AGILE: Adaptive Indexing for Context-Aware Information Filters

Jens-Peter Dittrich
jens.dittrich@inf.ethz.ch

Peter M. Fischer
peter.fischer@inf.ethz.ch

Donald Kossmann
kossmann@inf.ethz.ch

Institute of Information Systems
ETH Zurich
8092 Zurich, Switzerland
www.dbis.ethz.ch

ABSTRACT

Information filtering has become a key technology for modern information systems. The goal of an information filter is to route messages to the right recipients (possibly none) according to declarative rules called profiles. In order to deal with high volumes of messages, several index structures have been proposed in the past. The challenge addressed in this paper is to carry out *stateful* information filtering in which profiles refer to values in a database or to previous messages. The difficulty is that database update streams need to be processed in addition to messages. This paper presents AGILE, a way to extend existing index structures so that the indexes adapt to the message/update workload and show good performance in all situations. Performance experiments show that AGILE is overall the clear winner as compared to the best existing approaches. In extreme situations in which it is not the winner, the overheads are small.

1. INTRODUCTION

Information filter systems manage continuous streams of messages that must be routed according to rules or so-called profiles. Examples for such information filters are publish & subscribe systems [8] or Email spam filters.

In order to implement information filters, several methods have been proposed in the past. Examples are SIFT [21], LeSubscribe [9], or YFilter [7]. 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 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 paper 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 AGILE. 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 [16, 14, 15], 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 [19, 4], but with different goals and strategies.

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.

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.

1.2 Contribution Summary

This work makes four 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 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 3).

3. Based on a review on how existing methods can be adapted to support CIF in Section 4, Section 5 presents the generic algorithm `AGILE`. This algorithm extends best-of-breed index structures to automatically adapt to high update rates.

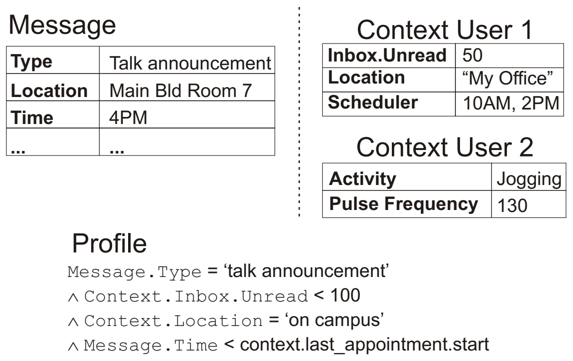
4. We present the results of comprehensive performance experiments in order to study the tradeoffs of `AGILE` (Section 6).

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". No message must be dropped or sent to the wrong user because a change in context has not yet been considered by the filter. This constraint rules out methods that update the context only periodically. In the following, we define the terms context, profile and message.

2.1 Context

In the literature on ubiquitous [20] and context-aware computing, a wide range of context definitions has been proposed. Most of these definitions focus on the notion of location ([6] gives an overview). 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 1 gives examples of the contexts of two users. User 1 is currently in her office, whereas the location of User 2 is unknown.



Message

Type	Talk announcement
Location	Main Bld Room 7
Time	4PM
...	...

Context User 1

Inbox.Unread	50
Location	"My Office"
Scheduler	10AM, 2PM

Context User 2

Activity	Jogging
Pulse Frequency	130

Profile

`Message.Type` = 'talk announcement'
$\wedge$ `Context.Inbox.Unread` < 100
$\wedge$ `Context.Location` = 'on campus'
$\wedge$ `Message.Time` < context.last_appointment.start

Figure 1: Example messages, contexts and profiles

Gathering context information is outside the scope of this paper and has been addressed, e.g., in the work on sensor networks [22], data cooking [11], context/sensor fusion [3], or the Berkeley HiFi project [5]. 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.

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 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.

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 define profiles as proposed in pub/sub-systems [9], using the disjunctive normal form (DNF) of atomic comparisons. This definition allows the use of context information in profiles in multiple ways: Message attributes can be compared with constants and with context attributes. The latter is the novel aspect of this work and a significant extension to the way profiles are defined in traditional pub/sub systems and information filters. This extension enables easier modeling, richer profiles and, as we will see, several opportunities for optimization.

