paths specified by flow rules.

In order to manage the spanning trees, the controller holds a set T of spanning trees (short *trees*). Each tree $t \in T$ covers a set of subspaces represented by a set of *dz-expressions* $DZ(t)$. A tree is a spanning tree for all publishers with advertisements covered by $DZ(t)$. All trees are disjoint which means for all trees t and t' $DZ(t) \cap DZ(t') = \oslash$. Resulting from this, an event is never forwarded along more than one tree.

Upon every new publisher p advertising on $DZ(p)$, the controller searches for every $dz_i \in DZ(p)$ if there is an overlap with an existing tree $t \in T$. The controller evaluates if the publisher can join existing trees and if new trees have to be created. For every $dz_i \in DZ(p)$ one out of three possible actions is performed:

1. If dz_i is *covered completely* by the spaces of one ore more existing trees p joins all covering trees.

2. If dz_i is *covered partially* by the spaces of one ore more trees p joins all covering trees and a new tree t_n is created where $DZ(t_n)$ is the set of dz representing the uncovered space.

3. If dz_i is *not covered* by any tree a new tree t_n is created with $DZ(t_n) = DZ(p)$.

The number of trees can quickly grow, especially when there are many advertisements with different ranges. To avoid a large number of trees, the controller can merge trees when the number of flows passes a threshold.

Along the spanning trees for the publishers, the flows rules to forward events to subscribers are installed. On every subscriber s subscribing with a subscription on $DZ(s)$ the controller searches for a set of trees T_s out of T with trees overlapping with $DZ(s)$.
If T_s is empty this means that there is no publisher publishing events for this subscription at this time. In this case the subscription is stored and activated as soon there are publishers relevant for this subscription.

In case T_s is not empty the controller establishes paths between the subscriber s and all publishers on all trees in T_s.

2.2.2 Maintenance of Flow Tables

For every publisher a route along the corresponding tree is calculated. A route is a path of flows installed on SDN switches forming a path through the network. In a second step flow rules are installed or modified to establish the path in the real network.

Every flow consists of matching fields and a set of out actions. As *PLEROMA* uses a fixed IP-multicast range to avoid conflicts with other services, the flows are matching on IP-addresses starting with the multicast prefix followed by the *dz-expression*.
For example for a subspace with the $dz_s = 101$ IPv4 corresponding address might be 225.160.0.0/11 when using a prefix 225.0.0.0/8. An event with the $dz_e = 101101$ would have the IP 225.180.0.0/14.
SDN switches support wildcard/masking operations. A flow rule on a subspace represented by the IP 225.160.0.0/11 matches packets with destination IP 225.180.0.0/14. This way flow rules can match events for certain subspaces and paths along spanning trees can be programmed.

2.3 Limitations of in-network Filtering

The necessary index length representing events and their dimensions increases linearly with the number of dimensions. As the number of available bits for the index is fixed, depending on the number of IP bits, in case of many dimensions, the events can not be represented with full precision. In this case the rate of falsely delivered event packets *false positive rate (FPR)* rises.
To solve this scalability problem, in [TKBR14] the concept of *Dimension Selection* is introduced. Every dimension is represented by a dimension in the index. The *dimension selection component* reduces the dimension to a smaller subset of dimensions which are more relevant for the in-network filtering and routing.

Using spatial indexing to generate *dz-expressions*, the dz length increases linearly with the number of event dimensions. Furthermore the number of *dz* to represent subscriptions and their routing paths also rises with the number of dimensions.
In a practical scenario both, the maximum length and the maximum number of *dz* is limited. By using IPv4 or IPv6 the number of IP bits is limited which causes a limitation for the dz length. The maximum IP range is 24 bits for IPv4 and 112 bits for IPv6. As the cost intensive TCAM memory is limited, in order of 40.000 to 180.000 rules [DK], the number of flow rules is limited too.
Both, IP bits and flow rule space can be lower if the network is shared with other services. The available IP range might be lower than the full IP-multicast range and flow rule space might be consumed by other services.
Consequently the precision of spatial index will decreases when the number of dimensions

increases. When the precision of the index decreases the in-network filtering is less precise and the number of events wrongly forwarded to subscriber, the *false positive rate (FPR)*, increases. With post filtering at the host side these events can be filtered out but a high FPR increases the network bandwidth. A high FPR can cause unnecessary network overloading.

Dimension Selection reduces this problem by selecting only a subset of dimensions to be used for spatial indexing. The goal is to find the set of dimensions that is most relevant to create a good, accurate index for good filtering. When leaving out less important dimensions more bits can be used to index the important dimensions more accurately. The best dimensions are the dimensions with the lowest number of *false positives* when used to create the index for in-network filtering.
The filtering effectiveness of dimensions mainly depends on three factors: distribution of events, selectivity of subscriptions and correlation of dimensions.

In [TKBR14] a basic approach to select dimensions is presented and evaluated. Evaluations show that a reduced set of dimensions can decrease the FPR.
The proposed algorithm which uses the set of active subscriptions and the set E^t of last events published. This data can be collected by the controller. The algorithm calculates a matrix W where every field $w_{i,j}$ is the number of subscriptions matched by the event $e_j \in E^t$ along dimension i. In a next step the variance along each row is calculated. For each dimension, the variance of the corresponding row can represent the importance of the dimension.
For handling correlated dimensions an algorithm is proposed calculating the covariance matrix from the variances. The covariance matrix indicates the correlation between dimensions. This covariance matrix is then eigendecomposed, the original dimension is transformed into a orthogonal basis. Based on the method presented in [MG04], the eigenvector with the largest eigenvalue represents the dimension which maximizes the variance. Finally those dimensions are selected which are most relevant in the selected eigenvector. The relevance of a dimension d_i is given by the absolute value of the i^{th} coefficient in the eigenvector.

The focus of this thesis is on improving the result of filtering by improving spatial indexing. Therefore two approaches are research - the dimension selection presented in Chapter 3 and the improved partitioning presented in Chapter 4.
Various algorithms to select dimensions are proposed, improving and extending the principle of dimension selection as presented in [TKBR14]. Different metrics and selection strategies are presented to improve the results of dimension selection.

Chapter 3

Dimensions Selection

As explained in Section 2.3, the *false positive rate (FPR)* can be reduced by selecting a subset of most promising dimensions to generate better *dz-expressions* with spatial indexing. When leaving out less promising dimensions, more bits can be used to index the information of the better dimensions more accurate. The basic idea of dimension selection was originally introduced in [TKBR14].

In this chapter various metrics for rating the relevance of dimensions and algorithms to select dimensions are presented. The proposed metrics try to predict how much a dimension can reduce the FPR when used to generate the *dz-expressions*. Based on these metrics the algorithms select the most promising dimensions.
Besides algorithms to select the best dimensions, algorithms to detect the best number of dimensions are proposed. The combination of detecting the best number of dimensions and the rating and selection of the most promising dimensions allows it to find the best set of dimensions to use for generating *dz-expressions* with spatial indexing.

There are a lot of different scenarios for dimensions with different distributions of events and subscribers. A good algorithm has to be universal - it should select the best dimensions in any possible scenario. This chapter will present different kinds possible scenarios for dimension selection and will improve the algorithms step by step to find more universal algorithms.

3.1 Event-based Selection

The two following figures show how the filtering can be improved by using only one dimension for the spatial indexing. Black rectangles represent subscriptions, every point represents an event.
In general, the goal of filtering is to decide if an event is inside a subscription or not. Wrongly classifications of events belonging to a subscription are *false positives*. The false positive rate represents the quality of the filtering.
Every red rectangle represents one index area. The filter decides that an event belongs to a subscription if the subscription contains or intersects the rectangle the event is in. If an event is in the same rectangle as a subscription but not inside the subscription it will be assigned to a subscription wrongly, it is then a *false positive*.

Figure 3.1 shows spatial indexing with four index bits with both dimensions, figure 3.2 shows indexing which leaves out the dimension with less variance. As you can see in Figure 3.1, filtering with an index with both dimension would have many false positives in the index areas 0110, 1100, 0011 and 1001. In these areas events that are inside the index area can not always be assigned to the right subscription.
This would be improved by indexing only with *Dimension 2*. *Dimension 1* is less relevant for filtering and when leaving it out the indexing precision along dimension 2 would be better. Figure 3.2 shows that the events can be assigned to subscriptions much more precisely. The false positive rate would be much better.

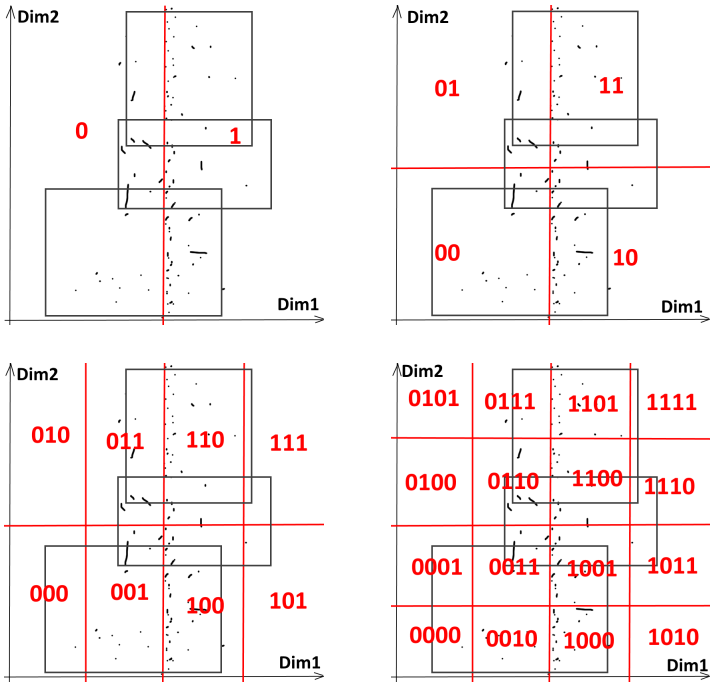


Figure 3.1: Spatial indexing, 4 bits, both dimensions

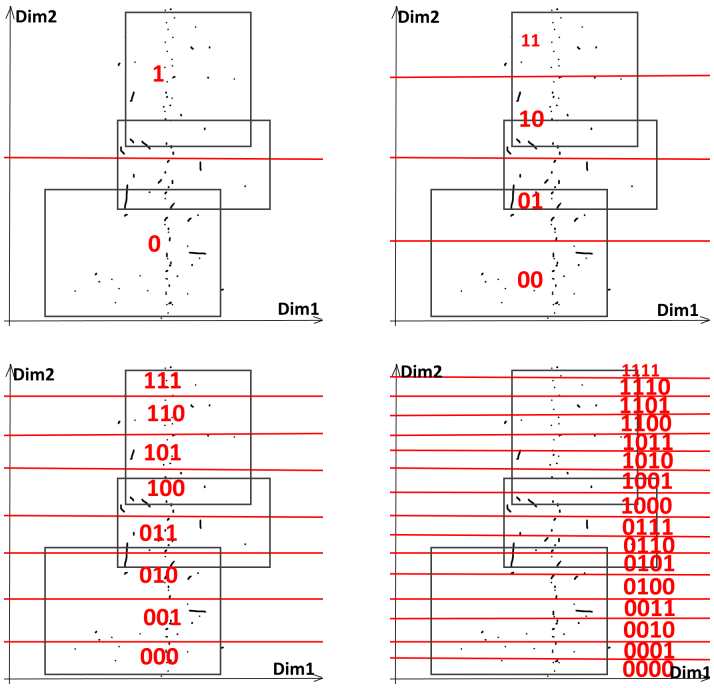


Figure 3.2: Spatial indexing, 4 bits, only dimension 2

3.1.1 Algorithm: Event-based Selection (EVS)

As explained in Section 3.1 the variance of events along dimensions is an important factor for the filtering effectiveness of a dimension. In this section an algorithm will be introduced that reduces the number of dimensions by selecting dimensions with higher variance of events. We call this algorithm *event variance-based dimension selection (EVS)*

The algorithm calculates for every dimension the variance of the events along this dimension. For this the mean of all event values in one dimension is calculated and the sum of the distances of all event values to the mean value is calculated. The square root of this value is the variance of event values along this dimension.
As input, parameter, the number of dimensions to select is given. For n dimensions to select for $0 < n < originalDimCount$ the algorithm selects the n dimensions with the highest variance.
Algorithm 3.1 shows the pseudocode of this algorithm. The function *CalcEvtVariances* calculates the event variances along the dimensions. Based on this the given number of dimensions with the highest variances is selected.

The advantage of this approach is the low calculation overhead. This algorithms time complexity is $\mathcal{O}(n_d * n_e)$ when n_d is the number of dimensions and n_e is the number of events. The complexity is linear dependent on the number of events and dimensions because the variance has to be calculated for all dimensions for all event values.
The disadvantage of this algorithm is that it is not universal. It can only analyze the event variances. It does not take into account the subscriptions and the relations between subscriptions and events. Subscription can also have an influence on the importance of a dimension but this algorithm only analyzes the events. In a scenario where the characteristics of subscriptions along dimensions is different from the characteristics of the events, this algorithm will not be able to find the best dimensions.

In the following sections new scenarios this algorithm can not deal with will be presented. Based on this algorithm, new algorithms will be proposed to solve theses problems.

Algorithm 1 Select Dimensions based on Event Variances (EVS)

```
1: function SelectDimensions(origDims[], selDimCount, events[])
2:     selDims[selDimCount]                          ▷ Array for selected dimensions
3:     variances[] ← CalcEvtVariances(origDims.length, events)
4:     for iSel = 0 to selDimCount do
5:                                                   ▷ Index of n-highest event variance
6:         dim ← GetNHighestValueIndex(iSel, variances)
7:         selDims[iSel] ← origDims[dim]
8:     end for
9:     return selDims
10: end function
11:
12: function CalcEvtVariances(dimCount, events[])   ▷ Calculate event variance along
    dimensions
13:     variances[dimCount]          ▷ Array with all event event variances along dimensions
14:     for iDim = 0 to dimCount do
15:         sum ← 0
16:         for iEvt = 0 to events.length do
17:             sum ← sum + events[iEvt].values[iDim]
18:         end for
19:         mean ← sum/events.length          ▷ Mean of all event values for this dimension
20:         var ← 0
21:         for iEvt = 0 to events.length do
22:             var ← (events[iEvt].values[iDim] − mean)^2
23:         end for
24:         variances[iDim] ← √var                ▷ Variance of events along this dimension
25:     end for
26:     return variances
27: end function
```

3.2 Subscription-based Selection

The last presented algorithm does not take into account subscriptions but it is obvious that subscriptions can have an influence on the dimension importance, independently of the event variance. This means that it is possible that the importance of a dimension can be different from the variance events.
Figure 3.3 shows a scenario where the importance of the dimensions is different from the variance of the events.

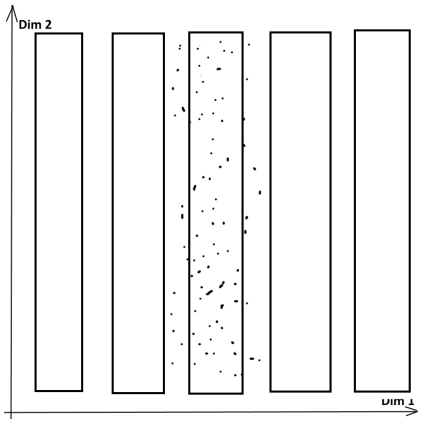


Figure 3.3: Variance of events and dimension importance different

In this scenario the algorithm based on event variance would choose *dimension 2* because of the high event variance along this dimension. However *dimension 2* is less important, the algorithm would choose the wrong dimension.
To illustrate this, figure 3.4 shows that an index based on *dimension 2* would not be good for filtering the events. When indexing and filtering with *dimension 2*, events would be assigned to all subscription but they only belong to one subscription. In this case false positive rate would be very high.

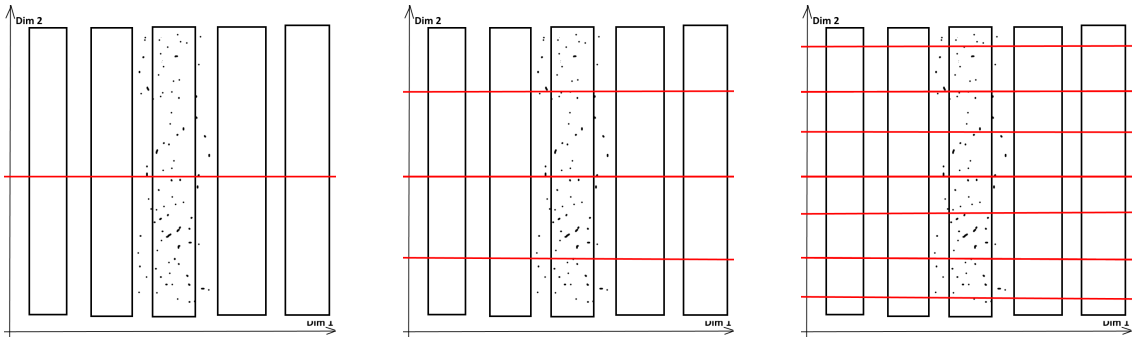


Figure 3.4: Spatial indexing, 3 bits, dimension 2

Figure 3.5 shows that an index based on *dimension 1* would be very good in filtering events. The events would only be assigned to the middle subscription. The false positive rate would be much lower compared to filtering with *dimension 2*.
A good selection algorithm should choose *dimension 1* in this scenario. This shows, that only analyzing the variance of events is not sufficient and that events, subscriptions and the relation between subscriptions and events has to be analyzed.

