Adaptive optimization techniques for context-aware information filters

Author(s):
Fischer, Peter Michael

Publication Date:
2006

Permanent Link:
https://doi.org/10.3929/ethz-a-005203442

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
Adaptive Optimization Techniques for Context-Aware Information Filters

A dissertation submitted to the
ETH ZURICH
for the degree of
Doctor of Sciences

presented by
PETER MICHAEL FISCHER
Diplom-Informatiker (Univ.) TU München
born 15th of April 1977
citizen of Germany

accepted on the recommendation of
Prof. Donald Kossmann, examiner
Prof. Michael Franklin, co-examiner

2006
Acknowledgements

First and foremost, I would like thank my advisor, Prof. Donald Kossmann. From the first day (actually starting back in the times when I was a Diploma student), he put trust in me and gave me the freedom to follow my ideas and work in the most effective manner. His enthusiasm to work on challenging research also motivated me to tackle the problems that occurred during my way to the doctorate. Whenever necessary, he gave me time for discussion and advice and spent uncountable hours with me to refine our work. He taught me to organize my thoughts into clean, well structured concepts and present them in a way that would provide the most impact. He showed me when to focus on specific details and when explore crazy new ideas, so that I could use my energy in the best possible way.

I would also like to thank Professor Michael Franklin, who served as co-referee for my doctorate. At the end of diploma studies (back in 2001), he gave me the opportunity to come to his group at UC Berkeley and spend six months there to do research for my Diploma Thesis. In these six months, when I worked with him and Yanlei Diao, he introduced me into the area of information filtering and raised my interest in this subject area. I enjoyed working in this area so much that I chose it as my thesis topic. Even though he has very limited time now, he agreed to take over the role as co-referee. He read and evaluated the thesis in a very short time, and again provided though-provoking and promising ideas for future research.

Without the students that contributed their part to this work, progress would never have been so swift, and also a lot of the excitement would have been missing. Therefore I would like to thank Dongseop Kwon, Nadine Schmidt, Mario Tadey and Julia Imhof for their ideas, their hard work and the interesting discussions we had.

Good research always benefits from a good, collaborative environment. This was especially true in my case, and therefore I would like to express my gratitude to be able work with such good colleagues in Munich, Heidelberg and Zurich. The spirit in our group has always been extremely friendly and supportive, and therefore it has been a joy to work with them. The support provided by my colleagues covered many areas, including (but not limited) many hours of proofreading papers and also this thesis.

Finally, I would like thank my family for their support and patience when I was focused working on my research. The last four years have not always been easy, but they gave me all the support I could wish for.
Seite Leer /
Blank leaf
# Contents

1 Introduction

1.1 Background and Motivation ........................................ 1
1.2 Purpose of this thesis ............................................. 3
1.3 Contributions .................................................. 4
1.4 Structure ..................................................... 5

2 Related work

2.1 Design Space for Information Filters ............................... 7
  2.1.1 Subject-Based Information Filters .......................... 7
  2.1.2 Predicate-Based Information Filters ....................... 8
  2.1.3 XML Filtering ........................................... 9
  2.1.4 XML Transformations .................................. 10

2.2 Standing Queries ................................................ 10
  2.2.1 Stream Processing ..................................... 10
  2.2.2 Continuous Query systems ............................... 11
  2.2.3 Trigger Processing .................................... 11

2.3 Context-Aware Computing ..................................... 11

2.4 Indexing Research ............................................. 12
  2.4.1 Profile Indexing ..................................... 12
2.4.2 Indexing for High Update Rates .................................. 13
2.4.3 Adaptive Indexing .................................................. 13
2.4.4 Bulk Index Operations ............................................. 13

I Stateless Information Filters ........................................ 15

3 Concept ................................................................. 17
  3.1 Introduction ...................................................... 17
    3.1.1 Use Cases .................................................. 17
    3.1.2 Contributions ............................................. 18
  3.2 Problem description ............................................ 18
    3.2.1 Messages .................................................. 19
    3.2.2 Profiles .................................................. 19
    3.2.3 Processing Model ........................................ 20

4 Analysis of the State of the Art .................................. 21
  4.1 Architecture .................................................... 21
    4.1.1 Indexes .................................................. 22
    4.1.2 Merge .................................................... 23
    4.1.3 Postfilter ............................................... 24
    4.1.4 Mapping other Information Filters to this architecture .. 24
  4.2 Implementation of the Test Bed ................................ 25
    4.2.1 Interval Skip lists ...................................... 25
    4.2.2 Marker Set Representation ............................... 28
  4.3 Cost Model ..................................................... 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1.2</td>
<td>Properties of AGILE Indexes</td>
<td>85</td>
</tr>
<tr>
<td>7.1.3</td>
<td>AGILE Algorithm</td>
<td>86</td>
</tr>
<tr>
<td>7.1.4</td>
<td>AGILE Indexes</td>
<td>87</td>
</tr>
<tr>
<td>7.1.5</td>
<td>Deescalation Policies</td>
<td>94</td>
</tr>
<tr>
<td>7.2</td>
<td>Cost Model Extension and Analysis</td>
<td>95</td>
</tr>
<tr>
<td>7.3</td>
<td>Performance Experiments and Results</td>
<td>98</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Experiment Setup</td>
<td>98</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Workload</td>
<td>99</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Experiment 1: Throughput in Steady State</td>
<td>100</td>
</tr>
<tr>
<td>7.3.4</td>
<td>Experiment 2: Vary $UpdAtt$</td>
<td>102</td>
</tr>
<tr>
<td>7.3.5</td>
<td>Experiment 3: Vary $\Delta U$</td>
<td>102</td>
</tr>
<tr>
<td>7.3.6</td>
<td>Experiment 4: Update Bursts</td>
<td>103</td>
</tr>
<tr>
<td>7.4</td>
<td>Conclusion</td>
<td>105</td>
</tr>
<tr>
<td>8</td>
<td>Batched Processing of Updates</td>
<td></td>
</tr>
<tr>
<td>8.1</td>
<td>Introduction</td>
<td>107</td>
</tr>
<tr>
<td>8.2</td>
<td>Extension of the Architecture</td>
<td>107</td>
</tr>
<tr>
<td>8.3</td>
<td>Methods</td>
<td>108</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Bulk Index Access</td>
<td>108</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Resulting Updates</td>
<td>109</td>
</tr>
<tr>
<td>8.4</td>
<td>Cost Model Extension and Analysis</td>
<td>111</td>
</tr>
<tr>
<td>8.5</td>
<td>Integrating Message and Update Batching</td>
<td>112</td>
</tr>
<tr>
<td>8.5.1</td>
<td>Interleaving of Message and Update Batches</td>
<td>112</td>
</tr>
<tr>
<td>8.5.2</td>
<td>Feedback with Batched Processing and AGILE</td>
<td>115</td>
</tr>
<tr>
<td>8.6</td>
<td>Performance Experiments and Results</td>
<td>116</td>
</tr>
</tbody>
</table>
10.1 Summary of the Thesis ........................................ 169
10.2 Future Work ................................................... 171
Zusammenfassung


Diese Arbeit trägt drei neue Aspekte zum Gebiet von Informationsfiltern bei: Skalierbarkeit in Richtung des Durchsatzes von Nachrichten, kontextsensitive Informationsfilter, die zusätzlichen Zustand zum Abgleich von Profilen und Nachrichten verwenden, und eine Studie zur Dienstgüte bei Informationsfiltern. Die Skalierbarkeit in Richtung des Durchsatzes von Nachrichten wird erreicht, indem Nachrichten in Gruppen und nicht, wie bisher, einzeln abgearbeitet werden. Dadurch werden die Kosten zur Bearbeitung einer einzelnen Nachricht reduziert. Kontextsensitive Informationsfilter erweitern bestehende zustandslose Informationsfilter, indem sie den Zustand von Kon-

In allen Gebieten, die diese Arbeit betrachten, werden eine theoretische Analyse sowie eine ausführliche Leistungsuntersuchung durchgeführt. Auf diese Weise können die Vorteile und Besonderheiten der Ansätze ausgewertet werden.

Abstract

In recent years, we have seen a shift in the way information is processed. Departing from the traditional paradigm in which information is first stored and then queried, we are quickly moving to a new paradigm in which new information is directly routed to the relevant recipients. This new paradigm is being adopted by several research communities, databases being only one of them. Information filters represent one of the key components of this new paradigm, as they loosely couple senders and receivers of data items. Receivers of information submit a profile of their interest to the information filter, while producers of information send messages to the information filter. The purpose of an information filter is then to match the messages to the profiles, so that the matching messages can be sent to the relevant receivers. Information filters are used in areas like application integration, personalized content delivery, networking monitoring and many other areas. Simpler versions of information filters are appearing as products on the market place, while research continues into several directions.

In order to enable the information filtering paradigm, techniques like profile indexing and stream processing are used. The main directions of research have been expressiveness of profiles, scalability in terms of profiles and distribution of information filters over networks.

This thesis contributes three new aspects to the area of information filtering: scalability in terms of message throughput, context-aware information filters that use state for the matching decision and a study of quality of service. Scalability in terms of message throughput is achieved by processing messages in batches instead of processing them one by one, thus reducing the cost of processing an individual message. Context-aware information filters augment existing, stateless information filters by including context state into the matching decision. Since this state receives updates, the key challenge in building such a context-aware information filter is to deal with high message rates and high update rates. The thesis addresses this challenge in two different ways: AGILE, a method to automatically adapt index accuracy to the workload parameters, and batched
processing of updates, where a set of updates is processed at once in order to reduce the cost. Quality of service becomes more and more important as information filters are used in settings where the load is unpredictable and might exceed the available resources. This work examines how state of the art approaches to implement quality of service apply to information filters. For the three areas contributed by this thesis, a theoretical analysis and an extensive performance study is provided, illustrating the benefits and trade-offs.

To sum up, this thesis contributes work to improve information filters by increasing the message throughput, including context state in the matching process and studying quality of service. The results provide further support for the adoption of information filters into the mainstream of information processing.
Chapter 1

Introduction

1.1 Background and Motivation

In recent years, we have seen a shift in the way information is processed. Departing from the traditional paradigm in which information is first stored and then queried, we are quickly moving to a new paradigm in which new information is directly routed to the relevant recipients. This new paradigm is being adopted by several research communities (databases being only one of them) and products are appearing on the market place: Tibco [1], Business Connector from SAP [2], BizTalk Server from Microsoft [6], or the Message Broker and bus from BEA [3], to mention just a few. In order to enable this new information filtering paradigm, techniques from areas such as event-based programming, publish and subscribe, continuous query processing, and information dissemination (push) are effected.

An information filter (as depicted in Figure 1.1) connects sources and sinks of information by the use of profiles. Parties interested in receiving information (sinks) submit a profile of their interest to the information filter, while parties interested in disseminating information send messages to the information filter. The purpose of an information filter is to match the messages to the profiles, so that the matching messages can be sent to the relevant subscribers. A message matches a profile if it contains values for all the attributes involved in predicates of the profile and these values meet the restrictions specified in these predicates. For instance, the message \([x=3,y=5,z=7]\) meets the profile \((x=3) \land (z>2)\), whereas it does not match the profile \((w \leq 0)\). In this thesis, context information is used to enhance the matching, thus context information needs to be kept in a context database inside the information filter. This database will receive
updates from outside sources. A more formal problem statement will be presented in Section 3.2.

To support information filtering, several research communities have made significant contributions. The main focus of these contributions has been on three areas:

1. **Expressiveness of Profiles**: While commercially available systems only support a very simple subject-based matching model [1], scientific research has steadily raised the bar. Predicate-based systems [44], whole document XML filtering [13] and, most recently, XML restructuring [40] have been the stepping stones.

2. **Scalability in Terms of Profiles**: While older systems often supported only a few thousands of profiles, current research systems support up to several millions of profiles for predicate-based systems [44], several hundreds of thousands profiles consisting of XML path expressions [39], and several tens of thousands profiles with complex XML expressions [39], all running on an off-the-shelf hardware.
1.2. Purpose of this thesis

3. **Distribution of Information Filters over the Network**: Since sources and sinks of information are often widely distributed, transferring all messages and profile subscriptions via a single, central information filter may impose a significant bottleneck and a single point of failure. Distributed information filters have been developed, which support routing protocols for both subscriptions and messages to minimize the network traffic and increase the scalability [22].

1.2 Purpose of this thesis

This thesis builds on top of the current state of the art in information filtering. It provides insights into the following issues that have — so far — not been sufficiently studied.

1. **Scalability in Terms of Message Throughput**: While the scalability in terms of profiles has been increased significantly by existing information filters, increasing the message throughput has always been just a byproduct. This thesis presents techniques to improve the throughput of an information filter by handling messages not one-by-one, but in larger batches. We call this technique **Batch Processing of Messages**.

2. **Context-Aware Information Filters**: Existing information filters are stateless, as they use an (almost) static set of profiles for matching. This thesis introduces an information filter that uses additional state for the matching decision, called "context".

3. **Quality of Service**: Research on matching algorithms has focused mostly on scalability and expressiveness. What is lacking, however, is a broader view on how these algorithms can provide specific service levels, deal with overload situations and changes in the workload. This thesis presents requirements and techniques for **quality of service** in information filters.

When considering the ideas and concepts examined in this thesis, adaptivity stands out. Its importance is highlighted by the following factors: Workloads processed by information filters are not static, but exhibit changes over time. Such changes are high and low arrival rate (burstiness), variations in types of events (messages, state update events) or changing skew over various parameters like the distribution of values in messages and state. A static algorithm is not able to handle these changes in the best possible
way. An algorithm with tunables could be set to the right parameters, but manual intervention is error-prone and difficult to perform in fast-changing workloads. Adaptivity driven by the workload is therefore the right choice. The following adaptive algorithms are presented in this thesis:

- Adaptive indexing to take advantage of changes in the workload, especially skew in the workload parameters.
- Adaptive batch size control for message batching to achieve optimal latency and throughput over changing arrival rates.
- Adaptive control of the message flow, the flow of state update events and the choice on how to process these events in order to ensure required quality parameters and optimize “free” parameters.

1.3 Contributions

In short, the main contributions of this thesis can be summarized as follows:

1. An architecture and an accompanying cost model was developed to capture the common parts of state-of-the art information filters. An implementation of this architecture serves as test bed for the experimental evaluation.

2. Algorithms to perform batched processing of messages were developed and examined analytically as well as experimentally.

3. A conceptual model, a problem description and several use cases for a stateful (aka context-aware) information filter (CIF) were devised. State of the art methods in information filtering, indexing and query processing were reviewed to survey their suitability for context-aware information filtering.

4. Two approaches to improve context update processing were designed, implemented and compared to the existing approaches and with each other, since none of the existing approaches is well suited for context-aware information filtering:

   - The first method is AGILE, an adaptive indexing method that adjusts itself to changes in the workload to reach the optimum performance.
1.4. Structure

• The second method is Batched processing of updates, utilizing similar ideas as batched processing of messages.

5. A set of quality of service requirements for information filters was established. Methods based on existing work and the work in this thesis are reviewed in respect to their suitability to enforce those requirements by an extensive performance study.

1.4 Structure

The remainder of this thesis is organized in the following:

• Chapter 2 describes related work.

• The first part of this thesis covers stateless information filters: Chapter 3 presents the problem description. Chapter 4 discusses the state of the art by showing our reference architecture with the methods to implement it and the cost model. Chapter 5 introduces batched processing of messages.

• The second part addresses context-aware information filters: Chapter 6 describes the concept with a problem description. The extensions to the architecture and a review of the state of the art processing methods are also in this chapter. Chapter 7 introduces AGiLE. Chapter 8 presents methods for batched processing updates as well as methods how integrate message and update batching.

• The third main part treats the issue of quality of service in information filters: Chapter 9 gives a problem description with relevant parameters. Existing methods to enforce these parameters are evaluated in an extensive performance study.

• Chapter 10 provides some concluding remarks and outlines future work.
Chapter 2

Related work

This chapter gives an overview of related work relevant to information filtering. Where applicable, later chapters will extend this related work by discussing specific issues.

The first section of this chapter presents the design space of existing information filters, following a survey in [38]. Following this section, an overview of other work using standing queries is presented. Since this thesis target context-aware information filters, the third section of this chapter covers related work in the area of context-aware computing. The last section presents relevant concepts in indexing, as information filters rely heavily on profile indexing.

2.1 Design Space for Information Filters

Figure 2.1 shows the design space of existing information filters, utilizing the same dimensions outlined in the introduction: expressiveness of profiles on the x axis and distribution on the y axis. Scalability in terms of profiles as an implicit goal of all the systems is not mentioned explicitly.

2.1.1 Subject-Based Information Filters

The first class of information filters are subject-based information filters. Labels from a pre-defined set of subjects are attached to a message by the publisher, e.g. “soccer” or “stock quote”. User specify their subject of interest by choosing them from the pre-
defined set. The matching decision on message and a profile is binary: Either “Match” or “No Match”. The expressiveness of profiles in this setup is fairly limited, because the message contents are not inspected.

Most commercial message brokers are in this class: TibCo [1], MQ Series by IBM [8], JMS by Sun Microsystems [5] and BizTalk by Microsoft [6] support distributed operations, while Oracle Advanced Queuing [7] is a centralized solution.

### 2.1.2 Predicate-Based Information Filters

The second class of information filters are predicate-based information filters. In contrast to the subject-based information filters (and similar to the following classes), the message contents are compared against profiles. Predicate-based filters represent the message contents as key/value pairs. Profiles are predicates over those key/value pairs, usually conjunctions or disjunctions of atomic predicates. Atomic predicates, in turn, are comparisons between the value of a message key and a constant. Most systems support equality, greater-than and smaller-than as comparison operations. An

---

If necessary, other formats such as plain text or HTML can be transformed into key/value pairs.
example of such a profile would be "Stock = IBM and Last Trade < 81 and Volume > 2 million". Similarly as with subject-based systems, matching is binary: If the predicates in a profile are evaluated to true using the message values, the message matches; otherwise it does not match.

Predicate-based information filters have seen a lot of interest in several research communities, but have – so far – not been adopted in the industry. Le Subscribe [44] and Xyleme [65] are centralized systems that employ advanced predicate indexing strategies and profile clustering in order to achieve scalability in terms of profiles. Gryphon [67] and Siena [22] are examples of distributed predicate-based information filters. These systems aggregate user profiles into so-called routing tables, which are used to forward messages over several steps to the correct recipients. This thesis focuses on this type of profiles, but adds the additional dimensions of scalability in terms of message throughput and state.

2.1.3 XML Filtering

The third class of information filters support XML filtering. The message content is interpreted as XML and used to match against profiles that are subsets of XPath [82] and XQuery [19]. In general, a profile is a restricted path expression that only supports certain classes of locations steps and some predicates on attributes or text nodes. As in the previous classes, matching is binary: If the path specified in a profile is contained in the document (including the predicates on attributes and text nodes), the document matches, otherwise it does not match.

Most of the recent work in the DB community regarding information filtering has been directed into this area, mostly on centralized approaches. XFilter [13] and Index Filter [21] build indexes on the path expressions, while XTrie [23], YFilter [39] and XMLETK [49, 50] also exploit common parts of the profiles. Research on distributed XML filtering was mostly performed by the networking community, with other goals than scalability in terms of profiles: [75] focuses on the reliability and in-order delivery for XML routing, while [24] aims to minimize space overhead at an XML router.
2.1.4 XML Transformations

The fourth class of information filters goes beyond binary matching of profiles and messages. Instead of just matching the full XML document, these systems exploit the structured nature of XML and extract fragments of the document that are transformed into a new document, according to the specifications in the profile. Therefore profiles are a larger subset of XQuery [19], especially \texttt{FOR}, \texttt{WHERE} and \texttt{RETURN} clauses. As a consequence, this class of information filters allows customizing the results a recipient gets, in terms of data selection as well as in terms of presentation. A version of YFilter [40] is the only known system to support XML transformations in a centralized system. In a distributed system, transformations can also be used to reduce the amount of data transferred, as unneeded parts of a XML document can be removed before forwarding them. ONYX [41] supports such distributed XML filtering, transformations and routing.

2.2 Standing Queries

Information filters are not the only area that uses the concept of standing queries against changing data. Stream processing, continuous query systems and trigger processing share this concept and also provide techniques relevant for this thesis.

2.2.1 Stream Processing

Processing continuous data streams with very high, often varying, rates has seen significant interest by the database community in recent years. Systems like TelegraphCQ [25], STREAM [63] and Aurora [10]/Borealis [9] have been developed as a result of this interest. Key issues that have been researched are shared processing [25, 63], approximation [63, 68], adaptivity [25] and quality of service [15, 68, 79].

Stream processing systems differ from the work presented in thesis in the following way: while streams processing systems are designed to handle very high message rates, the number of queries is usually much lower than the number of profiles in information filters. The queries typically contain aggregations that can be exploited to reduce the message rates early in the execution. The state considered in stream processing is more restricted than the one in context-aware information filters: stream process-
ing systems keep the state of messages within a certain window, while context-aware information filters keep the state of arbitrary contexts without limitations.

### 2.2.2 Continuous Query systems

Continuous query systems [80] are used to produce new results whenever an update of information occurs. These systems are often used in an Internet setting when a large amount of frequently changing data is monitored. Methods employed by continuous query systems include incremental view maintenance in OpenCQ [59], incremental grouping of join operators using query signatures [29], and selection placement after joins to increase sharing [28] in NiagaraCQ and adaptive routing of tuples according to selectivity in TelegraphCQ [60].

Continuous query systems often use a smaller number of queries with slightly more complex expressions (joins) than information filters. Continuous query systems are either stateless [28,29] or use a restricted type of state like windows [60] or the last data items [59].

### 2.2.3 Trigger Processing

Triggers have been part of conventional database for more than 15 years now [72,77,84]. They turn the database from a passive data store into a so-called "active database", as they provide complex rules to react on changes in the data and trigger (thus the name) actions based on that. The design goal of trigger processing inside database has been to provide a feature-rich programming environment, resulting in the inclusion of turing-complete trigger languages in modern database products. Scalability in terms of profiles has rarely been a major goal in this setting. A notable exception is [54], where trigger conditions are indexed. The strategies used are similar to the predicate indexes in this thesis.

### 2.3 Context-Aware Computing

This thesis presents the combination of scalable information filters with context awareness. The term "context aware" is used in many areas, but within the scope of this
thesis, the focus rests on ubiquitous computing [83]. [27] presents a survey about the current state in this area. Most of the current work in this area is directed towards localization with different means [48], location-based services [55] and the overall design of context-aware information systems [69].

Context-aware information push has been discussed in two recent works: [31] analyzes the human-computing interface (HCI) point of view of pushing context-related data. The context model is different to ours, as the context data is pushed to the recipient, while the work in this thesis uses context data for filtering. Spatial publish/subscribe is introduced and demonstrated in [30]. In that work, the notion of context is limited to location, and the number of profiles is limited to hundreds (compared to hundred of thousands in this thesis).

2.4 Indexing Research

Information filters are based on the concept of query indexing instead of data indexing; therefore research on indexes plays an important role when enhancing information filters. Concepts developed for other index types or for other applications can be leveraged in the context of information filtering.

2.4.1 Profile Indexing

A technique commonly used in information filters is to index profiles or queries. Most recently, a significant amount of research was performed in XML indexing for information filtering, as described in Section 2.1.3. Similarly, continuous query systems [29] or trigger processing systems [54] index expressions to speed up processing.

Since this thesis focuses on predicate-based profiles, interval indexes are an interesting choice. The area of interval indexes has seen a lot of interest, resulting in many different data structures. Due to the large amount of possible options, only a number of representatives is named here. For main-memory interval indexes, Interval Skips [53], IBS Trees [52], Segment Trees [71], Priority Search Trees [62] or Interval Trees [43] can be named. For disk-based interval indexes, [61] provides a good overview. Recent examples of such disk-based intervals indexes are RI Trees [56] or Interval B-Trees [14].
2.4. Indexing Research

2.4.2 Indexing for High Update Rates

Including context in information filters requires high updates rates, also affecting the indexes used to speed up profile matching. AGILE adopts some ideas from moving object databases [66, 70], but generalizes those ideas and applies them to a different application domain; i.e., information filtering. More specifically, it picks up the idea of Lazy Updates, proposed by [57] and generalized by [58]. Lazy Updates avoid index updates by extending the bounding area by a certain amount, and not changing the index when this bounding area is not exceeded. In contrast to AGILE, Lazy Updates only consider a single level (bounding areas at the leaves) and do not adapt to load changes.

A second approach of moving object indexes, the use of trajectories [70, 74], is not applicable, since predicting future values of a generic state is very challenging.

2.4.3 Adaptive Indexing

AGILE utilizes the concept of adapting the index to the workload issued against it. This concept is well-established in the literature. Two recent examples of adapting the index to the (query) workload are [78] and [33]. Both have different goals and strategies than AGILE. While AGILE adapts the index accuracy for individual profiles based on the update/probe ratio, [78] adapts the node size of a disk-based tree index to contain several disk pages according to the query access pattern. While [78] is applicable to almost any tree index (similarly to AGILE), [33] is optimized for spatio-temporal indexing. It adaptively partitions the time domain according to the query workload.

2.4.4 Bulk Index Operations

Batched message and update processing rely on bulk index operations. Bulk index operations have been studied in various contexts for database systems. The idea to bundle probes to indexes has been studied by Zhou and Ross [88]; the focus of that study, however, is to optimize processor cache hit rates and that work was carried out in a completely different context (traditional database index structures such as B-Trees, rather than information filters). Bulk update operations on indexes have been studied in [34, 47], bulk loading of (multi-dimensional) indexes in [17], and bulk join processing has been studied in [36].
Part I

Stateless Information Filters
Seite Leer / Blank leaf
Chapter 3

Concept

3.1 Introduction

This part presents stateless information filters. All existing information filters follow that paradigm. Therefore, the first part of this chapter presents the problem statement and an analysis how existing information filters are implemented. The second part presents how to improve the scalability in terms of message throughput by the use of message batching.

3.1.1 Use Cases

Stateless information filters are present in a lot of applications, including but not limited to:

- **Application Integration**: Instead of tightly coupling previously unrelated applications by the use of RPC, message brokers are used that send the exchanged data to the right application components and transform it on the fly to achieve a schema mapping.

- **Personalized News Delivery**: As the amount of news increases in both number of sources and the rate of news events, tailoring news to the interest of people becomes increasingly relevant. Consumers can specify their interests in a very specific way, and will receive only those news items that match to these interests, thereby reducing the effects of information overload.
3.1.2 Contributions

In order to examine stateless information filters, the chapters of this part make the following contributions:

1. Analyzing the current state of the art.
2. Establishing an architecture to capture the gist of existing information filter systems.
3. Implementing a test bed for this architecture and the extension in the following parts of this thesis.
4. Providing a cost model for the analysis of the tradeoffs in existing information filters.
5. Introducing the concept of batching and showing how it can be applied to various existing indexing schemes.
6. Proposing alternative batching strategies. The batching strategies differ in the way they group messages and in the way they index groups of messages.
7. Extending the existing cost model in order to identify the critical parameters that affect the performance of the alternative batching strategies and, thus, helps to examine performance trade-offs.
8. Presenting the results of performance experiments with different workloads and indexing schemes.
9. Devising a model that allows to define an upper bound for the latency of a message (in non-overload situations).

3.2 Problem description

The purpose of an information filter is to match messages and profiles. In a relational world, such messages are attribute/value pairs. Profiles are conjunctions of predicates such as equality, range or set containment [44]. A message matches a profile if it contains values for all the attributes involved in predicates of the profile and these values meet the restrictions specified in these predicates. For instance, the message
3.2. Problem description

Message

<table>
<thead>
<tr>
<th>Type</th>
<th>Talk announcement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td>Main Bid Room 7</td>
</tr>
<tr>
<td>Time</td>
<td>4PM</td>
</tr>
</tbody>
</table>

Profile

Message.Type = 'talk announcement'
Message.Location = 'Main Bid'
9AM < Message.Time < 6PM

Figure 3.1: Example of Profiles and Messages

[\{x=3, y=5, z=7\}] meets the profile consisting of the predicates \((x=3) \land (z > 2)\), whereas it does not match the profile \((w \leq 0)\). In an XML world, messages are XML documents, profiles are XPath [82] or XQuery [19] statements, and matches are defined accordingly.

3.2.1 Messages

A message is a set of attributes associated to values. For example, a message announcing a talk can be modeled as shown in Figure 3.1. (Often messages are delivered in XML. Nevertheless, the simplified attribute/value model is used in this thesis for reasons of clarity while maintaining generality. Approaches to apply the new algorithms presented in this part to XML are outlined in Section 10.2.)

3.2.2 Profiles

A profile is a continuous query specifying the information interests of a subscriber. A definition of profiles proposed in other pub/sub-systems [44] is used, specifying the disjunctive normal form (DNF) of atomic comparisons.

\[
\text{profile} := \text{conj} \lor \text{profile} \lor \text{conj} \\
\text{conj} := \text{pred} \lor \text{conj} \land \text{pred} \\
\text{pred} := \text{message.attr \ op \ constant} \\
\text{op} := < \mid > \mid = \mid \leq \mid \geq
\]
With this definition, predicates can be employed that compare message attribute values against constants. These predicates may be combined using conjunctions and disjunctions.

### 3.2.3 Processing Model

Figure 3.2 shows the processing of an Information Filter. The Information Filter keeps profiles of subscribers and receives an input stream of messages.

In order to deal with this input stream, an information filter must support the following method:

1. `handle_message(Message m)`:
   - Find all profiles that match the given message \( m \). Return this set of Profiles.
Chapter 4

Analysis of the State of the Art

In order to understand the mechanism and trade-offs of existing information filter systems, this chapter analyzes the methods used by such systems. An architecture capturing the respective approaches is derived. This architecture is then implemented in a test bed to study the effects on a real system and also to provide a reference point for the extensions proposed in the next chapters. As last part of the analysis, a cost model is established that highlights the main trade-offs.

4.1 Architecture

Figure 4.1 gives an overview of the architecture of an information filter, capturing the current state of the art. Such an information filter has three main components: (a) indexes, (b) merge, (c) postfilter. In addition, there is a queue that stores incoming...

![Information Filter Architecture](image-url)
messages while the filter is busy. In the following, the purpose of each component and suitable algorithms are described, completed by a mapping of existing information filters to the architecture.