An example for a profile is displayed in Figure 1. The profile asks for the delivery of messages of the type *talk announcement*, if the talk is today, the user is on campus, has less than 100 unread mails, and the talk starts before the last appointment.

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.

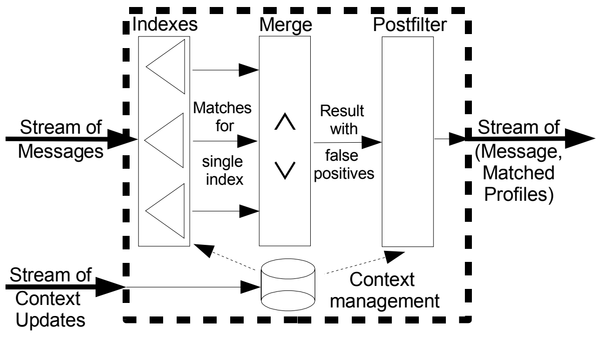

Indexes
Merge
Postfilter
Stream of Messages
Matches for single index
Result with false positives
Stream of (Message, Matched Profiles)
Stream of Context Updates
Context management

Figure 2: Architecture of a CIF

3. CONTEXT-AWARE INFORMATION FILTERS

This section introduces a processing model and a reference architecture for *Context-Aware Information Filters (CIF).*

3.1 CIF Processing Model

Figure 2 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. These two streams are serialized so that at each point in time either one message or one update is processed.

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.

3.2 CIF Architecture

As shown also in Figure 2, a CIF has four main components: (a) context management, (b) indexes, (c) merge, (d) postfilter. A similar architecture without context management has also been devised for (traditional, non-context-aware) information filters in [10]. In the following, the purpose of each component and suitable algorithms are described.

3.2.1 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. Both indexes and 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.

3.2.2 Indexes

Given a message, the information filter must find all profiles that match. This filtering can be accelerated by indexing the profiles or predicates of the profiles. The most important method supported by an index is `probe`, 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. Furthermore, an index provides `insert` and `delete` methods in order to register new profiles or delete existing profiles. As will be shown in Section 5, an index should also support an `update` method in order to deal with context updates. Indexes for profiles and predicates have been used in traditional information filtering systems and have been subject to extensive studies in literature; e.g., [21, 9, 13, 7].

An index can be classified by four different aspects:

- **Target:** *Value indexes* index the constants and values of attributes. The B-Tree, R-Tree [12], R*-Tree [2], Spatial/Moving Object Indexes [16] and Interval Indexes [13] 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 [7] and Data Guide [1]. For the model of Section 2, only value indexes are relevant. Integrating structure indexes into the `AGILE` framework is future work.

- **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 2 in a final postfilter step. By allowing false positives, the performance of index operations can be improved. The drawback are increased costs 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 values. These indexes are called *partial indexes* [18]. 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 5, 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.

3.2.3 Merge

As mentioned in Section 2.3, profiles are conjunctions and disjunctions of predicates. Since it is not efficient to use a high-dimensional index to cover all conjunctions and disjunctions [9], 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 [9, 10]. 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).

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). Since the implementation of the `merge` operation was not the focus of this work, we chose to exploit techniques described in [9] because these seemed to represent the state-of-the-art in this respect.

3.2.4 *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 based on the current state of the context. Of course, short-circuit evaluation for conjunctions and disjunctions 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. STATE-OF-THE-ART

This section describes existing approaches to implementing information filters and shows how these can be adapted for context-aware information filters (CIF).

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 3 shows pseudocode for the implementation of the `handle_message` and `update_context` operations based on the CIF processing model of Section 3.1. 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.

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 [9]. Figure 4 shows the pseudocode for this approach, where *AttI* depicts the

```
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 3: The `NOINDEX` Algorithm

```
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 4: The `EAGER` Algorithm

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 [9]. 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 expensive because it involves maintaining indexes, potentially with every context update.

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 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.

The idea of partial indexing goes back to Stonebraker [18], and was further studied in [17]. 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 5 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.