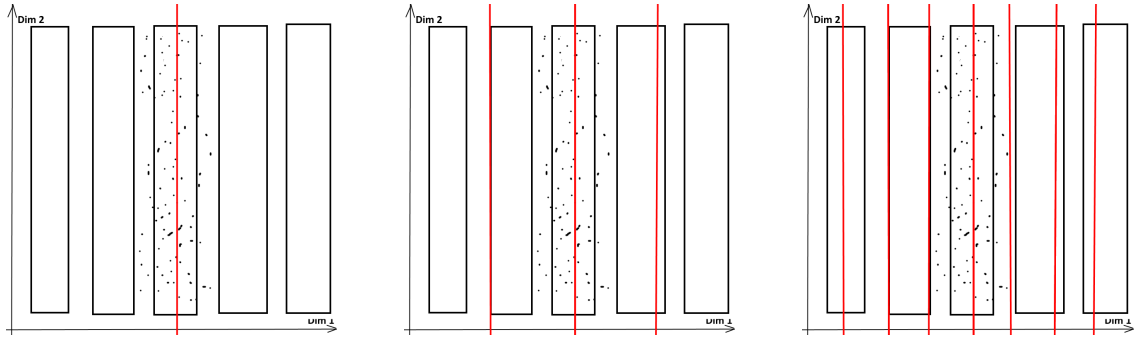


Figure 3.5: Spatial indexing, 3 bits, dimension 1

For developing improved algorithms new metrics to analyze subscriptions and event-subscription relations are needed. Those subscription metrics analyze how the subscriptions are distributed and how good the events can be correctly assigned to subscriptions correctly. Therefore we call those metrics *subscription selectivity* of the dimensions. Algorithms to calculate the subscription selectivity will be introduced
In section 3.2.1 an algorithm to select dimensions based on the *Subscription Selectivity* metric is presented. Sections 3.2.2, 3.2.2 and 3.2.2 present three different ways to calculate the subscription selectivity.

3.2.1 Selectivity-based Selection

Based on the *Subscription Selectivity* the dimensions can be rated and selected. For this an algorithm similar to the variance-based selection *EVS* can be used. Equivalent to the calculation of the variance, the subscription metrics will also be calculated for each dimension separately. The *Subscription Selectivity based dimensions Selection (SSS)* is illustrated in Algorithm 2.
Based on events and subscriptions the algorithm calculates for every dimension the *Subscription Selectivity*. The parameter *selDimCount* specifies the desired number of dimensions. The algorithm selects the desired number of dimensions with the highest *Subscription Selectivity*.

The time complexity of this algorithm depends on the complexity of the algorithm calculating the *Subscription Selectivity*. This is $O(n_d * n_e * n_s^2)$ for *SOS* or $O(n_d * n_e * n_s)$ for *SMS* and *SFS* when n_d is the number of dimensions, n_e the number of events and n_s the number of subscriptions.

Subscription Selectivity is not only a metric analyzing the subscription behavior. It also

Algorithm 2 Selecting based on Subscription Selectivity

```
1: function SelectDimensions(origDims[], selDimCount, events[], subscriptions[])
2:     selDims[selDimCount]                          ▷ Array for selected dimensions
3:     subSels[] ← CalcSubSelectivity(origDims.length, events, subscriptions)
4:     for iSel = 0 to selDimCount do
5:                                                   ▷ Index of n-highest dimension weight
6:         dim ← GetNHighestValueIndex(iSel, subSels)
7:         selDims[iSel] ← origDims[dim]
8:     end for
9:     return selDims
10: end function
```

considers the relations between subscriptions and events e.g. events matched by multiple subscriptions.

The following three sections present different approaches for calculating the *Subscription Selectivity*. For all those approaches this algorithm is used to select the dimensions, the calculation of the *Subscription Selectivity* is used by this algorithm through the call of the method **CalcSubSelectivity**.

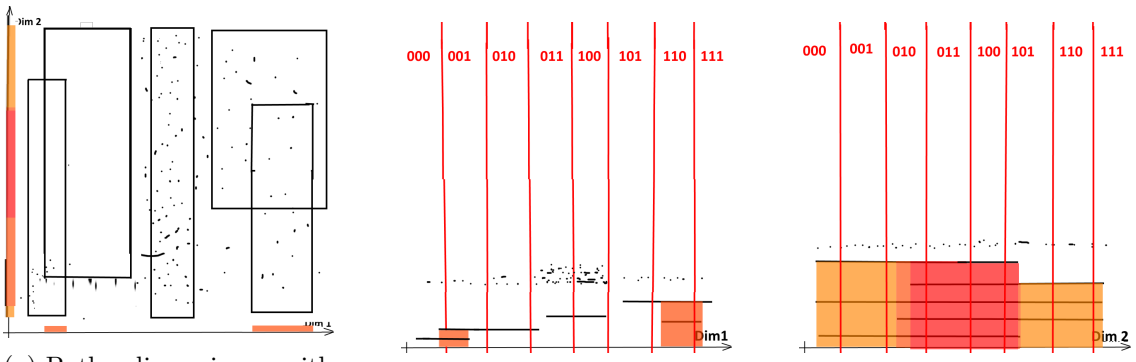
3.2.2 Overlap-based Selection (SOS)

In this section an approach for calculating the *subscription selectivity* based on subscription overlaps will be presented. The calculated subscription selectivity is used to select dimensions with the algorithm presented in the last section. We call the dimension selection based on this metric *subscription overlap based selection (SOS)* Subscription overlap in a dimension means that subscriptions match the same area along a dimension. The subscription overlap influences the selectivity, the filtering effectiveness of a dimension

Dimensions where subscriptions have a lot of overlap are less important for filtering. Figure 3.6 illustrates this. The majority of events are matched by all subscriptions on *Dimension 2* while only a few events are matched by more than one subscription on *Dimension 1.* Darker colors show areas on dimensions with higher overlap.

Filtering with Dimension 1 would have a low false positive rate. The events can be assigned to the correct subscriptions because most of the events are matched by only one subscription. In contrast when filtering with Dimension 2, the false positive rate would be very high. Many events are matched by multiple subscription on *Dimension 2* and many of these subscriptions don't match the event on *Dimension 1.* This means that many events would be assigned to subscriptions they are not matched by on all dimensions. These events are false positives because an event is only inside a subscription if it is matched by it on all dimensions.

Even when analyzing the variance of events and the overlap of subscriptions there are still scenarios that are not matched. The subscription coverage can be low and the variance of events



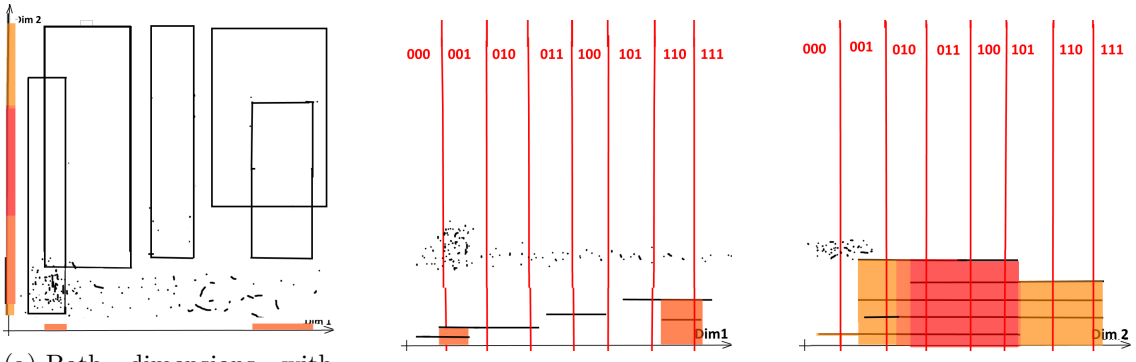
(a) Both dimensions with subscriptions, events and overlap

(b) Spatial Indexing with Dimension 1

(c) Spatial Indexing with Dimension 2

Figure 3.6: Example for dimensions selectivity and overlap

can be high but if the events are not matching all subscriptions the effect of the subscription overlap can be eliminated. In the end the overlap of events matched by subscriptions is what influences the dimension selectivity.
Figure 3.7 shows such a scenario. In *Dimension 1* the variance of events is high and the overlap of subscriptions is low while the variance in *Dimension 2* is low and the overlap high. However *Dimension 2* is good for filtering and *Dimension 1* is not.
The reason is that the majority of events is matched by only one subscription in *Dimension 2* but matched by more than one subscription in Dimension 1. Therefore Dimension 2 is better for filtering because the events can be better assigned to a single subscription.



(a) Both dimensions with subscriptions, events and overlap

(b) Spatial Indexing with Dimension 1

(c) Spatial Indexing with Dimension 2

Figure 3.7: Example for dimensions selectivity and overlap

The proposed algorithm, illustrated in Algorithm 3 calculates the *Subscription Selectivity* based on the overlap of subscriptions matching events. A value between 0.0 and 1.0 represents the degree of inverse overlap. High degree of overlap means bad filtering effectiveness so the *Subscription Selectivity* is the inverse of the degree of subscription overlap.

Algorithm 3 Calculate Selectivity based on Subscription Overlap

```
1: function CalcSubSelectivity(dimCount, events[], subscriptions[])
2:                                  ▷ Array with all subscription selectivities of the dimensions
3:     subSels[dimCount]
4:     for iDim = 0 to dimCount do
5:                         ▷ Set of events matched by at least one subscription at this dimension
6:         Set <> evtsCovAny
7:         for all evt ∈ events do
8:             for all sub ∈ subscriptions do
9:                 if sub.matchsAtDim(evt, iDim) then
10:                    evtsCovAny.add(evt) break
11:                end if
12:            end for
13:        end for
14:
15:        subOverlapSum ← 0
16:        for all subs ∈ subscriptions do
17:                                ▷ Set of events matched by this subscription at this dimension
18:            Set <> evtsCovThis
19:            for all evt ∈ evtsCovAny do
20:                if sub.matchsAtDim(evt, iDim) then
21:                    evtsCovThis.add(evt) break
22:                end if
23:            end for
24:                 ▷ Set of events matched by this and other subscriptions at this dimension
25:            Set <> evtsCovOverl
26:            for all sub2 ∈ subscriptions \ sub do
27:                for all evt ∈ events do
28:                    if evtsCovThis.contains(evt) & sub2.matchsAtDim(evt, iDim) then
29:                        evtsCovOverl.add(evt)
30:                    end if
31:                end for
32:            end for
33:        ▷ Subscription Overlap - fraction of events matched not only by this subscription
34:            subOverlap ← evtsCovOverl.size/evtsCovThis.size
35:                          ▷ Subscription Selectivity is mean of inverse Subscription Overlaps
36:            subOverlapSum ← subOverlapSum + subOverlap
37:        end for
38:        subSels[iDim] ← 1.0 − (subOverlapSum/subscriptions.length)
39:    end for
40:    return subSels
41: end function
```

For determining the relevant subscription overlap the algorithm, the algorithm calculates for each dimension for each subscription the number of events that are only matched by one subscription on a dimension. The fraction of all events matched by a subscription, that are also matched by other subscriptions, is the overlap factor of this subscription (between 0.0 and 1.0). Inverting this value gives the subscription selectivity of this subscription. The inverse of the average of all subscription overlaps on one dimension results the *Subscription Selectivity* of this dimension.

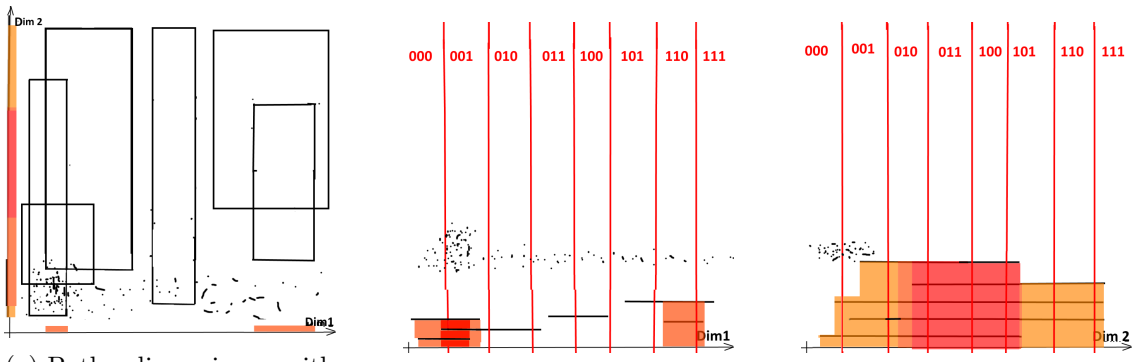
The major disadvantage of this algorithm is the time complexity. It is $O(n_d * n_e * n_s^2)$ when n_d is the number of dimensions, n_e the number of events and n_s the number of subscriptions. For every subscription and every event all other subscriptions have to be evaluated. This causes the quadratic dependency of the number of subscriptions.
One more disadvantage is the handling of overlap of many subscriptions. The number of overlapping can have an influence on the importance of a dimension and this algorithm can not analyze the number of overlaps. This algorithm can analyze the overlap of subscriptions but it does not analyze the number of overlapping subscriptions. If a subscription overlaps only with one other subscription the *Subscription Selectivity* would be the same as for many overlaps.
All in all this algorithm is a way to analyze events and subscriptions but there are problems with larger amounts of subscriptions.

Event Match Count-based Selection (SMS)

The next algorithm solves the two major problems of the *SOS* algorithm presented in Section 3.2.2 - the quadratic time complexity for subscriptions and the handling of multiple overlaps. We call the dimension selection based on this algorithm *subscription event match count-based selection (SMS)*
Figure 3.8 shows a two-dimensional scenario with multiple overlaps in both dimensions. Along all dimensions the majority of events is matched by more than one subscription. Therefore the last overlap calculation algorithm would calculate subscription selectivities close to zero for all dimensions.

To avoid this problem the proposed algorithm calculates the *Subscription Selectivity* based on the number of multiple subscriptions matching events along a dimension. The algorithm calculates for each dimension how many times events are matched by more than one subscription. Every time an event is matched by more than one subscription there is an overlap. Multiple matched subscriptions are taken into account by counting all matches instead of only one match like in the last algorithm.

The algorithm pseudocode of this algorithm is illustrated Algorithm 4. First all events matched by any subscription are collected. For all of these events the count of subscriptions match this event are counted. If the count is higher than one there is an overlap. In this case the number of overlaps, the number of matching subscriptions minus one, is added to the match sum.



(a) Both dimensions with subscriptions, events and overlap

(b) Spatial Indexing with Dimension 1

(c) Spatial Indexing with Dimension 2

Figure 3.8: Example for dimensions selectivity and overlap

Lastly the sum of matches is divided by the maximum number of possible overlaps to get a value between 0.0 and 1.0. The inverse of this is the final *Subscription Selectivity*.

Besides the fact that this algorithm is better in handling multiple overlaps its time complexity is better. It is $O(n_d * n_e * n_s)$ when n_d is the number of dimensions, n_e the number of events and n_s the number of subscriptions. The improvement over the last algorithm is, that the time complexity is linear dependent on the number of subscriptions instead of quadratic.

Algorithm 4 Calculate Event Match Counts

```
1: function CalcSubSelectivity(dimCount, events[], subscriptions[])
2:                                  ▷ Array with all subscription selectivities of the dimensions
3:     subSels[dimCount]
4:     for iDim = 0 to dimCount do
5:                         ▷ Set of events matched by at least one subscription at this dimension
6:         Set <> evtsMatchAny
7:         for all evt ∈ events do
8:             for all sub ∈ subscriptions do
9:                 if sub.matchsAtDim(evt, iDim) then
10:                    evtsMatchAny.add(evt) break
11:                end if
12:            end for
13:        end for
14:         ▷ Calculate how many times an event is matched by more than one subscription
15:        sumEvtOverl ← 0
16:        for all evt ∈ evtsMatchAny do
17:            evtMatch ← 0
18:            for all sub ∈ subscriptions do
19:                if sub.matchsAtDim(evt, iDim) then
20:                    evtMatch ← evtMatch + 1
21:                end if
22:            end for
23:            if evtMatch ≥ 1 then
24:                sumEvtOverl ← sumEvtMatch + (evtMatch − 1)
25:            end if
26:        end for
27:        maxOverl ← evtsCovAny.length ∗ (subscriptions.length − 1)
28:                ▷ Match Factor represents the fraction of events overlapping subscriptions
29:        matchFactor ← sumEvtOverl/maxOverl
30:                                    ▷ Subscription Selectivity is the inverse Match Factor
31:        subSels[iDim] ← 1.0 − matchFactor
32:    end for
33:
34:    return subSels
35: end function
```

False Event Match-based Selection (SFS)

This section shows an alternative metric for the *Subscription Selectivity*. Instead of analyzing overlaps of subscriptions matching events like the *SMS* selection, this algorithm counts how often an event is matched by a subscription on a dimension but not on all dimensions. We call a subscription match on a dimension but not on all dimensions a *False Match* and the selection based on this metric *subscription event false match-based selection (SFS)*.
The algorithm, illustrated in Algorithm 5, counts for every dimension how often events are matched by a subscription at this dimension. It also counts how often events matched by a subscription at this dimension are not matched by the subscription completely. An event is matched by a subscription completely when it is matched at all dimensions by it. An event subscription match at a dimension is a *False Match* if the event is not matched completely by the subscription. For every dimension the fraction of *False Matches* of the total count of matches is the *False Match Factor*. The inverse of the *False Match Factor* is the *Subscription Selectivity* of a dimension.