Typically, there are several indexes for different kinds of predicates of the profiles. For instance, there could be an index for predicates on the "sender" attribute of an incoming message, and there could be a separate index for predicates on the "product" attribute. Each index takes an individual message as input and returns a set of matching profiles. Since a profile may involve several predicates, the sets of profiles returned by each index need to be merged. Logically, the merging step carries out conjunctions and disjunctions in bulk by a set union or intersection. The result of the merging step is a set of profiles that match the message according to all predicates that are indexed. Processing the merge step can be optimized in several ways: selectivity ordering, special representations of the result sets (e.g. bitmap) and low-level optimizations (e.g. prefetching, cache awareness). Since a profile can involve additional predicates that are not indexed, a postfilter step is necessary in order to evaluate those predicates.

4.1.1 Indexes

The most important method supported by an index is probe, which will be invoked by the IF's handle_message function. probe takes a message as input and returns a set of profiles that potentially match that message. Furthermore, an index provides insert and delete methods in order to register new profiles or delete existing profiles.

Indexing to speed up finding matching predicates has been used in traditional information filters. These have been subject to extensive studies in the literature [39, 44].

An index can be classified by four different aspects:

- **Target**: Value indexes index the constants and values of attributes. The B-Tree, R-Tree [51], R*-Tree [16], Spatial/Moving Object Indexes [70] and Interval Indexes [53] are popular examples of value indexes. On the other hand, structure indexes index the structure, i.e., the type of the profile; for XML messages, the structure is represented by XPath expressions in the profiles. Examples of structure indexes are YFilter [39] and Data Guide [11]. In the context of this thesis, the focus is on value indexes. Integrating structure indexes with context-aware filtering is future work.
4.1. Architecture

- **Accuracy:** Depending on the index accuracy, probing an index can result in false positives; i.e., an exact index returns exactly those profiles that match a given message. In contrast, a fuzzy index may return a superset of the profiles that match. False positives are then eliminated in the architecture of Figure 6.6 in a final postfilter step. By allowing false positives, the performance of index operations can be improved. The drawback is increased cost for postfiltering.

- **Dimensionality:** A single index might cover all predicates of all profiles. Alternatively, there could be several indexes: each index covering only those predicates that involve a certain set of attributes or even one index per attribute.

- **Scope:** Indexes are typically used to index all values of a given attribute. These indexes are full indexes. Alternatively, indexing can be limited to certain value ranges. These indexes are called partial indexes [76]. Partial indexes could, for instance, be used to index values that are rarely updated but not to index values that are updated frequently. Similarly, only the most selective attributes are indexed for performance reasons.

4.1.2 Merge

As mentioned in Section 3.2.2, profiles are conjunctions and disjunctions of predicates. Since it is not efficient to use a high-dimensional index to cover all conjunctions and disjunctions [44], an individual index typically only covers one type of predicate (e.g., predicates on a specific attribute of a message). Therefore, several potential indexes are probed in order to process a message [44,45]. In other words, the merge operation takes several intermediate result sets of profiles as input and carries out conjunctions and disjunctions on those sets of predicates. For instance, consider a message announcing a talk for 5PM; assume that there are two indexes that index all predicates concerning attributes type of message and time of events, respectively. Then, both of these indexes are probed to process the talk announcement message and the merge step carries out unions and intersections on the two resulting sets of profiles in order to determine those profiles that match the message in both regards (type and time).

One may argue that merging is not necessary; if only one index is used with full scope on all attributes and values is used. This index, however, would be of high dimensionality. It has been shown in previous work [53] that such high dimensional indexes do not scale well in the number of dimensions and are thus not attractive for information filtering.
Several optimization techniques for this merge operation exist. One idea is to optimize the order in which intersection and union operations are applied. Another class of optimizations involves the algorithms and data structures used to implement the intersect and union operations (bit maps, compression, early stop, iterators and block-wise operation).

4.1.3 Postfilter

The last step of processing a message eliminates false positives. This step is necessary if inaccurate indexes are used or if the merge operation does not involve all kinds of predicates. The postfilter operation takes a set of profiles as input and checks which profiles match the message by reevaluating the predicates of the profiles. Of course, short-circuit evaluation for conjunctions (early stop if a predicate is false) and disjunctions (early stop if a predicate is true) is allowed in order to speed up the postfilter operation. Nevertheless, depending on the number of profiles that need to be checked, this step can be expensive.

4.1.4 Mapping other Information Filters to this architecture

As an example on how this architecture applies to system developed outside of this thesis, we review two existing information filter systems: YFilter [39] and Le Subscribe [44].

YFilter matches XML documents against a subset of XPath. This subset includes the location steps of child ("/"), descendant-or-self("//"), wildcards, nested paths and also value predicates at text nodes and attributes. YFilter uses a shared NFA as the index stage to do path/structure matching, returning the identifiers of profiles matched by a path as well as the location steps that were matched. Nested paths are treated individually in the index and are merged afterwards. The handling of value predicates is done in a postfilter stage, where the predicate values are compared to the respective location steps in the matched paths.

Le Subscribe matches attribute/value pairs against profiles that are similar to the ones used in the context of this thesis. It uses several indexes that support adaptive multidimensional clustering to exploit skew over several dimensions. Since each profile is indexed in only a single index, there is no merge. The remaining, non-indexed attributes are handled in a cache-optimized postfilter step.
4.2 Implementation of the Test Bed

When developing the test bed for the state of the art methods and the extensions developed in this thesis, the following aspects had to be considered:

- The test bed is a main-memory system. Similarly to almost all existing information filter systems, high matching speeds for a large number of profiles are not possible when using disk-based systems.

- Two different types of indexes are used: 1) Interval Skip Lists [53] represent the current state of the art for main-memory interval indexing and are therefore used to support range predicates. 2) Hash Tables are used for equality predicates. Since Interval Skip Lists (ISL) are not so widely known as Hash Table, some more details are presented in section 4.2.1.

- The representation of profile sets (aka "marker sets" [53]) plays an important role as large sets need to be retrieved, intersected and enumerated. Thus different variants of marker sets are discussed in Section 4.2.2.

4.2.1 Interval Skip lists

An ISL is a randomized, hierarchical index structure applicable to all ordered domains (e.g., numerical values, dates, and strings). Each identifier of a profile is associated with one or more ranges of values. Furthermore, each range is associated with a set of identifiers. Ranges are organized hierarchically so that all ranges covering a given value can be found more quickly (logarithmic complexity in the average case). Figure 4.2 gives an example of an ISL. In this figure, the intervals $a := [2;25]$, $b := [16;20]$, $c := [8;12]$, $d := [5;5]$, $e := [-\infty;16]$ are indexed. An ISL is a linked list where each node carries the value of an interval boundary. Nodes have different height (aka "level"), following a random distribution so that there are many nodes with few levels,
Chapter 4. Analysis of the State of the Art

Method ISL.findIntervals
Input: Value k
Output: Set of matching intervals S
Internal variables: Node x  // Current node to inspect
                   Int i  // Current level to inspect

(1) x := ISL->header
(2) S := ∅
(3) // Start at top level, step down
(4) For i := ISL.maxLevel DownTo 1 Do
(5)   // Search forward on the current level as far as possible
(6)     While (x->forward[i]≠NULL and x->forward[i]->key < k) Do
(7)       x := x->forward[i]
(8)     // Add interval markers on edge when going down
(9)       S := S ∪ x->markers[i]
(10)    // Scan at lowest level to the location where the key is
(11)   While (x->forward[0]≠NULL and x->forward[0]->key < k) Do
(12)     x := x->forward[0]
(13)   // If a node with k is not found, use markers on the edge,
(14) // otherwise use markers on that node
(15)    If (x->forward[0]≠NULL and x->forward[0]->key < k)
(16)       S := S ∪ x->markers[i]
(17)    Else
(18)       S := S ∪ x->eqMarkers[i]

Figure 4.3: The findIntervals Algorithm

and few with many levels. At each level, the ISL has a forward edge pointing to the next node of at least the same height.
Interval markers for an interval x are attached to the nodes and edges that are covered by x, e.g. markers for the interval c [8,12] are attached to node 8, node 12 (since both are inside the interval) and the highest edge connecting these nodes, in this case the one at bottom level. To provide a more concise notation, markers at edges originating from a certain node on a given level are written node[level], here 8[0]. The markers on edges are always placed at the highest possible edge, e.g. the marker between node 8 and node 20 is placed at 8[2] instead of 8[1] and 16[1]. All algorithms of the ISL expect this invariant to be maintained. This invariant leads to a staircase pattern of markers for a given interval: The markers start a low level, gradually step up to a higher level and then descend again. Interval a gives a good example: The first marked edge is 2[1],
4.2. Implementation of the Test Bed

Figure 4.4: ISL – findIntervals for \( k=18 \)

<table>
<thead>
<tr>
<th>Step</th>
<th>( x )</th>
<th>Forward checked</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Header</td>
<td>( H[2] )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>8[2]</td>
<td>( {a} )</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>8[1]</td>
<td>( {a} )</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>16[1]</td>
<td>( {a,b} )</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>16[0]</td>
<td>( {a,b} )</td>
</tr>
</tbody>
</table>

Table 4.1: Steps Executed when Using findIntervals (Fig 4.3) in Figure 4.4

followed by 8[2] and 20[1].

The most important task in message matching is to find the profiles for a certain value. For range predicates, this task can be rewritten into finding all the intervals covering a specific value. The algorithm findIntervals efficiently supports this operation in an ISL. Figure 4.3 presents the pseudocode of this operation, while Figure 4.4 and Table 4.1 show an example of this function.

The key idea of findIntervals is to traverse the list at the highest possible level. This traversal stops when the edge that is currently inspected either ends at a node with the search value or goes beyond this value. Then the markers at the node covering the value or at the edge are added to the result. After that the search continues at the next lower level. These steps are repeated until the lowest level has been covered. In the example, bold lines mark the edges that are inspected, while dotted boxes represent the marker sets added to the result. The search for \( k=18 \) starts at the highest level of the header node, thus the edge \( H[2] \) is inspected. Since the node at the end of the edge (8) is smaller than 18, traversal continues at the same level, now with 8 as the starting point. Edge 8[2] overshoots (and thereby covers) the search value, so \( a \) becomes a member of the result \( S \). The algorithm advances with inspecting 8[1] next, and since 16
< 18, 16[1] is the next step, adding \( b \) to \( S \). 16[0] is inspected last. As this edge holds an empty marker set, \( S \) remains \( \{a, b\} \).

### 4.2.2 Marker Set Representation

Efficiently representing the sets of markers inside the indexes and during processing is fairly important, since a significant share of the memory and the processing cost is used for the marker operations. Marker set implementations need to support at least the functions `setMarker` (to make an interval member of the marker set) and `getMarker` (to check if an interval is member if the marker set). For efficiency reasons, bulk combination (and, or) and operations to find the next set marker are also required. We reviewed and implemented four types of marker set implementations:

1. **Uncompressed bit maps**: A bit sequence represents the (numeric) profile space. If the marker set contains profile \( n \), the \( n \)-th bit is set, otherwise it is not set. The big advantage of uncompressed bit map are computationally cheap (actually constant) `setMarker` and `getMarker` operations and efficient bulk combination if the marker set space is fairly randomly populated. The main drawback is the large space overhead on sparsely populated marker sets, leading to high memory requirements and also slowing down (computationally cheap) operations due to cache misses.

2. **Compressed bit maps**: A bit sequence is used to represent the marker set. If there are sufficiently long sub-sequences with the same value (0 or 1), they are compressed using run-length encoding (RLE). Thus the `setMarker` and `getMarker` operations become more expensive, but the memory overhead is decreased significantly for almost empty or almost completely filled marker sets.

3. **Tree-based sets**: The profile identifiers are inserted into a balanced tree. Compared to the bitmap approaches, this approach carries higher memory overhead (pointers in the tree) and slower operations (always logarithmic cost).

4. **Hash-based sets**: The profile identifiers are inserted into a hash table. While the cost of `setMarker` and `getMarker` are low (constant), bulk operations do not work well, since no order of identifiers is provided. For densely populated sets, the space overhead is also quite big.
In the test bed, compressed bit maps are used for the lowest level of the ISL, while uncompressed bitmaps are used for higher levels and intermediate results. This follows an the results initial performance study.

4.3 Cost Model

To provide a good understanding for the qualitative analysis of the current state of the art and the impact of the extensions, a cost model for information filters is established here. Table 4.2 summarizes the notation used in this section.

Let $M'$ denote the sequence of messages. Then, the total cost of processing $M'$ is given by the following formula:

$$C_{\text{total}}(M') = \sum_{m \in M'} C_{\text{hm}}(m)$$

$C_{\text{hm}}(m)$ denotes the cost for matching a message against profiles, i.e., a call to handle_message;

For reasons of simplicity, we assume that the cost of handling an individual message is about the same. If this assumption does not hold (as in Section 5), we can break the sequence into sub-sequences that fulfill the assumption, and add up the cost for those subsequences. Therefore, the formula above can be rewritten into:

$$C_{\text{total}}(M') = |M'| \cdot C_{\text{hm}}(m).$$

The cost of a handle_message operation is given by

$$C_{\text{hm}}(m) = C_{\text{ind-pr}}(m) + C_{\text{merge}}(m) + \text{res}(m) \cdot C_{\text{post-filter}}$$

Here, $C_{\text{ind-pr}}(m)$ denotes the cost for an index probe with the given message $m$, while $C_{\text{merge}}(m)$ denotes the cost of merging. The number of profiles that are returned by the first two stages is denoted with res$(m)$. $C_{\text{post-filter}}$ gives the cost for a single post-filter operation.

The cost of an index probe $C_{\text{ind-pr}}(m)$ can be further broken down. It is

$$C_{\text{ind-pr}}(m) = |\text{attidx}| \cdot \left( O\left( \text{val}_i \cdot \log(\text{val}_i) \right) + \frac{iwl \cdot C_{\text{bv}}}{C_{\text{tree-search}} + C_{\text{tree-retrieval}}} \right)$$
for tree-like index structures (eg. B-Tree, Interval Skip List) and

\[ C_{\text{ind-pr}}(m) = |attIdx| \cdot (C_{\text{hash lookup}} + C_{\text{bv}}) \]

for hash tables. The cost of index probes depends on the number of indexed attributes \(|attIdx|\) and also consists of two parts: the lookup of the search value and the retrieval of the results. While the first is usually considered in the design of indexes, the latter is fairly important in real-life systems, especially if a large number of profiles is used in small dimensions of indexed values. \(C_{\text{tree search}}\) has logarithmic complexity on the size of the dimension indexed. \(C_{\text{tree retrieval}}\) depends on the distribution of marker set over the index structure: if marker sets are on the leafs only (as in a B+-Tree), \(lvl\) is 1. If internal nodes also carry marker sets (as in ISLs) \(lvl\) is the height of the tree, i.e. \(\log(wal)\). Since hash tables do not have intermediate levels, \(lvl\) is always 1 and can thus be factored out. \(C_{\text{bv}}\) represents the cost of retrieving and combining the results from the index. For uncompressed bitmaps, \(C_{\text{bv}}\) is linear to the number of profiles \(P\), for compressed bitmaps it is between \(P\) and the size of the intermediate result. An important tuning issue arises from this cost formula: When setting the fanout on trees with \(lvl > 1\), the right balance between search cost \(C_{\text{tree search}}\) and retrieval cost \(C_{\text{tree retrieval}}\) needs to be found, trading off linear scanning at fanout steps and number of partial results to combine for a result.

The cost for merging can broken down into

\[ C_{\text{merge}}(m) = (|attIdx| - 1) \cdot C_{\text{bv}}. \]

The merging cost depends on the number of indexed attributes and the cost of result set combination (which is linear to the number of profiles).

The cost for postfilter can broken down into

\[ C_{\text{merge}}(m) = |attP| \cdot C_{\text{Comp}}. \]

Thus postfiltering is linear to number of attributes that need to be postfiltered, and the cost of doing a comparison operation (which is bound by CPU, cache and memory speed).

**Summary of the cost model** The main trade-off of information filters is to make \(res(m)\) small enough so that \(C_{\text{hm}}(m) \ll P \cdot C_{\text{post-filter}}, \) i.e. the cost of handling a message using an index is smaller than doing it by checking all the predicates individually.
4.3. Cost Model

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>number of profiles</td>
</tr>
<tr>
<td>$Att$</td>
<td>number of attributes</td>
</tr>
<tr>
<td>$M'$</td>
<td>sequence of messages</td>
</tr>
<tr>
<td>$U'$</td>
<td>sequence of updates</td>
</tr>
<tr>
<td>$m$</td>
<td>message event</td>
</tr>
<tr>
<td>$u$</td>
<td>update event</td>
</tr>
<tr>
<td>$C_{total}(M', U')$</td>
<td>total cost to process message sequence $M'$ and update sequence $U'$</td>
</tr>
<tr>
<td>$C_{hm}(m)$</td>
<td>cost for handle_message of $m$</td>
</tr>
<tr>
<td>$C_{uc}(u)$</td>
<td>cost for update_context of $u$</td>
</tr>
<tr>
<td>$C_{ind-pr}(m)$</td>
<td>cost for an index probe with message $m$</td>
</tr>
<tr>
<td>$C_{merge}(m)$</td>
<td>cost for merging the individual index results of $m$</td>
</tr>
<tr>
<td>$C_{bv}$</td>
<td>cost of combining two bit vectors with profiles</td>
</tr>
<tr>
<td>$C_{post-filter}$</td>
<td>cost of a post-filter operation</td>
</tr>
<tr>
<td>$C_{store-up}$</td>
<td>cost of a context store update</td>
</tr>
<tr>
<td>$P_{update}(u)$</td>
<td>probability that an index update occurs for update $u$</td>
</tr>
<tr>
<td>$C_{ind-up}$</td>
<td>cost of an index update</td>
</tr>
<tr>
<td>$res(m)$</td>
<td>number of profiles that match $m$ after index and merge stage</td>
</tr>
<tr>
<td>$attidx$</td>
<td>set of indexed attributes, $</td>
</tr>
<tr>
<td>$attpf$</td>
<td>set of attributes to be postfiltered, $</td>
</tr>
<tr>
<td>$val_i$</td>
<td>size of dimension/attribute $i$</td>
</tr>
<tr>
<td>$C_{Comp}$</td>
<td>cost of a comparison in the postfilter</td>
</tr>
</tbody>
</table>

Table 4.2: Notation for Cost Model in Section 4.3

A key strategy to achieving this is to index the most selective attributes. The number of indexed attributes must be chosen to balance $res(m)$ and the cost of $C_{ind-pr}(m) + C_{merge}(m)$. For most state of the art systems, there is no overlap of indexed attributes and postfilter attributes, as they use accurate indexes $attidx \cap attpf = \emptyset$.

A note on the complexity of $C_{bv}$: While uncompressed bitmaps are linear to $P$ (and thus seem to defy the benefits of logarithmic search), the constants play an important role here. Uncompressed bitmaps carry very low computational overhead, thus only with very small $res(m)$ the methods linear to the size of the intermediate result (compressed bit maps, value lists) can achieve the performance of uncompressed bitmaps.
Seite Leer / Blank leaf
Chapter 5

Batched Message Processing

5.1 Introduction

5.1.1 Background and Motivation

To support information filtering, the database community has made significant contributions on indexing profiles (aka triggers, rules, queries) that determine how to filter and route incoming messages or events [12, 39, 44]. The indexing schemes differ in the kind of messages (XML or structured tuples) and profiles (keywords, predicates, complex queries) they support. They have, however, the common goal to achieve scalability in the number of profiles that can be indexed. Scalability in this dimension is very important because each user, device, or software component can specify several hundred profiles in order to determine which information is relevant. Based on the past work on indexing, the purpose of this chapter is to examine techniques that improve the scalability with regard to the number of incoming messages that can be processed (i.e. throughput). Rather than processing each message individually, the idea is to reorder and group a set of incoming messages and process this set of messages instead. We call this approach batching and, in principle, it can be applied with any existing indexing scheme.

Batching has two advantages. First, batching improves the throughput of an information filtering system, as we will see, up to an order of magnitude in certain cases. Second, batching significantly improves the behavior of a system if the arrival of messages is bursty; it can be argued that batching works particularly well during peak times in which temporarily more messages arrive than the system can handle. On the negative side,
Chapter 5. Batched Message Processing

5.2 Batching Strategies

5.2.1 Architecture Extensions

batching makes it more difficult to predict the latency of a message (time from arrival to output of the message). However, there are ways to control batching in such a way that the maximum latency can be constrained in non-overload situations. Another issue is that batching may reorder messages for optimization reasons, which could result in processing an earlier message after a message that arrived later. If maintaining the original order is required, there are efficient ways to do this. All results shown in this thesis always restore this order.

Figure 5.1: Information Filter (IF) Architecture

Figure 5.2: Batch-Enhanced IF Architecture

Considering the architecture laid out in Section 4.1 as well as in Figure 5.1, the following changes are applied to enable batching, as shown in Figure 5.2: The *input queue* is now an integral part of the filter, as it needs to be controlled by a new component, the
### 5.2. Batching Strategies

**batch control.** Its task is to collect a set of messages from the input queue and pass it on to the next stages as a batch. In order to improve the efficiency of those next stages, it may reorder and group the messages. In contrast to the traditional approach, the indexes now handle a complete batch at once. Therefore they do not return the set of matching profiles for a single message, but a set containing the union of matching profiles for all messages in the batch (called union set from now on). Those union sets are now merged using the existing merge algorithm and finally split up into the matches for individual messages using an improved postfilter. The result contains all matching profiles for messages in the batch.

Batching is beneficial because the index is probed and the merge is carried out only once for a batch of messages rather than for each message individually. On the index, savings to handle messages in a batch stem from two sources: a) Testing identical values requires just a single access. b) Messages in a batch can be ordered to optimize the access pattern (depending on the index type), to reduce search time or improve I/O-operations. For example, the cache efficiency of a B+-Tree is improved by buffering, as shown in [88].

The drawback is that postfiltering becomes more expensive, as the union set contains more profiles than the individual sets. The key to a good overall performance therefore is to keep the number of profiles in the union set as low as possible while still forming sufficiently big batches.

There are two other possible improvements from batching: The first is to index the messages to speed up postfiltering, as messages fulfilling a certain predicate can be found faster. The second is to improve the delivery of messages. These two advantages will be explained in more detail in Sections 5.2.3 and 5.2.4.

#### 5.2.2 Grouping Messages

The goal of batching is to save work on batched profile index accesses and merge while having low number of profiles in the union set. Just naively handling the largest possible batch is not a good strategy to achieve this goal. Instead, the batch needs to be broken up into smaller subsets that have a greater amount of similarity. The need for this improvement becomes clear by considering the effects of batching on the union set: A large set of messages is likely to contain matches for a large number of profiles, perhaps even all. If the union set contains all messages, the effort of probing the indexes and merging is wasted. Instead, the postfilter has to do all the filtering and becomes very expensive. Small batch sizes, on the other hand, severely limit the room
Chapter 5. Batched Message Processing

For improvements on batching.

To achieve the needed amount of similarity, the subsets are grouped by similarity on those attributes that are used by the predicate indexes. Each group is now being processed as a "minibatch". Since those minibatches are much more homogenous than the original batch, the number of profiles in the union set of each minibatch is much lower, which in turn allows for a more efficient postfilter operation. Compared to handling the full batch in one piece, cost savings on the batched stages will be lower, and grouping also has a certain cost. Additionally, message ordering is not preserved. Nonetheless, grouping messages represents the key to higher performance, as our experiments will show.

To actually perform the grouping, we need a method that fulfills the following requirements:

- Reaching a good balance on postfilter cost and probe and merge savings.
- Being fast enough as not to impose an overhead when handling thousands of messages per second
- Handling varying batch sizes
- Taking advantage of skew in message values
- Working for different profile workloads

There are many ways to group. Here are some alternatives. We will examine their tradeoffs in Section 5.5:
5.2. Batching Strategies

**fixed-size:** The complete batch is ordered and split into minibatches of a fixed size (e.g. 2 messages), as shown in Figure 5.3a). This method will most likely work well if the overall batch size does not change much, and the values of the messages have a uniform distribution. For a varying overall batch size, this method is bound to generate too small minibatches on very large batches (thus distributing similar messages to different minibatches) and too large minibatches on very small batches (thus grouping different messages into the same batch).

**fixed-number:** The complete batch is ordered and split into a fixed number of (equi-sized) minibatches of the complete batch (e.g. 3 batches), shown in Figure 5.3, b). This method takes into consideration that the size of minibatches has some dependency on the overall batch size, but it does not take advantage of the distribution of the messages. In addition, a fixed fraction might not find the optimum for all batch sizes.

**value-based:** The batch is split into groups of the same value or a value range on certain attributes. In turn, there are several ways how to determine those values or values ranges:

- distribution of values: uniform, skewed
- number of groups: fixed, variable over time (perhaps with adaptive control)
- attributes to use for grouping: all indexed attributes, only the first indexed attribute, any number of attributes in between
- correlation to index values: independent, correlated to the index
- correlation to message values: independent, correlated to message values

Value based approaches have the potential of capturing the actual properties of messages and profiles to a higher degree than the previously presented approaches. A large number of variants makes it difficult to choose the right approach, however. The two examples in Figure 5.3 show grouping on identical values of the first attribute (c) and grouping on the identical values on two attributes (d).

**hybrid approaches:** None of the approaches might be suitable for all message and profile workloads, so combinations might provide better results:

- Splitting the batch into 50 minibatches, but making the minibatches no smaller than 10 messages and no larger than 500 messages.
Combining all messages differing not more than 50 from the designated group value, but ensuring that a minibatch has at least 20 messages in it, otherwise merging it with the closest group.

By using such hybrid approaches, it might be possible to handle corner cases without introducing too much complexity.

A common drawback of all these methods is that they have tuning parameters. The sensitivity of those tuning parameters is examined in Section 5.5.5. An approach in order to determine these parameters automatically is also presented in this section.

5.2.3 Indexing Messages

In Figure 5.4, an index is built on the values of the second attribute of the messages. This index can be used in order to determine the matching messages for each profile in the postfilter step. Depending on the predicates used in the profiles, different approaches can be taken. For equi-predicates, hashing can be used. For range predicates, a possible strategy is to sort the messages and carry out a binary search, as it is easy to implement and analyze. More elaborate index structures are also possible.

5.2.4 Delivery

Processing messages in a batch returns all matching profiles for all messages. Depending on the requirements of delivery, we can split up this result. The first method is to get all profiles for a single message, as done in traditional information filters (Figure
5.2. Batching Strategies

5.5 a). This is beneficial if the filter only "tags" the messages, forwards them over a shared channel to later stages that do the actual delivery. The second method is to get all messages for an individual profile (Figure 5.5 b). This is useful in a unicast situation with direct delivery, as each client receives all relevant messages in a single transfer.

5.2.5 Summary of Batching Strategies

Considering the ideas outlined in this section, we can classify batching strategies along three dimensions:

1. **How are messages batched?** The alternatives are: no batching (Unbatched), minibatching (MiniB), full batching (Full). (Section 5.2.2)

2. **Is a profile index (Plx) used in the first two stages?** The alternatives are yes and no.

3. **Is a message index (Mlx) used in the postfilter stage?** The alternatives are yes and no. (Section 5.2.3).

Not all combinations within these three dimensions are either possible or useful: For example, in the "unbatched" cases, message indexing is impossible. Also not using a predicate index for "unbatched" will severely limit the performance. The combination of minibatching and message indexing without a predicate index has detrimental effects, as it actually reduces the efficiency of message indexing. Finally, the combination of full batching, profile indexing and message indexing will not be considered, as the resulting performance is not competitive.

Therefore, we will compare the following, detailed approaches in the rest of the chapter

**Unbatched/Plx−:** baseline, representing the current state of the art
Full/Plx/-: Apply a full batch at the profile indexes

Full/-/Mlx: Use only a message index to determine its efficiency (no profile index)

MiniB/Plx/-: Split batch into minibatches and apply them to the profile indexes

MiniB/Plx/Mlx: Generate message indexes on the individual minibatches to speed up postfiltering

5.3 Implementation Details of Batching

The largest modifications necessary for batching need to be done on the implementation of the access methods of the profile indexes. This section will now show in more detail how these modifications help to improve performance.

To understand the changes, first consider that any predicate index (regardless of the actual index type) stores profile identifiers matching certain values. Depending on the type of index structure, profiles for each value are stored on one location or spread over several places. If a probe is performed, the index is queried and returns this set, either directly or by combining the partial sets. Under most circumstances, the lookup itself can be performed at low cost, but handling possibly large sets of profile identifiers is expensive.

A first improvement can be done if the batch is arranged in a way so that identical values are next to each other, e.g. by lexicographically sorting the set. Using this arrangement, it is easy and cheap to perform only a single lookup for identical values.

If there are not only identical values in a batch, our strategy depends on the properties of the index structure. For indexes that do not have the notion of order or containment (such as hash tables), we are forced to perform a lookup for each distinct value and make the union of those results to retrieve the union result for the batch. If the index does in fact support such an order (such as B-Trees, or Interval Skip Lists) or containment (R-Trees), one can take advantage of it. The probed values in the batch need to be arranged in the corresponding pattern. Using this pattern to access the index inside the batch, the search cost can be reduced by continuing at the last probed value and also the cost of generating the result set by building it incrementally.

Since interval skip lists (ISL) are used in the test bed, the algorithm and implementation details for this data structure are shown here. For a conventional probe, the index is searched following the forward pointers in order to retrieve all intervals covering a
probed value. Starting from the highest level, the algorithm determines all pointers covering the probed value. The union of the intervals at that pointers form the result. Details and an example for this algorithm can be seen in Section 4.2.1.

A batched probe can continue its traversal by starting the search at the lowest possible forward pointer that had not been covered before. Only the marker sets on the newly examined edges need to be added to the result. Figure 5.6 shows the pseudocode for the method findIntervalUnion that extends the existing method to retrieve the union of matches for a set of probe values \( K \). The mode of operation of this algorithm and its benefits over the conventional approach can be seen in the following example: Figure 5.7 shows an ISL on which the values 9, 10, 11, 13, 16, 16, 17 and 23 shall be probed. The access pattern for repeated calls to the existing findInterval methods are shown in Table 5.1. For each key, the search starts at the highest level of the header, following by an examination of the forward edges and the copying of markers. In this example, the inefficiency of the conventional approach becomes very clear: For example, \( H[2] \) is traversed eight times, while the marker set at \( s[2] \) is copied seven times. On the other hand, the correct result (not including any spurious matches) is returned for each probe.