4.4 Lazy Updates, GBU

Lately, there has been work on moving object databases [16, 15] 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 make big jumps 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 [14]. 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`. Due to space constraints, it is not possible to describe GBU in full detail. The interested reader is referred to [14].

5. ADAPTIVE INDEXING: AGILE

This section presents the `AGILE` (**A**daptive **G**eneric **I**ndexing with **L**ocal **E**scalations) algorithm. We present the general idea of the algorithm in Section 5.1. The following subsections give a more formal and complete description of the approach and show how it can be applied to interval skips lists [13], the best known value index structure for information filtering.

5.1 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.

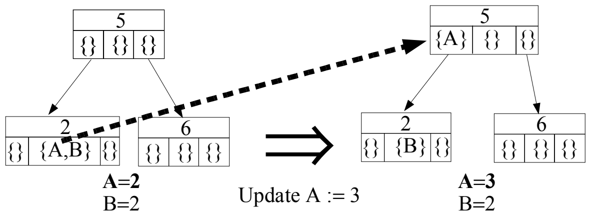


Figure 5: Escalate: A=2 → A =3

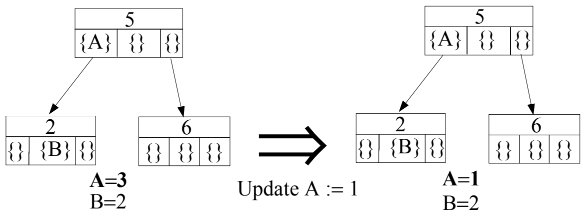


Figure 6: Cheap Update: A=3 → A =1

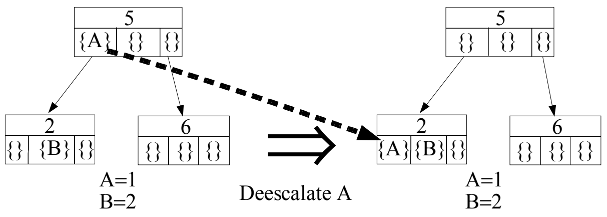


Figure 7: Deescalate A

Example. To demonstrate the key ideas of the approach, Figures 5 to 7 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 5) 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 5 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 5). 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 5 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. The advantage is that 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 6, 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 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, 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, 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 [13](Section 5.4). 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 5.5.

5.2 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. `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 1). What makes `AGILE` special is that it generalizes indexing and associates an identifier to a *set* of keys, denoted as $\mathcal{K}(i)$. In the left-hand tree of Figure 5, for instance, identifier A is associated to the set of keys $\{2\}$, i.e., $k(A) = \{2\}$. In the right-hand tree of Figure 5 (after escalation), identifier A is associated to the set of keys $]-\infty, 5[$.

Symbol	Meaning
i	Identifier of an object
$k(i)$	Key of object i
$\mathcal{K}(i)$	Set of keys associated with object i by the index
$\mathcal{E}(i)$	Set of keys associated with object i after escalation
$\mathcal{D}(i)$	Set of keys associated with object i after deescalation
Conditions	$k(i) \in \mathcal{K}(i)$, $\mathcal{E}(i) \supset \mathcal{K}(i)$, $\mathcal{D}(i) \subset K(i)$

Table 1: Overview of symbols in Sec 5.2

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

$$probe(k) = \{i | k \in \mathcal{K}(i)\}$$

In other words, the index returns identifier i when probed for key k if and only if k is in the set of keys $\mathcal{K}(i)$ 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 $k(i) \in \mathcal{K}(i)$.

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 $\mathcal{K}(i)$, then nothing needs to be done in order to implement the `update`; in other words, `update` becomes a no-operation in that case. Figure 6 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 $\mathcal{K}'(i) = \mathcal{K}(i) \cup \{k'(i)\}$.

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., $\mathcal{E}(i) \supset \mathcal{K}(i)$. 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., $\mathcal{D}(i) \subset K(i)$.

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, $\mathcal{E}(i)$ and $\mathcal{D}(i)$ can be any set of keys that fulfill the constraints $k(i) \in \mathcal{E}(i)$ and $k(i) \in \mathcal{D}(i)$, respectively. In practice, however, $\mathcal{E}(i)$ and $\mathcal{D}(i)$ are determined by the particular index structure used.