Like the algorithm presented in Section 3.2.2, this algorithm can deal with multiple subscription overlaps. The behavior of both subscriptions is very similar but in some scenarios this algorithm has better results. The time complexity of this algorithm is also $O(n_d * n_e * n_s)$ when n_d is the number of dimensions, n_e number of events and n_s number of subscriptions.

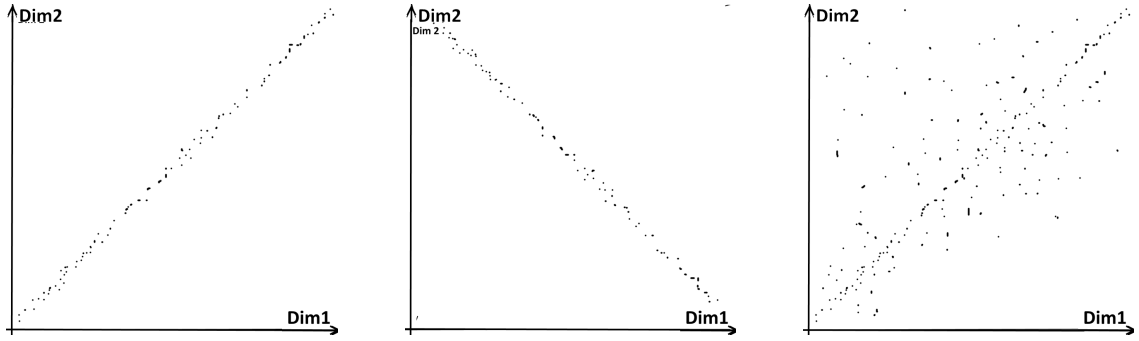
Algorithm 5 Calculate Event False Matches

```
1: function CalcSubSelectivity(dimCount, events[], subscriptions[])
2:                                  ▷ Array with all subscription selectities of the dimensions
3:     subSels[dimCount]
4:                         ▷ Calculate count of false subscription matches along this dimension
5:     matchCount ← 0
6:     falseMatchCount ← 0
7:     for iDim = 0 to dimCount do
8:         for all evt ∈ events do
9:             for all sub ∈ subscriptions do
10:                if sub.matchsAtDim(evt, iDim) then
11:                    matchCount ← matchCount + 1
12:                    if !sub.matches(evt) then
13:                        falseMatchCount ← falseMatchCount + 1
14:                    end if
15:                end if
16:            end for
17:        end for
18:          ▷ False Match Factor represents the fraction of false event subscription matches
19:        falseMatchFactor ← falseMatchCount/matchCount
20:              ▷ Subscription Selectivity is the inverse False Match Factor of this dimension
21:        subSels[iDim] ← 1.0 − falseMatchFactor
22:    end for
23:    return subSels
24: end function
```

3.3 Correlation-based Selection

The last sections introduced approaches to detect dimensions with high selectivity and ways to select dimensions based on the characteristics of events and subscriptions along dimensions. The metrics were calculated individually for every dimension.
A completely different factor for the importance of dimensions are correlations between dimensions. Correlation means that the behavior of dimensions is similar. A dimension can be represented by an other dimension if it is completely correlated with the other dimension, if its behavior is exactly the same.

Correlation of events between dimensions meas: When an event has a low/high value in dimension one it also has a low/high value in an other dimension. The value in one dimension can represent the value in the other dimension too. Figure 3.9(a) illustrates this case.
Another variant of correlation is the inverse correlation as shown in Figure 3.9(b). An event having a low/high value in Dimension 1 has a high/low value in Dimension 2. In realistic scenarios there is usually no pure correlation between dimensions. Figure 3.9(c) shows partial correlation of dimensions.



(a) Positive correlation of dimensions

(b) Negative correlation of dimensions

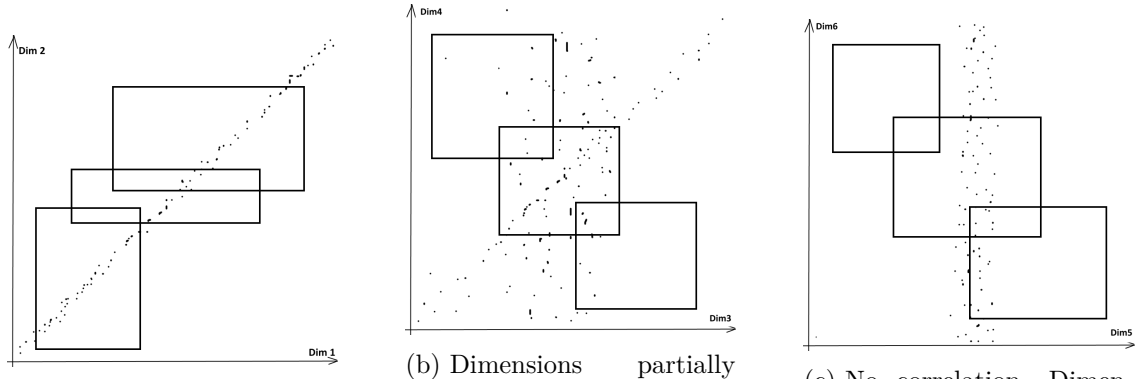
(c) Partial correlation of dimensions

Figure 3.9: Different scenarios of correlated dimensions

Correlation influences the importance of dimensions. A dimension that can be represented by an other dimension is redundant. Leaving out one of two correlated dimensions means no loss of information but more bits for more precise indexing for the other dimensions.
In case of completely correlated dimensions the importance of selecting both dimensions is zero, for partially correlated dimensions the importance is lower but not zero. Then, the dimension selection must be decided based on other factors like variance of events and overlap of subscriptions.
Dimension importance rating based on a combination of rating the individual importance of dimensions and correlation with other dimension will be more universal.

Figure 3.10 shows a scenario of six dimensions with different variances of events, subscription overlaps and correlations. Considering not all three factors will not lead to the best results.

The *dimensions 1 and 2* are highly correlated so only one of them should be selected. The event variance of both dimensions is similar. When looking at the subscription overlap, the overlap of *dimension 2* is lower so *dimension 2* is more important.
Dimension 3 and 4 are partially correlated but the event variance of Dimension 4 is higher. Therefore Dimension 4 should be selected out of the two correlated dimensions.
The *dimensions 5 and 6* are not correlated but the event variance of *dimension 5* is very low. Dimension 5 should not be selected.
All in all the dimensions 2, 4 and 6 should be selected for the best results. The challenge for the algorithm is not only to rate correlation, variance and overlap but also combining these metrics.



(a) Dimensions highly correlated, both high variance

(b) Dimensions partially correlated, Dimension 3 low variance, Dimension 4 high variance

(c) No correlation, Dimension 5 low variance, Dimension 6 high variance

Figure 3.10: Six dimensional scenario of different correlations and event variances

The following sections will present approaches to combine the algorithms for selecting dimensions based on their individual importance, presented in the last sections, together with algorithms detecting correlations between dimensions. Instead of only calculating for each dimension the selectivity, we combine the selectivities of dimension pairs together to measure the correlation between these dimensions. The combinations of selectivities between dimensions are put into a matrix, called *covariance matrix*, which can be used to detect dimension correlations.

The *covariance matrix* is a matrix of covariances measuring the relation of two random variables. In our case the random variables are the dimensions. The matrix is a $n_d * n_d$ matrix when n_d is the number of dimensions. The element at the position i,j is the covariance of the i^{th} and the j^{th} dimension.
Covariance matrices contain two types information: The relation between dimensions and the amount of variance in the dimensions. Both information are useful for our use case. The variance along the dimensions is an important factor for the selectivity of a dimension and the relation is important to find correlated dimensions.

In the following two subsections different ways to use covariance matrices for selecting di-

mensions will be presented.
Afterwards three different ways to calculate covariance matrices will be presented. The first way is based on the variance of events while the other two ways are no conventional covariance matrix
All three covariance calculation algorithms can be combined with both covariance based selection algorithms so there are six possible algorithm combinations to select dimensions.

3.3.1 Principal Component Analysis based Selection (PCS)

This algorithm selects the dimensions based on the *Covariance Matrix* and the *Principal component analysis (PCA)*. PCA is a mathematical method in the area of multivariate statistics to structure and simplify datasets by finding unimportant components that can be removed.

In [MG04] a scheme for selecting features with PCA is presented. The presented algorithm, illustrated in Algorithm 6 is based on this scheme. Features in our context means dimensions. The algorithm tries to find the set of principal features which is the set of principal dimensions.
PCA projects a set of vectors into an orthogonal space with minimal redundancies between the vectors. For this *Eigendecomposition* is used. In a first step the *Covariance Matrix* is calculated. Then the *Eigenvectors* ant their corresponding *Eigenvalues* are calculated. The principal components are *Eigenvectors* of the *Covariance Matrix*, sorted by their corresponding Eigenvalues. A higher Eigenvalue means higher importance of a component.
The approach from [MG04] then uses the most important component, the *Eigenvector* with the highest *Eigenvalues*, to find the most important dimensions. As the *Covariance Matrix* is a $n_d * n_d$ matrix when n_d is the number of dimensions, the *Eigenvector* has n_d elements. The i^{th} element of the component eigenvector represents the importance of the i^{th} dimension in this component.

For a given number of dimension the algorithm will return the dimensions with the highest values in the *Eigenvector* of the principal component. This way the algorithm tries to select a set of dimensions with high variances and low correlation.

Algorithm 6 Select Dimensions based on Principal Component Analysis

```
1: function SelectDimensions(origDims[], selDimCount, events[], subscriptions[])
2:     covMat[][] ← CalculateCovarianceMatrix(origDims.length, events, subscriptions)
3:                                  ▷ Calculate eigenvalues and eigenvectors of covariance matrix
4:     covEigenVecs[][] ← GetEigenVectors(covMat)
5:     covEigenVals[] ← GetEigenValues(covMat)
6:       ▷ Orthognal components are eigenvectors sorted by their corresponding eigenvalues
   (highest first)
7:     components[][] ← SortByEigenValues(covEigenVecs, covEigenVals)
8:                          ▷ Principal component is the eigenvector with the highest eigenvalue
9:     princComp[] ← components[0]
10:                              ▷ Select dimensions with highest value in principal component
11:    selDims[selDimCount]
12:    for iSel = 0 to selDimCount do
13:        dim ← GetNHighestValueIndex(iSel, princComp)        ▷ Index of n-highest
   event variance
14:        selDims[iSel] ← origDims[dim]
15:    end for
16:    return selDims
17: end function
```

3.3.2 Principal Feature Analysis based Selection (PFS)

An alternative algorithm to select features based on *PCA* is presented in [LCZT07]. Instead of using only one principle component multiple components are used. With clustering the algorithm tries to detect similar dimensions.

Like the *PCS* algorithm, this algorithm is based on *Principal Component Analysis*. Algorithm 7 shows the pseudocode of the algorithm.
First the principal components are calculated in the same way. Then instead of analyzing only the component with the highest eigenvalue, a set of most important components are used. The number of components to use is the same number as the number of dimensions to select. When selecting q dimensions from n_d dimensions, the most important components form the matrix A_q, a $q * n_d$ matrix with the most important components in the rows.
The column vectors of A_q then represent the values of the dimensions. Those column vectors are then clustered to q clusters using the *K-Means* algorithm as shown in Algorithm 8. From each of the q cluster the vector nearest to the clusters mean is chosen. This vector represents the cluster which is a group of potentially correlated dimensions.

For every chosen i^{th} vector the i^{th} dimension is selected. All q selected dimensions are returned as set of selected dimensions. By clustering the dimensions the algorithm is good in finding correlated dimensions but it can not analyze the variance of dimensions. This information is lost because the clusters are created by similarities of dimensions. The absolute values are not analyzed.

Algorithm 7 Select Dimensions based on Principal Feature Analysis

```
1:  function SelectDimensions(origDims[], selDimCount, events[], subscriptions[])
2:      covMat[][] ← CalculateCovarianceMatrix(origDims.length, events, subscriptions)
3:                                  ▷ Calculate eigenvalues and eigenvectors of covariance matrix
4:      covEigenVecs[][] ← GetEigenVectors(covMat)
5:      covEigenVals[] ← GetEigenValues(covMat)
6:        ▷ Orthognal components are eigenvectors sorted by their corresponding eigenvalues
    (highest first)
7:      components[][] ← SortByEigenValues(covEigenVecs, covEigenVals)
8:                  ▷ Get dominant components (number is selDimCount), forming Matrix A_q
9:      aq[][] ← SortByEigenValues(covEigenVecs, covEigenVals)
10:                                     ▷ Do clustering, output is indices of chosen features
11:     List <> selDimIndices ← DoClustering (origDims.length, selDimCount, domComps)
12:                                                 ▷ Get dimensions from dimension indices
13:     for iSel = 0 to selDimCount do
14:         dim ← selDimIndices[iSel]
15:         selDims[iSel] ← origDims[dim]
16:     end for
17:     return selDims
18: end function
```

Algorithm 8 KMeans Clustering to find the dimensions to select

```
1: function DoClustering(origDimsCount, selDimCount, components[][])
2:                                        ▷ Points to cluster are rows from the components
3:     points[origDimsCount]
4:     for iRow = 0 to origDimsCount do
5:         clusterPoint
6:         for iComp = 0 to components.length do
7:             clusterPoint.values[iComp] ←abs(components[iComp][iRow])
8:         end for
9:     end for
10:                                    ▷ Do KMeans clustering with selDimCount clusters
11:    clusters[] ← KMeansCluster(points, selDimCount)
12:    List <> selDimIndices
13:    for all clus ∈ clusters do
14:                                                ▷ Choose point next to cluster center
15:        clusPoint ← GetNearestPoint(points, clus.mean)
16:                                    ▷ Index of chosen cluster point is index of dimension
17:        clusPointIndex ← IndexOf(points, clusPoint)
18:        selDims.add(selDimIndices)
19:    end for
20: end function
```

3.3.3 Covariance Matrix from Events (CEV)

In the Section 3.1.1 an algorithm was presented that selects dimensions with the highest variance of events. This algorithm, calculating the covariance matrix for a *PCA* based selection, is a two-dimensional generalization of the variance calculation.

Input of this algorithm is an array with all event values. As shown in Algorithm 9, first the dataset is converted into an other dataformat, the *evtSamples* matrix a $n_e * n_d$ matrix where field i, j contains the i^{th} value of the j^{th} event.
For every column the mean is calculated and later used to calculate the variance. The covariance matrix is a quadratic $n_d * n_d$ matrix. Every field i, j is calculated from the variance of the column i of the *evtSamples* matrix multiplied with the variance of the column j, divided by $n_d - 1$. A Diagonal value of the matrix i, i is the variance of column i

The resulting covariance matrices can be used for the selection algorithms *PCS* or *PFS*. It contains information about variances of dimensions and similarities of event behaviors along dimensions.
The time complexity of this algorithm is $\mathcal{O}(n_d^2 * n_e)$ when n_d is the number of dimensions and n_e is the number of events. The complexity is quadratic dependent on the number of dimensions because the covariance is a $n_d * n_d$ matrix. For every field the variance of events along two dimensions has to be calculated.

Algorithm 9 Calculate Covariance Matrix from Event Variances (CEV)

```
1: function CalculateCovarianceMatrix(dimCount, events[])
2:                         ▷ Array of event value samples, events in rows, dimension in columns
3:     evtSamples[][] ←CalculateEventSamples(dimCount, events)
4:     sampleCount ← events.length
5:                                   ▷ Calculate Means of all samples on all dimensions
6:     means[dimCount]
7:     for iDim = 0 to dimCount do
8:         dimSum ← 0
9:         for iEvt = 0 to sampleCount do
10:            dimSum ← dimSum + evtSamples[iDim][iEvt]
11:        end for
12:        means[iDim] ← dimSum/events.length
13:    end for
14:                                                  ▷ Calculate covariance matrix
15:    covMat[dimCount][dimCount]
16:    for i = 0 to dimCount do
17:        for j = 0 to dimCount do
18:            covMat[i][j] ← CalculateCovariance(evtSamples[i], evtSamples[j], means[i],
19: means[j], sampleCount)
20:        end for
21:    end for
22:                                          ▷ Return calculated covariance matrix
23:    return (covSum/(n − 1))
24: end function
25:                                  ▷ Copy sample values from events into sample format
26: function CalculateEventSamples(dimCount, events[])
27:    evtSamples[dimCount][events]
28:    for iEvt = 0 to events.length do
29:        evt ← events[iEvt]
30:        for iDim = 0 to dimCount do
31:            evtSamples[iDim][iEvt] ← evt.values[iDim]
32:        end for
33:    end for
34:    return evtSamples
35: end function
36:
37:                                       ▷ Calculate covariance of two dimensions
38: function CalculateCovariance(samples1[], samples2[], mean1, mean2, n)
39:    covSum ← 0
40:    for i = 0 to samples1.length do
41:        v1 ← (samples1[i] − mean1)
42:        v2 ← (samples2[i] − mean2)
43:        covSum ← covSum + (v1 ∗ v2)
44:    end for
45:    return (covSum/(n − 1))
46: end function
```

3.3.4 Covariance Matrix from Event Match Counts (CMM)

As an improvement of the event variance based dimension selection, in Section 3.2 algorithms analyzing the selectivity of subscriptions were introduced. The advantage of these algorithms is, that they can also take into account the subscriptions and the relations between subscriptions and events.