In contrast, Table 5.2 illustrates the access pattern for a single call to findIntervalUnion with the sorted sequence \( K[9, 10, 11, 13, 16, 16, 17, 23] \) as parameter. This table also shows the state of the handled variable, which represents how far the probe has proceeded and, thus, where to continue probing. Figure 5.8 presents a "snapshot" of this progress when processing the value 13. As Table 5.2 indicates, the access to the first key (9) is identical to the the one shown in Table 5.1, yielding a result \( S \) of \( \{a, c, e\} \). Since 10 and 11 share all the edges with 9, there is no need to process them any more. For 13, the search starts at 12[0], and this is also the only edge that needs to be examined. Since this edge carries an empty marker set, \( S \) remains unchanged. For the first probe value of 16, only the node 16 needs to be inspected, adding \( b \) to \( S \). For 17, probing continues on 16[1], and for 23 on 20[2]. As one can see from these results, each forward edge is only examined once. As a consequence, the marker sets on those edges are only copied once. There are, however, false positives for the individual results which need to be postfiltered.
Method ISL.findIntervalsUnion

Input: Array of Value K
Output: Set of matching intervals S
Internal variables: Node x // Current node to inspect
                  Array of Edge handled // last edges already handled
                  Int i // Current level to inspect
                  Int idx // Current array key of K

1. S := 0
2. Set handled to the forward edges of ISL->header
3. Set idx so that all search values < handled[0]→key are skipped
4. While (idx < K.length) Do
5.   // Go up as far as needed to reach new nodes
6.   Set l so that handled[l+1]→key > K[idx]
7.   x := handled[l]
8.   // Continue traversal
9.   // Check current node
10. If x→key = K[idx]
11.   S := S U x→eqMarkers[i]
12.   idx := idx+1
13. Skip all search keys with the same value
14. Else // continue searching
15.   While l≥0
16.      While(x→forward[i]≠NULL and x→forward[i]→key < K[idx]) Do
17.         x := x→forward[i]
18.      If x→key ≠ dK[idx]
19.         S := S U x→markers[l]
20.         handled[l] := x→forward[l]
21.      If l = 0
22.         Skip all search values < x→key
23.         // These values are already covered by the edge
24.      Else
25.         S := S U x→eqMarkers[i]
26.         idx := idx+1
27.         handled[l] := x→forward[l]
28.         Skip all identical keys
29.      l := l-1

Figure 5.6: The findIntervalsUnion Algorithm
5.3. Implementation Details of Batching

Figure 5.7: Batched Probe on ISL: Before Start

<table>
<thead>
<tr>
<th>Key</th>
<th>Start: $x[i]$</th>
<th>Forwards examined</th>
<th>Markers copied</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>H[2]</td>
<td>H[2], 8[2], 8[1], 8[0]</td>
<td>8[2], 8[1], 8[0]</td>
<td>${a, c, e}$</td>
</tr>
<tr>
<td>10</td>
<td>H[2]</td>
<td>H[2], 8[2], 8[1], 8[0]</td>
<td>8[2], 8[1], 8[0]</td>
<td>${a, c, e}$</td>
</tr>
<tr>
<td>11</td>
<td>H[2]</td>
<td>H[2], 8[2], 8[1], 8[0]</td>
<td>8[2], 8[1], 8[0]</td>
<td>${a, c, e}$</td>
</tr>
<tr>
<td>13</td>
<td>H[2]</td>
<td>H[2], 8[2], 8[1], 8[0], 12[0]</td>
<td>8[2], 8[1]</td>
<td>${a, e}$</td>
</tr>
<tr>
<td>16</td>
<td>H[2]</td>
<td>H[2], 8[2], 8[1], 16[1]</td>
<td>8[2], 16</td>
<td>${a, b}$</td>
</tr>
<tr>
<td>16</td>
<td>H[2]</td>
<td>H[2], 8[2], 8[1], 16[1]</td>
<td>8[2], 16</td>
<td>${a, b}$</td>
</tr>
<tr>
<td>17</td>
<td>H[2]</td>
<td>H[2], 8[2], 8[1], 16[1], 16[0]</td>
<td>8[2], 16[1], 16[0]</td>
<td>${a, b}$</td>
</tr>
<tr>
<td>23</td>
<td>H[2]</td>
<td>H[2], 8[2], 20[2], 20[1], 20[0]</td>
<td>20[2], 20[1], 20[0]</td>
<td>${a}$</td>
</tr>
</tbody>
</table>

Table 5.1: Steps Executed when Using ISL findIntervals (Fig 4.3) in Figure 5.7

Figure 5.8: Batched Probe on ISL: Before Handling Search Value 13
### Table 5.2: Steps Executed when Using ISL findIntervalsUnion (Fig 5.6) in Figure 5.7

<table>
<thead>
<tr>
<th>Key</th>
<th>Start: ( x[i] )</th>
<th>Forwards examined</th>
<th>Markers copied</th>
<th>Handled</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>( H[2] )</td>
<td>( H[2], 8[2], 8[1], 8[0] )</td>
<td>( 8[2], 8[1], 8[0] )</td>
<td>( [12,16,20] )</td>
<td>( {a, c, e} )</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( {a, c, e} )</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( {a, c, e} )</td>
</tr>
<tr>
<td>13</td>
<td>( 12[0] )</td>
<td>( 12[0] )</td>
<td>( 12[0] )</td>
<td>( [16,16,20] )</td>
<td>( {a, c, e} )</td>
</tr>
<tr>
<td>16</td>
<td>( 16[1] )</td>
<td></td>
<td>( 16 )</td>
<td>( [16,16,20] )</td>
<td>( {a, b, c, e} )</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( {a, b, c, e} )</td>
</tr>
<tr>
<td>17</td>
<td>( 16[1] )</td>
<td>( 16[1], 16[0] )</td>
<td>( 16[1], 16[0] )</td>
<td>( [20,20,20] )</td>
<td>( {a, b, c, e} )</td>
</tr>
<tr>
<td>23</td>
<td>( 20[2] )</td>
<td>( 20[2], 20[1], 20[0] )</td>
<td>( 20[2], 20[1], 20[0] )</td>
<td>( [25,25,\text{NULL}] )</td>
<td>( {a, b, c, e} )</td>
</tr>
</tbody>
</table>
5.4 Extension of the Cost Model

5.4.1 Overview

The cost of processing a batch of messages can be measured in different units, but the most relevant units are

- CPU, as this affects the throughput
- memory, as it limits the scalability in terms of profiles or batch sizes
- response time/latency, as batching introduces “waiting times”.

This chapter will focus on an analysis of CPU and memory cost, as they are related to the batching methods. Latency estimates and methods to deal with it are described in Section 5.6, since they depend more on the traffic characteristics than on one of the different batching strategies.

5.4.2 CPU Cost

The cost model presented for existing information filters (Section 4.3) is not sufficient to cover batched processing, and therefore needs to be extended. Table 5.3 summarizes the notation used in this section. For simplicity, we assume that there is a single batch over \( M' \), so that the batch size \( bs = |M'| \)

For the existing model, the cost of handling this sequence of messages is

\[
C_{\text{total}}(M') = bs \cdot C_{\text{ind-pr}}(m) + bs \cdot C_{\text{merge}}(m) + bs \cdot \text{res}(m) \cdot C_{\text{post-filter}}
\]

**Full batching** introduces the cost to sort the batches: \( C_{\text{sort}}(M') \), and uses a single, batched index access \( C_{\text{batch-pr}}(bs) \). Thus, the cost of handling this sequence of messages is

\[
C_{\text{total}}(M') = C_{\text{sort}}(bs) + C_{\text{batch-pr}}(bs) + C_{\text{merge}}(mb) + bs \cdot \text{res \_ batch}(M') \cdot C_{\text{post-filter}}
\]

where \( C_{\text{merge}}(mb) \) is the cost of merging on the result of the complete batch and \( \text{res \_ batch}(M') \) is number of profiles matched in the batched probe.
The cost of the initial sorting can be expressed as

\[ C_{\text{sort}}(bs) = n_{\text{attidx}} \cdot bs \cdot \log(bs) \]

The cost of the batched probe on tree-structured indexes can be expressed as

\[ C_{\text{batch-pr}}(bs) = O(\log(val_i)) + bs \cdot lvl \cdot (1 - \text{overlap}) \cdot C_{bv} + (bs - 1) \cdot C_{\text{traversal}} \]

For the first value the location in the index must be searched, requiring logarithmic cost (based on the size of the dimension).

For all elements in the batch the marker sets need to be retrieved, if the have not yet been added. The cost of retrieval for a given value depend on the cost of retrieving an individual marker set \( C_{bv} \), the distribution of marker set over the index \( lvl \) and the amount of overlap \( \text{overlap} \) of the marker sets of two neighboring elements of the batch.

In turn, the cost of index traversal to the location of the next element in the batch \( (C_{\text{traversal}}) \) depends on the index structure and the distribution of values in the batch and in the index. In the worst case, it can be linear to \( val_i \), in the best case it is 0 (identical value of two neighbouring values in the batch).

Sorting the batching causes in fact superlinear cost in the number of messages, but in practice \( \log(bs) \) is small, and the constants of sorting are small compared to the constants of the rest (especially the retrieval). Similarly to \( \text{res(m)} \) in existing information filters, \( \text{res\_batch} \) has the largest impact on the overall cost. It is influenced by global selectivity, batch size and the similarity of message values. To perform the separation of the batch result into the individual results, postfiltering needs to consider the indexed attributes as well, meaning that \( \text{attidx} \subseteq \text{attpf} \).

For Minibatching, the cost of the initial lexicographical sort to prepare the minibatches needs to be added. For each minibatch, the cost formula is essentially the same as for Full Batching, but on a smaller batch size. Therefore the cost model for minibatching is:

\[
C_{\text{total}}(M') = C_{\text{sort}}(bs) + \sum_{mb_i \subseteq M'} (C_{\text{sort}}(|mb_i|) + C_{\text{batch-pr}}(|mb_i|) + C_{\text{merge}}) + |mb_i| \cdot \text{res\_batch}(mb_i) \cdot C_{\text{post-filter}}
\]

In general, the same arguments as for Full Batching hold true, but the most important part is that \( \sum_{mb_i \subseteq M'} \text{res\_batch}(mb_i) \ll \text{res\_batch}(M') \). Therefore, postfiltering cost is much lower for minibatching.
Summary of the Cost Model  The two main cost drivers for unbatched processing are operations on profile indexes and postfiltering. They continue to do so if batching is used. There are, however, changes on the individual cost drivers: For profile index operations and merge, clear savings can be achieved by processing a batch rather than each message individually. The actual savings on handling a batch of messages depends on the index structure and the message distribution in the batch. The more skewed the batch is, the higher the similarity of the messages in the batch will be, making batching more effective.

For postfiltering, the cost model does not change significantly, but the higher number of match candidates in the union set may have a negative impact. Using a message index on one attribute reduces the cost of evaluating this attribute from $O(\text{num\_msg})$ to $O(\log(\text{num\_msg}))$.

Although grouping and index building operations could have a superlinear cost, their overall cost is much smaller than the possible savings on the other operations.

5.4.3 Memory Requirements

To handle batches of messages, we need additional memory over what a unbatched information filter would need. The factors contributing to those memory requirements are:

1. storing the messages in the batch
2. indexing, grouping the messages
3. representing results for each message in the batch.

The first and the last are clearly linear to the number of messages, while the second can additionally cause logarithmic overhead. Considering that on high-throughput scenarios batches can easily consist of thousands or tens of thousands of messages, the available memory can become the limiting factor. In practice, most of the memory requirements very much depend on implementation and workload specifics and can be tuned accordingly.
### Symbol Definition

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P)</td>
<td>number of profiles</td>
</tr>
<tr>
<td>(Att)</td>
<td>number of attributes</td>
</tr>
<tr>
<td>(M')</td>
<td>sequence of messages</td>
</tr>
<tr>
<td>(U')</td>
<td>sequence of updates</td>
</tr>
<tr>
<td>(m)</td>
<td>message event</td>
</tr>
<tr>
<td>(u)</td>
<td>update event</td>
</tr>
<tr>
<td>(C_{\text{total}}(M', U'))</td>
<td>total cost to process message sequence (M') and update sequence (U')</td>
</tr>
<tr>
<td>(C_{\text{hmm}}(m))</td>
<td>cost for handle_message of (m)</td>
</tr>
<tr>
<td>(C_{\text{uc}}(u))</td>
<td>cost for update_context of (u)</td>
</tr>
<tr>
<td>(C_{\text{sort}}(mb))</td>
<td>cost of sorting a message batch (mb)</td>
</tr>
<tr>
<td>(C_{\text{ind-pr}}(m))</td>
<td>cost for an index probe with message (m)</td>
</tr>
<tr>
<td>(C_{\text{batch-pr}}(mb))</td>
<td>cost for probing a batch of messages (mb)</td>
</tr>
<tr>
<td>(C_{\text{merge}}(m))</td>
<td>cost for merging the individual index results of (m) (can be a result of single probe or a batch probe)</td>
</tr>
<tr>
<td>(C_{\text{post-filter}})</td>
<td>cost of a post-filter operation</td>
</tr>
<tr>
<td>(C_{\text{store-up}})</td>
<td>cost of a context store update</td>
</tr>
<tr>
<td>(p_{\text{update}}(u))</td>
<td>probability that an index update occurs for update (u)</td>
</tr>
<tr>
<td>(C_{\text{ind-up}})</td>
<td>cost of an index update</td>
</tr>
<tr>
<td>(C_{\text{traversal}})</td>
<td>cost to find the next relevant node</td>
</tr>
<tr>
<td>(\text{res}(m))</td>
<td>number of profiles that match (m) after index and merge stage</td>
</tr>
<tr>
<td>(\text{res_batch}(b))</td>
<td>number of profiles that match any (m) in (b) after index and merge</td>
</tr>
<tr>
<td>(\text{overlap})</td>
<td>ratio of overlap in the marker sets in an index</td>
</tr>
<tr>
<td>(\text{attidx})</td>
<td>set of indexed attributes, (</td>
</tr>
<tr>
<td>(\text{attpf})</td>
<td>set of attributes to be postfiltered, (</td>
</tr>
<tr>
<td>(\text{val}_i)</td>
<td>size of dimension/attribute (i)</td>
</tr>
<tr>
<td>(C_{\text{Comp}})</td>
<td>cost of a comparison in the postfilter</td>
</tr>
</tbody>
</table>

Table 5.3: Notation for Cost Model in Section 5.4.2
5.5 Performance Experiments and Results

Following the analysis in the last section, our goal is to measure the throughput improvements of the different strategies on various workloads, in order to determine the best strategy. Variants and tuning parameters for this method are also analyzed.

5.5.1 Experimental Setup

To validate our analysis, we extended our already existing C++ implementation of the information filter architecture with the new batching components, and modified the existing components accordingly. To index range queries, we use an interval skip list. The implementation we use was taken from Hanson's web site [53], and slightly modified to fit into our architecture. For point queries, we use a hash table implementation from the GHT library [4]. Both were extended as described in Section 5.3. For sorting values, we used the built-in \texttt{qsort()} method of the C runtime library. Overall, we needed about 3K lines of code to add the batch-related functionality.

The experiments were conducted on a Pentium 4 3.2 GHz with 2 GB of RAM running Linux 2.4. The programs were compiled using the standard GCC 3.3 provided by the Linux distribution.

5.5.2 Workloads

We created the following profile and message workloads and tested them against the individual batching strategies:

1. Range Queries (RQ): Profile consist of range queries. There are three different distributions of message values:
   (a) Zipf (RQ-Z),
   (b) Gaussian (RQ-G),
   (c) Uniform (RQ-U).

2. Point Queries PQ. Profiles consist of point queries, message values have Zipf distribution.
### Table 5.4: Workload Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>No of profiles</td>
<td>500K</td>
</tr>
<tr>
<td>AR</td>
<td>No of attr. RQ</td>
<td>8</td>
</tr>
<tr>
<td>AttPR</td>
<td>No of attr./profile RQ</td>
<td>8</td>
</tr>
<tr>
<td>AttMR</td>
<td>No attr. per msg (RQ)</td>
<td>8</td>
</tr>
<tr>
<td>AP</td>
<td>No of attr. PQ</td>
<td>32</td>
</tr>
<tr>
<td>AttMQ</td>
<td>No attr. per msg (PQ)</td>
<td>32</td>
</tr>
<tr>
<td>AttPP</td>
<td>No of attr./profile PQ</td>
<td>4</td>
</tr>
<tr>
<td>CO</td>
<td>No of indexed attr.</td>
<td>2</td>
</tr>
<tr>
<td>PD</td>
<td>distribution of profile pred.</td>
<td>uniform</td>
</tr>
<tr>
<td>VRR</td>
<td>Values range (RQ)</td>
<td>[0,10000]</td>
</tr>
<tr>
<td>VRP</td>
<td>Value range (PQ)</td>
<td>[1,35]</td>
</tr>
<tr>
<td>MD</td>
<td>Dist. of msg values</td>
<td>Zipf, uniform, gauss</td>
</tr>
<tr>
<td>BS</td>
<td>Batch size</td>
<td>1 - 100K</td>
</tr>
<tr>
<td>MBS</td>
<td>Minibatch size (number)</td>
<td>1 - BS/2</td>
</tr>
<tr>
<td>MBS</td>
<td>Minibatch size (range)</td>
<td>1 - 1250</td>
</tr>
</tbody>
</table>

In detail, the workload parameters were determined this way: For **RQ**, we used 8 possible attributes (AR), all of which were used in all profiles (AttPR) and messages (AttMR). The profile values were floating point numbers quantified to 3 significant digits. The value range (VRR) was 0 to 10000, corresponding to the experiments in Hanson's Skiplist work [53]. The selectivity of the attributes varied from about 3 percent for the most selective attribute to about 100 percent for the least selective (dummy) attribute. The actual overall selectivity (measured as percentage of profiles matching a document) varied from 0.13 percent (RQ-Z) to 0.15 percent (RQ-G); RQ-U falls in the middle with 0.147 percent. Expressed into other terms, this means that about 700 profiles matched each message, or a profile was matched by every 700th message. Message values were distributed uniformly over VD for **RQ-U** and the attributes of the other message workloads, if not determined otherwise. For **RQ-Z**, 50 values were drawn uniformly from VRR for the first four attributes (MD). The actual values for the attributes were then chosen from a Zipfian distribution over those values. For **RQ-G**, the first two attributes were drawn using a Gaussian distribution of median 5000 and standard deviation 250 and 350, respectively.
5.5. Performance Experiments and Results

Figure 5.9: Comparing Batching Strategies - Range Profiles - RQ-Z (Zipf) Message Distribution

For PQ, we used 32 possible attributes (AP), which were all used in the messages (AttMP). Profiles consisted of 4 attributes (AttPP), two of which were present on all profiles, the remaining two were chosen from the other 28. The values (VRP) were chosen from a range between 1 and 35 and quantified as integers, closely following the workload in [44]. The Zipfian distribution on all message attributes was based on all values. The selectivity was much lower. Using the Zipfian message workload, it is about $2.6 \times 10^6$, corresponding to 1.3 matching profiles per message or a profile being match by every 380,000th document.

All workloads consisted of 500K profiles (P). In both cases, indexing the two most selective attributes yielded the best performance for the unbatched case (CO). All profile values had a uniform value distribution (PD).

Batch sizes were varied from 1 to 100K messages per batch (BS). For all experiments, the minibatches within a single batch had either an equal number of messages or covered the same range of values (MBS). For minibatch sizes based on the number of messages, we varied the number of messages in minibatch between 1 and half the actual batch size. For minibatch sizes based on values, we changed the value range from 1 to 1250, as bigger values did not yield any benefits.
5.5.3 Comparison of Batching Strategies

Impact of batch size

The purpose of our first set of experiments is to compare the individual batching methods and determine their respective benefits. For each of these experiments we measured the maximum throughput the filter could handle for various batch sizes between 1
5.5. Performance Experiments and Results

and 10K. The unbatched method (Unbatched/Plx/-) represents the current state of the art and was compared against the batching methods presented in Section 5.2.5.

For the minibatching approaches, we chose a value-based partitioning; more details about it will be shown next section. The experiments were done with all the workloads defined in the last section. Figures 5.9, 5.10 5.11 and 5.12 show the results.

On RQ-Z (Figure 5.9), unbatched reaches a throughput of about 450 msg/sec. Full batching with Plx is not able to achieve a comparable performance; it quickly degrades when increasing the batch size, reaching just 20 msg/sec when the batch size exceeds 5000 messages. This drop in performance can be attributed to the very high number of profiles in the union set after the batched index operations, putting all the filtering work onto the postfilter.

Full batching using only a message index has very low performance on very small batch sizes (15 msg/sec at BS 1), but performs better at higher batch sizes. At a batch size of around 1250, the throughput is 50 percent higher than in the unbatched case (about 675 msg/sec). When further increasing the batch size, the performance begins to drop slightly, and falls below the unbatched performance at a batch size of 5000. The low performance at smaller batches and the increase at larger batch sizes correspond well to the expected behavior of index, as its efficiency increases with amount data indexed. When the batches get even larger, the batches do not fit into the L2 cache of the processor anymore, therefore the performance gradually declines. We verified this using the cache profiler valgrind [81].

Minibatching with a profile index (MiniB/Plx/-) shows a significant performance improvement over all the other approaches. At a batch size of 150, the performance is already twice as high as for unbatched. The gap grows with an increasing batch size: at 10000 messages the speedup is about 7.5. Bigger batch sizes result in a higher number of similar messages in the batch, which can be grouped together, and thus to higher throughput. Grouping does not add significantly to the overall cost, as the cost breakdown in the next section will show.

Finally, using message index in addition to minibatching and profile indexes (MiniB/Plx/MIx) leads to relatively similar results: On smaller batch sizes, the performance is slightly worse, for larger batch sizes, this approach catches up and matches (MiniB/Plx/-). This behavior can be explained by looking at the results of full batching with message index. Considering that the message index is now built and queried on the minibatches, the same pattern occurs again: On small minibatches, the efficiency of the message index is too low, when the sizes increases, the results become better.

Looking at those results, it can be easily concluded that the combination of full batching with profile and message indexes will not deliver competitive performance and is there-
The results for RQ-G (Figure 5.10) and RQ-U (Figure 5.11) confirm our results, but deliver lower overall gains for minibatching. With RQ-U, the maximum speedup we can achieve is around 2, with RQ-G it is around 6.5. In both cases, the approach of using a message index with minibatching and the profile indexes shows its benefits earlier.

PQ (Figure 5.12) completes the picture of minibatching on profile indexes (in some instances also with a message index) being the best approach. While unbatched already performs much better (2300 msg/sec) due to the cheaper index operations, minibatching is still able to improve this result by more than a factor of 4.

For the rest of this section and the next sections, we will use RQ-Z as reference workload.

Cost Breakdown

The performance numbers of the respective strategies are verified when looking at the individual cost factors contributing to the performance. For each of the above methods, we measured the time for sorting to group the minibatches, sorting to build the message indexes and sorting to optimize the index access pattern. We also measured the cost of accessing the predicates indexes, merging the results and accessing the message indexes. Finally we measured the time to postfilter the remaining profiles. For reasons of space, we only show the breakdown for the RQ-Z. Table 5.5 shows the cost factors in seconds for a batch size of 75, while table 5.6 shows them for a batch size of 5000. As
5.5. Performance Experiments and Results

<table>
<thead>
<tr>
<th></th>
<th>Ub./P/</th>
<th>Full/P/-</th>
<th>Full/-/M</th>
<th>Mini/P/-</th>
<th>Mini/P/M</th>
</tr>
</thead>
<tbody>
<tr>
<td>SortMB</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>SortPlx</td>
<td>0</td>
<td>0.01</td>
<td>0</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>Plx</td>
<td>14.28</td>
<td>1.97</td>
<td>0</td>
<td>5.51</td>
<td>7.87</td>
</tr>
<tr>
<td>SortMlx</td>
<td>0</td>
<td>0</td>
<td>0.01</td>
<td>0</td>
<td>0.01</td>
</tr>
<tr>
<td>Mlx</td>
<td>0</td>
<td>0</td>
<td>71.58</td>
<td>0</td>
<td>12.98</td>
</tr>
<tr>
<td>PostF</td>
<td>6.54</td>
<td>77.58</td>
<td>28.76</td>
<td>8.27</td>
<td>6.08</td>
</tr>
</tbody>
</table>

Table 5.5: Cost Breakdown of 5.9 BS 75

<table>
<thead>
<tr>
<th></th>
<th>Full/P/-</th>
<th>Full/-/M/</th>
<th>MiniB/P/-</th>
<th>MiniB/P/M</th>
</tr>
</thead>
<tbody>
<tr>
<td>SortMB</td>
<td>0</td>
<td>0</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>SortPlx</td>
<td>0.01</td>
<td>0</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>Plx</td>
<td>0.05</td>
<td>0</td>
<td>0.73</td>
<td>0.95</td>
</tr>
<tr>
<td>SortMlx</td>
<td>0</td>
<td>0.01</td>
<td>0</td>
<td>0.01</td>
</tr>
<tr>
<td>Mlx</td>
<td>0</td>
<td>15.91</td>
<td>0</td>
<td>2.83</td>
</tr>
<tr>
<td>PostF</td>
<td>462.79</td>
<td>14.9</td>
<td>3.29</td>
<td>2.45</td>
</tr>
</tbody>
</table>

Table 5.6: Cost Breakdown of 5.9 BS 5K

Both tables show, the sorting cost is almost negligible, prompting that more expensive grouping methods might be possible. As expected, the profile index operations (combined with merging) are reduced drastically by the batching methods. For Full Batching with profile index (Full/Plx/-), however, this reduction is overcompensated by the much higher postfiltering cost even on small batch sizes, resulting in a noncompetitive overall performance. For full batching with just a message index, one can see its low efficiency for small batch sizes by looking at the time it takes to do the index probes. As the batch size increases, the performance also improves. For (MiniB/Plx/-), the savings on the predicate index operations are not as big as seen on the full batching case, but the cost of postfiltering is low, with almost no increase over the unbatched case. At both batch sizes, using a message index incurs an additional cost, and does not help in speeding up the overall performance. For larger batch sizes, however, the investment into the message index pays off.

Scaleup for Larger BS

In this experiment, we further increased the batch size up to 100K messages (table 5.7), again using RQ-Z. We compared the methods based on minibatching (which show
the best performance) to Unbatched/Plx/- and observed that the speedup increased with a larger batch size. Furthermore, the additional message index is becoming more important with a growing batch size. In the extreme case, MiniB/Plx/Mix is faster than traditional unbatched processing by a factor of 15 and 15 percent faster than MiniB/Plx/-. 

5.5.4 Comparing Minibatching Methods

Following the discussion in Section 5.2.2, we compared the following four simple grouping methods to determine their suitability using RQ-Z: a) fixed number of messages (Fix No), b) fixed value range on the first attribute (Val Att 1), c) fixed (identical) value range on both indexed attributes (Val Att 2), d) fixed value ranges on both indexed attributes (second with bigger range than first) (Val Att 2+). All methods require the messages to be sorted lexicographically on the indexed attributes, as proposed in Section 5.3. Figure 5.13 shows the performance of the competing approaches. (Fix No) performs relatively well and beats both naive value-based approaches. Grouping based just on the values of the first attribute (Val Att 1) performs well for smaller batches, but levels off very soon, as the resulting minibatches get too large. In contrast, grouping on both attributes with the same value range (Val Att 2) tends to create too small minibatches,
which leads to generally lower performance, especially on smaller batch sizes. Based on these observations, the hybrid approach of using a bigger value range on the second attribute (Val Att 2+) looks most promising. As the graph shows, this is in fact the best approach, though not by a large margin. For the other message distributions, the differences are even smaller, thus making it impossible to declare a clear winner.

### 5.5.5 Sensitivity Analysis

As we stated in the previous sections, balancing the influences of message distribution, profile distribution and batch size to get the best grouping results requires tuning. The relevant parameter is either the number of messages per minibatch or the value range per minibatch. Figure 5.14 shows the effects of varying those on an overall batch size of 10000 for the previous experiment. For both approaches, the overall result is the same: A single throughput maximum can be determined from performance declines when increasing or decreasing the minibatch size. The approach grouping a fixed number of messages has lower overall maximum, and a lower minimum. Its advantage is that the decline is very gradual. The value-grouping approach, on the other hand, shows a very steep decline once the best possible size has been exceeded. Since it yields the better performance, it is nonetheless preferable.

Since both methods do not show any local minima, a machine learning technique like gradient descent can be applied to reach the best size. Slightly simplified, this means that if we decrease cost by changing the minibatch size into one direction (e.g. decreasing), we can further improve it by further changing it into the same direction until the cost increases. If the cost increases at the first attempt, we have to change our...
direction or we are already at the cost minimum.

5.5.6 Summary

Batching provides significant throughput enhancements, even under circumstances where messages have a limited amount of similarity. Increasing similarity and batch size further improve the result. The comparison of batching strategies shows Minibatching to be the clear winner for all message and profile workloads, with some additional improvements using message indexing. Both number- and range-based methods perform well, with slight advantages for the range-based approach. The more important factor is the actual size of a minibatch.

5.6 Latency Analysis

Latency is the time that it takes to process a message from the point where the message is put into the input queue until the point the message is ready for delivery to the individual profiles. The complete latency of processing a message or a batch of message would also have to include delivery. Since delivery depends on the particular architecture and system environment, we do not explicitly consider it in the rest of this Section.

5.6.1 Latency Model

Processing messages in batches can increase the latency, for two reasons:

1. messages need to be queued until a sufficiently big batch has been collected \( (\text{Avg. Queing}) \)

2. matching profiles for an individual message can only be determined when all messages have been processed \( (\text{Matching}) \)
For a steady-state system, the average time of these two factors can be estimated as follows:

\[ \text{Avg. Queuing} = \frac{\text{Batchsize}}{2 \cdot \text{Arrival Rate}} \]

\[ \text{Matching} = \frac{\text{Batchsize} \cdot \text{matching single}}{\text{speedup}} \]

The first formula describes that on a system where the arrival rate does not fluctuate much, the average latency for queuing is half the maximum latency caused to get the desired batch size. The second formula, in turn, describes the time to do the matching profiles. We model the performance benefits of batching by using the unbatched speed and applying a speedup factor that depends on the factors shown in the last Section.