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

Procedure AGILE.update_context
Input:   Context c, Attribute a, Value v
 (1)  For (i:=1 to Att)
 (2)      If (AGILEindex[i] indexes a ∧
                  (c.a ∉ AGILEindex[i].probe(v))
 (3)          AGILEindex[i].escalate(c, v)
 (4)      EndIf
 (5)  EndFor
 (6)  DataStore[c].a := v;
```

Figure 8: The AGILE Algorithm

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 *escalatations* 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.

5.3 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 8 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 4, Section 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 5.5.

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 4).

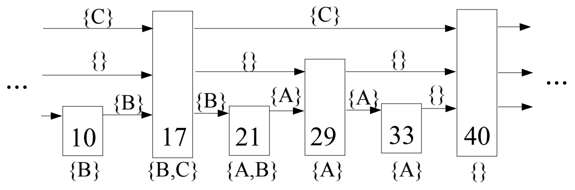


Figure 9: ISL A=[21;33], B=[10;21], C=[0;40]

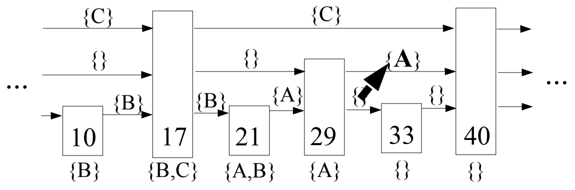


Figure 10: ISL Escalation: A=[21;33]→ A=[21;36]

5.4 AGILE Indexes

5.4.1 AGILE Interval Skip Lists

The best known value index structure for information filtering is the interval skip list (ISL) [13]. An ISL is a hierarchical index structure that is 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 9 gives an example for an ISL on a numeric domain: the number of items ordered in an incoming purchase order message. The three identifiers A, B, and C correspond to predicates of profiles of warehouses. Identifier A refers to the predicate 21 ≤ order.quantity ≤ 33, thereby specifying that Warehouse A is only interested in orders with quantities in that range. Identifier B refers to the predicate 10 ≤ order.quantity ≤ 21, Identifier C refers to the predicate 0 ≤ order.quantity ≤ 40. As a consequence, A is associated with the ranges [21;29] and [29;33]. Likewise, B is associated with the ranges [10;17] and [17;21]. Finally, C is associated with the ranges [0;17] and [17;40]. (The range marker for 0 is not shown in Figure 9. In addition, the skip list in figure 9 contains other intervals which are not explicity marked for ease of understanding, i.e. starting and ending at 17 and 33.) If a new order with, say, quantity = 25 arrives, this messages if filtered in the following way. First the top-level ranges are inspected in order to find the matching top-level range: that is [17;40]. Since C is associated with that range, C is immediately output as a matching profile. Next, the matching second-level range is found by inspecting the children of [17;40]: the result is range [17;29]. Since no identifier is associated with this range, no output is generated in this step. Finally, the matching bottom-level range is found among [17;29]'s children: that is [21;29]. Since A is associated with this range, A is output.

Figure 10 shows how an escalation can be implemented in an ISL. Predicate A is changed from [21;33] to [21;36], possibly due to an increase in the number of items on stock. As a result, A is no longer associated with the (bottom-level) range [29;33]. Instead, A is associated with the (second-level) range [29;40]. In general, an identifier is promoted to the lowest higher-level range that fully covers the new range. As for the simple binary tree in Section 5.1, escalations can result in false positives. After escalation, for instance, Warehouse A will be considered as a match for an order with `quantity = 39`. On the positive side, another update, say, from [21;36] to [21;39] 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).