The *Covariance Matrix from Event Match counts (CMM)* is a two-dimensional generalization of the *SMS* algorithm presented in 3.2.2. Instead of variances as the *CEV* calculation uses, this algorithm uses event subscription match counts, a metric for subscription overlap and selectivity. Algorithm 10 shows how the covariance matrix is calculated. A field i, j of the $n_d * n_d$ matrix is the sum of all events matched by subscriptions at the dimensions i and j. Every time an event is matched by a subscription at the dimensions i and j the *evtMatches* sum is incremented.

This algorithms time complexity is $\mathcal{O}(n_d^2 * n_e * n_s)$ when n_d is the number of dimensions and n_e is the event count and n_s the subscription count. For every field of the $n_d * n_d$ covariance matrix the matches of all events and subscriptions have to be counted.

Algorithm 10 Calculate Covariance Matrix based on Event Match Counts (CMM)

```
1:  function CalcCovarianceMatrix(dimCount, events[], subscriptions[])
2:      covMatrix[dimCount][dimCount]    ▷ Two-dimensional array for Covariance Matrix
3:      for i = 0 to dimCount do
4:          for j = 0 to dimCount do
5:    ▷ Calculate Fraction of all event subscription pairs that cover at the dimensions i and j
6:              evtMatchSum ← 0.0
7:              for all evt ∈ events do
8:                  evtMatches ← 0
9:            ▷ Calculate Fraction of all subscriptions that cover evt at the dimensions i and j
10:                 for all sub ∈ subscriptions do
11:                     if sub.matchsAtDim(evt, i) & sub.matchsAtDim(evt, j) then
12:                         evtMatches ← evtMatches + 1
13:                     end if
14:                 end for
15:                 evtMatchRate ← evtMatches/subscriptions.length
16:                 evtMatchSum ← evtMatchSum + evtMatchRate
17:             end for
18:             evtMatchAvg ← evtMatchSum/events.length
19:     ▷ Matrix value is inverse of the fraction of event subscription pairs covering at i and j
20:             covMatrix[i][j] ← 1.0 − evtMatchAvg
21:         end for
22:     end for
23:     return covMatrix
24: end function
```

3.3.5 Covariance Matrix from False Event Matches (CFM)

An alternative subscription selectivity metric was the *SFM* introduced in Section 3.2.1. Instead of only counting how often events are matched, it is counted how often events are matched at a dimension but not on all dimension.
The algorithm, shown in 11, counts the number of events matched by subscriptions and the false matches. For every field i, j of the $n_d * n_d$ covariance matrix the sum of all events matched by subscriptions at the dimensions i and j is counted. It is also is counted how many matches are *false matches*. This means that an event is matched by a subscription at the dimensions i and j but not on all dimensions. The fraction of events that are false matched out of the sum of all events is the *false match rate*. The inverse of the total false match rate of two dimensions i and j is the value of the covariance matrix.
This algorithm also has a time complexity of $\mathcal{O}(n_d^2 * n_e * n_s)$ when n_d is the number of dimensions and n_e is the event count and n_s the subscription count.

Algorithm 11 Calculate Covariance Matrix based on False Event Matches

```
1:  function CalcCovarianceMatrix(dimCount, events[], subscriptions[])
2:      covMatrix[dimCount][dimCount]      ▷ Two-dimensional array for Covariance Matrix
3:      for i = 0 to dimCount do
4:          for j = 0 to dimCount do
5:                          ▷ Calculate Fraction of false matched events at the dimensions i and j
6:              evtFMSum ← 0.0
7:              for all evt ∈ events do
8:                  evtMatches ← 0
9:                  evtFalseMatches ← 0
10:         ▷ Calculate Fraction of all subscriptions that cover evt at the dimensions i and j
11:                 for all sub ∈ subscriptions do
12:                     if sub.matchsAtDim(evt, i) & sub.matchsAtDim(evt, j) then
13:                         evtMatches ← evtMatches + 1
14:                         if Not sub.matchsAtDim(evt) then
15:                             evtFalseMatches ← evtFalseMatches + 1
16:                         end if
17:                     end if
18:                 end for
19:                 falseMatchRate ← evtFalseMatches/evtMatches
20:                 evtFMSum ← evtFMSum + falseMatchRate
21:             end for
22:             evtFMAvg ← evtFMSum/events.length
23:                                 ▷ Matrix value is inverse of the fraction of false matches
24:             covMatrix[i][j] ← 1.0 − evtFMAvg
25:         end for
26:     end for
27:     return covMatrix
28: end function
```

3.4 Evaluation-based Selection

Evaluation-based selection means that the algorithm directly finds the best scenario by trying out all possible sets of selected dimensions and choosing the best. Although the last algorithms have good results in many scenarios. all previously presented algorithms can't find the perfect set of dimensions in all possible kinds of scenarios.

The evaluation based algorithms try out all possible combinations of dimensions by simulating the event filtering and evaluating the false positive rate and choosing the combination with the lowest false positive rate.
On one hand those algorithms are very universal but on the other hand simulating the event filtering is very slow. Therefore the algorithms are much slower than the other algorithms based on mathematical models.

3.4.1 Brute Force Selection Algorithm (BES)

The *Brute force Evaluation-based dimension Selection (BES)* tries out all possible combinations of dimensions for a given number of dimensions and chooses the best combination. Algorithm 12 shows the basic structure of the algorithm.
Trying out means a complete simulation of the filtering of all events. With the set of dimensions to simulate the filtering with the dz-expressions for all events and subscriptions are generated. Then the filtering of the event dzs for the subscription dzs is performed. Events dedicated to wrong subscriptions are *false positives*. The rate of false positives is then used to evaluate the filtering quality that can be achieved with the given set of dimensions for the given scenario.
Finally the set of dimensions with the lowest *false positive rate* is returned.

This algorithm will always find out the best solution for the dimension selection problem, finding out the set of dimension with the best filtering effectiveness for the given dataset of subscriptions and events.
The disadvantage of this algorithm is, like for all brute force algorithms, that it is very slow, it has exponential time complexity. For every simulation run for n_e events and n_s subscription the complexity is $e * s$ because every event has to be filtered for every subscription. For every count of dimensions the algorithm must evaluate all possible combinations. This means for n_d dimensions the algorithm must evaluate for all n possible counts of dimensions all possible combinations.
Evaluating out all possible combinations has exponential runtime - for the dimension count m with $n_d > m > 0$ there are ${n_d}^m$ combinations. All in all the time complexity of the brute force approach is $\mathcal{O}(n_e * n_s * n_d * {n_d}^{n_d})$

Algorithm 12 Select Dimensions based on Brute Force Evaluations (BES)

```
1: List <> selectedDims                          ▷ Temporary storage for reduced dimensions
2: dimIndicesTmp[origDims.length]    ▷ Temporary storage of selected dimension indices
3: bestFpRate ← 1.0                    ▷ Temporary storage for best false positive value
4:
5: function SelectDimensions(origDims[], selDimCount, events[], subscriptions[])
6:     selectedDims ← origDims
7:                                                    ▷ Start brute force recursion
8:     BruteForceSelect(origDims, selDimCount, events, subscriptions,0)
9:                                                    ▷ Return selected dimensions
10:    return selectedDims
11: end function
12:
13:          ▷ Recursive function for brute force finding the best dimension combination
14: procedure BruteForceSelect(origDims[], selDimCount, events[], subscriptions[],
    depth)
15:    for iDim = 0 to origDims.length do
16:        dimIndicesTmp[depth] ← iDim;
17:        if depth < (selDimCount − 1) then
18:                                                ▷ Go deeper in recursion
19:            BruteForceSelect(origDims, selDimCount, events, subscription, depth + 1)
20:        else
21:                              ▷ Reached max recursion depth, evaluate selected dimensions
22:            List <> selectedDimsTmp
23:            for iSel = 0 to selDimCount do
24:                selectedDimsTmp.add(dimIndicesTmp[iSel])
25:                    ▷ Simulate and measure false positive rate with this set of dimensions
26:                simuFpRate ← Simulate (selectedDimsTmp, events, subscriptions)
27:                if simuFpRate < bestFpRate then
28:                    bestFpRate ← simuFpRate
29:                    selectedDims ← selectedDimsTmp
30:                end if
31:            end for
32:        end if
33:    end for
34: end procedure
```

3.4.2 Greedy Selection Algorithm (GES)

The *Greedy Evaluation-based dimension Selection (GES)* algorithm is also based on evaluations of dimension sets and very similar to the brute force strategy - it also evaluates different combinations of dimensions but instead of trying out every permutation the runtime is improved significantly by using a greedy strategy.
Instead of trying out all permutations the algorithm removes step by step one dimension. First, for n_d dimensions, the setting with all n_d dimensions is evaluated. Then all settings with a different dimension removed are evaluated. The setting with the best evaluation result is chosen and one dimension is removed finally.
This way incrementally all dimensions are removed step by step until only one dimension is left. All dimension counts are evaluated and the best count is chosen.
Evaluations comparing the GES algorithm with the brute force algorithm show that the results are nearly as good as the result of the *BES* approach.
The time complexity of the greedy algorithm is $\mathcal{O}(n_e * n_s * {n_d}^2)$. For all dimension counts m with $n_d > m > 0$ there are m combinations for removing the next dimension. Time complexity of the algorithm is quadratic depending on the number of dimensions but not exponential like as for the *BES* algorithm.

Algorithm 13 Select Dimensions based on Greedy Evaluations (GES)

```
1:  function SelectDimensions(origDims[], selDimCount, events[], subscriptions[])
2:      List <> selectedDims                          ▷ Temporary List for reducing dimensions
3:      selectedDims.addAll(origDims)
4:                                                    ▷ Reduce dimensions step by step greedily
5:      while selectedDimsims.size > selDimCount do
6:          bestDimIndex ← 0
7:          bestDimFp ← 1.0
8:                          ▷ Find Dimension whichs absence improves false positive rate most
9:          for iDim = 0 to selectedDims do
10:             List <> selectedDimsTmp ← copy(selectedDims)
11:             selectedDimsTmp.removeAt(iDim)
12:                     ▷ Simulate and measure false positive rate with this set of dimensions
13:             simuFpRate ← Simulate (selectedDimsTmp, events, subscriptions)
14:                     ▷ Remember this dimension if its absence improves false positive rate
15:             if simuFpRate < bestDimFp then
16:                 bestDimIndex ← iDim
17:                 bestDimFp ← simuFpRate
18:             end if
19:         end for
20:                     ▷ Remove Dimension whichs absence improves false positive rate most
21:         selectedDimsTmp.removeAt(bestDimIndex)
22:     end while
23:     return selectedDims
24: end function
```

3.5 Finding best Dimensions Count

All algorithms presented up to now can only find the best dimensions for a fixed number of dimensions. However the best number of dimensions can vary, depending on the number of IP bits and the selectivity of the individual dimensions. In a scenario with many dimensions with high selectivity the best number of dimensions is higher than in a scenario with less dimensions with high selectivity. The number and distribution of events and subscriptions can also influence the best number of dimensions. This section will show two different ways to find out the best number of dimensions based on the presented algorithms. One way is based on the *Principal Components Analysis (PCA)* and another way is a modified version of the greedy *GES* algorithm.

3.5.1 PCA-based Dimension Count

This approach is based on the principal component analysis which transforms an n-dimensional space, specified by the covariance matrix, into an orthogonal space of eigenvectors. The principal components are eigenvectors of the covariance matrix, sorted by their corresponding eigenvalues.
The basic idea of this approach is that the most relevant components can represent the system with very accurately. The eigenvalue of a component represents, broadly speaking, the amount of importance of a dimension. In scenarios with more selective dimensions the number of important components will be higher. The higher the number of important components with higher eigenvalues, the more components are needed so that the sum of their eigenvalues is above a certain threshold. To determine the best number of dimensions it is calculated how much of the most important components together have a sum of eigenvalues which have a ratio of the sum of all eigenvalues above a specified threshold.
Algorithm 14 shows the algorithm to determine the number of dimensions. Input for the algorithm is are the eigenvalues of the principal components in descending order and the threshold, a value between 0.0 and 1.0. First the sum of all eigenvalues is calculated. In a second step the minimum number of most important components with the necessary a sum of eigenvalues is returned.

3.5.2 Evaluation-based Dimension Count

The greedy algorithm *GES* can easily modified to find the best dimension count. For that it is useful that this algorithm reduces the number of dimension step by step.
In Algorithm 15 the algorithm is illustrated. As the original *GES* algorithm for every count every dimension is removed and the set of remaining dimensions is evaluated. The difference is that the algorithm remembers the count and set of dimensions with the best *false positive rate*. After all counts of dimensions were evaluated the best set of dimensions of all possible dimension counts is returned.
The amortized time complexity of this algorithm is also $\mathcal{O}(n_e * n_s * {n_d}^2)$, like for the unmodified greedy algorithm.

Algorithm 14 Algorithm to determine dimension count based on PCA

```
1: function DetermineDimCount(sortedEigenVals[], threshold)
2:                                                  ▷ Calculate sum of eigenvalues
3:     eigenValSum ← 0
4:     for i = 0 to sortedEigenVals.length do
5:         eigenValSum ← eigenValSum + sortedEigenVals[i]
6:     end for
7:     eigenValThr ← eigenValSum * threshold
8:     eigenValSumTmp ← 0
9:                                  ▷ Determine components to get over threshold
10:    for i = 0 to sortedEigenVals.length do
11:        eigenValSumTmp ← eigenValSum + sortedEigenVals[i]
12:        if eigenValSumTmp ≥ eigenValThr then
13:                                  ▷ Return number of components return i + 1
14:        end if
15:    end for
16: end function
```

Algorithm 15 Select Dimensions based on Greedy Evaluations

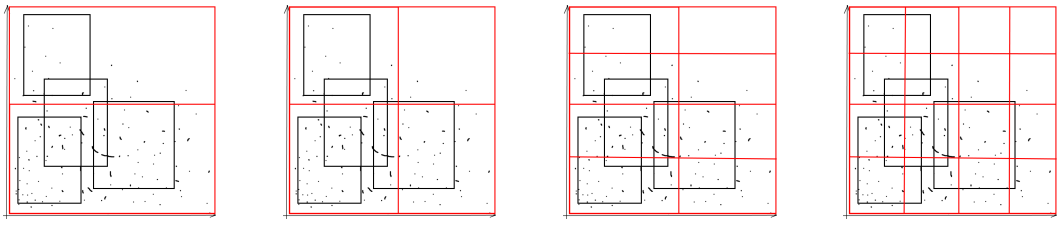
```
1: function SelectDimensions(origDims[], events[], subscriptions[])
2:     List <> selectedDims                          ▷ Temporary List for reducing dimensions
3:     selectedDims.addAll(origDims)
4:                                  ▷ Variable for false positive rate for all dimension counts
5:     bestFpTotal ← Simulate (selectedDims, events, subscriptions)
6:                                                  ▷ Reduce dimensions step by step greedily
7:     while selectedDims.size > 1 do
8:         bestDimIndex ← 0
9:         bestDimFp ← bestFpTotal
10:                    ▷ Find Dimension whichs absence improves false positive rate most
11:        for iDim = 0 to selectedDims do
12:            List <> selectedDimsTmp ← copy(selectedDims)
13:            selectedDimsTmp.removeAt(iDim)
14:                ▷ Simulate and measure false positive rate with this set of dimensions
15:            simuFpRate ← Simulate (selectedDimsTmp, events, subscriptions)
16:                ▷ Remember this dimension if its absence improves false positive rate
17:            if simuFpRate < bestDimFp then
18:                bestDimIndex ← iDim
19:                bestDimFp ← simuFpRate
20:            end if
21:        end for
22:                                ▷ Only remove dimension if this improves false positive rate
23:        if bestDimFp < bestFpTotal then
24:                ▷ Remove dimension whichs absence improves false positive rate most
25:            selectedDimsTmp.removeAt(bestDimIndex)
26:            bestFpTotal ← bestDimFp
27:        end if
28:    end while
29:                                                ▷ Return List of reduced dimensions
30:    return selectedDims
31: end function
```

Chapter 4

Event Space Partitioning

The last chapter explained how the results of spatial indexing can be improved with better input data, by selecting the most relevant dimension. This chapter will show a way to improve the spatial indexing itself.
As explained in section 2.1.2, the spatial index is created by partitioning the event space Ω step by step in the middle. This works well as long as the events are distributed equally over the event space and the subspace partitions are balanced regarding the number of events in each partition.

However if the events are not equally distributed, the events are not balanced across the partitions. Figure 4.1 shows such a scenario. Recall that event filtering is done by checking if an event is in the same partition as a subscription. If a subscription covers/intersects the same partition as an event, the event is forwarded to the subscriber.
In the illustrated scenario, even with a 4-bit index it is not possible to do accurate event filtering. While the partitions in the upper right are mostly useless for filtering, as there are only few events that can be filtered with these partitions, the partitions in the lower left are overloaded. In the lower right there are many events too but not many subscriptions to filter the events for. As a result of the poor filtering effectiveness the *false positive rate* would be high which leads to a higher network traffic.