The average latency is then \( \text{Avg. Queuing} + \text{Matching} \), the maximum latency \( 2 \cdot \text{Avg. Queuing} + \text{Matching} \). The maximum latency occurs for a message that is queued as the first, thus taking twice the average time in queuing and delivery. The minimum latency, on the other hand, is for a message that is immediately processed in a batch, and consists only of \( \text{Matching} \).

### 5.6.2 Policies to Control Latency

In some applications, it might be necessary to limit the latency of all messages. This can be done by controlling the batch size, and thus, the time a message is queued. Using the formulae from the previous Section, the latency can be constrained to \( \text{acceptable latency} \) by choosing a batch size according to the following formula:

\[ \text{batchsize} < \frac{\text{acceptable latency}}{\left( \frac{1}{\text{Arrival Rate}} + \frac{\text{matching single}}{\text{speedup}} \right)} \]

Obviously, constraining the batch size also limits the throughput benefits that can be achieved by batching. Table 5.8 shows the possible speedups when a certain latency is tolerable. The setting from RQ-Z in Section 5.5 is used, and as expected, the maximum possible throughput grows when allowing more latency. For a latency of 10 ms, the speedup is merely a factor of 1.1. At 1000 ms, the speedup is already 2.28, further growing to 9.2 at 10000ms.
5.6.3 Latency Improvements for Bursty Traffic

So far, our discussion has been focused on the adverse effects of batching on the latency of an information filter, especially when a system is dealing with a constant arrival rate. In most real-life scenarios, however, traffic is anything but constant. More typically, longer periods of relatively low traffic alternate with bursts of messages that arrive at much higher rates, causing the systems to be temporarily overloaded. Examples of this are mobile phone systems becoming unavailable on New Years Eve as everybody calls, web sites breaking down on the sudden interest of people on the information they provide (think of disasters!), or mail servers being hit by a spam attack.

Batching is particularly effective here. Traditionally, on a traffic burst, the messages are being queued up until the resources to store them are exhausted. At this point the system either breaks down or has to discard messages, neither of which is a very desirable behavior. In any case, there is a significant increase in latency since all the messages queued up need to be handled first before a newly arrived message can be handled. Often, this additional latency will persist even after the end of the burst - until the queue has been cleared. While it is often possible to lessen such effects by significantly overprovisioning the resources, this approach is very costly. In contrast to that, adaptive batching is a much more effective way: The system can accommodate the normal traffic without batching. At a burst, the system switches over to batched operation, thereby increasing its throughput capabilities and "leveling" the burst. As messages are queuing up already, batching does not cause any additional latency, but reduces it by emptying the queues much faster. As table 5.9 shows, those savings can be significant. We took RQ-Z workload from the first experiment in Section 5.5 and computed the average and maximum latencies for temporary bursts with a rate of 3000
5.7. Related Work

msg/sec while the system was able to handle about 450 msg/sec without batching. The duration of the burst is varied between 10 ms and 10000 ms. While there is not much benefit at a burst of 10 ms (only the maximum latency goes down by about 20 percent, performance benefits of batching increase sharply with the duration of the burst. At a burst length of 10 seconds, both the average and maximum latencies are more than ten times lower.

A more detailed analysis of latency and other quality of service parameters is given in Section 9.

5.7 Related Work

Our work and the benefits it provides are based on the significant work in the database community on information filtering. This work includes index structures and merge strategies for publish/subscribe systems, scalable trigger processing, and continuous query processing. Examples are LeSubscribe [44], Siena [12], XML filters such as YFilter [39], and work on interval skip lists for predicate indexing [53]. Furthermore, techniques to optimize indexes for moving object indexes (e.g., [74]) are related and can be exploited. The purpose of our work is also to show how these systems can be extended in order to make use of batching.

Batched processing has been studied in various contexts for database systems and in particular for operations on indexes. A related idea to bundle probes to indexes has been studied by Zhou and Ross [88]; the focus of that study, however, is to optimize processor cache hit rates and that work was carried out in a completely different context (traditional database index structures such as B-Trees, rather than information filters). Batched update operations on indexes has been studied in [34,47], bulk loading of (multi-dimensional) indexes in [17], and bulk join processing has been studied in [36].

One of the key ideas of batched processing of messages is to treat messages just like profiles and compute a join between messages and profiles and exploit all traditional join processing techniques such as partitioning and dynamic indexing. Such ideas have been exploited at several occasions in the context of data dissemination systems. One prominent example is the PSoup system [26].
5.8 Conclusion

This chapter presented and studied a novel technique for information filter systems: batching. On a high level of abstraction, the idea presented here is fairly straightforward. Rather than processing each message individually, a whole set of messages is processed. The advantage is that the cost for index probing can be reduced significantly and that additional benefits can be achieved during the postfilter and message delivery phases of an information filter. On the negative side, postfiltering can become more expensive. In order to exploit the benefits and limit the extra cost during postfiltering, we proposed minibatching; minibatching takes a potentially large number of messages and then forms smaller groups of messages that are very similar. The performance experiments show that this approach results in significant throughput gains as compared to traditional, unbatched processing, as carried out by state of the art information filters. Furthermore, the performance experiments demonstrate the stability of this approach towards different profile workloads, indexing techniques, value distributions of the messages, and tuning parameter settings.

There are several avenues for future work. First, we would like to evaluate batching for a larger class of pub/sub techniques (e.g., YFilter). Furthermore, we plan to investigate more sophisticated ways to group messages into minibatches; for instance, such techniques could be based on feedback from the postfiltering step in order to adjust the size and value ranges of the minibatches. Finally, we plan to incorporate our techniques into existing publish & subscribe and message broker products.
Part II

Context-Aware Information Filters
Seite Leer / Blank leaf
Chapter 6

Concept

6.1 Introduction

In order to implement information filters, several methods have been proposed in the past. Examples are SIFT [85], LeSubscribe [44], or YFilter [39]. The focus of all that work was on the development of scalable index structures in order to group and index profiles. A major shortcoming of the existing approaches is, however, that they are very inefficient if profiles refer to values in a database that are subject to change. We call such a database a context. For instance, a profile could indicate that a specific message containing a purchase order is only relevant for a warehouse if the warehouse has enough items in stock. The number of items in stock can change rapidly and updating the indexes in such a situation is very costly using one of the existing techniques.

This chapter presents Context-aware Information Filters (CIF). In contrast to traditional information filters, a CIF has two input streams: (a) a stream of messages (e.g. orders) that need to be routed and (b) a stream of context updates such as the new values of items in stock. This way, a CIF provides a unified solution to tailor information delivery for the routing of messages and to manage context information.

The challenge of building a CIF is that the two goals to route messages and record context updates efficiently are in conflict. As mentioned above, traditional approaches to indexing profiles are very efficient in routing messages, but are very inefficient when it comes to processing context updates. Context updates can be implemented most efficiently if there are no profile indexes at all; in this case, however, message filtering becomes prohibitively expensive. To close this gap, we present two approaches (1) AGILE and (2) batched processing of updates.
AGILE is a generic way to extend existing index structures in order to make them resilient to context updates and achieve a high message throughput at the same time. AGILE adopts some ideas from moving object databases [58,66,70], but generalizes those ideas and applies them to a different application domain; i.e. information filtering. The effectiveness of making index structures adaptive has also been shown in [33,78], but with different goals and strategies.

Batching processing of updates applies the idea of bulk processing to context updates, in a similar way as it is applied to messages in Section 5.

6.1.1 Use Cases for CIF

To establish better understanding of what a context is and what the benefits of a context-aware information filter are, a few use cases are sketched in the following:

**Message broker with state** : A message broker routes messages to a specific application and location. One example (stated above) is sending an order message to the warehouse that has the item in stock and is closest to the customer. Each message can change the state of the receivers and affects future routing decisions dynamically.

**Generalized location-based services** : With an increased availability of mobile, yet network-connected devices, the possibilities for personalized information delivery have multiplied. So far, those services mostly use very little context information, such as the location of the device. A more general solution is to extend those systems to a more elaborate context. As an example, a researcher could be interested in announcements of talks on certain topics. However, the researcher is only interested in such announcements if she is on campus, has less than one hundred unread Emails, and the talk is before her last appointment on that day.

**Stock brokering** : Financial information systems require sending only the relevant market updates to specific applications or brokers. Often, relevance is determined by the stocks in the portfolio. Stocks may change rapidly for day-traders that buy and sell at extremely high rates to take advantage of small and transient price differences. In this example, the portfolio represents the context, which must be updated frequently.
6.2 Problem Statement

To sum up, some contexts have a high update rate (inventory, portfolio), others have a low update rate (location of the warehouse), but many have varying, "bursty" update rates (location of a person, portfolio data). All examples involve a high message rate and a large number of profiles. Skipping updates in order to reduce update rates has to be avoided because it leads to costly errors in information filtering.

6.1.2 Contribution Summary

This chapter and the following in this part make these major contributions:

1. We introduce the concept of a Context-Aware Information Filter (CIF), and define its special requirements in terms of high context update rates as well as high message rates (Section 6.2).

2. We introduce a CIF-architecture in which intermediary filter stages are allowed to generate false positives as trade-in for higher update rates. To ensure correctness, false positives are eliminated in a separate post-filtering step (Section 6.3.2).

3. Based on a review on how existing methods can be adapted to support CIF in Section 6.4, Section 7.1 presents the generic algorithm AGILE. This algorithm extends best-of-breed index structures to automatically adapt to high update rates.

4. We present the results of comprehensive performance experiments in order to study the trade-offs of AGILE (Section 7.3).

5. We introduce algorithms for batched processing of updates to speed up handling of updates in Section 8.

6. We present the performance results for batched processing and compare it against the results for AGILE in Section 8.6.

6.2 Problem Statement

The main issue for context-based information filters can be summarized as follows: "Given a large set of profiles, high message rates and varying rates of context updates, provide the best possible throughput of messages". If possible, no message should be
dropped or sent to the wrong user because a change in context has not yet been considered by the filter. In the following, we define the terms context, profile and message.

### 6.2.1 Context

In the literature on ubiquitous [83] and context-aware computing, a wide range of context definitions has been proposed. Most of these definitions focus on the notion of location (an overview is given in [37]). For the purpose of this work, we use the following, more general definition: A context is a set of attributes associated with an entity; the values of those attributes can change at varying rates. Figure 6.1 gives examples of the contexts of two users. User 1 is currently in her office, whereas the location of User 2 is unknown.

Gathering context information is outside the scope of this thesis and has been addressed, e.g. in the work on sensor networks [87], data cooking [48], context/sensor fusion [20], or the Berkeley HiFi project [35]. The only assumption that is made in this work is that the values of an attribute of a context can change and that these changes are triggered by a stream of context updates.

### 6.2.2 Messages

A message is a set of attributes associated to values. For example, a message announcing a talk can be modeled as shown in Figure 6.1. (Often messages are delivered in XML. Nevertheless, we use the simplified attribute/value model in this paper for reasons of understandability while maintaining generality. Approaches to apply the new algorithms presented in this part to XML are outlined in Section 10.2.)
6.2. Problem Statement

6.2.3 Profiles

A profile is a continuous query specifying the information interests of a subscriber. Expressions in profiles can refer to a static condition or a dynamic context. Static conditions change relatively seldom, since they specify abstract interests. In contrast, context information can change frequently. We extend the definition of Section 3.2.2 to allow additional types of comparison, but still using the disjunctive normal form (DNF) of atomic comparisons:

\[
\text{profile} := \text{conj} \lor \text{profile} | \text{conj} \\
\text{conj} := \text{pred} | \text{conj} \land \text{pred} \\
\text{pred} := \text{val} \ op \ \text{val} \\
\text{op} := < | > | = | \leq | \geq \\
\text{val} := \text{constant} | \text{context.attr} | \text{message.attr}
\]

This definition allows the use of context information in profiles in multiple ways: Message attributes can be compared with constants and with context attributes. Profiles can refer to zero or many contexts, and contexts can be referred to by zero to many profiles (see Figure 6.2), providing an N:M-mapping. Contexts belonging to other entities can be referred to when allowed by privacy controls. Furthermore, comparisons between context attributes and static values are possible, or even between two different context attributes. In the scope of this work, these types of comparisons are not considered in detail.

These extensions are the novel aspect of this work and a significant extension to the
way profiles are defined in traditional pub/sub systems and information filters. They enable easier modeling, richer profiles and, as we will see, several opportunities for optimization.

An example for a profile is displayed in Figure 6.1. The profile asks for the delivery of messages of the type talk announcement, if the talk occurs “today”, the user is on campus, has less than 100 unread mails, and the talk starts before the last appointment.

\[
\begin{align*}
\text{message.type} &= \text{‘talk announcement’} \\
\wedge \text{message.activity.date} &= \text{today}() \\
\wedge \text{context.location} &= \text{‘on campus’} \\
\wedge \text{context.inbox.unread} &< 100 \\
\wedge \text{message.end} &< \text{context.last.appointment.start}
\end{align*}
\]

If messages and/or context information are represented in XML, then profiles involve XPath expressions. Again, we use the simplified attribute/value model for ease of presentation and without loss of generality.

### 6.2.4 CIF Processing Model

Figure 6.3 shows the processing of a CIF. The CIF keeps profiles of subscribers and context information. The CIF receives two input streams: a message stream and a context update stream. Conceptually, these two streams are serialized so that at each point in time either one message or one update is processed, providing a clear consistency model. The actual implementation can process events in parallel in order to improve performance if the result is equivalent to this consistency model.

In order to deal with the two input streams, a CIF must support the following methods:

1. `handle_message(Message m)`:
   Find all profiles that match the given message m, considering the current context state. Return this set of Profiles.

2. `update_context(Context c, Attribute a, Value v)`:
   Set the attribute a of context c to the new value v, i.e. \( c.a := v \). All profiles referencing this context must consider this new value.
6.3 CIF Architecture

The changes in the requirements and the working model (compared to a traditional IF) also have a significant impact on the architecture. Managing the context data and processing the stream of updates needs to be taken into consideration. The first issue to consider is if there should be separate filters for static and context-based profile parts or an integrated filter. The second issue is how to actually extend a filter to cater for context management and update processing requirements.

6.3.1 Separated vs Integrated Context Handling

A first, naïve approach to build a CIF would be to build separate filters (with different optimizations) for the static parts and the context-enabled parts of the profiles, as shown in Figure 6.4. Messages would be filtered according to the static profile parts in a first step, resulting in a (much smaller) superset of matches for each message. In a second step, context information is used to further reduce this intermediate result to the actually matching set. For this step, an update-optimized system can be used. Such a design is based on the assumption that the context-enabled profile parts will have many updates and that there are few profiles actually using context information, so that a separate, update-optimized system is a good choice. There are, however, a number of drawbacks to this approach:

- If the context-enabled profile parts provide the most selective parts (e.g. loca-
tion for location-based profiles), the intermediate result after the static parts will be large. This will reduce the overall effectiveness, since a (traditional) update-optimized system is much more sensitive to the number of possibly matching profiles than a traditional information filter.

- Skew in the context update distribution cannot be leveraged: If there are contexts that are updated rarely, they will be treated like contexts updated extremely often when in fact they should be treated more like static profiles.

- Highly selective and often updated context-enabled profile parts cannot be catered well: By placing them on the static filter, a high update cost is incurred. By placing them on the update-optimized filter, their selectivity is wasted.

- Changes in the context update distribution cannot be used. A number of selective,
rarely updated context-enabled profiles might have been manually placed into the static filter. If these profiles see a change in their update pattern, this placement should be reversed.

Architectures that perform filtering of context-enabled profile parts first or process both types in parallel with an additional merge step suffer from similar problems.

Consequently, we propose an information filter architecture that integrates static and context-enabled profile parts into a single filter, as shown in Figure 6.5. Since static and context-enabled profile parts are handled and controlled by the same filter, the drawbacks of separating both can be avoided. The challenge when building such an integrated system is that traditional information filters are not designed to handle significant update loads, while update-optimized system do not scale well in terms of profiles or message throughput.

6.3.2 Architecture Extensions for CIF

The information filter architecture introduced in Section 4.1 needs to be extended to handle context management and the stream of incoming context updates. Some components are added (Context Management), while others are modified to extend their capabilities (Indexes, Merge, Postfilter). This is similar to the approach used when extending the architecture for message batching (Section 5.2.1). Message batching is not considered here, but the interaction of message batching with updates will be evaluated conceptually and with performance experiments in Section 8.

As shown also in Figure 6.6, a CIF now has four main components: (a) context management, (b) indexes, (c) merge, (d) postfilter.
Chapter 6. Concept

Context management

The first component manages context information. It stores the values of static attributes and values of context attributes which are used in predicates of profiles. Any context change is recorded by this component. This component interacts heavily with indexes and postfiltering, which both consume this information. Indexes as well as postfiltering require values of constants and context attributes in order to evaluate predicates of profiles for incoming messages. Often, the context manager can keep all relevant context information in the main memory, thereby using a hash table as a data structure. If context updates are rare and the context information is very large, a conventional disk-based relational database is also an option.

Indexes

The role of the indexes, accelerating the filtering by indexing the profiles or predicates of the profiles, remains unchanged, but now they not only use values of (almost) static profiles, but also context values (see previous section). The current state of the context (as it would be used in the comparison) is indexed. As before, the index needs to support the probe method, which is invoked by the CIF's handle_message method. probe takes a message as input and returns a set of profiles that potentially match that message. As will be shown in Section 7.1, an index should also support an update method in order to deal with context updates. For completeness, the aspects used to classify indexes (see Section 4.1.1) are repeated here, as they will be needed in the rest of this chapter:

- **Target**: Value indexes or structure indexes. For the model of Section 6.2, only value indexes are relevant. Integrating structure indexes into the AGILE framework is future work.

- **Accuracy**: Does the index return false positives or will it only give the exactly matching profiles? False positives are then eliminated in the architecture of Figure 6.6 in a final postfilter step. By allowing false positives, the performance of index operations can be improved. The drawback is the increased cost for postfiltering.

- **Dimensionality**: A single index might cover all predicates of all profiles. Alternatively, there could be several indexes: each index covering only those predicates that involve a certain set of attributes or even one index per attribute.
6.4 State of the Art Approaches

- **Scope:** Indexes are typically used to index all values of a given attribute. These indexes are full indexes. Alternatively, indexing can be limited to certain values. These indexes are called partial indexes [76]. Partial indexes could, for instance, be used to index constants or context values that are rarely updated but not to index values that are updated frequently.

As will be shown in Section 7.1, the key idea to implementing adaptive context-aware information filters is to control the accuracy and scope of indexes. With regard to the target and dimensionality, context-aware information filters operate in the same way as traditional (non-context-aware) information filters.

**Merge**

The role of merging is unchanged, since the extensions in the profiles did not affect the composition of predicates by conjunctions and disjunctions.

**Postfilter**

The last step of processing a message eliminates false positives. Similarly to the indexes, postfilter is extended to use values from the context management. For the following discussion, it is important that updating the context management (and thus the postfilter) is significantly cheaper than updating the indexes. In contrast, the postfilter does not scale so well with the number of profiles to filter compared to the index (linear vs. logarithmic complexity).

6.4 State of the Art Approaches

This section describes existing approaches to implementing information filters and shows how these can be adapted for context-aware information filters (CIF). For each approach, we provide pseudocode for the handle_message and update_context operations of our information filter processing model and thus, show how the approach fits into the general CIF architecture described in the previous section.
Function NOINDEX.handle_message
Input: Message m
Output: Set of matching profiles RES
(1) RES := postfilter(m, <all profiles>)
(2) return RES

Procedure NOINDEX.update.context
Input: Context c, Attribute a, Value v
(1) DataStore[c].a := v;

Figure 6.7: The NOINDEX Algorithm

6.4.1 No Index

The brute-force approach is to use no index at all. As a result, the index and merge components are trivial: they do nothing. All the work is carried out in the postfilter operation. Figure 6.7 shows pseudocode for the implementation of the handle_message and update_context operations based on the CIF processing model of Section 6.2.4. The main advantage of the NOINDEX approach is that the update_context operation is cheap, since no indexes need to be maintained. On the negative side, the handle_message operation is expensive because the postfilter operation is applied to all profiles.

6.4.2 Eager Full Indexing

The opposite to the NOINDEX approach is an approach that makes aggressive use of indexes and keeps all indexes up-to-date and 100 percent accurate. We call this approach EAGER. This approach represents the traditional approach taken in information filtering systems [44]. Figure 6.8 shows the pseudocode for this approach, where Attrl depicts the number of indexes. The big advantage of EAGER is that the handle_message operation is as cheap as it can get, thereby exploiting the state-of-the-art index structures for information filters [44]. If indexes exist for all attributes of a message (fullyIndexed evaluates to TRUE), the postfilter step can be avoided altogether. For performance reasons, usually only attributes for selective predicates are indexed so that a postfilter step is necessary; however, postfiltering is applied to a small set of profiles only, rather than to all profiles, as it is done in the NOINDEX approach.

The big disadvantage of the EAGER approach is that the update.context operation is ex-
6.4. State of the Art Approaches

Function **EAGER.handle_message**

**Input:** Message m

**Output:** Set of matching profiles RES

(1) RES := merge(index[1].probe(m), ..., index[AttI].probe(m))

(2) If (NOT fullyIndexed(m))

(3) RES := postfilter(m, RES)

(4) EndIf

(5) return RES

Procedure **EAGER.update.context**

**Input:** Context c, Attribute a, Value v

(1) For (i:=1 to AttI)

(2) If (index[i] indexes a)

(3) index[i].remove(c.a)

(4) index[i].insert(c.a,v)

(5) EndIf

(6) EndFor

(7) DataStore[c].a := v

Figure 6.8: The EAGER Algorithm

Expensive because it involves maintaining indexes, potentially with every context update.

6.4.3 Partial Indexing

The idea of partial indexes is to reduce the cost of the update.context operation by reducing the scope of an index. If an update is outside the scope of an index, then the index need not be updated. For instance, assume the index involves the context.inbox.unread attribute (see example of Figure 6.1), and the scope is constrained to [0; 50]. Then a context update from 100 unread Emails to 101 unread Emails need not be reflected in the index because profiles associated to that context are not being indexed by the partial index. The drawback of this approach is that for a probe operation, all non-indexed values must be processed in a brute-force manner.

Figure 6.9 shows the pseudocode for the handle_message and update.context operations of PARTIAL. The hope is that most messages will be in scope so that the partial
Function PARTIAL_INDEX.handle_message
Input: Message m
Output: Set of matching profiles RES
(1) RES := merge(partindex[si].probe(m), ..., partindex[sn].probe(m))
    with m in scope for index[s_i], 1 ≤ s_i, n ≤ AttI
(2) If (NOT fullyIndexed(m) OR NOT Fullscope(m))
(3) RES := postfilter(m, RES)
(4) EndIf
(5) return RES

Procedure PARTIAL_INDEX.update_context
Input: Context c, Attribute a, Value v
(1) For (i:=1 to AttI)
(2) If (partindex[i] indexes a AND c.a is in scope)
(3) partindex[i].remove(c.a)
(4) EndIf
(5) If (partindex[i] indexes a AND v is in scope)
(6) partindex[i].insert(c.a,v)
(7) EndIf
(8) EndFor
(9) DataStore[c].a := v;

Figure 6.9: The PARTIAL Algorithm

indexes can be used in order to carry out the handle_message operation as efficiently as in the EAGER approach. Furthermore, the hope is that most context updates will be out of scope so that the partial indexes need not be updated and the update_context operation is as efficient as in the NOINDEX approach.

The idea of partial indexing goes back to Stonebraker [76], and was further studied in [73]. Stonebraker studied partial indexing in the more traditional database context (rather than for information filtering). Obviously, the most important issue is how to define the scope of a partial index. Section 7.1 shows how to exploit the idea of partial indexing and at the same time automatically set the scope of a partial index based on the message and context update workload.


6.4.4 Lazy Updates, GBU

Lately, there has been work on moving object databases [66, 70] and the basic insight of that work is that updates often exhibit a high degree of locality. For instance, people often change their location within a building and they do not suddenly make quick, long-distance moves to totally different places. This observation can be exploited in order to implement index updates more efficiently and, thus, reduce the cost for the update_context operation. The idea is that updates that remain within the bounding box of a leaf node of an index are not propagated to non-leaf nodes of the index; propagation only occurs if the new value is outside of the bounding box of the old value. If propagation is necessary, then locality is also exploited as much as possible. The most recent example of such an approach is GBU on R-Trees [58]. The approach can be applied to enhance both eager and partial indexing. For simplicity, we will only apply this approach to eager indexing and refer to the resulting approach as GBU.

Figure 6.10 shows the pseudocode for GBU. In that code, we used terminology defined in [58]; please, refer to [58] for details. The cost of the handle_message operation is comparable to the cost of the handle_message operation of the EAGER approach: it is slightly higher because it always requires a postfilter step. The cost of the update_context operation, however, stays much lower in cases in which the update stays within the same bounding box or between sibling nodes in the R-Tree.

Depending on the model of bounding boxes, such indexes can generate false positives, thus — following our model —, false positive elimination is required.

The pseudocode in Figure 6.10, one can see that for message handling, the result of the index operation needs to be postfiltered, which is a quite straightforward operation. The update, however, is relatively complex. If the new value is inside the current bounding box around the old value, no index update needs to be done. If the new value is outside, the algorithm tries to find a sibling bounding box that is nearby. If this also fails, the algorithms walks up the tree until it finds a common parent for both the old and the new value. Starting from this parent, a conventional update is performed.

Another approach from moving object indexes [70, 74] cannot be generalized to arbitrary context data, as it assigns (linear) trajectories to objects. For arbitrary contexts, finding a trajectory is next to impossible.
Function GBU.handle_message
Input: Message m
Output: Set of matching profiles RES
(1) RES := merge(GBUindex[i].probe(m), ..., 
    GBUindex[AttI].probe(m))
(2) RES := postfilter(m, RES)
(3) return RES

Procedure GBU.update_context
Input: Context c, Attribute a, Value v
(1) For (i:=1 to AttI) do
(2) If (GBUindex[i].indexes(c.a) and 
    NOT GBUindex[i].contained(c.a, v))
(3) If (GBUindex[i].boundsExtendableTo(c.a, v))
(4) GBU.extendBounds(c.a, v)
(5) ElseIf (GBU.siblingAvailable[i](c.a, v))
(6) GBU.moveSibling[i](c.a, v)
(7) Else
(8) parent = SearchParent(v, c.a)
(9) parent.remove(c.a, oldValue)
(10) parent.insert(c.a, v)
(11) EndIf
(12) EndIf
(13) EndFor
(14) DataStore[c].a := v;

Figure 6.10: The GBU Algorithm
Chapter 7

AGILE

7.1 Adaptive Indexing: AGILE

This section presents the AGILE (Adaptive Generic Indexing with Local Escalations) algorithm. We present the general idea of the algorithm in Section 7.1.1. The following subsections give a more formal and complete description of the approach and show how it can be applied to interval skips lists [53], the best known value index structure for information filtering.

7.1.1 General Idea

The key idea of AGILE is to dynamically reduce the accuracy and scope of an index if context updates are frequent and to increase the accuracy and scope of an index if context updates are seldom and handle_message calls are frequent. This way, AGILE tries to act in the same way as the NOINDEX approach for update.context operations and like the EAGER approach for handle_message operations, thereby combining the advantages of both approaches. In order to do so, AGILE generalizes techniques from PARTIAL and GBU.

The operation to reduce the accuracy is called escalation; it is triggered by context updates in order to make future context updates cheaper. The operation that increases the accuracy of an index is called deescalation; it is triggered by handle_message events in order to make future message processing more efficient. Both operations are carried out in the granularity of individual index entries. This way, the index remains accurate for
profiles that are associated with contexts that are rarely updated and the index moves profiles that are associated with contexts that are frequently updated out of scope. As a result, AGILE only escalates and deescalates as much as necessary and can get the best level of accuracy for all profiles.

AGILE is generic and can be applied to any index structure; in particular, it can be used for the index structures devised specifically for information filtering. It works particularly well for hierarchical index structures.

Example To demonstrate the key ideas of the approach, Figures 7.1 to 7.3 show how escalation and deescalation work for a simple binary search tree. The binary tree shown in those figures could, for instance, be part of a message broker which routes order messages to a warehouse, if the warehouse has enough items in stock. For this purpose, the number of items available are the keys (represented by integers), whereas the warehouses are the identifiers (represented by capital letters). At the beginning (left-hand tree in Figure 7.1) both warehouses A and B have two items in stock.
In order to implement AGILE on a binary tree, the structure of a node is extended. In addition to the key \( k \), every node has three sets of identifiers:

- **left**: this is a set of escalated identifiers (i.e., profiles) which are associated with the key range \([-\infty; k[\]
- **right**: this is a set of escalated identifiers (i.e., profiles) which are associated with the key range \([k; +\infty[\]
- **exact**: the set of non-escalated identifiers which are associated with \( k \)

**Example Escalation** Figure 7.1 shows how an identifier, \( A \), is escalated. This operation is triggered by increasing the stock of Warehouse \( A \) by one; i.e., a context update from two to three. Rather than carrying out an insert and delete on the binary tree, the escalation moves the Identifier \( A \) up to the left set of the parent node (Figure 7.1). What this means is that \( A \) has less than five items in stock, but the index does not capture the precise value anymore. When a new order for four items arrives, then the index returns \( A \) as a possible warehouse to fulfill the order. In fact, \( A \) is not able to fulfill the order (it only has three items in stock), but the index at this point is not accurate enough to detect this, and thus \( A \) must be filtered out in the postfilter step of the CIF. Warehouse \( B \) is not considered as a possible warehouse to fulfill the order because \( B \) is in the exact set of Node 2 so that the index has accurate knowledge of the number of items in stock for \( B \). Likewise, Warehouse \( A \) is not a candidate warehouse for an order that asks for five, six, or more items because the index knows that \( A \) has less than five items in stock.

**Example Cheap Update** Obviously, escalations as shown in Figure 7.1 make the handle_message operation more expensive: the index is less accurate, resulting in false positives that must be filtered out in the postfilter step. On the other hand context updates become cheaper. Now, consider that two items are taken out of stock of Warehouse \( A \). As a result, Warehouse \( A \) has only one item left. As shown in Figure 7.2, the index need not be adjusted at all in order to reflect this change and, thus, the update_context operation is as cheap as for the NOINDEX approach in this case.