5.4.2 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.

5.5 Deescalation Policies

As mentioned in Section 5.3, it is important to control when deescalations occur. There are several different policies 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 8 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. *always* is a special case of *fixed* with $FP = 1$. In practice, a good value of FP is 1000 (Section 6).

3. **Auto**: *auto* operates like *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 deescalations should be rare (large FP). On the other hand, if many false positives are detected as part of `handle_message` operations, then deescalations 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 this paper.)

We will study the trade-offs of the different approaches in Section 6. Clearly, more elaborate policies are conceivable and we plan to study such policies as one avenue for future work; the experiments presented in Section 6, however, indicate that the simple policies described here work very well for a large range of workloads.

6. 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 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.

6.1 Software and Hardware used

The information filter used for the performance experiments was built using the architecture described in Section 3. In order to implement the individual components, the following design choices were made:

1. **Context Management:** The 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 [9]. In addition, there are links which directly connect each profile to its corresponding context.

2. **Indexes:** As an index structure, Interval Skip Lists [13] (ISL) were used for all index-based approaches (`EAGER, GBU, AGILE`). ISL are the state-of-the-art indexes to index predicates for information filtering. The ISL implementation used is from [13]. Not all predicates of all profiles were indexed, but only the most selective attributes (as proposed in, e.g., [9]) so that postfiltering was always necessary. Depending on the cho-

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

Table 2: Workload parameters

sen approach, however, more or less profiles had to be postfiltered.

3. **Merge:** The Merge component was implemented making use of (compressed) bitmaps in order to represent sets of profiles.

4. **Postfilter:** This step was implemented by applying each profile of the intermediate set to each message individually. Short-circuiting was used if a profile consisted of conjunctions and disjunctions of predicates. Furthermore, we laid out the contents of the predicates in memory in such a way that the access locality (processor cache hits) was improved.

Based on this infrastructure, we implemented the following approaches: `NOINDEX` (Section 4.1), `EAGER` (Section 4.2), `GBU` (Section 4.4) and `AGILE` (Section 5). 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 software was implemented in C++. 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 of context updates (depending on the workload parameter settings) before the system throughput was measured.

6.2 Workload

When selecting the workloads to test the different methods, we followed the requirements derived from the Use Cases (Section 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 2. 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[1]. 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 (*AttI*). The number of profiles (*P*) was 500,000 for all experiments.