(a) Normal Partition 1-bit dz (b) Normal Partition 2-bit dz (c) Normal Partition 3-bit dz (d) Normal Partition 4-bit dz

Figure 4.1: Normal Partitions

To solve this problem, to improve the indexing results, it is desirable to improve the partitioning so that the partitions are more balanced, so that there is more filtering granularity where it is needed. This chapter will present an approach for spatial indexing based on balanced partitioning taking into accounts events and subscriptions.

4.1 Improved Partitioning

In [CS04] an approach to improve the *content space partition* of the presented system *Kyra* is proposed. This approach was simplified and mapped to be used in a system like *PLEROMA*.

The algorithm from *Kyra* uses a simple methodology: Partitioning into non-overlapping balanced zones. Instead of always partitioning the event space in the middle, as presented in Section 2.1.2, partitioning is always done at the position of the workload median.
For determining the workload median a metric to calculate the workload of events is needed. The workload of an event subspace is calculated as the sum of all numbers of matching subscriptions of all events with the following formula:
$workload_{subspace} = \Sigma(matches_e)$

Figure 4.2 shows an example for improved partitioning. Instead of partitioning always in the midpoint every partition is done at the median of event loads which depends on the number of events and subscriptions matching events. This case demonstrates how the indexing can be improved this way. In the important lower left of the diagram the subspaces are much more granular than in the upper right. This way the in-network filtering would be much more efficient which reduces the *false positive rate (FPR)*.

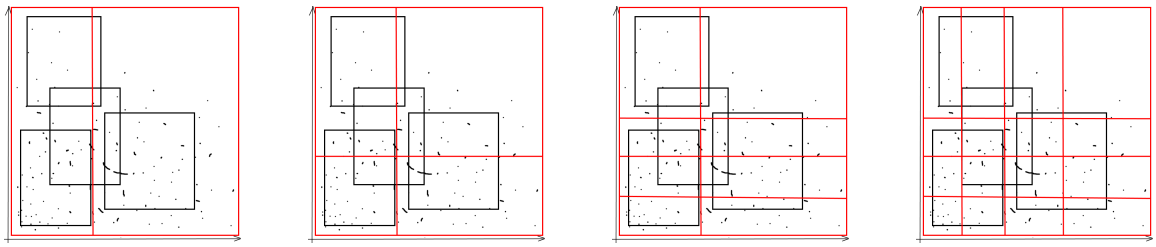


Figure 4.2: Partitions event and subscriber based

Algorithm 16 shows a simplified, unoptimized implementation of the event load median calculation. In a system like *PLEROMA* the calculation of the partition positions based on the medians can be done at the controller.
A list of all partition positions can then used to generate the *dz-expressions*. This list can be distributed from the controller to all hosts. All *dz* for forwarding *flow rules* and for all event *packet headers* must be generated consistently using the same partition list.

Algorithm 16 Unoptimized algorithm to find median of event workloads between given borders

```
1: function FindEventMedian(origDims[], events[], subscriptions[], dimIndex, low, up)
2:     List <> sortedEvtLoads                ▷ List with event workloads, sorted by event value
3:     sumLoads ← 0
4:                                                          ▷ Calculate event workloads
5:     for all evt ∈ events do
6:                         ▷ Add event load (number of matching subs) together with event value
7:         if evt.values[dimIndex] ≥ low & evt.values[dimIndex] ≤ up then
8:             subMatches ← 0
9:             for all sub ∈ subscriptions do
10:                if sub.matchs(evt) then
11:                    subMatches ← subMatches + 1
12:                    sumLoads ← subMatches + 1
13:                end if
14:            end for
15:                        ▷ Add event load (number of matching subs) together with event value
16:            sortedEvtLoads.add(newEvtLoad(subMatches, evt.values[dimIndex]))
17:        end if
18:    end for
19:                                                  ▷ Sort event loads by event value
20:    sortedEvtLoads.sortByEvtValues()
21:                                  ▷ Sum up event loads until workload median is reached
22:    medianPos ← low
23:    sumLoadsTmp ← 0
24:    for all evtLoad ∈ events do
25:        sumLoadsTmp ← sumLoadsTmp + evtLoad.Load
26:        if sumLoadsTmp > (sumLoads/2) then
27:            medianPos ← evtLoad.EvtValue
28:            break
29:        end if
30:    end for
31:    return medianPos
32: end function
```

4.2 Combining with Dimension Selection

Both, dimension selection and improved partitioning can improve spatial indexing and reduce the *FPR*. It would be desirable to combine them both with an even lower *FPR* as a result. Indeed both techniques can be combined very good and evaluations show that a combination of both reduces the *FPR* more than the single techniques.

In chapter 3 two main algorithm types were presented: Selection algorithms-based on mathematical models and evaluation-based selection. The combination with the improved partitioning is slightly different for both types of algorithms.

When combining the improved indexing with an algorithm-based on a mathematical model such as *correlation-based selection*, both approaches work completely independent. In a first step the dimension selection algorithm reduces the number of dimensions. Then in a second step the partitions for the reduced dimension subset is generated. All *dz-expressions* in the system are then generated with the reduced set of dimensions and the pre-calculated partitions.

The combination with evaluation-based dimension selection algorithms allows optionally to use the improved partitioning during the dimension selection. As the evaluation-based algorithms simulate the in-network filtering for every combination of reduced dimensions, the improved partitioning can be used in every evaluation step.
In this case the algorithm can be find the very best dimension set taking into account the improved partitioning. However the disadvantage of this combination is, that every evaluation step becomes slower because for every evaluation step the partitioning has to be calculated. Alternatively the evaluation-based algorithm can be used independently from the improved partitioning and combined in the same way as described for the mathematical model-based algorithms.

All in all both approaches can be combined easily with good results. They can even us collected data together. Both use active subscriptions and events collected in a certain timeframe and as both modify the *dz* generation, both must update the whole system, which includes redeployment of all flows.
When both algorithms work synchronously the can use the same data and update the system together at the same time.

Chapter 5

Evaluation of Algorithms

In this chapter the evaluations of all proposed algorithms will be presented. Recall the goal of this thesis is to improve *spatial indexing* for publish/subscribe systems like *PLEROMA* to reduce the *false positive rate (FPR)* and therewith the network traffic.
The algorithms will be evaluated how good they can reduce the FPR. Furthermore the performance of the algorithms will be evaluated because there is always a tradeoff between FPR reduction and run time of the algorithms.

All evaluations are performed in a test application simulating the filtering of pub/sub system like *PLEROMA*. This application will be described in the first section of this chapter. In the second section the results of the evaluations will be presented.

5.1 Experimental Setup

The evaluation application consists of four main components: The *data generator* component, the *dimension selector* component, the *pub/sub simulator* component and the *evaluator* component.

In a first step, the *data generator* generates subscriptions and events randomly. For simulating the *in-network filtering*, publishers are not necessary, the simulation only simulates how events are filtered to subscribers. All data is randomly generated, a fixed initial seed for better comparability of results can be specified.
Events are always generated inside subscriptions. The reason for this is on the one hand, that only events that are matched by at least one subscription are relevant for the simulation, on the other hand, that in cases with many dimensions, the chance that an event is inside a subscription is very low. Events are randomly generated by picking a random location inside a random subscription.
Subscriptions can be generated with two different basic patterns: *Uniform* distribution and *zipfian* distribution. When using uniform distribution, the events are generated uniformly over the event space, when using zipfian distribution they are generated in so-called buckets. The behavior of dimensions can be configured in various ways. Depending on the type of experiments, the variance of subscriptions and events, the selectivity of subscriptions and the correlation between dimensions can be configured. Average size and distribution of subscriptions is also configurable.

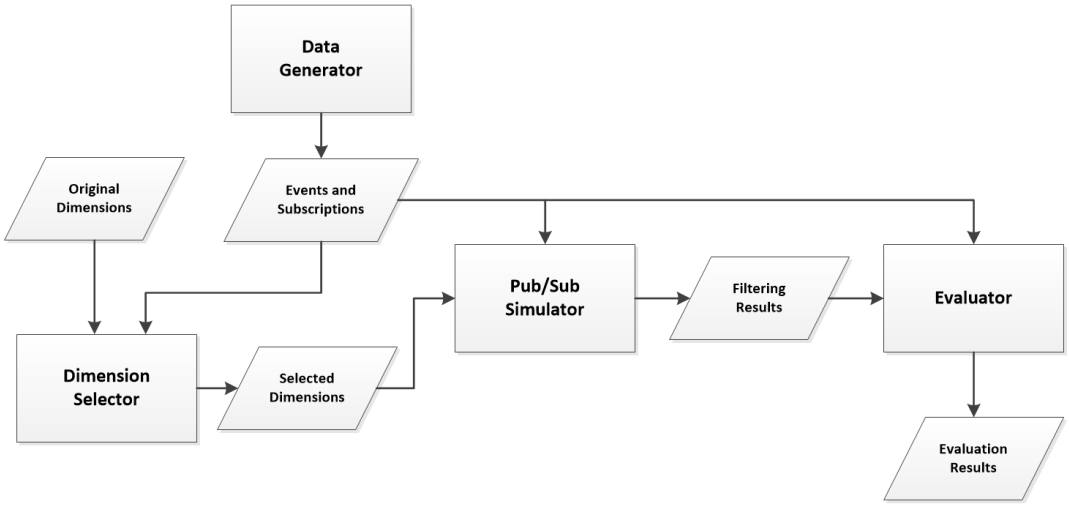


Figure 5.1: Architecture of Simulation Application

All in all it is possible to generate various scenarios for simulating different scenarios showing the behavior of the algorithms under different circumstances.

Based on the events and subscriptions, the *dimension selector* selects a subset of dimensions. As selection algorithm any algorithm presented in Chapter 3 are available. For different experiments the desired algorithm can be configured.

The *pub/sub simulator* simulates the in-network filtering based on the selected dimensions and the generated events and subscriptions. For every event and subscription, the *dz-expressions* are generated. As described in Chapter 2, in-network filtering is performed by looking for every event, for every subscription, if one of the subscription *dz* matches the event *dz*. A *dz-expression* dz_s matches dz_e if dz_s is equal to dz_e or a prefix of dz_e.
For all events the in-network filtering is performed, for every event is filtered to all subscriptions. The result of the *pub/sub simulator* is the filtering result, for every event to which subscription they were forwarded by the filter, based on the *dz-expressions*.
If selected dimensions allow better *spatial indexing*, the simulated event filtering is more accurate.

In a final step, the filtering result is evaluated. Therefore, the *evaluator* checks for all events every subscription the filter would forward them to. Besides the filtering input, the second input data for the evaluator are the events and subscriptions. By testing if all event values are inside the bounds of a subscription it can be checked if a subscription is really interested in an event.
Every event that is filtered to a subscription correctly is a *true positive*. Events that are wrongly filtered to a subscription, that is not interested in this event, is a *false positives*. Events that are not filtered to a subscription that would be interested in the event, so-called *false negatives*, are impossible by design.

The evaluation result we are interested in most is the *false positive rate*. It is calculated as the rate of filtered events that are falsely filtered:

$$fprate = \frac{|falsepos|}{|falsepos|+|truepos|}$$

For the following evaluation measurements, for every evaluation run, the *dimension selector* will reduce the number of dimensions step by step. Every dimension count will be evaluated. For a evaluation with n dimensions all numbers of dimensions from n,$n-1$...1 will be evaluated, including n dimensions which means the unreduced data. This way the effectiveness of the algorithms and the best number of dimensions to select can be found out.

The evaluations are done with different numbers of subscribers. To show a wide range of scenarios, most of the scenarios are evaluated with 200, 400, 1000, 2000, 4000 and 10,000 subscribers. The number of events is 10,000 for all experiments.
The event space range, the range of event values and subscription bounds, is from 0 to 10,000 for every dimension. For all experiments the maximum length of *dz-expressions* is 24. This is aligned to the number of available bits when using IPv4-multicast ranges and IP fields in the packet header to filter events.

5.2 Evaluation Measurements

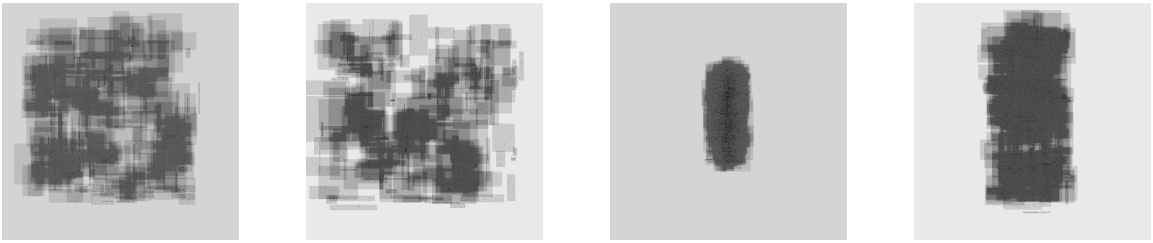
This section will present the evaluation results of all presented algorithms. Every type of algorithm is evaluated in a separate subsection.
For the algorithms different scenarios will be evaluated to show the strengths and weaknesses of the algorithm. The description of a evaluation consists of the description of the scenario/data generation and the evaluation of the false positive rate and other metrics.
The data generation will be illustrated with graphics showing the distribution of subscriptions and events over the event space. Every diagram shows the distributions over two dimensions. In these the subscriptions are plotted as rectangles and events plotted as green.
False positive rates will be shown in diagrams showing how the FPR increases or decreases with the number of selected dimensions.

5.2.1 Event-based Selection

The first introduced algorithm was the *Event Variance-based dimension Selection (EVS)*. EVS selects dimensions with a higher variance of events and removes dimensions with lower variance.

As a first evaluation, to show the effect of EVS, a scenario of eight dimensions with uniformly distributed subscriptions and random dimensions selectivity is chosen. The selectivity of dimensions is modified by changing the distribution of subscriptions. A dimension with widely distributed subscriptions is more selective as a dimension where all events have the same value. Events are randomly generated inside the subscriptions. Figure 5.2 shows this

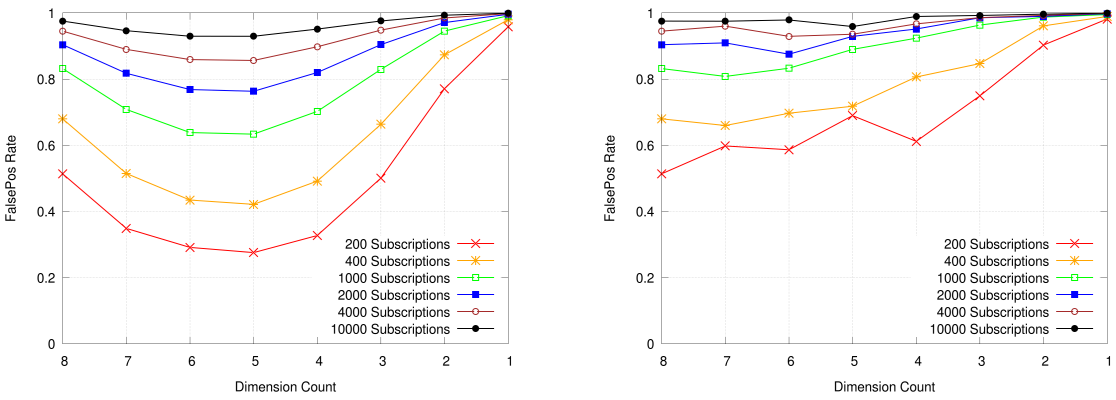
type of scenario for 200 subscriptions.
Every graphic shows two dimensions, all four graphics together show all dimensions. The distribution of subscriptions and therewith the selectivity is random. Dimensions 1, 2, 3, 4 and 8 have a higher selectivity while the dimensions 5, 6 and 7 have a lower selectivity.



(a) Dimensions 1, 2 (b) Dimensions 3, 4 (c) Dimensions 5, 6 (d) Dimensions 7, 8

Figure 5.2: Distribution of 200 subscriptions: Uniform distribution, random selectivity

This scenario now is evaluated. The goal of the algorithm is to find the more selective dimensions. Figure 5.3 shows the *false positive rate (FPR)* resulting from the dimension reduction. On the x-axis the dimension count is plotted. 8 means no dimension reduction, the other counts down to 1 are for the dimensions selected by the algorithm.
For comparison and to show that the algorithm is really selecting the right dimensions and not random dimensions, the figure also shows the evaluation of an algorithm selecting random dimensions.
As you can see the EVS algorithm can reduce the FPR significantly while the random algorithm does not. For higher subscription numbers the FPR is in general higher but the algorithm can always reduce the FPR. All in all, in this scenario the EVS algorithm is an improvement for the event filtering.



(a) EVS (b) Random dimension selection

Figure 5.3: False positive rate for: Uniform distribution, random selectivity

The next scenario uses zipfian distribution of subscriptions instead of uniform distribution. Subscriptions are generated in so-called buckets to generate the hotspot like behavior. Every bucket is an area where the subscriptions are generated in. For our scenario we use five buckets.
Figure 5.4 shows the zipfian distributed subscriptions with random selectivity of dimensions. This distribution is an attempt to generate more realistic data than uniformly distributed subscriptions.