**Example Deescalation** Figure 7.3 shows how deescalation is performed. This operation is triggered if the handle_message operation is called several times for orders of,
say, three or four items and Warehouse A was returned by the index as a potential candidate and had to be filtered out by the postfilter step. If this happens, AGILE decides to deescalate the index entry for Warehouse A in order to improve the performance of future calls to the handle_message operation. As shown in Figure 7.3, deescalation involves adjusting the index such that an entry from a left or right set is moved into the appropriate set of a lower node. Deescalating from a left or right set of a leaf node involves inserting a new leaf node and moving the identifier into the exact set of this new node. In the example shown in Figure 7.3, A is placed into the left set node of Node 2. After the deescalation, Warehouse A will no longer be considered a possible warehouse to handle orders for two or more items.

As mentioned above, the advantages of AGILE are that it effectively combines the advantages of the NOINDEX and EAGER approaches in a fine-grained way. It can deal with workloads in which certain contexts are frequently updated by escalating the entries for those contexts: in the extreme case to the root of the data structure or even outside of the scope of the index. Likewise, AGILE is suitable for workloads in which context updates are seldom and many messages need to be processed; in this case, no escalations take place and AGILE behaves just as traditional information filtering systems. On the negative side, AGILE does incur certain overheads. One is that depending on the index structure used, the memory requirements can increase. This is definitely true for the binary tree used in the example above or for B+-Trees; it is, however, not true for the interval skip list [53](Section 7.1.4). Another overhead is incurred by escalations and deescalations. For this reason, it is very important to control when escalations and deescalations take place. Alternative approaches to controlling these
movements are described in Section 7.1.5.

7.1.2 Properties of AGILE Indexes

As mentioned above, the escalate and deescalate operations can be implemented on any index structure, and thus AGILE can be used to extend any index structure for context-aware information filtering. Formally, every index maps each key \( k \) to a set of identifiers \( \{i\} \). This mapping is returned by the probe operation of an index, i.e. \( \text{probe}(k) \rightarrow \{i\} \). Vice versa, every index internally associates an identifier \( i \) to its key \( k \). We refer to this key \( k \) as \( k(i) \) (Table 7.1). What makes AGILE special is that it generalizes indexing and associates an identifier to a set of keys, denoted as \( K(i) \). In the left-hand tree of Figure 7.1, for instance, identifier \( A \) is associated to the set of keys \( \{2\} \), i.e., \( k(A) = \{2\} \). In the right-hand tree of Figure 7.1 (after escalation), identifier \( A \) is associated to the set of keys \( ] - \infty; 5[ \).

Based on this generalization to sets of keys, the probe operation is defined as follows.

\[
\text{probe}(k) = \{i | k \in K(i)\}
\]

In other words, the index returns identifier \( i \) when probed for key \( k \) if and only if \( k \) is in the set of keys \( K(i) \) associated to \( i \). Clearly, this generalization of indexing can result in false positives for each \( k \neq k(i); \) i.e., the key of identifier \( i \) is not \( k \). In order to avoid false negatives, it is crucial to make sure that \( k(i) \in K(i) \).

The insert and delete operations of an index are not modified and are the same as in the basic (non AGILE) version of the index. However, AGILE allows efficient implementations of the update operation that assigns a new value to an identifier. One special case is the following: if the new value of \( i \), \( k'(i) \), is already an element of \( K(i) \), then nothing needs to be done in order to implement the update; in other words, update becomes a no-operation in that case. Figure 7.2 shows an example for this kind of update. Depending on the index structure, it might be possible to find other cheap ways to implement the update operation: for instance, it might be cheap to implement \( K'(i) = K(i) \cup \{k'(i)\} \).

What are escalations and deescalations in this framework? Both operations re-assign a new set of keys to an identifier. For an escalation, the new set of keys, \( \mathcal{E}(i) \), is typically a superset of the old set of keys; i.e., \( \mathcal{E}(i) \supset K(i) \). Deescalation is the inverse operation which makes the index more accurate. For a deescalation, the new set of keys, \( \mathcal{D}(i) \) is always a subset of the old set of keys; i.e., \( \mathcal{D}(i) \subseteq K(i) \).
Escalations are carried out as a result of update_context operations in order to make future calls to update_context cheaper. Deescalations are carried out as a result of a handle_message operation in order to reduce the number of false positives for future calls to handle_message. Both are very general concepts and in theory, $E(i)$ and $D(i)$ can be any set of keys that fulfill the constraints $k(i) \in E(i)$ and $k(i) \in D(i)$, respectively. In practice, however, $E(i)$ and $D(i)$ are determined by the particular index structure used.

The basic idea of AGILE is not new. In some sense, R-Trees and other indexes for spatial data apply a similar approach. In R-Trees, identifiers are associated with bounding boxes which can also be interpreted as sets of keys. Also subset semantics are used in order to implement the probe operation and false positives occur when a path of an R-Tree is inspected although it does not contain any relevant data. The difference is that AGILE uses escalations and deescalations in order to control the accuracy of the index, whereas an R-Tree does not adjust its accuracy depending on the update/probe workload.

7.1.3 AGILE Algorithm

Based on the escalate and deescalate operations, we are now ready to present the algorithms for the handle_message and update_context operations. Figure 7.4 shows the pseudocode of these two methods. The algorithm for the handle_message operation is almost the same as the algorithm for the EAGER approach (Figure 6.8, Section 6.4.2). The only difference is that a special implementation of the postfilter operation is used (Lines 2 to 9). If a profile is detected to be a false positive (i.e., test in Line 3 fails for one of the predicates of the profile), then a DEpolicy function (Line 5) determines whether to deescalate the index for that profile. The function that tests the profile (test) returns 0 if the profile matches the message. If not, test returns a reference to the index for the predicate that failed; this index returned the profile as a false positive and therefore is a candidate for deescalation (Line 6). Since deescalation is an expensive operation, it is not necessarily a good idea to carry it out whenever a false positive occurs. Alternative policies that determine when to deescalate are described in Section 7.1.5.

The algorithm for update_context is straightforward. In the first step (Lines 2 to 5), it checks whether an escalation is necessary. In the second step (Line 6), the context is updated, just as in every other CIF approach (Section 6.4).
7.1. Adaptive Indexing: AGILE

Function AGILE.handle_message

Input: Message m
Output: Set of matching profiles RES

1) RES := merge(AGILEindex[1].probe(m), ...
   AGILEindex[N].probe(m))

2) ForEach p in RES
3) f := test(p, m)
4) If (f > 0)
5) RES := RES \ {p}
6) If (DEpolicy(p))
7) AGILEindex[f].deescalate(p)
8) EndIf
9) EndFor
10) return RES

Procedure AGILE.update_context

Input: Context c, Attribute a, Value v

1) For (i:=1 to Attr)
2) If (AGILEindex[i] indexes a ∧
   (c.a \notin AGILEindex[i].probe(v))
3) AGILEindex[i].escalate(c, v)
4) EndIf
5) EndFor
6) DataStore[c].a := v;

Figure 7.4: The AGILE Algorithm

7.1.4 AGILE Indexes

AGILE Interval Skip Lists

The information filter testbed uses interval skip lists (ISL) [53] to index range predicates, as they the best known value index structure for information filtering.

Figure 7.5 repeats the example for an ISL on a numeric domain that has been introduced in Section 4.2.1, indexing the intervals a[2;25], b[16;20[, c[8;12], d[5,5], e[-inf;16]. These intervals represent predicates inside the profiles, e.g. the sizes of orders each
Figure 7.5: ISL Before Escalation

Figure 7.6: ISL Escalation: $c = [4,15] \rightarrow c = [8,12]$

warehouse is willing to server. The markers for each interval follow a staircase pattern, since markers for a given interval are always placed to the highest level possible.

Figure 7.6 shows the result of escalation on this ISL. Interval $c$ is changed from $[8;12]$ to $[4;15]$, possibly due to an increase in the number of items on stock. As a result, $c$ is no longer associated with the (bottom-level) range $[8;12]$. Instead, $c$ is associated with the (second-level) ranges $[2;8]$ and $[8;16]$. In general, an identifier will be promoted to the lowest higher-level range that fully covers the new range. As for the simple binary tree in Section 7.1.1, escalations can result in false positives. After escalation, for instance, Warehouse $c$ will be considered as a match for an order with quantity $= 3$. On the positive side, another update, say, from $[4;15]$ to $[3;16]$ is cheap because the index need not be changed.

Deescalations are carried out in an analogous way: An identifier is moved from a higher-level range to one (or more) lower-level ranges. At the bottom-level, a deescalation potentially involves generating a new range marker (i.e., a range split).

To get a better understanding of the tradeoffs and design issues when actually implementing escalation (and, in turn de-escalation) on an existing index structure, the algorithm for escalations on an ISL is now presented in more detail. The design of the algorithm is driven by three main factors:

1. The changes to existing parts of an escalated interval should be as small as pos-
sible in order to avoid unnecessary setting and unsetting of markers.

2. The index should not be modified more than necessary, i.e., no additional nodes inserted and existing nodes only deleted if they become unused.

3. The invariants of ISL (markers at the highest possible level) should be maintained, so that the other, existing methods on the ISL such as `findInterval` for probing continue to work as before.

For these reasons, the escalation is not performed by searching the next full interval further up the ISL that would cover the new values (especially since this is quite hard to implement efficiently on an ISL), but by extending the current interval at its limits. More precisely, higher edges at the limits are sought that cover the new limit and also touch or cover the existing interval, so that the staircase pattern is maintained. Figure 7.7 shows the candidates for escalated edges on both sides of the interval \( c[8;12] \), not yet considering the actual extension value. Since, in the extreme case, an escalation to \([-\text{inf};+\text{inf}]\) might be possible, the ISL has been modified to always contain two nodes with -\text{inf} and +\text{inf}, respectively. These nodes have a high height to span their connecting edges which provide the escalation target \([-\text{inf};+\text{inf}]\) over all the other nodes.

Given this set of edges and nodes, which can be computed by incrementally searching from the old interval limits, the algorithms climbs it, starting from the lowest level and continuing until an edge is found that fulfills the requirement. This climbing is done individually for each side, as long as the invariants are maintained. In the escalation example of Figure 7.6, the search can stop after one level on both sides, as shown in Figure 7.8. Since the chosen escalation edge from 8 to 16 (also denoted as \( 8[1] \), since it originates from 8 and is at level 1) overlaps the edge from 8 to 12 (\( 8[0] \)), the marker at \( 8[0] \) needs to be removed to maintain the ISL invariant.

In general, there are eight cases on how the edges on both sides can be escalated, how they overlap the existing interval and the escalated edge on the other side of the interval. In order to improve clarity, those eight cases are not shown as shown in a complete ISL, but in more abstract way. Figure 7.9 shows this abstract notation for the example of Figure 7.8. The white upright bars represent the nodes that limit the interval before the escalation. In the example of Figure 7.8, these nodes are 8 and 12. The dotted horizontal bar covering them illustrates the existing markers for this interval. The shaded triangles depict the new values, in the example 4 and 15. They both exceed the limits of the existing interval, so escalated edges on both sides are used. These edges are shown as solid horizontal bars, spanning 2 to 8 and 8 to 16. The shaded areas of
Figure 7.7: Escalation: Edge and Node Candidates for $c = [4,15] \rightarrow c := [8,12]$

Figure 7.8: Escalation: Chosen Edges and Nodes for $c = [4,15] \rightarrow c := [8,12]$

Figure 7.9: Simplified Description of Figure 7.8

The existing markers are the parts that are overlapped by the new, escalated edges and need to be removed in order to maintain the ISL invariant. In the example, this is the edge 8 to 12. The unshaded areas of the existing markers remain the same.

Figure 7.10 covers all the possible escalation cases in this abstract notation. To The simplest case is shown in Figure 7.10a: If neither new value (depicted as triangle) exceeds the boundaries of the existing indexed interval (shown as upright rectangles), the existing index markers (depicted as dotted bar on top of the existing interval) can be kept, and nothing needs to be changed. If an extension is needed only on the left, the newly found edge (shown as solid, shaded bar) can either overlap just a part of
7.1. Adaptive Indexing: AGILE

the existing interval (Figure 7.10b) or the complete existing interval (Figure 7.10c). If an extension is needed on both sides, three cases are possible: The edge found on the left also overlaps the value on the right (and also a possible right edge that is lower) (Figure 7.10d), both edges only overlap parts of the interval (Figure 7.10e) and, at last variant, the edge found on the right also overlaps the value on the left (and also a possible left edge that is lower) (Figure 7.10f). For extensions on the right, the cases are symmetric to the extension on the left: complete overlap of newly found edge (Figure 7.10g) and partial overlap (Figure 7.10h).

The pseudocode of the escalate functions (Figure 7.11) shows how the algorithm deals with all of those cases: If an extension on the left is needed (line 1), the old node is searched and removed if no interval is using it any more. Now the candidates for an escalated edge are collected by searching incrementally for the new left value (line 4). The resulting vector is climbed up, until a matching edge is found (lines 5-8). If this edge covers the complete existing interval, all markers of the interval are removed (line 10). If not, just the markers overlapped by the edge are removed (line 12). The marker on the new left node is set, but the new edge is not yet placed since it might be overlapping by an edge found on the right side of the interval. For the right side, the procedure is almost identical (lines 14-29), with some small differences: The check for an escalated right edge only needs to be done if the escalated left edge would not yet cover the new right value as well. On the other hand, if a new right edge is found it can be placed immediately, since no left edge would overlap it. After checking the right side, the decision on whether to place the left edge can be made (lines 30-33).

For clarity, the pseudocode leaves out some aspects that are relevant when actually implementing the algorithm, among them:

- The deletion of old nodes changes the list, so the border/coverage nodes need to be reset.
- To maintain the invariant, the highest edge from node into the existing interval needs to be used (in Figure 7.8 8[1] instead of 12[0]).
- Deletion of nodes is done via ownership counting.
- The escalated values need to be stored in the interval \( I \) to allow quick containment checks, so this interval needs to be updated.

Overall, the code for this functions spans about 250 LOC, including comments. For de-escalate, the concepts and the implementation effort is about the same.
Figure 7.10: Escalation Cases in an ISL
7.1. Adaptive Indexing: AGILE

Method ISL.escape
Input: Interval I, Value left, Value right
Internal variables: Node leftNewNode, leftOldNode
Node rightNewNode, rightOldNode
Array of Edge UpdateVec
Int i, leftLevel, rightLevel

1) If left < (I→left)
2) leftOldNode, UpdateVec := search(I→left)
3) remove leftOldNode if not needed any more
4) leftNewNode, UpdateVec := search_incremental(left, UpdateVec)
5) For i:=0 to ISL→level Do
   6) If UpdateVec[i]→forward[i]→key > I→left
      leftNewNode := UpdateVec[i]; leftLevel := i
      break
7) If leftNewNode→forward[leftLevel]→key > I→right
8) removeMarkers(leftOldNode,I)
9) Else
10)   removeMarkersPartial(oldLeft, left→forward[leftLevel]→key, I)
11) newLeft→eqMarkers ∪ {I→ID}
12) If (right > I→right and (newLeft !:= NULL and
13)      right > newLeft→forward[leftLevel]→key) or (newLeft=NULL))
14)   rightOldNode := search(right)
15) remove rightOldNode if not needed any more
16) rightNewNode, UpdateVec := search_incremental(right, UpdateVec)
17) For (i:=0 To ISL→maxLevel Do
18)   If update[i]→key < I→right;newRight := update[i]→forward[i]
19)      rightLevel := i
20)     break
21) If UpdateVec[rightLevel]→key > I→left
22) removeMarkersPartial(UpdateVec[rightLevel], rightOldNode→key, I)
23) Else
24)   removeMarkers(oldLeft, I)
25) update[rightLevel]→markers[rightLevel] ∪ {I→ID}
26) update[rightLevel]→eqMarkers ∪ {I→ID}
27) rightNewNode→eqMarkers ∪ {I→ID}
28) If (newLeft!:= NULL and (newRight=NULL or
29)      (newRight!=NULL and newLeft→key < UpdateVec[rightLevel]→key)))
30)   newLeft→markers[leftLevel] ∪ {I→ID}
31) newLeft→forward[leftLevel]→eqMarkers ∪ {I→ID}
32) Figure 7.11: The escalate Algorithm - Simplified
Other AGILE Index Structures

We plan to study AGILE for a large variety of index structures as part of future work. In the following, a brief sketch of how AGILE can be applied to the most popular index structures is given:

1. **Hash Table**: An escalation is implemented by associating an identifier with the whole domain of values. Effectively, this means deleting the identifier from the hash table and keeping it in a separate list of identifiers that are returned for every probe. Deescalations are implemented by re-inserting the identifier into the hash table and deleting it from the 'escalate' list.

2. **B-Tree, B⁺-Tree, R-Tree**: As for binary search trees, special buffers must be implemented for each node in order to implement escalations and deescalations. Logically, an escalation is implemented by moving an identifier into the buffer of its parent. Deescalations are implemented by moving an identifier to a child node. There are several ways to implement the buffers and, thus, escalations and deescalations. We will explore their tradeoffs as part of future work.

7.1.5 Deescalation Policies

As mentioned in Section 7.1.3, it is important to control when deescalations occur. Several different policies are conceivable in order to decide when to deescalate an index. Ideally, an index should be deescalated if the cost for the deescalation is lower than the cost of eliminating false positives in the postfilter step of future handle_message operations. Since it is impossible to look into the future, we list some simple heuristics in the following:

1. **Always**: Every false positive encountered by the postfilter triggers a deescalation. While this strategy is trivial to implement (the DEpolicy function of Figure 7.4 returns true for every call), it is obviously sub-optimal: in update-intensive workloads, indexes are deescalated and possibly re-escalated before the cost of deescalation is amortized.

2. **Fixed**: A fixed number of false positives $FP$ is ignored until a deescalation is performed. Again, this strategy is easy to implement; the DEpolicy operation keeps a counter and returns true whenever the counter is 0 modulo $FP$, and false
otherwise. \emph{always} is a special case of \emph{fixed} with $FP = 1$. In practice, a good value of $FP$ is 1000 (Section 7.3).

3. \textbf{Auto}: \emph{auto} operates like \emph{fixed} and ignores a certain number of false positives $FP$ before a deescalation is triggered. The difference is that the value of $FP$ is adjusted dynamically according to the following formula:

$$FP = C \cdot \frac{\# \text{updatesPerformed}}{\# \text{falsePositives}}$$

In other words, if many context updates are carried out, then descalations should be rare (large $FP$). On the other hand, if many false positives are detected as part of \texttt{handle message} operations, then descalations should become more frequent (small $FP$). Here, $C$ is a constant; in our prototype, we set $C = 3000$ and that worked very well for all workloads that we have encountered so far. We experimented with other values for $C$, but the sensitivity was very low. Therefore, we do not show the results in more detail.

We will study the trade-offs of the different approaches in Section 7.3. Clearly, more elaborate policies are conceivable and we plan to study such policies as one avenue for future work; the experiments presented in Section 7.3, however, indicate that the simple policies described here work very well for a large range of workloads.

### 7.2 Cost Model Extension and Analysis

In order to get a better understanding on the cost of our algorithm, we now extend the cost model to include the changes introduced by \texttt{AGILE}. Table 7.2 summarizes our notation.

In Section 4.3, the cost for handling a sequence of messages $M'$ was given as:

$$C_{\text{total}}(M') = \sum_{m \in M'} C_{\text{hm}}(m)$$

Now, a full sequence of events also includes updates, where $U'$ the stream of updates to be processed. Thus, the total cost of is given by the following formula:

$$C_{\text{total}}(M', U') = \sum_{m \in M'} C_{\text{hm}}(m) + \sum_{u \in U'} C_{\text{uc}}(u)$$
\(C_{hm}(m)\) denotes the cost for matching a message against profiles, i.e., a call to `handle_message`; \(C_{uc}(u)\) denotes the cost for a call to `update_context`. This means, the total cost is determined by the sum of the cost of all `handle_message` and `update_context` operations.

The cost formula of a `handle_message` operation is similar to the one in Section 4.3 and is given by

\[
C_{hm}(m) = C_{ind-pr}(m) + C_{merge}(m) + res(m) \cdot C_{post-filter} \cdot p_{de-esc} \cdot C_{de-esc}
\]

Here, \(C_{ind-pr}(m)\) denotes the cost for an index probe with the given message \(m\). The number of profiles that are returned by the latter index probe is denoted with \(res(m)\). \(C_{post-filter}\) gives the cost for a single post-filter operation. Recall, that `escalate` increases the probability of creating false positives. Therefore, the more of the values get escalated in the index structure, the more will the cost for `handle_message` be dominated by the product \(res(m) \cdot C_{post-filter} \cdot p_{de-esc} \cdot C_{de-esc}\). Since the feedback of false positives can cause de-escalations, the cost of a de-escalation \(C_{de-esc}\) needs to be taken into account with a given probability \(p_{de-esc}\). Note, that \(res(m)\) is minimal for `EAGER` and maximal for `NO_INDEX` (=number of profiles), while \(p_{de-esc}\) is 0 for any method other than `AGILE`. To remove the false positives created by reduced accuracy from the intermediate results, postfiltering needs to consider the indexed attributes as well, meaning that \(\text{attidx} \subseteq \text{attpf}\)

The cost for an `update_context` operation is given by

\[
C_{uc}(u) = C_{store-up} + p_{update}(u) \cdot C_{ind-up}
\]

Here, \(C_{store-up}\) denotes the cost for an update on the context store; \(p_{update}(u)\) denotes the probability that the update \(u\) will trigger an update operation on the index structure; and \(C_{ind-up}\) the cost of such an update. Recall, that `de-escalate` increases exactness of the index and therefore, the probability of index updates is increased. Therefore, the more of the values get de-escalated in the index structure, the more will the cost for context updates be dominated by the product \(p_{update}(u) \cdot C_{ind-up}\). Note, that \(p_{update}(u)\) is minimal for `NO_INDEX` and maximal for `EAGER`. \(C_{store-up}\) can be considered a constant (depending on CPU speed, memory speed and cache hit rate). It is also important to consider that \(C_{store-up} \ll C_{ind-up}\), so that the index updates clearly dominate the overall cost. \(C_{ind-up}\) can be expressed as follows:

\[
C_{ind-up} = O(\log(val_i)) + C_{markerset} + p_{delNode} \cdot C_{del-node} + p_{insNode} \cdot C_{ins-node}
\]
7.2. Cost Model Extension and Analysis

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m$</td>
<td>message event</td>
</tr>
<tr>
<td>$u$</td>
<td>update event</td>
</tr>
<tr>
<td>$C_{\text{total}}(M', U')$</td>
<td>total cost to process message sequence $M'$ and update sequence $U'$</td>
</tr>
<tr>
<td>$C_{\text{hm}}(m)$</td>
<td>cost for handle message of $m$</td>
</tr>
<tr>
<td>$C_{\text{uc}}(u)$</td>
<td>cost for update context of $u$</td>
</tr>
<tr>
<td>$C_{\text{ind-pr}}(m)$</td>
<td>cost for an index probe with message $m$</td>
</tr>
<tr>
<td>$C_{\text{post-filter}}$</td>
<td>cost of a post-filter operation</td>
</tr>
<tr>
<td>$C_{\text{store-up}}$</td>
<td>cost of a context store update</td>
</tr>
<tr>
<td>$p_{\text{update}}(u)$</td>
<td>probability that an index update occurs for update $u$</td>
</tr>
<tr>
<td>$C_{\text{ind-up}}$</td>
<td>cost of an index update</td>
</tr>
<tr>
<td>$p_{\text{de-esc}}$</td>
<td>probability that a de-escalation is performed</td>
</tr>
<tr>
<td>$C_{\text{de-esc}}$</td>
<td>cost of de-escalation</td>
</tr>
<tr>
<td>$C_{\text{markerset}}$</td>
<td>cost to set/unset markers for profiles</td>
</tr>
<tr>
<td>$p_{\text{delNode}}$</td>
<td>probability of a index node deletion</td>
</tr>
<tr>
<td>$p_{\text{insNode}}$</td>
<td>probability of a index node insertion</td>
</tr>
<tr>
<td>$C_{\text{del-node}}$</td>
<td>cost of a index node deletion</td>
</tr>
<tr>
<td>$C_{\text{ins-node}}$</td>
<td>cost of a index node insertion</td>
</tr>
<tr>
<td>$\text{res}(m)$</td>
<td>number of profiles that match $m$</td>
</tr>
<tr>
<td>$\text{val}_i$</td>
<td>size of dimension/attribute $i$</td>
</tr>
</tbody>
</table>

Table 7.2: Notation for Cost Model in Section 7.2

There first summand denotes the cost for searching the update value in the index. $C_{\text{markerset}}$ denotes the cost to place and remove the markers, while $C_{\text{del-node}}$ and $C_{\text{ins-node}}$ denote the cost of removing and inserting a node. The probability if a node needs be inserted or deleted is denoted by $p_{\text{delNode}}$ and $p_{\text{ins-node}}$. For EAGER, $p_{\text{delNode}}$ and $p_{\text{insNode}}$ are always 1, since the existing entry needs to be removed and a new entry needs to be placed. For AGILE, this depends on how densely populated the index is and widely distributes the values are. $C_{\text{de-esc}}$ has the same cost formula as $C_{\text{ind-up}}$, but the probabilities of insertion/deletion are different.

**Summary of the Cost Model** We conclude, that the path to good performance is to keep $\text{res}(m)$, $p_{\text{update}}(u)$ and $p_{\text{de-esc}} \cdot C_{\text{de-esc}}$ in balance. Balancing is steered by the feedback strategy as explained in Section 7.1. The probabilities are influenced by the
locality of updates, the overlap of probes and updates in an index and the ratio of probes and updates.

7.3 Performance Experiments and Results

In order to study the advantages and disadvantages of AGILE in the context of CIF, we implemented AGILE and compared its throughput for various workloads with the throughput of traditional approaches to implement information filters (Section 6.4). This section presents the results of these experiments.

The two factors determining the performance of AGILE are the number of false positives for message probes and the probability that a context update results in an escalation. Both these factors should be small and depend on the distribution of attribute values in messages, the locality of context updates, and the mix of new messages and context updates.

7.3.1 Experiment Setup

To perform the experiments, we extended the test bed (developed in Section 4.2) to conform to the architecture described in Section 6.3.2:

The Context Management data store, which manages all context information, was implemented as an in-memory table that maps (context, attribute) pairs to values. This is similar to what other information filter systems do [44]. In addition, there are links which directly connect each profile to its corresponding context.

Based on this infrastructure, the following approaches were implemented: NOINDEX (Section 6.4.1), EAGER (Section 6.4.2), GBU (Section 6.4.4) and AGILE (Section 7.1). Both the fixed and auto deescalation polices were used for AGILE. The parameter $FP$ was varied between 10 and 10,000. PARTIAL was not considered as part of this experimental performance evaluation because of the difficulty to configure this approach (i.e. setting the scope of each index depending on the workload).

All experiments were performed on a 3.2 GHz Pentium 4 machine with 2 GB of RAM running Linux 2.4. At the beginning of an experiment, all indexes were fully deescalated for AGILE; escalations happened then as part of context updates in the workload. In all experiments, there was a warm-up phase with 500 messages and a varying number
7.3. Performance Experiments and Results

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Att</td>
<td>Overall number of attributes, used in messages contexts</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>and profiles</td>
<td></td>
</tr>
<tr>
<td>CB</td>
<td>Percentage of profiles referring to context attributes</td>
<td>90</td>
</tr>
<tr>
<td>Attl</td>
<td>Indexed attributes</td>
<td>2</td>
</tr>
<tr>
<td>P</td>
<td>Number of profiles</td>
<td>500,000</td>
</tr>
<tr>
<td>Val</td>
<td>Values for messages, contexts and constants</td>
<td>0–10,000</td>
</tr>
<tr>
<td>Msg</td>
<td>Messages per Run</td>
<td>10,000</td>
</tr>
<tr>
<td>ΔU</td>
<td>Maximum Distance of context updates</td>
<td>1–2,500</td>
</tr>
<tr>
<td>UP</td>
<td>Context Updates per Profile</td>
<td>1–10,000</td>
</tr>
<tr>
<td>UpdAtt</td>
<td>Percentage of updates on indexed attributes</td>
<td>0; 25–100</td>
</tr>
</tbody>
</table>

Table 7.3: Workload Parameters

of context updates (depending on the workload parameter settings) before the system throughput was measured.

7.3.2 Workload

When selecting the workloads to test the different methods, we followed the requirements derived from the Use Cases (Section 6.1.1): The number of profiles is high, most profiles refer to contexts. Low, high and varying context update rates are studied. In addition to these application-specific parameters, we evaluated the impact of parameters that are derived from the expected behavior of the methods, such as the locality of updates or distribution of updates over attributes.

We created messages, profiles and context values according to the parameters shown in Table 7.3. Both messages and contexts were sets of attribute/value pairs. The overall number of attributes (\(Att\)) in both cases is 8. Profiles contain only conjunctions of simple predicates. We also experimented with profiles containing disjunctions. However, the throughput results were almost identical for the same filtering selectivity. Therefore, we do not present the results of those experiments here. Predicates, in turn, specify an epsilon environment around a constant or a context value. 90 percent of the profiles refer to context attributes (\(CB\)), thus putting the emphasis strongly on the contexts. The selectivity of profiles on individual attributes was chosen in a way that there was a global selectivity order among the attributes. For the index-based methods, we put indexes on predicates involving the two most selective attributes (\(Attl\)). The number of profiles (\(P\)) was 500,000 for all experiments.
Figure 7.12: Exp. 1, Normalized Troughput, Vary \( \Delta U \)

\[ \Delta U = 150, \ UpdAtt = 25, \ FP = 1,000 \]

The values used in message attributes, context attributes and constants (Val) are of type float and are taken uniformly from the range \([0; 10,000]\); here, we followed Hanson’s experimental setup [53]. Similar results can be expected for other data types and domains. To determine the sensitivity on the locality of updates, the distance between the old and new value of a context update was varied uniformly in the range \([-\Delta U; +\Delta U]\); \(\Delta U\) varied from 1 to 2500. We quantified all values to three relevant digits (Q) in order to create a reasonably large number of different values.

The distribution of updates over the attributes (UpdAtt) was uniform, issuing about 25 percent of the updates on attributes of indexed predicates. In Experiment 3, where the impact of this parameter was specifically analyzed, we varied this ratio from 0 to 100 percent.

To test the adaptivity of AGILE towards evolving workloads, we used a constant message number (Msg) of 10,000 Messages per turn, while using different rates of context updates. In order to characterize the update load, we used the unit (Context)“Updates per Profile” (UP), taking the number of profiles into account.