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 [13]. 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 $[-\Delta U; +\Delta U]$; ΔU 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.

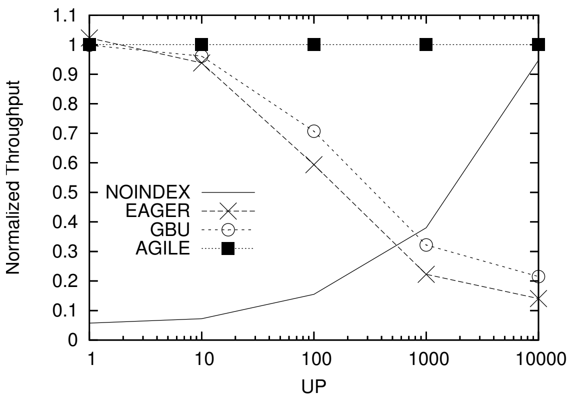


Figure 11: Exp1, Normalized Troughput, Vary UP
ΔU=150, *UpdAtt*=25, FP=1,000

6.3 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). ΔU was set to [-150;+150], and $FP = 1000$.

Figure 11 shows the relative throughput, normalized to the throughput of `AGILE`. Table 3 shows the absolute through-

[1]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 in this paper.

UP	1	10	100	1,000	10,000
NOINDEX	27	28	28	21	9.5
EAGER	484	366	106	13	1.4
GBU	472	375	125	19	2.1
AGILE	472	390	178	58	10.1

Table 3: Exp. 1, Throughput [Msg/sec], Vary *UP*

UP	1	10	100	1,000	10,000
NOINDEX	0	0	0	0	0
EAGER	0.024	0.24	2.4	24	240
GBU	0.024	0.23	2.2	20	180
AGILE	0.024	0.22	1.4	4	11

Table 4: Exp. 1, Index Updates [Mio.], Vary *UP*

UP	1	10	100	1,000	10,000
NOINDEX	4,000	4,000	4,000	4,000	4,000
EAGER	12	12	12	12	12
GBU	12	12	14	19	32
AGILE	13	19	72	250	700

Table 5: Exp. 1, Prof. to Postfilter [Mio.], Vary *UP*

put 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 4) and the number of profiles that need to be inspected in the postfilter operation (Table 5). As expected, the number of index updates increases for all index-based methods with an increasing context update rate (Table 4). 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 postfiltered increases with an increasing update rate for AGILE (Table 5) 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.

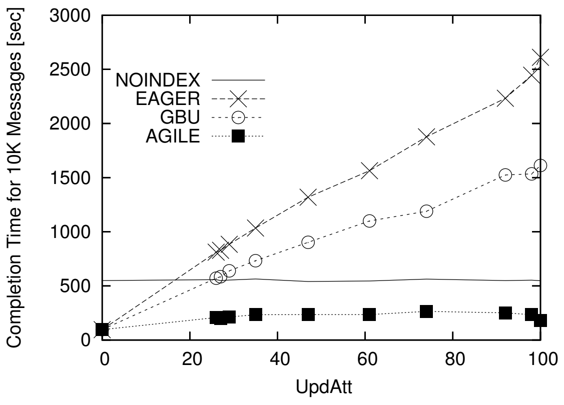


Figure 12: Exp. 2, Completion Time, Vary *UpdAtt*
ΔU=150, *UP*=1,000, *FP*=1,000

6.4 Experiment 2: Vary *UpdAtt*

The second experiment studies the impact of varying the distribution of updates to indexed and non-indexed attributes (*UpdAtt*). Figure 12 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 is also 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.

6.5 Experiment 3: Vary ΔU

Both GBU and AGILE take advantage of the locality of context updates. Therefore, parameter ΔU impacts their performance. Figure 13 shows the completion time for varying ΔU from very high update locality (ΔU close to 0) to very low update locality (ΔU = 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 ΔU 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.

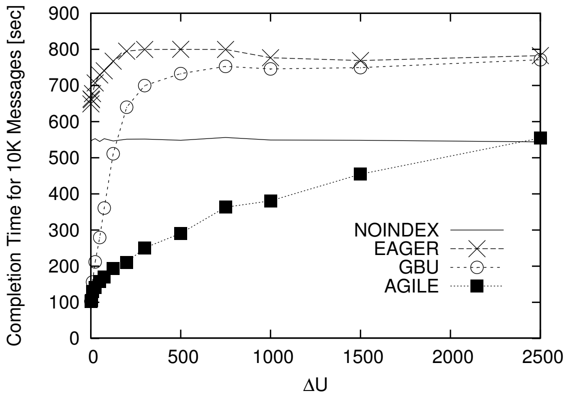


Figure 13: Exp. 3, Completion Time, Vary ΔU
UP=1,000, *UpdAtt*=25, *FP*=1,000

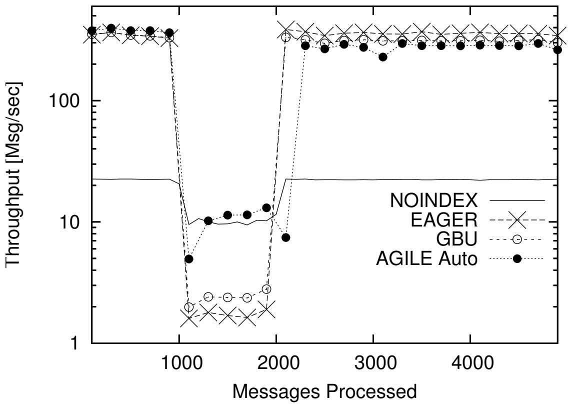


Figure 14: Exp. 4, Throughput, Update Burst
ΔU=150, *UpdAtt*=25

6.6 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 14 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 approach in this situation. In the other phases, it shows (almost) the same performance as `EAGER` and `GBU` which are the best options if

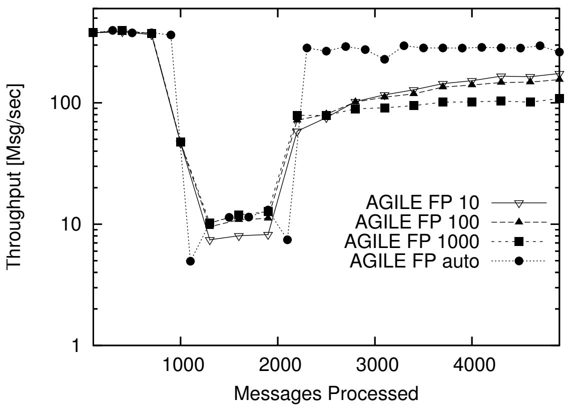


Figure 15: Exp. 4, Vary FP on AGILE
ΔU=150, *UpdAtt*=25

Alg/ FP	`NOINDEX`	`EAGER`	`GBU`	`AGILE`			
				Auto	10	100	1,000
Msg/sec	18	8.8	12.15	39.8	30.25	36.7	37.4

Table 6: Exp. 4, Average throughput

update rates are low. The average throughput for the whole workload (i.e., considering the total time to process all 5,000 messages) is shown in Table 6, and again it can be seen that `AGILE` is the overall best approach.