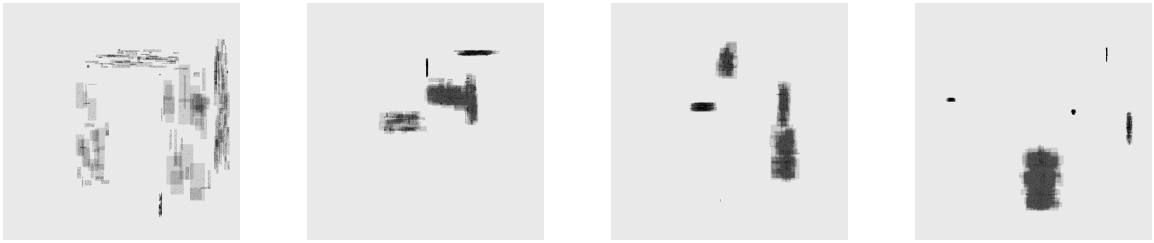
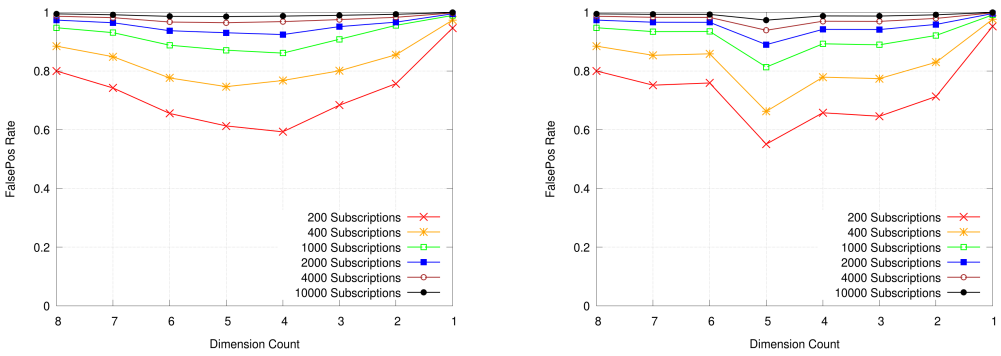


Figure 5.4: Distribution of 200 subscriptions: Zipfian distribution, random selectivity

The selectivity for zipfian distribution is modified by changing the distribution inside the buckets. In a dimension with lower selectivity the subscriptions are generated more close to the bucket center, for high selectivity they are distributed all over the buckets.
Figure 5.5 shows that the EVS algorithm is not able to deal with this type of selectivity. As it measures the variance of events over the whole dimension its metric can not detect the subscription distribution inside the buckets.
The selection accuracy of EVS in this scenario is very similar to the accuracy of the random selection algorithm. For this scenario the dimension selection can reduce the FPR but not as good as for uniform distribution.



(a) EVS (b) Random dimension selection

Figure 5.5: False positive rate for: Zipfian distribution, random selectivity

The last two scenarios showed how EVS can reduce dimensions for scenarios with varying dimension selectivity, with uniform and zipfian distribution. Both scenarios are extreme cases because they include dimensions with very high and very low variance.
An other extreme case are scenarios with no varying selectivity of dimensions. In the following

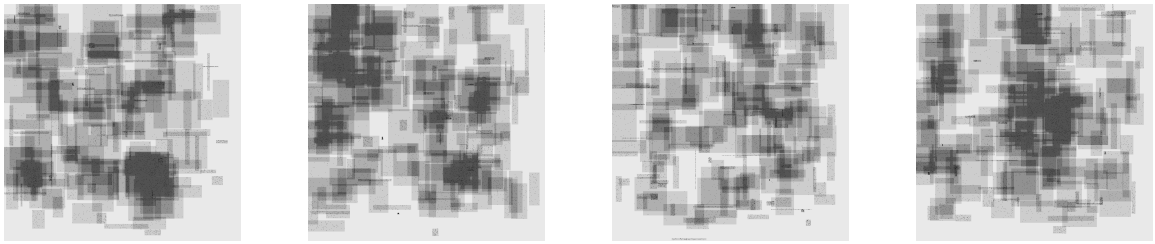


Figure 5.6: Distribution of 200 subscriptions: Uniform distribution, uniform selectivity

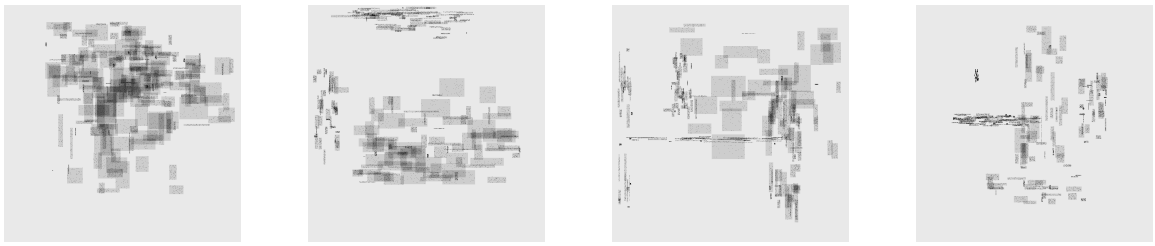
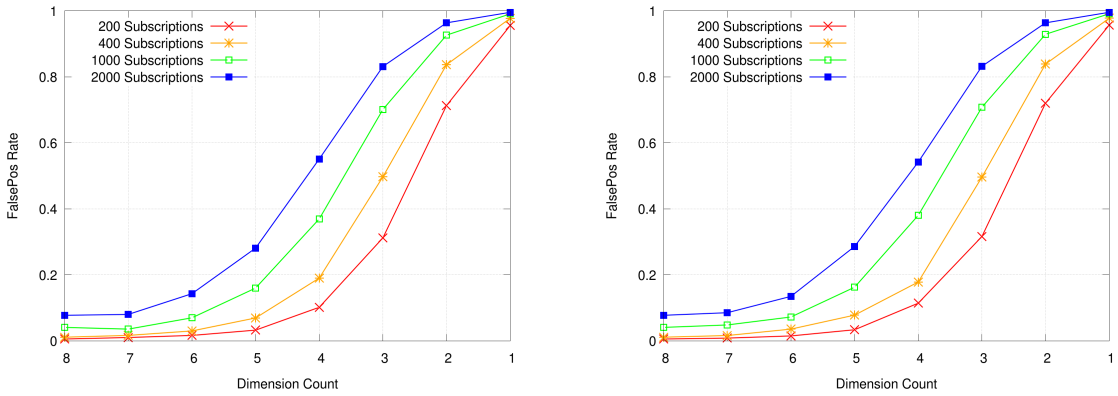


Figure 5.7: Distribution of 200 subscriptions: Zipfian distribution, uniform selectivity

two scenarios with no varying selectivity are evaluated, with uniform and zipfian distribution. The Figures 5.6 and 5.7 show these scenarios. In both the subscriptions are distributed over the full range, there is no difference in selectivity between the dimensions.

As the Figures 5.8 and 5.9 show, in these scenarios the results of EVS are very similar to the results of the random selection.
The reason for this is obvious. As there is no measurable difference in variance between the dimensions, EVS can not decide which dimensions to select.

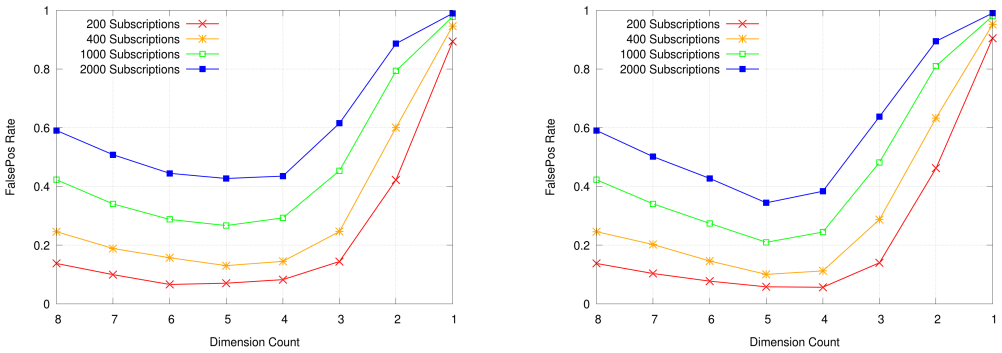


(a) EVS

(b) Random dimension selection

Figure 5.8: False positive rate for: Uniform distribution, uniform selectivity

All in all the EVS algorithm can reduce the *false positive rate* by selecting dimensions but there are scenarios where it has problems to detect the right dimensions to select.



(a) EVS (b) Random dimension selection

Figure 5.9: False positive rate for: Zipfian distribution, uniform selectivity

An advantage of EVS is its run time. For processing 1000 subscriptions and 20,000 events for eight dimensions it needs < 0.01 seconds to select the dimensions and its time complexity depends linearly on the number of subscriptions and events.

5.2.2 Event and Subscription-based Selection

To avoid the problems of EVS, in Section 3.2, algorithms based on the selectivity of subscriptions were introduced.
Three algorithms were presented, the *subscription overlap-based selection (SOS)*, the *subscription event match count-based Selection (SMS)* and the *subscription false event match based selection (SFS)*.
Performance measurements showed that SOS is much more resource intensive than the other algorithms. Its time complexity depends quadratically on the number of subscriptions. For processing 1000 subscriptions and 20,000 events for eight dimensions SOS needs $\sim$ 800 seconds. SMS and SFS have run time depending linearly on the number of subscriptions and events, SMS needs $\sim$ 1.5 seconds and SFM $\sim$ 2.5 seconds to process the same amount of data.

The following scenario shows a weakness of EVS: When the subscription selectivity is independent from the event variance, it is not able to detect the dimensions to select with its metric. Figure 5.10 shows this scenarios. The dimensions selectivity of subscriptions is random but the event variance is more or less constant.

Figure 5.11 shows a comparison of the dimension selection results for EVS, SOS and SMS. EVS fails in this scenario. It is not able to detect the dimensions that should be selected.
SOS is an improvement over EVS but it is also not able to deal with this scenario very good and due to the time complexity it was not possible to evaluate SOS with higher number of subscriptions than 400.
SMS is definitively much better in selecting the best dimensions in this scenario. It can reduce the number of false positives significantly.

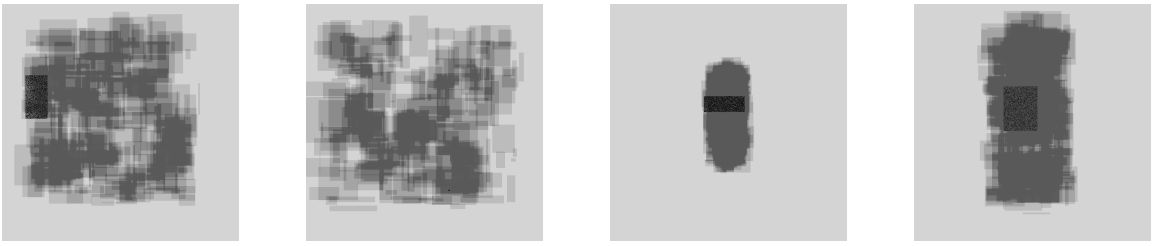
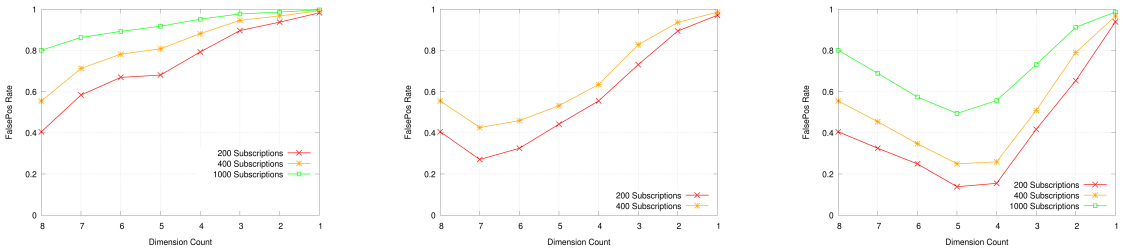


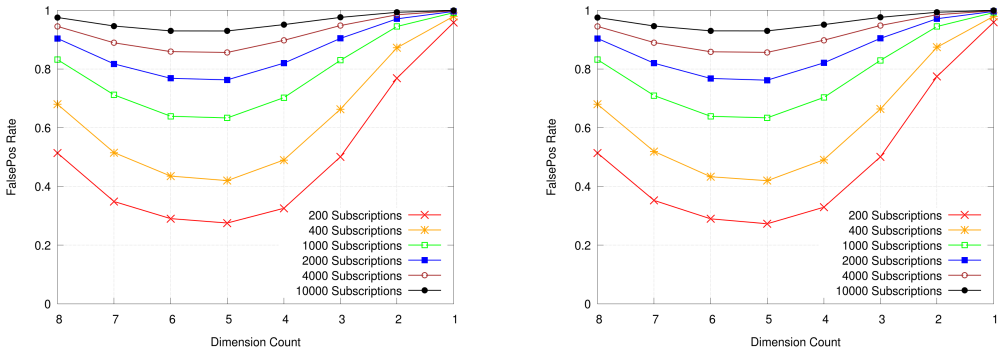
Figure 5.10: Distribution of 200 subscriptions, constant event variance



(a) EVS false positive rate (b) SOS false positive rate (c) SMS false positive rate

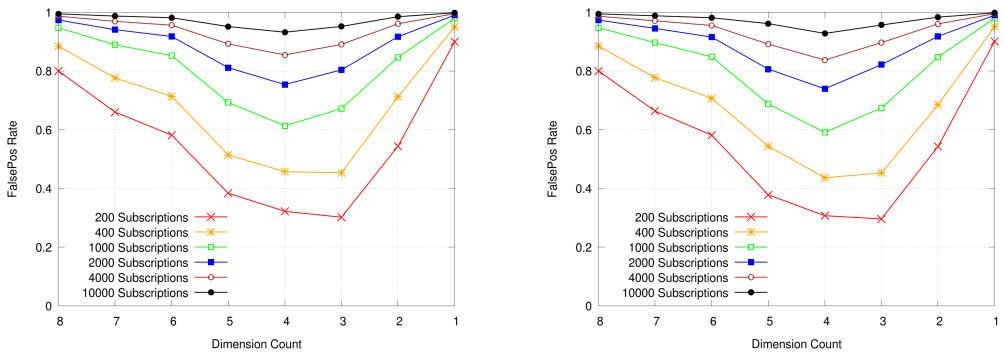
Figure 5.11: Comparison of EVS, SOS and SMS for fixed event variance

As the results of SOS are worse than of SMS and it is very slow we will not continue evaluating it.
The Figures 5.12 and 5.13 show the results of the dimension selections of the algorithms SMS and SFS. The scenarios are equal to the scenarios used in the last section - uniform and zipfian distribution with random dimension selectivity.
Both algorithms have nearly the same results, the *false positive rate* with SFS is a little lower.



(a) False positive rate of SMS (b) False positive rate of SFS

Figure 5.12: False positive rate of SMS and SFS for: Uniform distribution, random selectivity



(a) False positive rate of SMS

(b) False positive rate of SFS

Figure 5.13: False positive rate of SMS and SFS for: Zipfian distribution, random selectivity

All in all the algorithms SMS and SFS have very similar and good results. SMS is a little faster while SFS is a slightly more accurate. Both algorithms can deal with a wide range of scenarios and achieve a good reduction of the *false positive rate* by selecting dimensions.

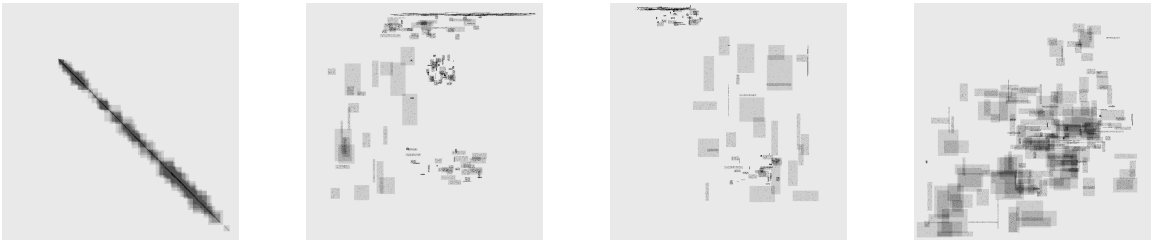
5.2.3 Correlation-based Selection

Besides selecting dimensions with higher individual selectivity there is one more factor to consider: Correlations between dimensions. Correlation of dimensions a and b means that subscriptions and events with high/low values in dimension a will also have high/low values on dimension b. When two or more dimensions are correlated it is enough to select only one of these because the behavior of one dimension can reflect the behavior of all correlated dimensions.
Therefore in Section 3.3 algorithms were introduced to detect correlated dimensions. Two different algorithms to select *features*, in our context dimensions, were introduced: *Selection-based on Principal Component Analysis (PCS)* and *Selection based on Principal Feature Analysis (PFS)*.
Both algorithms do their calculations based on a *covariance matrix*. To calculate the covariance three algorithms were introduced: *Covariance matrix from events (CEV)*, *Covariance matrix from event match counts (CMM)* and *Covariance matrix from false matches (CFM)*. All three algorithms are based on previously presented algorithms. CEV is based on the event based EVS algorithm and CMM and CFM are based on the subscription selectivity metrics SMS and SFS.