### 7.3.3 Experiment 1: Throughput in Steady State

The first experiment studied the throughput of the alternative approaches. The context update rate (UP) was varied from 1 (very few updates) to 10,000 (many updates). \(\Delta U\) was set to \([-150; +150]\), and \(FP = 1000\).

Figure 7.12 shows the relative throughput, normalized to the throughput of AGILE. Table 7.4 shows the absolute throughput results. It can be seen that AGILE has the best
performance in this experiment. For low update rates (UP=1), EAGER, the traditional (non-context-aware) approach to implement information filters, slightly outperforms AGILE, but the margin is small (2 percent). For all other workloads, AGILE is the clear winner; in the best case, it has a 2.5 times higher throughput than the best other alternative. For each competing approach, there is at least one workload for which AGILE outperforms that alternative by an order of magnitude.

The other algorithms follow the expected pattern. NOINDEX is not competitive at low update rates: due to the lack of indexes that support message probes, it is almost 20 times slower than the index-based methods. For high update rates, NOINDEX becomes increasingly attractive, but even at very high context update rates (UP=10,000), it is outperformed by AGILE. EAGER shows a good performance for low update rates. However, its performance deteriorates quickly with a rising number of updates. For very high update rates (UP=10,000), it is outperformed by AGILE by a factor of 10. GBU is slower than EAGER for low update rates due to its additional postfiltering overhead, but it performs better than EAGER when the update rate increases. Nevertheless, GBU is
also clearly outperformed by AGILE.

A more detailed understanding of these results can be gained by looking at the number of executed index updates (Table 7.5) and the number of profiles that need to be inspected in the postfilter operation (Table 7.6). As expected, the number of index updates increases for all index-based methods with an increasing context update rate (Table 7.5). Due to escalations, this number increases much slower for AGILE than for the other index-based approaches. GBU is better than EAGER in this respect because GBU is designed to take advantage of update locality. Nevertheless, GBU is not as competitive as AGILE. Due to escalations, the number of profiles that need to be post-filtered increases with an increasing update rate for AGILE (Table 7.6) and it is much higher as for the other index-based approaches. However, for high update rates, it is more important to control the costs of context updates than the costs of postfiltering.

7.3.4 Experiment 2: Vary $UpdAtt$

The second experiment studies the impact of varying the distribution of updates to indexed and non-indexed attributes ($UpdAtt$). Figure 7.13 shows the total time used to execute a workload of 10,000 messages and 500 Mio. updates ($UP=1000$). Obviously, the performance of NOINDEX is independent of the $UpdAtt$ parameter. AGILE also exhibits very robust because it adapts to the update workload. This observation indicates that AGILE reduces the complexity for choosing the right attributes to index (the physical database design problem which is known to be very difficult) by taking the update ratio from the list of things to worry about. The performance of EAGER and GBU suffers severely, if context updates hit indexed attributes; for those approaches choosing the right attributes to an index is much more critical than for AGILE.

7.3.5 Experiment 3: Vary $\Delta U$

Both GBU and AGILE take advantage of the locality of context updates. Therefore, parameter $\Delta U$ impacts their performance. Figure 7.14 shows the completion time for varying $\Delta U$ from very high update locality ($\Delta U$ close to 0) to very low update locality ($\Delta U = 2,500$ which is 25 percent of the whole scope of possible attribute values). $UP$ was set to 1,000 in this experiment. Again, the performance of NOINDEX does not depend on this parameter and NOINDEX is used as a baseline. Since EAGER does not exploit the locality of updates, the performance of EAGER is also also independent
of the $\Delta U$ parameter. EAGER slightly benefits from a high locality of updates due to processor cache effects and because those updates only have localized impact on the index structure. The performance of GBU, which was particularly designed for a high locality of updates, and AGILE depend strongly on this parameter. With decreasing locality, GBU quickly shows almost the same behavior as EAGER. Due to its ability to escalate the index gradually, AGILE's performance decreases more slowly with a decreasing locality of updates, up to the point at which AGILE behaves like NOINDEX.

### 7.3.6 Experiment 4: Update Bursts

In the previous experiments, we tested streams of messages and updates with a fixed update rate. We now turn to an experiment that studies how AGILE adapts when there
are bursts of context updates. The workload studied has the following characteristics:

1. 1,000 messages with 0.1 Updates per Profile (corresponding to $UP=1$)
2. 1,000 messages with 100 Updates per Profile (corresponding to $UP=1000$)
3. 3,000 messages with 0.3 Updates per Profile (corresponding to $UP=1$)

Figure 7.15 shows the throughput at different moments in time; the throughput is computed for every batch of 100 messages. It can be seen that the message throughput drops during the update burst (between Message 1,000 and Message 2,000) because message probes and context updates compete at this time. The message throughput increases again after the update burst is over. Again AGILE is the overall winner; during the burst, it shows (almost) the same performance as NOINDEX which is the best
7.4 Conclusion

Information filtering has matured to a key information processing technology. Various optimization techniques and index structures have been proposed for various kinds of applications, data formats and workloads. Considerable progress has been made in this area in the recent past. At the same time, it has become clear that these systems and, therefore, the corresponding indexes must be context-aware. This paper picked up this challenge by providing simple extensions to existing index structures for information filtering systems. We called this approach AGILE for Adaptive Generic
Indexing with Local Escalations. The proposed extensions are universal and can, in theory, be applied to any index structure. The key idea is to adapt the accuracy and scope of an index to the workload of a context-aware information filter or more generally, to information filtering and message routing with state. In periods in which updates are seldom, an AGILE index behaves almost like its traditional counterpart, with only slight overheads. In periods in which context updates are frequent, AGILE automatically restructures the index in such a way that updates become cheap.

Performance experiments showed that AGILE can improve the message throughput of a context-aware information filter by as much as one order of magnitude, compared to traditional approaches to implementing information filtering systems. Furthermore, performance experiments showed that AGILE has further advantages: it is robust to poor physical design (e.g., too aggressive indexing), and it can gradually adjust to changes in the locality of updates as well as changes in the update rates. Furthermore, AGILE is able to deal with workloads with bursts.

The first results of our performance experiments with AGILE are very promising and open up many avenues for future work. This work was geared towards data dissemination applications and consequently, the implementation was based on interval skip lists, the best-known index structure for those applications [53]. As part of future work, we plan to apply AGILE to more traditional database workloads; e.g., transaction processing and the TPC-C benchmark. This work will involve applying AGILE to B+-Trees; there are several alternative ways to do that and we plan to explore these alternatives. We also plan to study other index structures such as R-Trees and hash tables (in memory and extensible hash tables on disk). There are also many details of the framework that deserve more investigation; simple deescalation policies worked very well for interval skip lists and data dissemination with information filters, but for other index structures and applications more elaborate policies might be needed. Furthermore, an escalation/deescalation policy can have better control to which levels in a hierarchical index structure to escalate and deescalate, respectively. This way, it might be possible to achieve even better performance.
Chapter 8

Batched Processing of Updates

8.1 Introduction

This chapter presents a second approach to provide faster update processing for context-aware information filters: batched processing of updates. Similar to idea of batched message processing (presented in Section 5), batched processing of updates combines several events (in this case updates) and processes them together in order to improve efficiency. The concept of batch processing of context updates is some ways related to bulk index updates [17, 34, 47], but the work in this chapter focuses on in-memory index structures for information filtering instead of disk-based index structures for traditional database workloads.

In the first part of this chapter, methods to perform batched updates are presented and analyzed. In the second parts, the interaction of batched update processing with message processing inside the CIF is discussed, while the third part contains an extensive performance evaluation.

8.2 Extension of the Architecture

The architecture of a CIF introduced in Section 6.3.2, Figure 6.6, needs to be extended by a queue to collect the set of updates that will be processed in bulk. Figure 8.1 shows two possible approaches to add this queue: before the context management (Figure 8.1a), and after the context management (Figure 8.1b). Placing the queue before context management (Queue Before) allows saving context management update
operations and will provide a clear consistency model when batched handling of messages and updates is used (see Section 8.5.1). Placing the queue after the context management and directly before the indexes (Queue After) allows easier optimizations of bulk operations on the indexes and avoids performing updates in the non-indexed updates twice (in the queue and in the context management).

8.3 Methods

The performance improvements of update batching are based on two orthogonal concepts. Both concepts rely on the fact that a sequence of updates is available. These concepts are bulk index access and the usage of a resulting set.

8.3.1 Bulk Index Access

The idea of a bulk index access stems from the concepts of message batching: Performing each index access in a set individually causes a lot of unnecessary search operations and marker set retrievals (see Figure 5.1). A single optimized access is able to avoid these redundant operations.

A bulk update operation should therefore traverse the index in an optimal pattern, avoiding unnecessary search operations and improving cache utilization. In addition, the bulk index access should avoid restructuring the index as much as possible, since that would cause a significant cost. An important share of the update cost is caused by the marker set operations. These marker set operations cannot be well performed in bulk, leaving
8.3. Methods

the optimization of node insertion and node removals.

The method updateIntervalsBulked (shown in Figure 8.2) implements these concepts on an ISL by bringing all the changed values—before and after the update—into the traversal order of the index. The index is searched incrementally from the location of the previous update. If a value is a new (updated to), an insertion into the index is performed. In case of an old value (updated from), the corresponding node is removed from the index. Identical values only need to be handled once; values that are removed and inserted can be ignored. When a removed value is the left border of an (old) interval, the markers belonging to this interval are removed; the node on the right border will be removed later, when it is in the traversal order. In turn, if an inserted node is the right border of a new interval, the markers for that interval are inserted. This access runs against the traversal order, but inserting the markers before the right node causes a lot of bookkeeping overhead and requires a significant, intrusive change to the ISL code. This algorithm implements an EAGER (Section 6.4.2) strategy. We did not implement batched updates for AGILE (Section 7.1.4), since AGILE already reduces the number of node insertions and deletions, thus leaving not much room for the optimization approach of batched updates. In addition, both updateIntervalsBulked and escalate are relatively complex algorithms with many special cases. Combining them would create an ever bigger function that would be very hard to debug.

8.3.2 Resulting Updates

The main idea of resulting updates is to only apply a single update for a specific profile and attribute out of the sequence of updates. The most common case would be the last update, but for specific scenarios also an average value or a momentum value could be used. In Figure 8.3, the first attribute is updated to 5 in \( U_1 \) and updated to 9 in \( U_5 \), so the resulting update value for the first attribute is 9. Overall, batching turns six updates into one resulting update. The method of collecting resulting updates is particularly effective if there is skew among the updates, otherwise it has to bear the overhead of checking for existing values. The implementation of the update queue uses a hash-table instead of a FIFO queue. On insertion, the inserted profile/attribute is checked if a previous update already exists. If not, it is inserted into the hash table. Otherwise, the existing value in the hash table is replaced by the new value. On queue flush, the hash table is traversed and all entries are returned in no specified order. This concepts allows superseding an existing update with a more recent one in \( O(1) \) and flushing the queue in \( O(\text{resulting updates}) \), but causes higher "constant" cost for both operations.
Chapter 8. Batched Processing of Updates

Method ISL.updateIntervalsBulked

Input:  
- Array of Interval I
- Array of Value L
- Array of Value R

Internal variables:  
- Array of Value allvalues // All modified values
- Array of Interval existing // Contains the old values
- Array of Edge update // Current position in the ISL
- Int idx // Counter
- Node cNode // Node currently modified

(1) existing := I.clone()
(2) For idx:= 0 to I.length
(3)   I[idx]->left := L[idx]
(4)   I[idx]->right := R[idx]
(5) EndFor
(6) allvalues := all values in I and existing
(7) sort allvalues ascending
(8) update = ISL.header
(9) For idx:= 0 to allvalues.length
(10)    cNode := search_incremental(value[idx],update)
(11)    If value[idx] from existing
(12)       If value[idx] is from a left side
(13)          removeMarkers of Interval belonging to value[idx]
(14)     EndIf
(15)    cNode->ownerCount := cNode->ownerCount - 1
(16)    If cNode->ownerCount = 0
(17)     remove(cNode)
(18)    Else // new value
(19)       If cNode not in ISL
(20)          insert(cNode,value[idx])
(21)     EndIf
(22)    cNode->ownerCount := cNode->ownerCount + 1
(23)    If value[idx] is from a right side
(24)          placeMarkers of Interval belonging to value[idx]
(25)     EndIf
(26) EndIf
(26) EndFor

Figure 8.2: The updateIntervalsBulked Algorithm
8.4 Cost Model Extension and Analysis

Following the cost model established in Section 4.3 and extended towards updates in Section 7.2, the cost for a sequence of update events $U'$ can be expressed as

$$C_{\text{total}}(U') = \sum_{u \in U'} C_{uc}(u)$$

where $C_{uc}(u)$ is the cost for a single context update $u$. (All identifiers used here are listed in Table 8.1.)

For conventional update strategies, the cost of an update can be assumed to be constant, so the cost of all updates can be rewritten to

$$C_{\text{total}}(U') = |U'| \cdot C_{\text{store-up}} + |U'| \cdot C_{\text{update}}(u) \cdot C_{\text{ind-up}}$$

where $C_{\text{store-up}}$ is the cost of a context management update and $C_{\text{ind-up}}$ is the cost of an index update.

For batched updates with the queue before the context update (Queue Before), the cost of processing $U'$ can be expressed as

$$C_{\text{total}}(U') = |U'| \cdot C_{\text{enq}} + |U'| \cdot C_{\text{deq}} + |U'| \cdot C_{\text{store-up}} + \sum_{u \in U'} C_{\text{update}}(u) \cdot C_{\text{batch-up}}$$

where $C_{\text{enq}}$ and $C_{\text{deq}}$ are the cost of insertion and removal of a single element on the queue. $\sum_{u \in U'} C_{\text{update}}(u) \cdot C_{\text{batch-up}}$ is the cost of doing a bulk index update.

For batched updates with the queue after the context management (Queue After), the cost formula is

$$C_{\text{total}}(U') = |U'| \cdot C_{\text{store-up}} + |U'| \cdot C_{\text{update}} + |U'| \cdot C_{\text{deq}} + |U'| \cdot C_{\text{update}} + \sum_{u \in U'} C_{\text{update}}(u) \cdot C_{\text{batch-up}}$$
where $U_{idx}'$ is the sequence of updates on indexed attributes and $\text{upd.res}$ now denotes the number resulting updates for the indexed attributes.

The cost of the bulk index update is

$$C_{\text{batch-up}}(bs) = C_{\text{sort}}(bs) + O(\log(val_i))$$

$$+ bs \cdot (C_{\text{traversal}} + C_{\text{markerset}} + p_{\text{delNode}}C_{\text{del-node}} + p_{\text{insNode}}C_{\text{ins-node}})$$

where the first term expresses the cost of bringing the updates in traversal order, while the second term expresses the search for the smallest update value in the index. Besides the cost of traversing the index to the next update value ($C_{\text{traversal}}$), the rest of the cost formula is the same as for conventional index updates. The probabilities $p_{\text{delNode}}$ and $p_{\text{insNode}}$ are changed due to the different access pattern.

Summary of the Cost Model: The efficiency of update batching largely depends on the value of $\text{upd.res}$. It needs to be significantly smaller than $U'$ to offset the cost of the queue operations. For bulk index updates, the cost of bringing the values into traversal order needs to be outbalanced by improved traversal cost and lower probabilities of node insertions and deletions.

## 8.5 Integrating Message and Update Batching

Instead of using message batching or update batching alone, combining them is the most promising approach. There are, however, some aspects that need to be considered. The first is how to interleave the batches, the second is the role of false positive feedback on message batching and AGILE.

### 8.5.1 Interleaving of Message and Update Batches

The experiments for message batching in Section 5.5 show that large batch sizes are necessary to achieve significant speedups. The experiments in Section 8.6 will show that the same is true for update batching. In practice, however, messages and updates do not arrive in large groups of one type, but are interleaved, thus rarely providing opportunities to form big batches. When maintaining this arrival order, the performance will therefore not be better than the unbatched methods, even at the extremes of the message/update ratio spectrum.
To achieve higher speedups, the original order has to be given up so that batches with a sufficient size can be formed. The drawback is that now different matching results will be produced, since messages are matched against a different state. Figure 8.4 shows different execution orders: the first represents the unbatched order in which the execution is carried out in arrival order of events. In the second model messages are executed first (i.e. the state is the same as before the first message), while in the third model updates are executed first. The fourth shows one possible approach where a certain overlap is allowed. As an example how those models can influence
the matching results, Table 8.2 shows the matches for the following workload: There are two profiles $P_1$, $P_2$, with one attribute $A$ each. Each profile is an open interval with integer values. The initial profile settings are: $A_{P1} < 12$, $A_{P2} > 13$. There are the following profile updates: $U_1: A_{P1} < 10$, $U_2 A_{P2} > 11$, $U_3 A_{P2} > 15$, $U_4 A_{P1} < 11$ and the following messages: $M_1 A = 11$, $M_2 A = 13$, $M_3 A = 15$, $M_4 A = 11$. As one can see from Table 8.2, the matching results are different for each execution order.

Restoring the results of the arrival order by keeping the relevant state for a specific message and postprocessing the results was considered, but we concluded that the overhead of keeping the specific states and doing individual postprocessing would offset all performance benefits gained by batching. Therefore, this concept was not evaluated further.

From an architecture point of view, combining message and update batching requires the Queue Before approach to maintain the context management in a consistent state while messages are processed. Otherwise, incoming updates would change the context state while the message matching is done, leading to undefined results.
8.5. Integrating Message and Update Batching

8.5.2 Feedback with Batched Processing and AGILE

Batched processing of messages and AGILE leverage the same concept: Both reduce the cost at the index stage by returning a superset of the actual result and using a post-filter to remove the false positives from this superset. There is, however, a significant difference: AGILE uses the feedback of false positives to control the index accuracy, while batched message processing just removes the false positives. This would no be a problem, if batched message processing would not increase the number of false positives compared to the (unbatched) AGILE approach. Especially profiles that are slightly escalated get a significantly larger number of false positives. In unbatched AGILE, false positives would only occur at the escalated areas, meaning that there are only few false positives. For batched message processing, these profiles receive a large number of false positives as they are part of the common intermediate result, but are only relevant to few actual profiles. This large number of false positives would lead to a large number of spurious de-escalations, even attempted de-escalations on non-escalated profiles. Deciding which false positives should lead to de-escalation and which should be ignored is therefore an important issue.

Batched processing has another impact on AGILE: since the index should not change during a batched access, the reaction to changes in the workload (escalations and de-escalations) can only be applied at the end or the beginning of a batch. Therefore the reactions to changes in the workload are slowed down.

Batched processing of updates also has some effects on AGILE: both AGILE and Batched processing of updates aim to avoid index updates; AGILE by varying the index accuracy and only updating the context data management, Batching by propagating the resulting update. The resulting set approach removes the escalation “strength” information, since only one update is applied at the index, no matter if originally there was a single update or 1 million updates.

The current implementation uses the following approaches to deal with those issues: The large number of additional false positives is reduced by only propagating at maximum a single false positive per profile on a batch, while additional false positives are discarded. This reduction of false positives can be controlled by controlling the feedback rate. If a de-escalation is attempted on a non-escalated profiles, this attempt is discarded before actually accessing the index. This approach works reasonable well to deal with the large number of additional false positives, but it could be further enhanced to deal with the other issues mentioned in this section.
8.6 Performance Experiments and Results

This section presents the results of several performance evaluations to determine the efficiency of Bulk Index Updates, Resulting Updates, Batched Updates with Individual Message Processing and the Batching Both.

8.6.1 Experimental setup

The information filter testbed was run on a Linux 2.6 system with a single 2.2 GHz AMD Opteron processor and 4 GB of main memory, with the extensions described in this chapter. The programs were compiled using the standard GCC 3.3 provided by the Linux distribution. Since the focus of this work is on the impact of the matching engine on the different QoS parameters, we excluded the cost of I/O and message parsing from the measurements. An event stream was pre-generated according to the respective workload settings (see Section 8.6.2) using a separate workload generator and stored to disk. The same event stream was processed by the information filter for the alternative approaches.

8.6.2 Workloads

Table 8.3 describes the parameters of the workloads used in this section, which are similar to the parameters used in Section 5.5 and Section 7.3. \( P \) is the number of profiles (or intervals) used in the experiments. It is varied from 5,000 to 500,000. The number of attributes of profiles, messages and contexts \( (\text{Att}) \) is 1 in the bulk index experiments and 8 in the other experiments. The number of indexed attributes \( (\text{AttI}) \) is 1 for the bulk index experiments and 2 in the other experiments. The percentage of update directed to the indexed attributes \( (\text{UpdAtt}) \) is generally 25, but varied in specific experiments to go up to 100 percent. Values for messages, contexts and constants are of type float, quantified to three digits and originating from the range 0 to 10,000. The distribution of message values \( (\text{MD}) \) is skewed similar to the experiments in Section 5.5. The locality of updates \( (\Delta U) \) is 0 to 150 from the original value and also no locality. When explicitly set, the number of updates per experiment \( (\text{Upd}) \) is varied from 5,000 to 500,000, while the number of messages per experiment is set to 500 and 50,000. The ratio of updates to message is determined by \( \text{MUP} \): message per update per profile, factoring out the number of profiles. The batch size \( (\text{BS}) \) was varied between 1 and 10,000.
8.6. Performance Experiments and Results

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>Number of Profiles</td>
<td>5,000-500,000</td>
</tr>
<tr>
<td>$Att$</td>
<td>Overall number of attributes, used in messages, contexts and profiles</td>
<td>1,8</td>
</tr>
<tr>
<td>$AttI$</td>
<td>Indexed attributes</td>
<td>1,2</td>
</tr>
<tr>
<td>$UpdAtt$</td>
<td>Percentage of updates on indexed attributes</td>
<td>25-100</td>
</tr>
<tr>
<td>$Val$</td>
<td>Values for messages, contexts and constants</td>
<td>0–10,000</td>
</tr>
<tr>
<td>$UpdProf$</td>
<td>Distribution of updates over profiles</td>
<td>uniform, skewed</td>
</tr>
<tr>
<td>$MD$</td>
<td>Distribution of message values</td>
<td>Zipf</td>
</tr>
<tr>
<td>$UD$</td>
<td>Distribution of updates over profiles</td>
<td>uniform, skewed</td>
</tr>
<tr>
<td>$ΔU$</td>
<td>Maximum Distance of updates</td>
<td>0–150, no locality</td>
</tr>
<tr>
<td>$Upd$</td>
<td>Updates per Experiment</td>
<td>5,000–500,000</td>
</tr>
<tr>
<td>$Msg$</td>
<td>Messages per Experiment</td>
<td>500, 50,000</td>
</tr>
<tr>
<td>$MUP$</td>
<td>Updates per Message per Profile</td>
<td>0.5–1000</td>
</tr>
<tr>
<td>$BS$</td>
<td>Batch size</td>
<td>1–10,000</td>
</tr>
</tbody>
</table>

Table 8.3: Workload Parameters

Figure 8.5: Bulk Index Updates - Scaling $P$ - Throughput

8.6.3 Update Batching - Bulk Index Access

The first set of experiments is suited to show the benefits of Bulk Index Update and the sensitivity to certain workload parameters.
Chapter 8. Batched Processing of Updates

Figure 8.6: Bulk Index Updates - Measuring Operations - Throughput

Figure 8.7: Bulk Index Updates - Measuring Operations - Operations

Figure 8.8: Bulk Index Updates - Scaling BS - Throughput
Impact of Profile Number

The first experiment tests the sensitivity of Bulk Index Update to the number of intervals \( P \) and gives a general overview of the performance benefits of Bulk Index Update. The number of intervals \( (P) \) is varied from 5,000 to 500,000, while the number of updates \( Upd \) is kept at 5,000, all in a single batch. The intervals are uniformly distributed over the value range; updates are uniformly distributed over profiles, and with no locality in updates.

Figure 8.5 shows the results for unbatched EAGER and Bulk Index Update, with \( P \) on the \( x \) axis and the number of index updates per second on the \( y \) axis. Unbatched achieves about 14,000 index updates per second for \( P \) greater than 10,000. Bulk Index Update achieves about 20,000 index updates per second for \( P \) between 25,000 and 150,000; for larger values, it slightly decreases its performance. For small \( P \), both unbatched and bulk index operations take a performance hit, as a large percentage of the index is modified, causing many index readjustments.

The shrinking performance benefit of Bulk Index Update for large \( P \) is examined in more detail in the following experiment, as shown in Figure 8.6. In this experiment, \( P \) is varied from 5,000 to 25,000, while all the other parameters are the same as in the previous experiment. The graphs show the results for unbatched EAGER and bulk index update, this time with their relative throughput on the \( y \) axis (normalized to unbatched). Bulk index updates is twice as fast at \( P = 5000 \), and its relative performance decreases to a factor 1.4 at 12,500, after which it remains almost constant.

A clear explanation of this behavior can be seen in Figure 8.7, which shows the number of node insertion/deletion operations needed. Unbatched and Bulk Index Update require the same amount of insertions and deletions, therefore only a single line is shown. The number of operations decreases with increasing \( P \), reducing the optimization potential of bulk operations. This reduction is caused by the fact that quantification limits the number of different values in the index, meaning that for larger \( P \) existing nodes will be more and more re-used. The remaining benefit for bigger \( P \) is caused by the improved access pattern. An analysis in the CPU and cache profiler valgrind [81] shows that the cost of Bulk Index Update is completely dominated by the marker set operations.
Impact of Batch Size

This experiment is suited to show the impact of the batch size ($BS$) on the performance of Bulk Index Update. The workload settings from the previous experiments are kept, with three changes: $P$ is fixed at 200,000; $Upd$ is 50,000 and $BS$ is varied between 1 and 25,000. Figure 8.8 presents the results, with $BS$ on the x axis and the number of updates per second on the y axis.

Unbatched is obviously unaffected by batch sizes and maintains a index update rate of about 14,500 updates per second. For Bulk Index Update, the speed increases with increasing batch size. For $BS$ smaller than 250, unbatched is faster. The speed of unbatched increases quickly to about 20,000 updates per second at $BS$ 2,500, after that the performance improvement level off, only increasing the update rate to about 21,000 updates per second for $BS$ 25,000.

Additional experiments showed that there is no strong correlation between the bulk update speed and the other workload parameters.

8.6.4 Update Batching - Resulting Set

The next set of experiments is geared towards showing the efficiency of the resulting set method. The skew in the workload is varied in different ways to create the potential of reducing the number index updates, but also stress the information filter in different ways.

The workloads uses 50,000 profiles ($P$), with a uniform distribution of intervals. 8 attributes are present in the messages, contexts and profiles ($Att$), while 2 attributes are indexed ($AttI$). Locality of update value ($\Delta U$) was set to an epsilon environment of 150 per individual update, and 500,000 updates ($Upd$) were applied.

Attribute Skew

In the first experiment of this subsection, the skew among the attributes was increased towards the indexed attributes, increasing the index update load but also providing more possibilities to reduce the number of resulting updates. Figure 8.9 shows the throughput results (on the y axis) of Unbatched EAGER, Batching with Queue Before, Queue After without Bulk Index Updates and Queue After with Bulk Index Updates. Skew is
8.6. Performance Experiments and Results

![Graph showing throughput and attribute skew](image)

Figure 8.9: Resulting Set - Attribute Skew - Throughput

![Graph showing resulting updates and attribute skew](image)

Figure 8.10: Resulting Set - Attribute Skew - Resulting Updates

varied by setting the Zipf parameter of the attribute distribution from unskewed via 0.01 (equivalent to unskewed) to 2 (representing a very skewed distribution).

The performance of *Unbatched* decreases as the skew increases, since more and more updates need to be performed on the indexes (similar to the results shown if Figure 7.13). Expressed in numbers, the 95,000 updates per second are performed at an unskewed distribution, while 32,000 updates per second are performed at a Zipf parameter of 2. All batching methods show the same trend: The performance decreases when increasing the skew from unskewed to Zipf parameter 1.0, but increases when further increasing the skew. *Queue Before* and *Queue After without Bulk Index Access* show almost the same performance access until Zipf 1.0, after that *Queue Before* benefits more from the skew (170,000 updates/sec at unskewed, 135,000 updates/second at Zipf 0.5). *Queue After* with Bulk index updates provides an additional speedup, reaching...
250,000 updates per second for an unskewed workload and 200,000 updates/second for Zipf 1.0.

Figure 8.10, showing the number of resulting updates after the queues on the y axis, provides the explanation: For the unskewed workload 280,000 out 500,000 updates "survive" the reduction in Queue Before. This reduction corresponds almost directly the execution speeds improvement without Bulk Index Update. When increasing the skew, the number of resulting updates decreases, but this effect is initially overcompensated by skewing the updates towards the indexed attributes. This can be seen especially well on the curve for Queue After. The initial number of results is lower, since only the updates on the indexed attributes pass through the queue. The increase of resulting updates shows up between unskewed and Zipf 0.75. After that, the reduction to resulting updates becomes effective enough to reduce the number of updates on the index, leading to a speedup.

Attribute and Profile Skew

In the second experiment, not only the skew in the updates towards the attributes is increased, but also the skew towards the profiles. The Zipf parameter is changed simultaneously for both parameters, e.g. attribute skew 0.75, profile skew 0.75. Figure 8.11 show the performance results. Unbatched sees a similar decrease in update speed as in the previous experiment. The performance of the batching, on the other hand, increases significantly with increasing skew. Queue before is now the best approach, reaching almost 2 million updates per second at Zipf 2. Figure 8.12, showing the number of resulting updates explains this sizeable speed improvement: the number of resulting updates decreases significantly with increasing skew, down to 1000 resulting updates at Zipf 2.0. The additional benefit of Queue Before over Queue After with Bulk Index Updates can be explained that eliminating updates at the queue is more efficient than performing a context database update.

8.6.5 Combined Results - No Message Batching

The third set of experiments shows the results when combining various strategies to improve update speed with individual message processing. A special focus is put on the comparison of AGILE to different batching approaches. In contrast to the next set of experiments, only a performance evaluation is done, but no error analysis for the
batching methods that can lead to deviating results. The workload consists of 50,000 profiles \((P)\) and 500 messages per run \((Msg)\). Updates have a skew in attributes and profiles, the Zipf parameter is 1.0 (as in the last set of experiments). Message values are skewed.

**Impact of Update Rate**

In this experiment, the impact of different update rates on different approaches is examined by varying number of updates determined by \(MUP\) between 0.5 and 100. Figure 8.13 shows the results for *Unbatched AGILE*, Batching with update batch processing for every message (*Flush on Every Message* – gives correct results) and *Batching with Fixed* (update) *Batch Sizes*: 50,000 and 200,000. The x axis shows \(MUP\), the y
axis the relative message throughput normalized to the results of *Unbatched EAGER* (not shown for readability).

*AGILE* is the best approach for small $MUP$ (=high update rates): it achieves at 20-times speedup over Unbatched EAGER, as it shifts the index update load to the postfilter. When $MUP$ is increasing (and the update ratio increasing), the benefit is reduced. Still, a factor of 5 is reached at $MUP$ 100. *Flush on Every Update* shows the same trend, but with much lower throughput. At higher update rates, the batches of updates between the messages get larger, so that there is more room to reduce the number of resulting updates. Batching with Fixed Update Batch Sizes performs generally better for higher $MUP$. The variant with 200,000 updates per batch even beats *AGILE* by a small margin on $MUP$ 50 and 100, but will introduce errors.

Figure 8.14 shows ratio of resulting updates of batching approaches for this experiment, providing additional explanations for the performance of the different batching variants: *Flush on Every Update* has the least efficiency in reducing the ration of resulting updates, especially at higher $MUP$: 0.28 at $MUP$ 0.1; 0.73 at $MUP$ 100. The approaches with fixed update batch size are more efficient, seeing some more improvement at smaller $MUP$.

**Impact of Batch Size**

Figure 8.13 shows are more detailed analysis of the impact of the batch size ($BS$). The settings of the last experiment are kept, with $MUP$ at 0.5 and 100 respectively. $BS$ is changed from 1 to 100,000. The figure again shows the message throughput relative to Unbatched EAGER.

The speedup increases for larger batch sizes. At small batch sizes, the speedup is bound to the batch size, but at large batch sizes, the workload with a higher update loads benefits more. The speedup increases up to factor of 5 and 7, respectively.

### 8.6.6 Combined Results - Message Batching

The last set of experiments in this section highlights the performance and error results when combining message batching and update batching. The impact of the error rate, the impact of different batch sizes, the errors created by reordering messages and updates for batching and the impact of different execution orders are presented.
8.6. Performance Experiments and Results

Figure 8.13: Batching Updates - Throughput

Figure 8.14: Batching Updates - Resulting Updates

Figure 8.15: Batching Updates - Scaling BS
The results were measured over a workload of 50,000 messages, and all parameters other than the profile/context update rate followed the general workload outlined in the previous set of experiments.

**Sensitivity to Update Rate**

Figure 8.16 shows the relative message throughput of batching for a wide range of profile/context update rates. For batching, the batch size (BS) was set to 10,000 messages, allowing five executions of a batch, and the execution order of *Messages First* was used. AGILE was used in combination with batched update processing in order to also study how well it handles being used in a batched setting.

*Unbatched EAGER* achieves a throughput of about 630 msg/sec at *MUP* 1 million (very few updates). Up to *MUP* 10,000 it performs well, after that the message throughput degrades quickly, reaching only a throughput of less than 2 msg/sec at *MUP* 10 (large number of updates). *Unbatched AGILE* adapts better to increasing update load and reaches about 14 msg/sec at *MUP* 10. *Batching* reaches a throughput of almost 3,500 msg/sec at *MUP* 1 million, around 1500 msg/sec at *UP* 10,000 and 20 msg/sec at *UP* 10. While *EAGER* and *AGILE* introduce no errors, *Batching* will introduce errors, especially at high update rates.
Impact of Batch Size

To study the impact of the batch size, $BS$ was modified from 5 to 10,000 for three $MUP$ settings, corresponding to low, medium and high update rates. All other workload parameters were kept at the same settings as the previous experiments.

The results show that there is indeed a significant impact of the batch size to the performance benefits on all update rates. The important difference is the batch size required to achieve better results than unbatched AGILE, and the maximum speedup reachable at the maximum batch size: For the low update rate of $MUP$ 1 million (Figure 8.17) batching is better for $BS$ greater than 60 and achieves almost a 7-fold speedup at BS 10,000. For the medium update rate of $MUP$ 10,000 (Figure 8.18) batching better for $BS$ greater than 100 and achieves a speedup of 3.5 times at BS 10K. At the high update rate of $MUP$ 100 (Figure 8.19) batching is only better for $BS$ greater than 1000 and reaches just about a 2 times speedup at BS 10K.

Comparing Batch Execution Orders

The final set of experiments compares the different batch execution orders (see Section 8.5.1) in their throughput and with regard to their respective error rates at different batch sizes. The workload settings are the same as in the previous experiment, only the batch execution orders are changed: in addition to Messages First also Updates First is used.

Figure 8.20 shows throughput results for different batch sizes and execution orders: Figure 8.20a is a combination of the figures in the last experiment, showing message through normalized to Unbatched AGILE, with Messages First execution order at $MUP$ 1 million, 10,000 and 100. Figure 8.20b, in turn, uses the same setup, but the Update First execution order. The results are almost identical; the only difference is that update first slightly slower at BS 10000. This is an artifact of not having to perform feedback and handling escalated profiles and the first batch of Message First, which the small number of batch executions cannot fully hide.

Figure 8.21 shows the errors of both approaches at varying $MUP$ and $BS$, split up into false positives (FP) and false negatives (FN). Figure 8.21a shows false positives for the Messages First execution orders. the x axis shows $BS$, the y axis the percentage of errors in the overall result. As in the previous, the results for $MUP$ 1 million, 10,000 and 100 are shown.
Figure 8.17: Batching Both - Scaling BS - MUP 1M, Msg First

Figure 8.18: Batching Both - Scaling BS - MUP 10K, Msg First

Figure 8.19: Batching Both - Scaling BS - MUP 100, Msg First
The error rates increase with update rate, batch size. For \textit{MUP} 1 million (and thus very few updates), no errors are seen at small batch sizes. At big batch sizes the error rate increases up to 0.08 percent false positives. For \textit{MUP} 10,000, the rate of false positives increases from 0.009 to 4 percent errors. Finally, for \textit{MUP} 100, the rate of false positives increases from 0.8 to 65 percent errors.

Figures 8.21b, 8.21c and 8.21d show that the results are almost identical for false negatives and the Update First batch execution order.

From performance and error point of view, one can conclude that both execution orders are identical. Errors rates are fairly tolerable at small update rates, but become an important problem at larger update rates and larger batch sizes.

### 8.6.7 Experiment Summary

The results of the experimental evaluation can be summed up as follows: \textit{Bulk Index Updates} provide an incremental improvement by factor 1.3 to 2 over \textit{EAGER}. \textit{Resulting Updates} can achieve significant speedups, but requiring a significant skew and large batch sizes in order to do so. When combining individual message processing and batched update processing, batched update batching only exceeds \textit{AGILE} by small margin at lower update rates and large batch sizes. At high update rates \textit{AGILE} is better. \textit{AGILE} also does not introduce errors into the result. When combining batched message processing and batched update processing, significant performance gains over the whole update range are achieved, but high error rates occur at higher update rates. The experiments also show that the execution order of batches does not affect
This chapter introduced methods to perform batched update operations and also reviewed the combination of batched updates with individual messages and batched messages. The methods used to implement batched update operation are bulk index operations and the use of resulting updates. When combining batched execution, consistency models corresponding to execution order of batches were considered as well as the role of AGILE feedback in batched execution. The performance benefits of batched update processing largely depend on the skew in the updates. When combining the batched execution approaches, significant performance improvements can be
achieved, but at the cost of high error rates at higher update rates.

For future work, we mainly plan to look into more refined control policies when using AGILE in combination with batched updates. The main goals are to handle feedback even better and allow quicker reactions to changes in the load. In addition, we are considering to look into batch control policies that are able affect the batch size so that the benefits of unbatched AGILE and batched AGILE can be exploited in the best possible way.
Part III

Quality of Service for Information Filters
Chapter 9

Information Filter Quality of Service

9.1 Introduction

Under the name of message broker [1] or message queuing system [5, 8], information filters are now an important part of information processing, often used for data integration. The database community has aided this success by developing matching algorithms that support expressive profile languages like XPath [21, 39, 50] and are scalable to millions of profiles [44, 65] while maintaining high message throughput rates. What is lacking, however, is a wider view on how these algorithms can provide specific service levels, including timely delivery and error-free processing if the available resources are limited.

This chapter contributes the following aspects to the area of quality of service (QoS) in information filters:

1. it defines QoS parameters for information filters;

2. it reviews (qualitatively) the suitability of existing processing architectures and algorithms devised in the database community to support the QoS parameters;

3. it gives an extensive performance study of these architectures and algorithms with regard to QoS parameters.
9.2 QoS of Information Filters

An information filter, as presented in the previous sections and depicted in Figure 9.1), connects sources and sinks of information by the use of profiles. The information filter keeps track of the profiles, and uses additional state (aka context) for the matching decision. The updates in the profiles (un-/re-subscribe or state change) compete with the messages for timely processing, and load variations can push the information filter to its limits.

This chapter targets context-aware information filters, but the results are equally applicable to stateless information filters.

9.2.1 QoS Scenarios

Existing evaluations of information filters mostly target the cost aspect: how much is the cost (in time or CPU) to process a message or a subscription change. There are, however, many situations where cost is just one aspect among others. The usefulness (or quality) of an information filter “service” needs to be judged by many, often competing, aspects. We will first present some scenarios with specific requirements in this section, and generalize them in Section 9.2.2.

Most services do not have a constant load, but rather massive changes in their utilization. Provisioning those services for peak load is -under most circumstances- not possible at acceptable cost. So while fulfilling all possible requirements at low load is not a problem, a tradeoff has to be made for higher loads. Some of these requirements
are "hard", so that they must be fulfilled, while others can be treated with lower priority. The following examples show some very different, yet typical scenarios of information filter quality of service:

- **E-science**: A big scientific instrument (like the particle accelerators at CERN) sends results of a running experiment at very high rates. An information filter is used to only keep the data that is relevant for specific areas of interest. No errors are allowed to occur, as important information about not yet discovered subatomic particles might get lost. The arrival of events does not have to be timely, however.

- **Sports results**: During a sports event like a soccer or a basketball game, the current result is transmitted when it changes (e.g. a goal or point is scored). Messages with the result can be dropped, if new results are coming in quickly (e.g. many points in the same minute).

- **Location based service**: Messages with offers are sent to a subscriber based on his/her current location. If there is a delay in processing the messages, it is actually better to filter based on the new location than filter on the old location (which might be hard to reach again).

- **Load balancing**: Incoming jobs (=messages) are placed on the server that is the best fit (lowest CPU load, enough free memory). Placing a short-running job on a system that has somewhat outdated load statistics might result in sub-optimal utilization, but the impact is limited. Updating the load statistics every microsecond will be more costly overall.

Each of these scenarios needs different QoS requirements, so in practice most of them are solved by building a specialized, ad-hoc system or massive overprovisioning. The goal of this work is to study how already existing processing algorithms for information filters can be used for those different requirements, and if there is one single method that would cover all the requirements.

### 9.2.2 QoS Requirements

In the scenarios outlined in the last section, the main tradeoff is always between "speed" and "accuracy". In general, the requirements of applications can therefore be characterized by the following parameters:
• **Latency:** Latency here is defined as the time between message creation and message arrival at the designated target. In the scenarios above, soccer, load balancing and location based services require low latency.

• **Jitter:** Jitter is the change in latency between events, more specifically between two consecutive messages. It is important in the e-science scenario to avoid overloading later stages with burst of messages.

• **Errors:** Two types of errors can occur in an information filter:

  – **False negatives:** False negatives are matching messages that are not delivered to a subscriber. In the scenarios above, the e-science example is extremely sensitive to false negatives, since lost messages cannot be recovered.

  – **False positives:** False positives are non-matching messages that are delivered to a subscriber. False positives are again important in e-science scenario in order to avoid flooding the subscribers with spurious data. In contrast to false negatives, false positives can be handled by the subscriber using an additional filter step.

Besides those main requirements, there are other relevant parameters: availability, message order and content errors. For all these parameters, there exist solutions orthogonal to the processing algorithms devised by the database community: Availability can be ensured by techniques like replication and failover. Message order (which is given up by some processing algorithms to improve the throughput) can be restored at a fairly low cost. Content errors (i.e. the message content is destroyed) are normally not a problem affecting the filtering system, but more the network. Solutions are error-correcting encoding, checksumming and retransmitting.

Processing in the filter is not the only factor influencing the QoS parameters in an information filter, but there are also external effects: Latency and jitter are affected by network latency, capacity and the delivery policies. Errors can be caused by packet loss in the network and delays in delivering profile updates (so that a profile is not yet active or active for too long). These effects influence all the processing algorithms in the same way, so they need not to be taken into consideration in the rest of the study.

An important aspect of QoS requirements is the granularity. With coarse-grained QoS, on the one hand, all messages, updates and profiles have the same requirements. With fine-grained QoS [18], on the other hand, certain sets of messages, updates and...
profiles have different QoS requirements, e.g. some have profiles that require strict latency while others do not allow errors. Since this work is a study of the current state of the art, and no information filter methods to establish fine-grained QoS currently exist, the focus will be on coarse-grained QoS.

9.3 Information Filter Techniques

9.3.1 Components of an Information Filter

The right-hand side of Figure 9.2 (shaded in blue) gives an overview of the architecture of an information filter, as laid out in section 4.1, and extended in Sections 5.2.1, 6.3.2, and 8.2. Such an information filter has four main components: (a) indexes, (b) merge, (c) postfiltering, (d) context management.

9.3.2 Extensions for QoS Control

To support the QoS parameters of latency, jitter and errors, a system needs to take control over the flow of messages and updates (similar to queuing disciplines in networking like random early detection [46]) and also control the activities of the filter. The left-hand side of Figure 9.2 (shaded in yellow) shows queues which keep the incoming messages and updates until they are processed. The QoS policy can access these queues and also steer the filter for specific operations.
9.3.3 QoS Control Approaches

While there are many policies to enforce certain QoS parameters on queues of events, we focus on three main approaches that are applicable on information filters and represent the current state of the art:

1. **Traditional processing**: process all messages individually in the order they arrive. This method is used in nearly all existing publish/subscribe systems, among others Siena [22] and Gryphon [67] and commercial message broker systems like Tibco [1], JMS [5], Oracle Streams [7], Websphere MQ [8] and Microsoft BizTalk [6].

2. **Shedding**: discard events to keep QoS bounds. This is used in networking (e.g. [46]) and has recently gained a lot of attention in stream processing, including Aurora [79], LoadStar [32] and STREAM [15].

3. **Batching**: handle events in bulk to speed up processing: This is also used in networking (e.g. Nagle's algorithm [64]). Sections 5 and 8 describe batching approaches in the context of information filtering.

These three approaches will be further discussed in the following sections.

9.4 Traditional Processing

9.4.1 General Idea

The general idea of traditional processing is to *not* exert any control over the messages and updates in the queues. All events are kept in the queues until they are processed, and the processing order of events is the same as their arrival order. There are, however, approaches that adapt the filter component itself to optimize the throughput (and thus, in the traditional processing modelling, the latency) on changes in the message/update ratio, as presented in Section 7. The changes in the ratio are used to automatically shift the cost of filtering and updating between the indexes and the postfilter. The method without this kind of load adaptation will be called **Traditional EAGER**, the method with load adaption **Traditional AGILE** (as in [42]).
9.4.2 QoS Evaluation

A qualitative evaluation of the traditional approach leads to the following results: Since no events are discarded and the order of the events stays the same, there will be neither false positives nor false negatives. In turn, there are no guarantees on latency: If the arrival rate of events is lower than the processing capabilities of the filter, the latency will be very low. If the arrival rate exceeds the capacity, the filter will be overloaded and there will be a backlog of events. Jitter will be relatively low, since the changes in processing time for between two consecutive messages are relatively small (even at the begin or at the end of a burst).

9.4.3 Control Parameters, Adaptivity

There are no control parameters on traditional processing; it will always process all events in the given order.

9.5 Load Shedding

9.5.1 General Idea

The main idea of load shedding is to discard events, when processing within the QoS bounds is not possible anymore. This strategy is used heavily in networking (e.g. [46]) and has also seen a good deal of interest in the area of data streams [15, 32, 79].

9.5.2 Algorithmic Details

When looking into the details of possible shedding policies, the following issues need to be considered:

- **What to shed**: Messages alone, updates alone or both messages and updates?
- **When to shed**: How does the system detect that it needs to shed events?
• **Which specific events to shed:** Which events are taken out of the queue: the last arrived, some random events or specific events (e.g. semantic shedding [79])?

In the context of this work, we decided to use a combination of cost-based heuristics and a random shedding policy to cater for the above issues: When the filter is ready to process new events, it determines the available time from the latency bound, the current time and the arrival timestamp of the message as well as the number of outstanding updates. The filter "knows" (as set by the administrator or by self-monitoring) a (close) upper bound of the cost of processing a single message or a single update (in Section 9.7, 1850 µsec for a message and 80 µsec for updates were used, derived from observing the filter). From this information, the filter determines if all events can be processed. If the time is not sufficient, it randomly drops as many events needed until the bound can be reached (again based on the cost). On the issue what to shed, we evaluate all variants: *Shed messages, Shed updates, Shed both*.

### 9.5.3 QoS Evaluation

Load shedding is the only method (inside the scope of this work) that can provide hard bounds on latency (unless the wrong type of events is shed, e.g. *Shed updates* in a message burst). In turn, it does not provide any error bounds. *Shed messages* leads to false negatives (due to the dropped messages during an overload situation). The two other shedding methods lead to false negatives and false positives during and also after the burst, as updates are shed. Since the shed updates are not applied, the state of the filter after the burst is different compared to the traditional execution, an effect we call "error propagation". In terms of jitter, shedding is very similar to traditional processing: the processing in the filter is at the same speed, and the removal of events does not cause significant variations in queuing times.

### 9.5.4 Control Parameters, Adaptivity

For all shedding methods, the targeted latency bound is the main control parameter. For *Shed both*, the preference to shed messages or to shed updates can be adapted.
9.6. Batch Processing

9.6.1 General Idea

The concept of batching is to combine several events or operations (of the same type) and do a combined processing. This is more efficient than processing them individually. In the context of databases, bulk index operations are a very typical example [17]. This technique is also widespread in networking, for example Nagle's algorithm in TCP [64] which adaptively combines smaller packets generated by interactive applications into larger packets to reduce the protocol overhead (320 percent instead of 4000 percent).

While the improved throughput can reduce the overall latency, batching can also have adverse effects on the quality of service: One aspect is that batching needs a minimum number of events per batch to be more effective than individual processing; otherwise its overhead will slow it down. Collecting a sufficiently large batch can introduce additional latency. Another aspect is that the stream of incoming events is usually fairly mixed in terms of messages and updates, but batches can only be formed for messages and updates separately. So batching approaches need to reorder the flow of events, introducing errors from the different execution order of messages and updates.

9.6.2 Algorithms for Message Batching

The main message batching algorithm introduced in Section 5.2 is shown on the left hand side of Figure 9.3 and can be summarized as follows: The batch of messages is reordered and clustered into "minibatches" of similar messages. Therefore, in the
Figure 9.4: Batching Consistency Models

example, messages 2, 5, 3, 8, 6 are put into one batch, since the first attribute of each message has the value 1. Similarly, messages 4, 7, and 1 are put into the same batch. Each "minibatch" is probed as a whole against the indexes (these accesses can be highly optimized) and split up into the results for the individual messages by the postfilter. The savings at the index access exceeds the additional cost during postfiltering and restoring message order by far, thus increasing the efficiency of the filter.

9.6.3 Algorithms for Update Batching

The main idea of batched updates (as presented in Section 8) is to only apply the last update for a specific profile and attribute, as described in Section 8.3. In Figure 9.3, right hand side, the first attribute receives an update to 5 in $U_1$ and later receives an update to 9 in $U_9$, so the resulting update value for the first attribute is 9. Overall, batching turns six updates into one resulting update. When applying the updates, the index access can be optimized by ordering the access pattern. The method of collecting resulting updates is particularly effective if there is skew among the updates, otherwise it has to bear the overhead of checking for existing values.

9.6.4 Consistency Models

As noted in Section 8.5.1, combining message and update batching requires giving up the original ordering between messages and updates to create batches with a sufficient size. When maintaining this order, the performance will not be better than the traditional methods, even at the extremes of the message/update ratio spectrum. The drawback
is that now different matching results will be produced, since messages are matched against a different state. Figure 9.4 shows different consistency models, based on the different execution orders provided by Figure 8.4. The first represents the traditional model in which the execution is carried out in arrival order of events. In the second model messages never “see” future state (i.e. the state is the same as before the first message), while in the third model messages never “see” outdated state. The fourth show one possible model where a certain overlap is allowed.

9.6.5 QoS Evaluation

Batching can neither give a hard bound on latency nor guarantee that there are no errors. However, latency under heavy load will be fairly low since batching can speed up the processing of events. Compared to load shedding, batching has much stronger guarantees on errors. At the end of a batch, the state is the same as in traditional execution, so there is no error propagation. Additionally, the borders of batches form a limit on how much the state can diverge from the state in traditional execution: So by limiting the batch size the temporal error can be bound. Batching introduces jitter: the first message on the next batch has to wait all the time until the previous batch has been processed. The different execution orders allow specifying an error model, e.g. location-based services would use no outdated state to against the current location of a subscriber instead of his/her location when the message arrived. As all the different models perform equally on the QoS requirements, the most fitting model for an application can be chosen without worrying about its impact on the QoS parameters.

9.6.6 Control Parameters, Adaptivity

The size of batches can be limited to guarantee that a certain temporal error is not exceeded. In general, the batch size is automatically adapted to the current load by taking all events that have arrived out of the queue when the filter is ready to process.

9.7 Experiments

Based on the qualitative QoS analysis in the last sections, the approaches Traditional (both EAGER and AGILE), Shedding (Shed messages, Shed updates and Shed
both) and Batching are evaluated. As stated in Section 9.2.2, Latency, Jitter and Errors (false negatives (FN) and false positives (FP)) are the relevant QoS requirements. The goal of the experiments is to show how well each of the approaches supports the QoS requirements for different workloads.

The QoS requirements become important if the information filter becomes overloaded. Many real-life workloads can cause overload since they exhibit bursty load patterns: rush hours, major sports events like the soccer world cup, denial-of-service attacks, major disasters, to name just a few. Provisioning information systems that always can handle the peak of a burst is costly or—in most cases—impossible.

We therefore designed our experiments to contain various types of bursts. The steady state of the workload was set in a way that it can be easily handled by the information filter, while the bursts exceed the processing capacity. Message bursts represent the situation where a lot of information becomes available in short time (e.g. news reports on a major crisis), while update bursts represent the situation where the profile state is modified at lot (e.g. people moving during rush hours). A combined burst of messages and updates represent the commonly observed situation that major events (many messages) also changes the interest of people (many updates), so they are “tuning in” to those news. In addition to the results during the burst, the results after the end of a burst are measured, since high latencies (due to a backlog of unprocessed events in the queues) and errors after the actual burst also have a negative effect on QoS.

9.7.1 Methodology and Setup

To perform the analysis, we used our existing information filter implementation testbed. To support the different QoS policies outlined in Section 9.4, 9.5 and 9.6, we extended this system with queues for the incoming events and the respective control policies. To collect updates we use regular FIFO queues (traditional and shedding) and hash-table based queues that allow superseding an existing update with a more recent one in $O(1)$ and flushing the queue in $O(|\text{resulting updates}|)$, as presented in Section 8.3.2. The system is in implemented in C++ and was run on a Linux 2.6 system with a single 2.2 GHz AMD Opteron processor and 4 GB of main memory.

Since the focus of this work is on the impact of the matching engine on the different QoS parameters, we excluded the cost of I/O and message parsing from the measurements. An event stream was pre-generated according to the respective workload settings (see Section 9.7.2) using a separate workload generator and stored to disk.
The same event stream was processed by the information filter for the alternative approaches (Traditional/Shedding/Batching) and their respective settings. Since producing arrival rates over a 150K events per second with low variation and also excluding the I/O cost from measurements is difficult to achieve at the same time (among other due to the limited accuracy of time measurement and the operating system scheduler impact), we used a different approach to measuring time-related parameters than direct measurement: Each event gets a timestamp that corresponds to its designated arrival according to the workload requirements. The time to process a message, the updates between messages, a batch of messages or a batch of updates is measured using the `gettimeofday()` OS method which provides an accuracy of about 1 μsec. If processing previous events finished before the arrival time, the arrival time is counted as start for the next processing, otherwise the finishing time of the previous processing. From this starting time and the actual processing cost the new finishing time is computed, and a new cycle of time measurement and computation is started.
9.7.2 Workload

The parameters to create the profiles, messages and updates are shown in table 9.1. To simulate changes in the workload, the arrival rates for messages (MR) and updates (UR) are changed. The same burst duration (BD) of 10 seconds is used in all workloads containing a burst to give comparable results. The following workloads were used:

- Steady state: MR 150 messages/sec, UR 7500 updates/sec
- Message burst: Burst with BD 10 seconds, MR 1,500 msgs/sec, UR 7,500 updates/sec
- Update burst: Burst with BD 10 seconds, MR 150 msgs/sec, UR 150,000 updates/sec
- Combined burst: Burst with BD 10 seconds, MR 800 msgs/sec, UR 115,000 updates/sec

The other parameters follow the settings in the previous chapters: Both messages and contexts are sets of attribute/value pairs. The number of profiles (P) is 500,000. The overall number of attributes (Att) is 8. Profiles contain only conjunctions of simple predicates. Predicates, in turn, specify an epsilon environment around a constant or a context value. The selectivity of profiles on individual attributes was chosen in a way that there was a global selectivity order among the attributes. We put indexes on predicates involving the two most selective attributes (AttI). The values used in message attributes, context attributes and constants (Val) are of type float and are taken uniformly from the range [0; 10,000]. We quantified all values to three relevant digits in order to create a reasonably large number of different values. The distribution of updates over the attributes (UpdAtt) is uniform, issuing about 25 percent of the updates on attributes of indexed predicates. The distribution of updates over profiles (UpdProf) is also uniform. The distribution of message (MD) value was skewed following a zipf distribution. In the provisioning experiment, each measurement was done with a run of 50,000 messages, while the number of updates per profile were changed from 0.05 updates to 5,000 updates (UP). Batches in the size of 10,000 messages (BS) were used. For shedding, the experiments using a latency bound (Lat) of 5 seconds are shown, while we also did experiments with latency bounds ranging from 100 milliseconds to 10 seconds.
9.7. Experiments

9.7.3 Experiment 1: Provisioning

The first experiment provides a reference point for provisioning the information filter on the given general workload. Figure 9.5 shows the overall throughput for all methods for a wide range of profile/context update rates.

The throughput was measured over a workload of 50,000 messages, and all parameters other than the profile/context update rate followed the general workload outlined above. For batching, the batch size was set to 10,000 messages, allowing five executions of a batch. Traditional EAGER, which represents the current state of the art in most information filters, achieves a throughput of about 630 msg/sec at \( UP 0.05 \) (very few updates). Up to \( UP 5 \) it performs well, after that the message throughput degrades quickly, reaching only a throughput of less than 2 msg/sec at \( UP 5000 \) (large number of updates). Traditional AGILE adapts better to increasing update load and reaches about 14 msg/sec at \( UP 5000 \). Batching (here batching with no future state, all other variants show almost identical results) reaches a throughput of almost 3,500 msg/sec at \( UP 0.05 \), around 1500 msg/sec at \( UP 5 \) and 20 msg/sec at \( UP 5000 \). While EAGER and AGILE introduce no errors, batching will introduce errors, especially at high update rates. With shedding, the message throughput varies according to the acceptable latency and the resulting update shedding: If no latency bound is used, the message throughput will be similar to the Traditional EAGER or Traditional AGILE. If there is a very tight latency bound, and Shed updates is used, the message throughput with high update ratio can be kept at the same level as for a low update ratio at the cost many errors, as seen in Section 9.7.6. Figure 9.5 therefore shows an area which shedding covers, not a single line.
9.7.4 Experiment 2: Steady State

The second experiment shows a workload without bursts to provide a baseline for the steady state. The workload parameters were chosen so that all competing approaches can easily cope with the load: 150 messages/sec and 7500 updates/sec (so that on average a profile receives an update about every minute). This message/update ratio corresponds to the UP5 setting in Experiment 1.

Figures 9.6, 9.7 and 9.8 show the arrival time of the messages on the x axis and the latency for each message on the y axis. All methods have steady latency since they are able to handle the load. Traditional (Figure 9.6) and all variants of shedding (Figure 9.7) have an average latency of about 3 milliseconds, the maximum latency is 27 milliseconds. Batching (Figure 9.8) has an average latency of 50 milliseconds, the maximum latency is at 150 milliseconds. The evaluation for jitter (not shown here for space reasons) reflects those results: Traditional and Shedding have low jitter, batching
9.7. Experiments

Figure 9.8: Experiment 2: Latency for Steady State - Batching

has higher jitter. Since there is no overload, shedding does not create any errors. Batching shows some very minor errors, 34 false positives and 20 false negatives out of about 1.4 Million overall matches.

Shedding performs like Traditional here, since both can process an incoming event once it arrives. The small fluctuations for both are caused by slight variations of message and update arrivals (as it would be in a real workload) and the influence of external “background noise” such as the operating system. The higher peak and average latency of batching are caused by the overhead batching has to carry on very small batches (especially caused by update batching). This overhead causes batching to fall back in processing until a sufficiently large batch has been collected. With this larger batch batching is faster than traditional, and catches up with the flow of events. This, in turn, means that the next batches are small, slowing down batching again. If low latency in steady state and batching are required, a more refined QoS control policy could bypass batching queues and batching controls at such low arrival rates, only switching to batching when needed.

9.7.5 Experiment 3: Message Burst

The third experiment is suited to show the effect of an overload caused by a high message arrival rate on the different QoS methods. The workload now consists of three phases:

1. **Steady state**: 10 seconds with $MR$ 150 msgs/sec, $UR$ 7500 updates/sec (as in Experiment 2),
2. **Message Burst**: BD 10 sec. with MR 1500 msgs/sec UR 7500 updates/sec, a ten-fold increase in message arrival rate compared to the previous phase,

3. **Decay**: 80 seconds with MR 50 messages/sec, UR 2500 updates/sec, to show the return to steady state.