The time complexity of *PCS* and *PFS* itself is constant for a constant number of dimensions and can be ignored. Mainly the algorithm run time is caused by the calculation of the covariance matrix calculation. Like EVS on which is based, CEV is very fast, for processing 1000 subscriptions and 20,000 events for eight dimensions it needs only < 0.1 and scales linear with the number of subscriptions, events and dimensions. CMM needs for the same input data ~ 7.5 seconds and CFM needs ~ 9.5 seconds. Both are scaling linearly too.

In this section we evaluate how good the algorithms can select dimensions in scenarios with correlated dimensions. The results of the algorithms are compared to algorithms that do not analyze the correlations of dimensions.

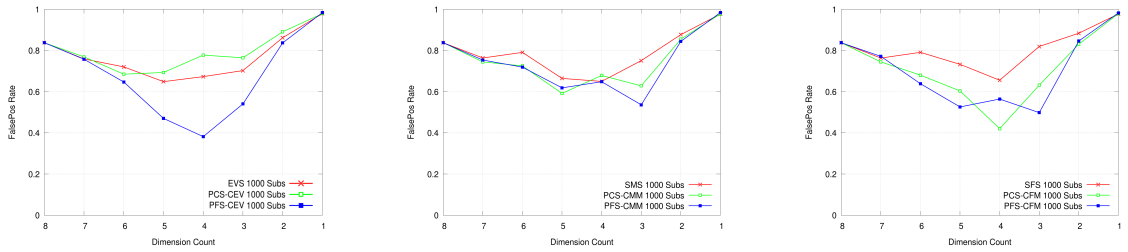
First a scenario with dimensions with pairwise 90 percent correlation between each others is evaluated. This means that the values of four pairs of random dimensions are in average 90 percent equal. The distribution is zipfian based and the selectivity of dimensions is constant. Figure 5.14 shows such a scenario for 200 subscriptions. In the figure is shown that fore example the dimensions 1 and 2 are one of four correlation pairs.



(a) Dimensions 1, 2 (b) Dimensions 3, 4 (c) Dimensions 5, 6 (d) Dimensions 7, 8

Figure 5.14: Distribution of 200 subscriptions, zipfian distribution, constant selectivity, correlation 90%

The results for this type of scenario, with 1000 subscriptions, are shown in Figure 5.15. Three figures show the results of event variance based, match count based and false match based selections. In each figure the results of the PCS and PFS algorithms, for each of the three covariance calculation algorithms, are shown. For comparison in every figure also the results of an algorithm not handling correlations between dimensions is shown.
Two algorithm combinations are the best in this scenario: PFS selection with a covariance matrix calculated from CEV and the combination of PCS with CFM.

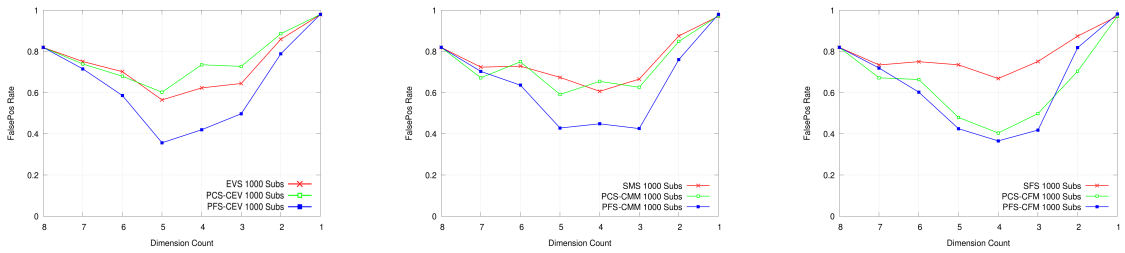


(a) Event variance based selection (EVS and CEV) (b) Match count based selection (SMS and CMM) (c) False match based selection (SFS and CFM)

Figure 5.15: Comparison of subscription selectivity selection, PCS and PFS for correlation 90%, 1000 subscriptions

An other type of correlation is inverse correlation. Correlation of dimensions a and b means that subscriptions and events with high/low values in dimension a will have low/high values on dimension b. Figure 5.16 shows the results of the same algorithms for a scenario with

zipfian distribution, constant dimension selectivity and 90 percent inverse correlation. For this inverse correlation scenario all PFS algorithms have good results but also the combination of PCS and CFM.



(a) Event variance based selection (EVS and CEV)

(b) Match count based selection (SMS and CMM)

(c) False match based selection (SFS and CFM)

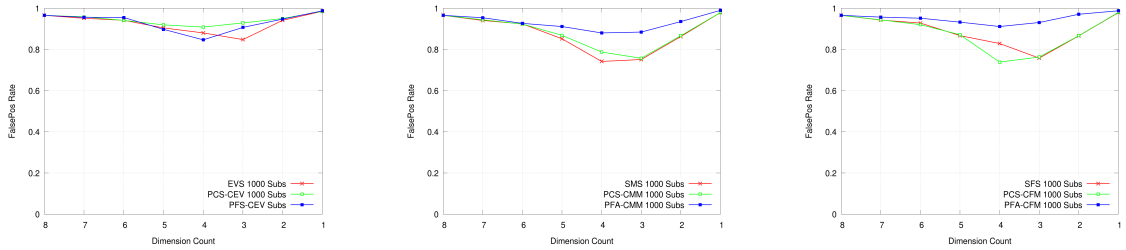
Figure 5.16: Comparison of subscription selectivity selection, PCS and PFS for inverse correlation 90%, 1000 subscriptions

Both scenarios are extreme cases as there is very high correlation between dimensions and the selectivity of dimensions is constant. Therefore the next scenario has random selectivity between dimensions and random correlation. Random correlation means that still pairs of dimensions are correlated but with a random amount of correlation instead of constant 90 percent. This scenario is illustrated in Figure 5.17.

Figure 5.17: Distribution of 200 subscriptions, zipfian distribution, random selectivity, random correlation

In Figure 5.18 the results of the algorithms for random correlation are plotted. All event variance based algorithms, shown in (a), are not good in this scenario. The main reason for this might be that they can not analyze the selectivity of dimensions with zipfian distribution, as shown in Section 5.2.1. PFS selection is worse for this type of scenario, only SMS, SFS and the combination of PCS with CMM or CFM is good for this scenario.

In summary, the combination of the selection algorithm PCS together with the covariance calculation CFM is most universal. This combination had good results in all three scenarios: Constant correlation, constant inverse correlation and inverse correlation. PFS based selection is good in detecting correlated dimensions but it cannot analyze the selectivity of dimensions and therefore is only useful for extreme cases with highly correlated dimensions.



(a) Event variance based selection (EVS and CEV)

(b) Match count based selection (SMS and CMM)

(c) False match based selection (SFS and CFM)

Figure 5.18: Comparison of subscription selectivity selection, PCS and PFS for random correlation, 1000 subscriptions

5.2.4 Evaluation-based Selection

In Section 3.4 algorithms that are based on evaluating and finding the best dimension set instead of selecting based on a mathematical model.
Two algorithms were presented: The *brute force evaluation based selection (BES)* and the *greedy evaluation based selection (GES)*.

Both algorithms evaluate possible sets of dimensions and are therefore very resource consuming. For processing 1000 subscriptions and 20,000 events for eight dimensions BES would need about ~ 8 hours. GES is much faster as its time complexity is linear depending on the number of dimensions instead of exponentially. The GES algorithm needs only ~ 400 seconds for the same data set.

As the BES algorithm is very slow, it is only possible to evaluate it in a smaller scenario. The following Figure 5.20 shows that the results of BES and GES are very similar. Brute force based selection trying all possible combinations is less than one percent better as greedy dimension selection. However as GES is much faster than BES it is the better alternative.

Figure 5.20 shows the results of the dimension selection using the GES algorithm in the scenarios with random dimension selectivity with uniform and zipfian distribution, as introduced in Section 5.2.1.
As you can see GES is better than all mathematical model based algorithms. As it is based on evaluations and not on al model it is very universal, it can be used for any type of scenario.

All in all the evaluation based methods can reduce the *false positive rate* more than the other algorithms but their disadvantage is the runtime. The BES algorithm can not be used in a practical scenario as it is very slow but the GES algorithm is much faster and is nearly as good as the BES.

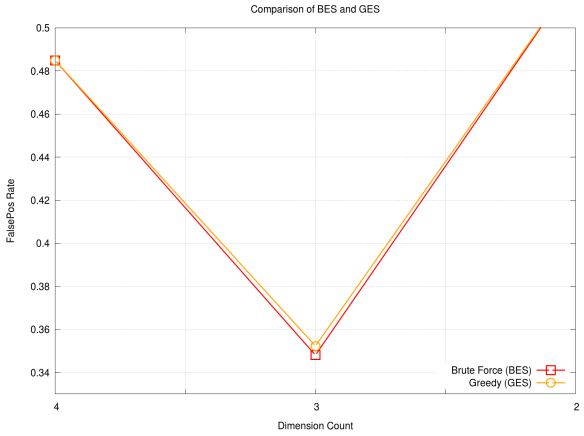
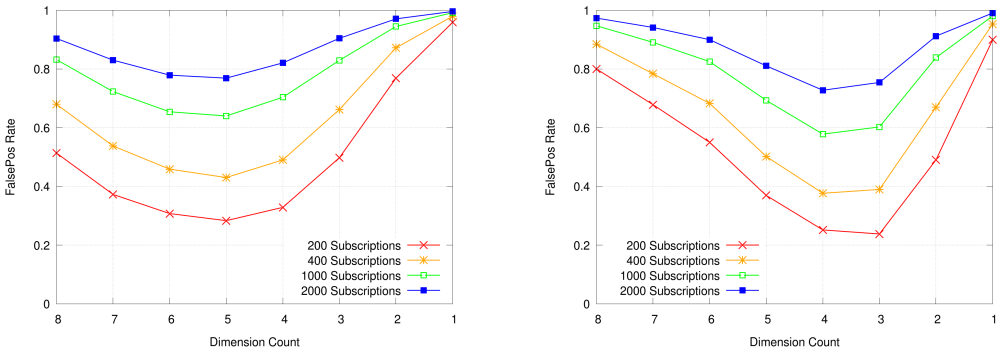


Figure 5.19: Architecture of Simulation Application



(a) GES for uniform distribution

(b) GES for zipfian distribution

Figure 5.20: GES false positive rates for: Uniform and zipfian distribution, random selectivity

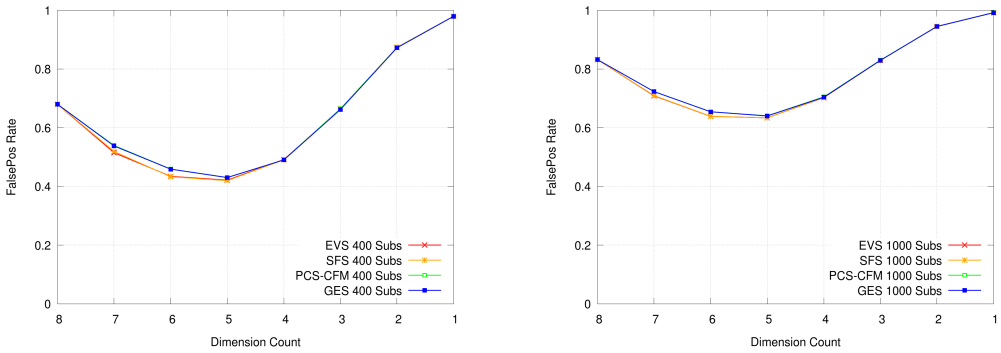
5.2.5 Dimension Selection Comparison

In this section a comparison of four representative algorithms is presented:

1. EVS - the fastest selection algorithm
2. SFS - the most accurate subscription selectivity based algorithm
3. PCS-CFM - the most universal correlation based algorithm
4. GES - the faster evaluation based algorithm

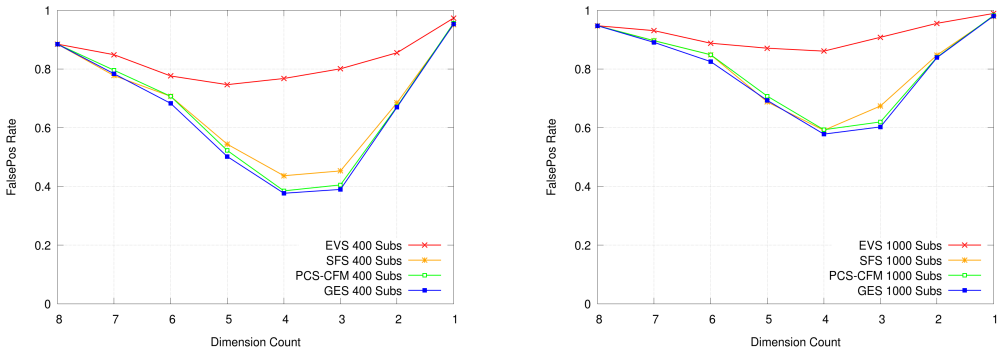
These algorithms are compared for four different scenarios for 400 and 1000 subscriptions: Uniform and zipfian with random selectivity, zipfian with 90 percent correlation and zipfian with random selectivity and random correlation.

The first comparison is for uniform distribution



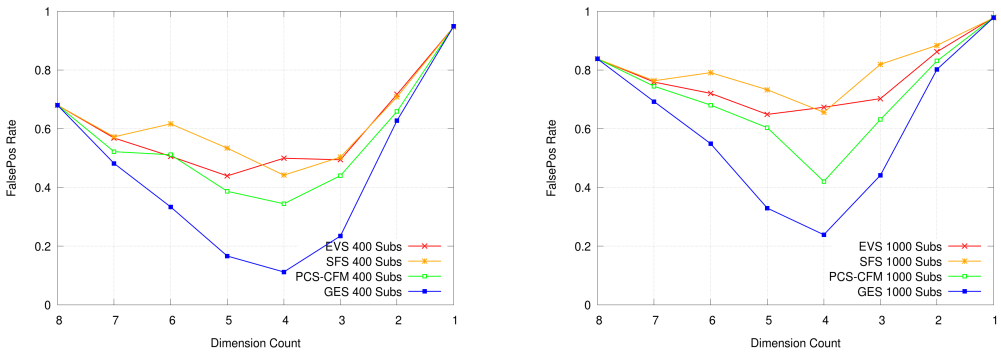
(a) 400 Subscriptions (b) 1000 Subscriptions

Figure 5.21: Comparison for: Uniform distribution, random selectivity



(a) 400 Subscriptions (b) 1000 Subscriptions

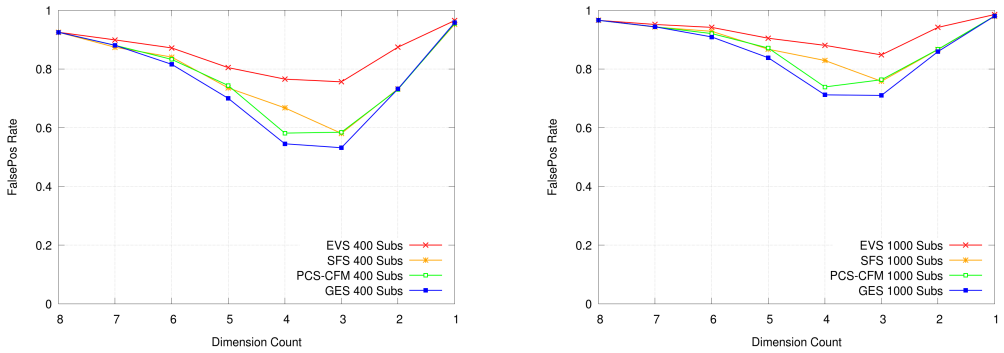
Figure 5.22: Comparison for: Zipfian distribution, random selectivity



(a) 400 Subscriptions (b) 1000 Subscriptions

Figure 5.23: Comparison for: Zipfian distribution, constant selectivity, 90 percent correlation

In summary the GES algorithm is the most universal algorithm. For all four comparison scenarios it had the best results. Anyway the PCS selection algorithm with a covariance from the CFM algorithm is also very universal and accurate, for correlated and for uncorrelated



(a) 400 Subscriptions (b) 1000 Subscriptions

Figure 5.24: Comparison for: Zipfian distribution, random selectivity, random correlation

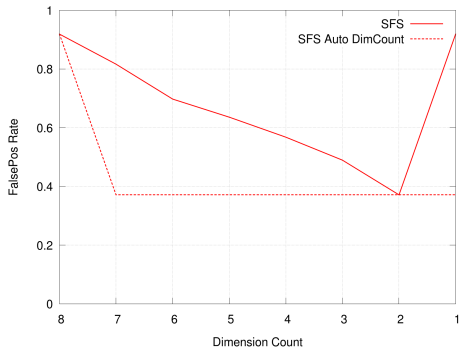
data. For uniformly distributed subscriptions without correlation all algorithms, including EVS, had similarly good results.

5.2.6 Best Dimension Count

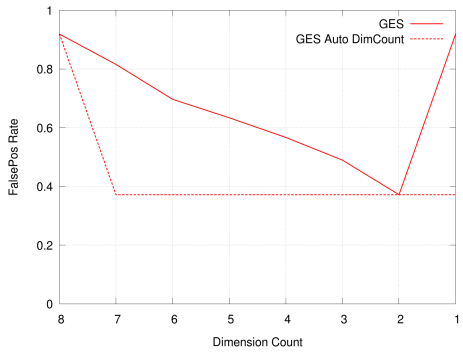
All presented algorithms need the number of dimensions to select as input. However this number can vary depending on the number of selective dimensions and other factors.
Therefore in Section 3.5 two approaches were presented to select the best number of dimensions. In this section we evaluate the PCA-based dimension count, presented in Section 3.5.1 together with the SFS dimension selection algorithm and the GES algorithm detecting the dimension count, presented in Section 3.5.2.
The PCA-based dimension count determination uses a fixed threshold to determine the principal components with highest accuracy. For this evaluation we chose 0.85 for best results.
The following diagrams show the evaluation results of the algorithms for all dimension counts and with a constant, automatically selected dimension count. The solid line shows the selection results for all dimension counts, the dashed line shows the results for a automatically chosen dimension count.

Figure 5.25 shows a scenario with two selective dimensions and uniformly distributed subscriptions. The automatically determined dimension count is clearly very accurate, it always has the same result as the best dimension count.
In Figure 5.26 a scenario with four selective dimensions and uniformly distributed subscriptions is evaluated. In this case the PCA-based dimension count with SFS does not always find the best dimension count. GES with automatic dimension count always finds the best dimension count by design.

The Figures 5.27 and 5.28 show the results for zipfian distribution with two and four selective dimensions. Same as for the last figures GES always finds the best dimension count and the PCA based dimension count only for the scenario with two selective dimensions.

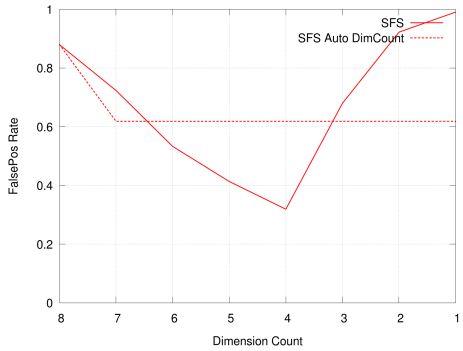


(a) SFS and SFS with automatic dimension count

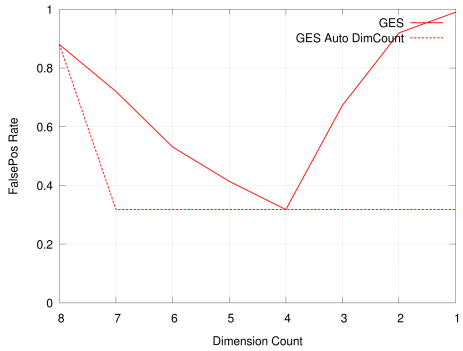


(b) GES and GES with automatic dimension count

Figure 5.25: Results of automatic dimension count for: Uniform distribution, two selective dimensions



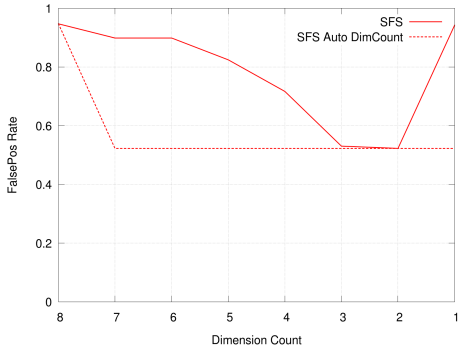
(a) SFS and SFS with automatic dimension count



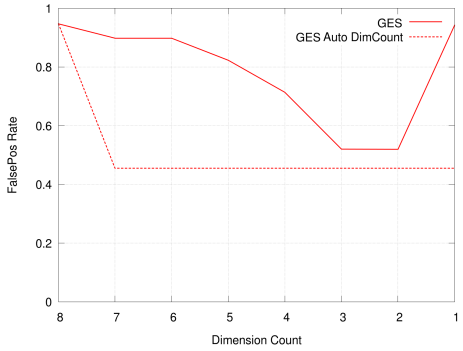
(b) GES and GES with automatic dimension count

Figure 5.26: Results of automatic dimension count for: Uniform distribution, four selective dimensions

All in all the PCA based dimension count determination can not find the best dimension count for all scenarios. To achieve this more intelligent strategies than a fixed threshold are needed. Anyway this approach was able to find the best dimension count ± 1.
The GES algorithm for automatic dimension count always finds the best number of dimensions.

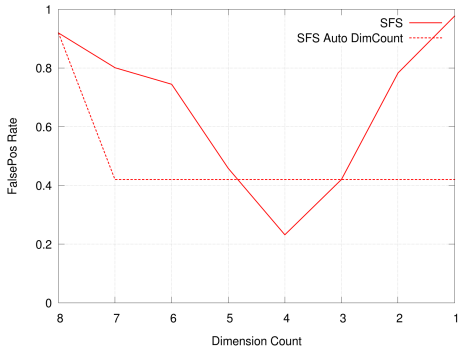


(a) SFS and SFS with automatic dimension count

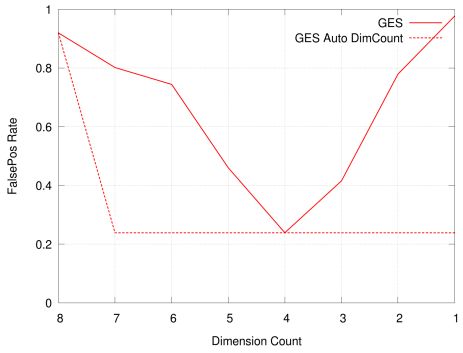


(b) GES and GES with automatic dimension count

Figure 5.27: Results of automatic dimension count for: Uniform distribution, two selective dimensions



(a) SFS and SFS with automatic dimension count



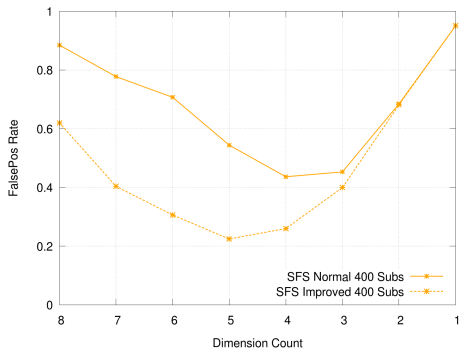
(b) GES and GES with automatic dimension count

Figure 5.28: Results of automatic dimension count for: Uniform distribution, four selective dimensions

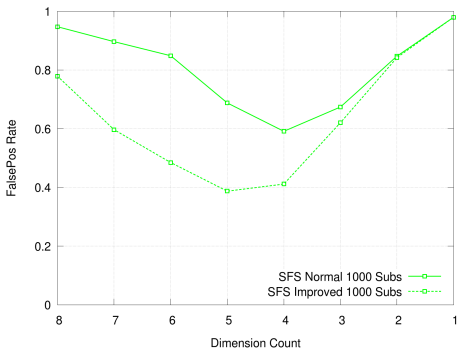
5.2.7 Dimension Selection and Improved Partitioning

An additional approach to increase the bandwidth efficiency by reducing the *false positive rate (FPR)* is the improved partitioning of the event space, presented in Chapter 4. This approach optimizes the generation of the spatial index for scenarios with unevenly distributed subscriptions.

The figures show the combination of this approach with the two algorithms SFS and GES, as described in Section 4.2. The evaluation scenario is again zipfian distribution with random dimension selectivity, for 400 and 1000 subscriptions.
For both algorithms, SFS and GES, the combination with the improved partitioning decreases the false positive rate significantly again. The combination of both algorithms can reduce for 400 subscriptions the original FPR from 88% to 23% with SFS and 24% with GES. The FPR of 95% for 1000 subscriptions can be reduced to 39% with SFS and to 42% with GES. This means a reduction of false positives of $\sim 75\%$ for 400 subscriptions and $\sim 60\%$ for 1000 subscriptions.

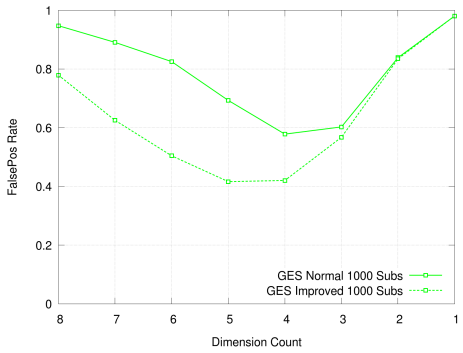


(a) SFS with improved partitioning, 400 subscriptions

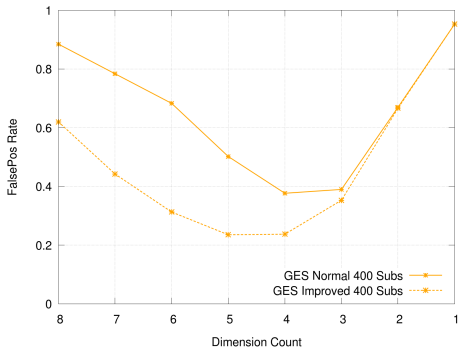


(b) SFS with improved partitioning, 1000 subscriptions

Figure 5.29: SFS with and without improved partitioning for zipfian distribution



(a) GES with improved partitioning, 400 subscriptions



(b) GES with improved partitioning, 1000 subscriptions

Figure 5.30: GES with and without improved partitioning for zipfian distribution

Chapter 6

Conclusion and Future Work

While SDN allows content-based routing with line-rate performance, the limited filtering resources available on switches decrease the efficiency of the *in-network* filtering and thereby decrease the bandwidth efficiency.

In this thesis ways to increase the bandwidth-efficiency by using the available filtering resources more efficiently were presented. In the system model, this thesis is based on, for each transmitted information packet an index is generated, called *dz-expression (dz)*. The *dz* is stored in the packet header and used for routing decisions on the SDN switches. Thus its accuracy is crucial for the precision of the in-network filtering. The maximum length of this index depends on the network protocol and the number of filtering rules depends on the available TCAM memory on the switch. Both limits the precision of the in-network filtering. Therefore this thesis addressed the problem of bandwidth-efficient routing by improving the generation of *dz-expressions*. For best filtering results a dz must contain as much filtering relevant information as possible and the dz must represent the information to filter as good as possible. Two approaches to generate more representative dz-expressions were presented: Better selection of information event dimensions that should be used to generate the dz and an improvement of the dz generation itself.

For selecting the most relevant dimensions to generate the index various different strategies were developed. Two main factors were considered: Selectivity of dimensions, which means how good an dimension can be used for filtering and correlation of dimensions, if it is possible that an dimension can be represented by an other and therefore left out. Taking into account these factors, the strategies were developed and evaluated. Two dimension selection algorithms stand out: The *greedy evaluation based selection (GES)* which is most universal but slower and the *principal component based selection using covariance from false matches (PCS-CFM)* which is not as universal as GES but faster.
In addition approaches to detect the best number of dimensions to select for indexing were developed. Depending on the scenario the best number of dimensions to select can vary. Evaluations have shown that one of the two proposed algorithms can always find the best number and the other at least the best number ± 1. With a combination of the determination of the best number of dimensions to select and the selection of the best dimensions, the best set of dimensions can be selected.

Furthermore an approach to improve the generation of *dz-expressions* itself was presented

and evaluated. This approach improves the partitioning of the n-dimensional event space of dimensions, done by the *spatial indexing*. The approach improves the filtering accuracy which was proven by evaluations.

Both approaches, the improved selection of dimensions and improving the event space partitioning can be combined. The evaluation of both approaches combined showed that the combination leads to even higher filtering precision. In summary the proposed strategies can determine the ideal number of dimensions, select the dimensions best for filtering and improve the generation of the dz-expressions used for filtering. A combination of the strategies presented in this thesis can increase the bandwidth efficiency of content-based pub/sub systems in SDN networks significantly. The number of false positives can be reduced by up to 75%, depending on the scenario.

The best of the presented dimension selection algorithms need a significant amount of time to process the input data and select the best dimensions. As their processing time scales scales linearly with the size of the input data, such as events and subscriptions, the processing time could be problematic for large scale systems. A possible approach to improve the algorithm run time is to reduce the amount of input data. Strategies to reduce the amount of input data without loosing relevant information are needed.
Moreover the evaluations showed that higher numbers of subscriptions increase the number of false positives. Strategies to deal with many subscriptions should be developed or existing approaches could be adapted.

Bibliography

[BBQV07] R. Baldoni, R. Beraldi, L. Querzoni, A. Virgillito. Efficient Publish/Subscribe Through a Self-Organizing Broker Overlay and its Application to SIENA. *The Computer Journal*, 50:444–459, 2007.

[BCM+99] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, D. C. Sturman. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. In *Proceedings of the 19th IEEE international conference on distributed computing systems (ICDCS)*. 1999.

[BFPB10] S. Bianchi, P. Felber, M. G. Potop-Butucaru. Stabilizing Distributed R-Trees for Peer-to-Peer Content Routing. *IEEE Transactions on Parallel and Distributed Systems*, 21:1175–1187, 2010.

[CDNF01] G. Cugola, E. Di Nitto, A. Fuggetta. The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27:827–850, 2001.

[CJ11] A. K. Y. Cheung, H.-A. Jacobsen. Green Resource Allocation Algorithms for Publish/Subscribe Systems. In *Proceedings of the 31st international conference on distributed computing systems (ICDCS)*. 2011.

[Com12] O. M. E. Committee. *Software-defined Networking: The New Norm for Networks*. Open Networking Foundation, 2012.

[CRW01] A. Carzaniga, D. S. Rosenblum, A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19:332–383, 2001.

[CS04] F. Cao, J. P. Singh. Efficient Event Routing in Content-based Publish-Subscribe Service Networks. In *Proceedings of IEEE INFOCOM 2004*. IEEE, Hong Kong, China, 2004.

[DK] F. Dürr, T. Kohler. Comparing the Forwarding Latency of OpenFlow Hardware and Software Switches. Tecnical report 2014/04. *Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany*.

[FCMB06] L. Fiege, M. Cilia, G. Muhl, A. Buchmann. Publish/Subscribe Grows Up: Support for Management, Visibility Control, and Heterogeneity. *IEEE Internet Computing*, 10:48–55, 2006.

[Fou] O. N. Foundation. OpenFlow Switch Specification.

[GSAA04] A. Gupta, O. D. Sahin, D. Agrawal, A. E. Abbadi. Meghdoot: Content-Based Publish/Subscribe over P2P Networks. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on middleware*, pp. 254–273. Springer-Verlag New York, Inc., 2004.

[JCL+10] H.-A. Jacobsen, A. K. Y. Cheung, G. Li, B. Maniymaran, V. Muthusamy, R. S. Kazemzadeh. The PADRES Publish/Subscribe System. In *Principles and Applications of Distributed Event-Based Systems*. IGI Global, 2010.

[KDT13] B. Koldehofe, F. Dürr, M. A. Tariq. Event-based systems meet software-defined networking. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (DEBS)*. 2013.

[KDTR12] B. Koldehofe, F. Dürr, M. A. Tariq, K. Rothermel. The Power of Software-defined Networking: Line-rate Content-based Routing Using OpenFlow. In *Proceedings of the 7th MW4NG Workshop of the 13th International Middleware Conference*, pp. 1–6. ACM, 2012. doi:10.1145/2405178.2405181. URL `http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2012-41&engl=0`.

[LCZT07] Y. Lu, I. Cohen, X. S. Zhou, Q. Tian. Feature selection using principal feature analysis. In R. Lienhart, A. R. Prasad, A. Hanjalic, S. Choi, B. P. Bailey, N. Sebe, editors, *ACM Multimedia*, pp. 301–304. ACM, 2007. URL `http://dblp.uni-trier.de/db/conf/mm/mm2007.html#LuCZT07`.

[MG04] A. Malhi, R. X. Gao. PCA-based feature selection scheme for machine defect classification. *IEEE T. Instrumentation and Measurement*, 53(6):1517–1525, 2004. URL `http://dblp.uni-trier.de/db/journals/tim/tim53.html#MalhiG04`.

[Pie04] P. Pietzuch. *Hermes: A Scalable Event-Based Middleware*. Ph.D. thesis, University of Cambridge, 2004.

[PRGK09] J. A. Patel, E. Rivière, I. Gupta, A.-M. Kermarrec. Rappel: Exploiting interest and network locality to improve fairness in publish-subscribe systems. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 53:2304–2320, 2009.

[TKBR14] M. A. Tariq, B. Koldehofe, S. Bhowmik, K. Rothermel. PLEROMA: A SDN-based High Performance Publish/Subscribe Middleware. In *In Proceedings of the ACM/IFIP/USENIX Middleware Conference*. ACM press., 2014. doi:10.1145/2663165.2663338. URL `http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2014-67&engl=0`.

[TKK+11] M. A. Tariq, B. Koldehofe, G. G. Koch, I. Khan, K. Rothermel. Meeting subscriber-defined QoS constraints in publish/subscribe systems. *Concurrency and Computation: Practice and Experience*, 23:2140–2153, 2011.

[TKKR09] M. A. Tariq, B. Koldehofe, G. Koch, K. Rothermel. Providing Probabilistic Latency Bounds for Dynamic Publish/Subscribe Systems. In *Proceedings of the 16th ITG/GI conference on kommunikation in verteilten systemen (KiVS)*. Springer, 2009.

[TKKR12] M. A. Tariq, B. Koldehofe, G. G. Koch, K. Rothermel. Distributed Spectral Cluster Management: A Method For Building Dynamic Publish/Subscribe Systems. In *Proceedings of the 6th ACM international conference on distributed event-based systems (DEBS)*. 2012.

[TKR13] M. A. Tariq, B. Koldehofe, K. Rothermel. Efficient content-based routing with network topology inference. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (DEBS)*. 2013.