*Shedding* is set to a latency limit of 5 seconds (*Lat*).

Figures 9.9, 9.10 and 9.11 show the latency for traditional, shedding and batching. Traditional (Figure 9.9) sees a steep increase of latency with the beginning of the burst, further growing until the end of the message burst to almost 26 seconds. After the end of the burst, traditional returns fairly slowly to the normal latency level. The system reaches the steady state again about 60 seconds after the end of the burst. Shed messages and Shed both (Figure 9.10) see a similar increase of latency until the latency limit of 5 seconds is reached. At that point, the QoS control becomes active and keeps
9.7. Experiments

![Graph showing latency per message (ms) vs. time at message arrival (ms) for Experiment 3: Latency for Message Burst - Batching]

Figure 9.11: Experiment 3: Latency for Message Burst - Batching

![Graph showing jitter (ms) vs. time at message arrival (ms) for Experiment 3: Jitter for Message Burst - Traditional (any)]

Figure 9.12: Experiment 3: Jitter for Message Burst - Traditional (any)

![Graph showing jitter (ms) vs. time at message arrival (ms) for Experiment 3: Jitter for Message Burst - Shedding (Messages, Both)]

Figure 9.13: Experiment 3: Jitter for Message Burst - Shedding (Messages, Both)
the latency at this level by discarding events. After the end of the burst, shedding recovers fairly quickly, taking about 10 seconds to return to normal latency. Since shedding discarded all messages that would stay for too long, only a backlog of unprocessed events in the queues with the size of the latency limit needs to be processed. Shed updates is not able to achieve the latency bound, as the cost of message processing
Experiments

9.7. Experiments

alone is already too much for the filter (not shown here). \textit{Batching} (Figure 9.11) starts off with a higher latency in the first phase (as explained in Experiment 2), and turns in a sawtooth pattern on latency during the burst, as larger batches are formed and processed with the higher arrival rate of events. The maximum latency is 4.3 seconds, the return to normal latencies at the end of the burst is in the order of 1-2 seconds, as the remaining queued events are processed at a high speed.

Figures 9.12, 9.13 and 9.14 augment those results with a jitter analysis. On the y axis the difference in latency from a message to the previous message is shown. For \textit{traditional} (9.12) and \textit{shedding} (9.13), there is not much jitter, since the increase in latency during the burst is distributed over a number of messages, so that the individual difference is fairly small. The same happens at the end of the burst. \textit{Batching} (9.14) has a generally higher jitter level outside the burst (as explained in Experiment 2) and some very strong jitter during the burst, up to the maximum latency. This high jitter occurs at the border between two batches: The first message that is not part of the previous batch needs to wait only for the completion of processing of the previous batch, but also on the completion of the batch where it is added (as explained in section 9.6.5).

Figure 9.16: Experiment 3: Errors for Message Burst - Shed Both
The errors introduced by processing this burst are shown in Figures 9.15, 9.16 and 9.17, split into false positives (spurious matches) and false negatives (missing matches). Each graph shows the errors for groups of 50 messages on the y axis. Since traditional does not introduce any errors, no error graphs for this methods are presented. There are, however, significant differences between the shed messages and shed both strategies in terms of errors, so they are shown in Figure 9.15 as well as 9.16, respectively. Shed messages has a large number of false negatives during the burst (the shedded messages), and no errors after the burst. When summing up the errors over the whole experiment, there are about 260,000 false negatives (compared to about 1.25 million matches overall), distributed over 9800 shedded messages. Shed both has almost similar pattern of false negatives during the burst, but also introduces a smaller number of false positives and false negatives after the burst. This effect can be explained by the shedded updates during the burst (error propagation). The overall number of errors over the whole experiments is about 255000 FNs and 1700 FPs, with about 9600 shedded messages. For batching, (Figure 9.17), some errors occur during the burst (180 FP and 194 FN when summed up), and there are no errors after the burst.
9.7.6 Experiment 4: Update Burst

The fourth experiment shows the effects of a burst of consisting of updates. Again, the workload is split into three phases:

1. **Steady state**: about 10 seconds of 150 msgs/sec, 7500 updates/sec, (as in Experiment 2),

2. **Update Burst**: BD 10 sec. with MR 150 msgs/sec, UR 150,000 updates/sec, an increase of updates by a factor of 20 (each profile is updated every 3 seconds)

3. **Decay**: 80 seconds of 50 msgs/sec, 2500 updates/sec, to show the return to steady state

*Shedd*ing is set to a latency limit of 5 seconds (*Lat*). Figures 9.18, 9.19, 9.20 and 9.21 show the latency for the two variants of *Traditional, EAGER* (9.18) and *AGILE* (9.19) as well as for the shedding policies (9.20). *Traditional EAGER* quickly reaches a maximum latency of about 13 seconds, and takes about 30 seconds to return to steady state. *Traditional AGILE* adapts to change in update ratio, speeds up processing of updates and therefore only reaches a maximum latency of 3.3 seconds. The return to steady state is a bit faster than *Traditional EAGER*, but not much, since *AGILE* has to re-tune itself to the new workload with a higher message ratio. *Shed updates* and *Shed both* see an increase of latency until the limit, and a faster return to the original latency than the traditional methods due to the smaller backlog of events in the queues. Even though they use AGILE as the underlying index update method, they cannot fully take advantage of the index retuning as shedding updates negatively influences the adaptation to higher update rates in AGILE. *Shed messages* is not able to keep the latency bound (not shown here). *Batching* (Figure 9.21 shows the familiar sawtooth pattern again, resulting in a maximum latency of 3.4 seconds. The latency stays somewhat higher after the end of the burst due to the effects on the underlying AGILE index update strategy. The jitter evaluation shows similar results as in the previous experiments, so the graphs are not shown due to space constraints. The error evaluation shows that both shedding approaches (*Shed updates* (Figure 9.22) and *Shed both* (Figure 9.23) show the same behavior of producing false negatives and false positives after the begin of the burst due to the shedded messages. (about 22,000 FP and 22,000 FN to 800,000 overall matches). *Batching* (Figure 9.24) has again a much smaller number of errors, 488 false positives and 444 false negatives in total.
Figure 9.18: Experiment 4: Latency for Update Burst - Traditional (EAGER)

Figure 9.19: Experiment 4: Latency for Update Burst - Traditional (AGILE)

Figure 9.20: Experiment 4: Latency for Update Burst - Shedding (Updates, Both)
9.7. Experiments

In the fifth experiment the effect of a combined burst of messages and updates is shown, as many real-life scenarios might exhibit this behavior. Yet again, the workload is split in three phases:
1. **Steady state**: about seconds of 150 msgs/sec, 7500 updates/sec (as in Experiment 2)

2. **Combined Burst**: BD 10 seconds with MR 800 msgs/sec, UR 115,000 updates/sec, an increase in the message rate by more than a factor of 5 and an increase in the update rate by more than a factor of 15 (each profile is updated every 5 seconds),

3. **Decay**: 50 seconds of 50 msgs/sec, 2500 updates/sec, to show the return to steady state.

The latency bound for shedding \((Lot)\) is again 5 seconds.

The latency evaluation (Figures 9.25, 9.26, and 9.27) shows a similar picture as the results of the previous experiments: `Traditional` (Figure 9.25) reaches a maximum latency of about 19.5 seconds, returning slowly to steady state afterwards. `Shed messages` and `Shed both` (Figure 9.26) limit the latency to the designated latency and returns faster...
to normal latency levels as there is no big backlog of events in the queues. Shed Updates is not able to achieve this latency bound, reaching about 19 seconds latency with a target of 5 seconds, since the messages alone already cause an overload. Batching (Figure 9.27) exhibits a sawtooth pattern with a maximum latency of 5.6 seconds. In evaluation of errors, shed messages (Figure 9.28) has a larger number of shedded
messages during the burst (7500) than *shed both* (6900), but the latter has a tail of errors after the end of the batch (Figure 9.29). *Batching* again has a fairly small number of errors during the batch (2609 FP, 2706 FN) and no errors after the batch (Figure 9.30).

### 9.7.8 Additional Experiments

We performed more experiments to show the impact of setting the control parameters to the individual methods, and also study longer bursts. For brevity, we do not show the results here, but they can be summarized as follows: Setting different latency bounds on shedding shows that it is effective to keep the latency under control by discarding events. The same overall results were observed, with increasing/decreasing number of errors for lower/higher latency bounds. On longer bursts, the number of errors for *shedding* and *batching* increases. More events need to be discarded to keep the bounds,
and the batches get larger and thus also the deviation in state. When a limit on batch time is set, and bursts are sufficiently long for batching to reach that limit, overall latency becomes worse for the bounded batches compared to the unbounded batches, but overall errors are smaller.

9.7.9 Experiment Summary

Overall, the experiments show that traditional and shedding keep the strong bound they promise (errors and latency, resp.), but perform fairly weak on the missing parameter (latency and errors, resp.). Jitter is not an issue for either of these methods. Batching does not give a hard bound on latency, but can improve the overall latency compared to traditional, as events are processed faster; errors during the burst are possible. When compared to shedding with Lat set to the latency batching achieves, batching produces fewer errors during overload and no errors after the overload. Batching causes significant jitter, as it produces results in groups.
Figure 9.29: Experiment 5: Errors for Combined Burst - Shed Both
Figure 9.30: Experiment 5: Errors for Combined Burst - Batching
Chapter 9. Information Filter Quality of Service

9.8 Related Work

Considering quality of service is well-established in networking area, on the conceptual (like DiffServ [18]) as well as in off-the-shelf products like home routers. The use of shedding as QoS control mechanism is used in the design of many network protocols (see [86] for an overview) and has lately received a significant amount of interest in stream processing system like Aurora [79], LoadStar [32] and STREAM [15].

Batch processing as a concept is also used in network processing (e.g. Nagle's algorithm [64] to reduce the overhead of many small packets). There exists also some recent work in information filtering, speeding up the processing of messages [45]. The use of batching on overload can be compared to the use of approximative query processing on overload in data stream processing (DataTriage [68]), as both increase the throughput by reducing the processing cost while allowing errors in well-defined bounds.

Existing publish/subscribe systems like SIENA [22], Gryphon [67], Tibco [1], Sun JMS [5], Oracle Streams [7], Websphere MQ [8] and Microsoft BizTalk [6] use the traditional approach in their processing. Gryphon applies some methods of congestion control (notify the publishers to reduce their sending frequency) to ensure lower latencies.

The database community has done a significant amount of work on information filter matching algorithms, among others Le Subscribe [44], Xyleme [65] for the attribute/value data model and XFilter [13], XTrie [23], YFilter [39], XPush [50] and Index Filter [21] for XML.

9.9 Conclusion and Future Work

The impact of filtering algorithms on the quality of service of information filters has — so far — been an uncharted territory. This work defines QoS requirements, reviews (qualitatively) the suitability of existing methods and does an extensive performance study. The main QoS requirements are latency, jitter and errors. The methods reviewed are traditional processing, load shedding and batching. Workloads included steady state and bursts of different type.

The results of the qualitative analysis and the performance study show that there is no clear winner: Traditional and Shedding each satisfy one parameter completely (error
and latency, respectively), but are weak on the missing parameter (latency and error, respectively). Batching is able to perform well on both latency and error in overload situations by increasing the efficiency of processing. However, it cannot give a hard bound on latency and errors, and introduces much larger jitter.

In terms of numbers, load shedding results in a much larger number of errors than batching when set to latency bounds that are similar to the ones achievable by batching; sometimes several orders of magnitude during the burst. Compared to traditional, batching could reduce the latency by a factor of five on messages-heavy bursts, while introducing a relatively small number of errors.

Since extending an existing information filter (which already supports batching) with queues and the implementations of the QoS methods could be done with a small effort (overall about 500 additional or changed LOC), supporting all the methods in the same information filter is not a big challenge.

Such a system could then be extended by a "frontend" that maps QoS requirement specifications on the method that is the closest fit. There is, however, no seamless transition between all the methods, making it difficult to find the right methods for certain specifications. While adapting the latency bound works between shedding and traditional and adapting the maximum batch size works between traditional and batching, there is no obvious transition between batching and shedding. Combining batching and shedding with a hybrid method that allows a gradual transition is therefore one of the possible next steps in the area of QoS support of information filters.

Another direction of research that can be based on this work are models and techniques for fine-grained QoS guarantees, handling different QoS requirements on the same filter. In networking, this is actually the most common setting, as a router differentiates between low-latency traffic (Voice over IP) and bulk traffic (P2P). The interesting issue when designing fine-grained QoS on information filters is that—for some granularity—determining which QoS group an incoming messages belongs to is already as expensive as the main filtering: When a given profile requires minimizing latency, determining if a message is affected by the requirements means finding out whether the profile is matched by the message – which is the overall matching problem.
Seite Leer / Blank leaf
Chapter 10

Conclusion

10.1 Summary of the Thesis

The main paradigm of information processing is quickly shifting from “store and query” towards “route, filter, and process on the fly”. Information filters are one of the building blocks when implementing this paradigm, as they enable a loose coupling of information providers and information consumers. This loose coupling is established by the use of profiles, which information consumers submit to the information filter. Information providers send their data items (“messages”) to the information filter, without having to know the consumers. Information consumers receive messages matching to their specifications without having to know the producers.

The software industry has picked up this trend and offers information filter products like message brokers or message queuing systems. Several research communities (databases among them) have also been contributing to this paradigm shift.

This thesis presents three novel aspects of information filtering:

- **Batched Processing of Messages**: Instead of processing each message individually, a set of messages is processed at once. Processing messages in bulk results in a throughput benefit of an order of magnitude over state of the art information filters.

- **Context-Aware Information Filtering**: In contrast to existing information filters, which are stateless, our system uses state information in the matching. The definition of state ("context") is very generic and can serve many, more restricted, state
models such as windowed state or historical data. The key challenge when building such a "context-aware" filter is to deal with the combination of high messages rates and high context update rates. This thesis presents two approaches to deal with this challenge: Adaptive Indexing controlled automatically by the workloads characteristics (named AGILE) and Batched Processing of Updates. Both approaches are compared with existing approaches to deal with high message and high updates rates. AGILE exhibits performance competitive to specialized methods at extreme update to message ratios, and clearly exceeds their performance in the middle ground. It quickly adapts when changes occur. Batching updates exploits skew very well, but can introduce errors as context updates are delayed to form bigger batches.

- **Quality of Service**: Existing information filters have been designed with the goal of improving the expressiveness of profiles or the scalability in terms of profiles. Since information filters become part of open information processing infrastructure, they need to deal with unpredictable load and limited resources, leading to overload. In such situations, a tradeoff needs to be made between several service requirements. This thesis introduces and reviews quality of service requirements for information filters. Existing methods to guarantee these requirements are evaluated in an extensive performance study.

The most prominent concepts when designing the algorithms in thesis are adaptivity and an information filter architecture that allows shifting the load between the components.

Adaptivity is needed to deal with changes in the workload like arrival rate, value distribution or the ratio of updates and messages. Static algorithms will not be able to achieve the best possible performance, while tunable algorithms suffer from the need of manual adaptation. The algorithms introduced in this thesis adapt automatically to load changes by using feedback and load monitoring.

An architecture that allows shifting the load provides the foundation for adaptivity. Information filters use profile indexes and individual evaluation of predicates, which have different performance characteristics: Profile indexes work well for evaluating many profiles at once, but carry high update cost. Individual evaluation has low update cost, but does not scale well with the number of profiles. Shifting the load between those two components at fine granularity provides the opportunity to capture changing load characteristics very precisely.

To sum up, information filters will become more and more important as the paradigm
how information is processed changes. The techniques developed in this thesis might help to make information filters suitable for more applications and also provide a starting ground for further research.

10.2 Future Work

Information filtering provides many avenues for future work. Based on the contributions of this thesis, the following directions seem promising:

- **Batched Processing on Other Data Types**: The batching algorithms developed in this thesis target relational data in the form of key/value-pairs. Since more and more data is transferred in XML, it is worthwhile to look into techniques how to combine batched processing with XML filtering or even XML transformations. A possible approach would be building a summary structure for a set of documents and perform the matching on this summary structure.

- **AGILE on Other Index Structures**: AGILE was implemented on a main-memory interval index in this thesis. Since the concept of AGILE is generic, we plan to apply it to other index structures, especially on disk-based indexes like B⁺-Trees or spatial indexes like R-Trees. Disk-based index structures pose a special challenge, as escalation interacts with the database buffer: escalated entries require space in the inner nodes of the tree, thus reducing the fanout. A escalation strategy for disk-based index thus has to take the buffer utilization into account, and cannot naively escalate. For XML indexes, ideas from existing XML indexes with variable accuracy can be exploited.

- **Refined Control Policies**: In the scope of this thesis, AGILE uses a fairly straightforward feedback policy. Similarly, batch sizes are determined by flushing all available events. More elaborate control policies may be developed that improve adaptation to load changes when AGILE and batch operations are combined, or specific QoS requirements are needed. Similarly, feedback policies could be used to tune other aspects of information filters, such as attribute order or the clustering of batches.

- **Advanced Quality of Service**: Quality of service was reviewed for a coarse-grained setting and existing methods in this thesis. One direction of future work for QoS is to establish requirements and algorithms for fine-grained QoS, thus allowing to
support different QoS policies on the same information filter. Another direction is to develop algorithms that combine load shedding with batched processing of updates so that the benefits of both approaches can be exploited and a gradual transition is possible.

- **Distributed Context-Aware Information Filters**: In this thesis, context-aware information filters were studied for a centralized information filter. Using context information in distributed information filters provides a number of research opportunities, since a new stream of state update events needs to be taken into account. The location where the state should be kept—as close to the subscriber as possible or distributed over the network—is also an interesting issue.

- **Context-Aware Information Filtering over Sensors Networks**: This thesis and most existing work have considered information filtering on fairly powerful systems that route messages at high rates for a large number of profiles. The rise of sensor networks brings a different scenario for information filtering into focus: small-scale systems organize themselves into networks and exchange information. Context-aware information filtering provides opportunities to optimize the filtering process. To work efficiently the physical context need to be considered as well, optimizing for parameters such as network properties, available processing power and energy.
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Steps Executed when Using <code>findIntervals</code> (Fig 4.3) in Figure 4.4</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Notation for Cost Model in Section 4.3</td>
<td>31</td>
</tr>
<tr>
<td>5.1</td>
<td>Steps Executed when Using ISL <code>findIntervals</code> (Fig 4.3) in Figure 5.7</td>
<td>43</td>
</tr>
<tr>
<td>5.2</td>
<td>Steps Executed when Using ISL <code>findIntervalsUnion</code> (Fig 5.6) in Figure 5.7</td>
<td>44</td>
</tr>
<tr>
<td>5.3</td>
<td>Notation for Cost Model in Section 5.4.2</td>
<td>48</td>
</tr>
<tr>
<td>5.4</td>
<td>Workload Parameters</td>
<td>50</td>
</tr>
<tr>
<td>5.5</td>
<td>Cost Breakdown of 5.9 BS 75</td>
<td>55</td>
</tr>
<tr>
<td>5.6</td>
<td>Cost Breakdown of 5.9 BS 5K</td>
<td>55</td>
</tr>
<tr>
<td>5.7</td>
<td>Speedup for Bigger BS vs Unbatched</td>
<td>56</td>
</tr>
<tr>
<td>5.8</td>
<td>Maximum Speedup for Given Latency</td>
<td>60</td>
</tr>
<tr>
<td>5.9</td>
<td>Latencies for Bursty Traffic (in ms)</td>
<td>60</td>
</tr>
<tr>
<td>7.1</td>
<td>Overview of Symbols in Section 7.1.2</td>
<td>84</td>
</tr>
<tr>
<td>7.2</td>
<td>Notation for Cost Model in Section 7.2</td>
<td>97</td>
</tr>
<tr>
<td>7.3</td>
<td>Workload Parameters</td>
<td>99</td>
</tr>
<tr>
<td>7.4</td>
<td>Exp. 1, Throughput [Msg/sec], Vary UP</td>
<td>101</td>
</tr>
<tr>
<td>7.5</td>
<td>Exp. 1, Index Updates [Mio.], Vary UP</td>
<td>101</td>
</tr>
<tr>
<td>7.6</td>
<td>Exp. 1, Profiles to Postfilter [Mio.], Vary UP</td>
<td>101</td>
</tr>
</tbody>
</table>
List of Tables

7.7 Exp. 4, Average Throughput ........................................... 105
8.1 Notation for Cost Model in Section 8.4 ................................. 113
8.2 Matches for Different Execution Orders in Batching (see Figure 8.4) . . 114
8.3 Workload Parameters ...................................................... 117
9.1 Workload and Tuning Parameters ...................................... 147
List of Figures

1.1 Information Filtering ........................................ 2

2.1 Design Space of Existing Information Filters – Adapted from [38] .... 8

3.1 Example of Profiles and Messages ................................ 19

3.2 Processing Model of a Information Filter ................. 20

4.1 Information Filter Architecture ................................. 21

4.2 Interval Skip List (ISL) .................................. 25

4.3 The findIntervals Algorithm .................................. 26

4.4 ISL – findIntervals for k=18 .............................. 27

5.1 Information Filter (IF) Architecture ........................ 34

5.2 Batch-Enhanced IF Architecture ............................. 34

5.3 Minibatching Strategies ..................................... 36

5.4 Message Index on Second Attribute .......................... 38

5.5 Delivery Options for Batching ................................. 39

5.6 The findIntervalsUnion Algorithm ............................ 42

5.7 Batched Probe on ISL: Before Start ........................ 43

5.8 Batched Probe on ISL: Before Handling Search Value 13 .... 43
## List of Figures

<table>
<thead>
<tr>
<th>Figure No.</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.9</td>
<td>Comparing Batching Strategies - Range Profiles - RQ-Z (Zipf) Message Distribution</td>
<td>51</td>
</tr>
<tr>
<td>5.10</td>
<td>Comparing Batching Strategies - Range Profiles - RQ-G (Gaussian) Message Distribution</td>
<td>52</td>
</tr>
<tr>
<td>5.11</td>
<td>Comparing Batching Strategies - Range profiles - RQ-U (Uniform) Message Distribution</td>
<td>52</td>
</tr>
<tr>
<td>5.12</td>
<td>Comparing Batching Strategies - PQ</td>
<td>54</td>
</tr>
<tr>
<td>5.13</td>
<td>Competing Grouping Approaches for Minibatching</td>
<td>56</td>
</tr>
<tr>
<td>5.14</td>
<td>Tuning MBS</td>
<td>57</td>
</tr>
<tr>
<td>6.1</td>
<td>Example Messages, Contexts and Profiles</td>
<td>68</td>
</tr>
<tr>
<td>6.2</td>
<td>Possible Relations between Contexts and Profiles</td>
<td>69</td>
</tr>
<tr>
<td>6.3</td>
<td>Processing Model of a CIF</td>
<td>71</td>
</tr>
<tr>
<td>6.4</td>
<td>Architecture of a CIF: Separate Static and Context Filtering</td>
<td>72</td>
</tr>
<tr>
<td>6.5</td>
<td>Architecture of a CIF: Integrated Filtering</td>
<td>72</td>
</tr>
<tr>
<td>6.6</td>
<td>Architecture of a CIF: Extensions over Conventional IF</td>
<td>73</td>
</tr>
<tr>
<td>6.7</td>
<td>The \textit{NOINDEX} Algorithm</td>
<td>76</td>
</tr>
<tr>
<td>6.8</td>
<td>The \textit{EAGER} Algorithm</td>
<td>77</td>
</tr>
<tr>
<td>6.9</td>
<td>The \textit{PARTIAL} Algorithm</td>
<td>78</td>
</tr>
<tr>
<td>6.10</td>
<td>The \textit{GBU} Algorithm</td>
<td>80</td>
</tr>
<tr>
<td>7.1</td>
<td>Escalate: A=2 $\rightarrow$ A=3</td>
<td>82</td>
</tr>
<tr>
<td>7.2</td>
<td>Cheap Update: A=3 $\rightarrow$ A=1</td>
<td>82</td>
</tr>
<tr>
<td>7.3</td>
<td>Deescale A</td>
<td>82</td>
</tr>
<tr>
<td>7.4</td>
<td>The \textit{AGILE} Algorithm</td>
<td>87</td>
</tr>
<tr>
<td>7.5</td>
<td>ISL Before Escalation</td>
<td>88</td>
</tr>
<tr>
<td>7.6</td>
<td>ISL Escalation: $c=[4,15] \rightarrow c=[8,12]$</td>
<td>88</td>
</tr>
</tbody>
</table>
7.7 Escalation: Edge and Node Candidates for c = [4,15] \to c := [8,12] ...... 90
7.8 Escalation: Chosen Edges and Nodes for c = [4,15] \to c := [8,12] ...... 90
7.9 Simplified Description of Figure 7.8 ........................................ 90
7.10 Escalation Cases in an ISL ................................................... 92
7.11 The escalate Algorithm - Simplified ........................................ 93
7.12 Exp. 1, Normalized Troughput, Vary UP .................................. 100
7.13 Exp. 2, Completion Time, Vary UpdAtt .................................... 103
7.14 Exp. 3, Completion Time, Vary \Delta U .................................... 103
7.15 Exp. 4, Throughput, Update Burst ......................................... 104
7.16 Exp. 4, Vary FP on AGILE .................................................... 104

8.1 Extensions for Update Batching ................................................. 108
8.2 The updateIntervalsBulked Algorithm ..................................... 110
8.3 Usage of Resulting Set in Batch ............................................. 111
8.4 Batching Execution Orders ..................................................... 114
8.5 Bulk Index Updates - Scaling P - Throughput .......................... 117
8.6 Bulk Index Updates - Measuring Operations - Throughput .......... 118
8.7 Bulk Index Updates - Measuring Operations - Operations .......... 118
8.8 Bulk Index Updates - Scaling BS - Throughput ......................... 118
8.9 Resulting Set - Attribute Skew - Throughput ............................. 121
8.10 Resulting Set - Attribute Skew - Resulting Updates .................. 121
8.11 Resulting Set - Attribute + Profile Skew - Throughput ............ 123
8.12 Resulting Set - Attribute + Profile Skew - Resulting Updates .... 123
8.13 Batching Updates - Throughput ............................................ 125
8.14 Batching Updates - Resulting Updates .................................... 125
8.15 Batching Updates - Scaling BS ........................................ 125
8.16 Batching Both - Batched vs Unbatched - Throughput ............. 126
8.17 Batching Both - Scaling BS - MUP 1M, Msg First .................. 128
8.18 Batching Both - Scaling BS - MUP 10K, Msg First ................ 128
8.19 Batching Both - Scaling BS - MUP 100, Msg First ................ 128
8.20 Comparing Batch Execution Orders - Throughput .................... 129
8.21 Comparing Batch Execution Orders - Errors .......................... 130

9.1 Information Filtering ..................................................... 136
9.2 Information Filter Architecture – Supporting QoS ..................... 139
9.3 Examples of Message and Update Batching ............................ 143
9.4 Batching Consistency Models .......................................... 144
9.5 Experiment 1: Provisioning ............................................. 149
9.6 Experiment 2: Latency for Steady State - Traditional ................. 150
9.7 Experiment 2: Latency for Steady State - Shedding (any) ............. 150
9.8 Experiment 2: Latency for Steady State - Batching .................... 151
9.9 Experiment 3: Latency for Message Burst - Traditional (any) ....... 152
9.10 Experiment 3: Latency for Message Burst - Shedding (Messages, Both) 152
9.11 Experiment 3: Latency for Message Burst - Batching ............... 153
9.12 Experiment 3: Jitter for Message Burst - Traditional (any) ......... 153
9.13 Experiment 3: Jitter for Message Burst - Shedding (Messages, Both) 153
9.14 Experiment 3: Jitter for Message Burst - Batching ................... 154
9.15 Experiment 3: Errors for Message Burst - Shed Messages .......... 154
9.16 Experiment 3: Errors for Message Burst - Shed Both ............... 155
9.17 Experiment 3: Errors for Message Burst - Batching ............... 156
List of Figures

9.18 Experiment 4: Latency for Update Burst - Traditional (EAGER) ....... 158
9.19 Experiment 4: Latency for Update Burst - Traditional (AGILE) ........ 158
9.20 Experiment 4: Latency for Update Burst - Shedding (Updates, Both) ... 158
9.21 Experiment 4: Latency for Update Burst - Batching ................. 159
9.22 Experiment 4: Errors for Update Burst - FP Shed Updates ............ 159
9.23 Experiment 4: Errors for Update Burst - FP Shed Both ............... 160
9.24 Experiment 4: Errors for Update Burst - Batching ................. 161
9.25 Experiment 5: Latency for Combined Burst - Traditional (any) ...... 161
9.26 Experiment 5: Latency for Combined Burst - Shedding (Messages, Both) 162
9.27 Experiment 5: Latency for Combined Burst - Batching ............... 162
9.28 Experiment 5: Errors for Combined Burst - Shed Messages .......... 163
9.29 Experiment 5: Errors for Combined Burst - Shed Both ............... 164
9.30 Experiment 5: Errors for Combined Burst - Batching ............... 165
Seite Leer / Blank leaf
Bibliography


Curriculum Vitae  
Peter Michael Fischer

Citizenship: German

Date of Birth: 15th of April 1977 in Munich, Germany


May 2003 – August 2004 Research assistant in the group of Database Systems, Department of Computer Science, Universität Heidelberg. PhD student advised by Prof. Kossmann

May 2002 – April 2003 Research assistant in the group of Database Systems and Knowledge, Bases, Department of Computer Science, TU München. PhD student advised by Prof. Kossmann

Nov. 1997 - April 2002 MSc student in Computer Science, secondary subject Economics at TU München

October 2001 – March 2002 Visiting Student at the Database Group at UC Berkeley (Prof. Franklin)

Fall 2000 – Summer 2002 Participation and graduation at the Center for Digital Technology and Management (CDTM) of LMU München and TU München

September 1999 – March 2000 SOCRATES/ERASMUS exchange semester at the Queen’s University of Belfast

1996 – 1997 Mandatory civilian service („Zivildienst“) at the university hospital Grosshadern/Munich, Med. Klinik I (cardiology)

1996 Scholarship of the Bavarian State („Bayerischen Begabtenförderung“)

1987 – 1996 High School: Wilhelmsgymnasium München, Munich, Germany

1983 – 1987 Primary School: Grundschule an der Ostpreußenstraße, Munich, Germany