One interesting aspect of this experiment is to show how quickly `AGILE` can adapt to such bursts and to the end of such a burst. Figure 14 shows the throughput of `AGILE` if *auto* is used as a deescalation policy (Section 5.5). Looking closely at Figure 14, it can be seen that at the beginning of the burst, it indeed takes `AGILE` some time to adjust to the new workload characteristics so that for a short period of time, it shows sub-optimal performance. Also, it takes `AGILE` some time to adjust to the new workload characteristics when the burst is over: at this point, deescalations must be carried out in order to increase the accuracy of the indexes. In all, this experiment shows that `AGILE` is very well capable to adapt to changes in workload characteristics.

Figure 15 and Table 6 show how alternative deescalation strategies fare in this experiment. Indeed, *auto* outperforms *fixed* in this experiment, but the differences are not large. Furthermore, the *fixed* policy is not very sensitive to the right setting of its tuning parameter FP.

7. 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 **A**daptive **G**eneric **I**ndexing with **L**ocal **E**scalations. 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 [13]. 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.

Acknowledgements. We would like to thank Dongseop Kwon for the discussions at the beginning of this projects. We would also like to thank our colleague Cristian Duda for proofreading.

8. REFERENCES

[1] S. Abiteboul. Querying Semi-Structured Data. In *ICDT*, 1997.

[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*, 1990.

[3] R. R. Brooks and S. Iyengar. *Multi-Sensor Fusion: Fundamentals and Applications in Software.* Prentice Hall, 1997.

[4] H.-J. Cho, J.-K. Min, and C.-W. Chung. An Adaptive Indexing Technique Using Spatio-Temporal Query Workloads. *Information and Software Technology*, 46(4):229–241, 2004.

[5] O. Cooper, A. Edakkunni, M. J. Franklin, W. Hong, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, and E. Wu. HiFi: A Unified Architecture for High Fan-in Systems. In *VLDB*, 2004.

[6] A. K. Dey. Understanding and Using Context. *Personal and Ubiquitous Computing Journal*, 5(1):4–7, 2001.

[7] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *TODS*, 28(4):467–516, 2003.

[8] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[9] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In *SIGMOD*, 2001.

[10] P. M. Fischer and D. Kossmann. Batched Processing for Information Filters. In *ICDE*, 2005, to appear.

[11] A. R. Golding and N. Lesh. Indoor Navigation Using a Diverse Set of Cheap, Wearable Sensors. In *3rd International Symposium on Wearable Computing*, 1999.

[12] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984.

[13] E. Hanson and T. Johnson. Selection Predicate Indexing for Active Databases Using Interval Skip Lists. *Information Systems*, 21(3):269–298, 1996.

[14] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In *VLDB*, 2003.

[15] B. C. Ooi, K. L. Tan, and C. Yu. Frequent Update and Efficient Retrieval: An Oxymoron on Moving Object Indexes? In *Web Information Systems Engineering (Workshops)*, 2002.

[16] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD*, 2000.

[17] P. Seshadri and A. N. Swami. Generalized Partial Indexes. In *ICDE*, 1995.

[18] M. Stonebraker. The Case for Partial Indexes. *SIGMOD Record*, 18(4):4–11, 1989.

[19] Y. Tao and D. Papadias. Adaptive Index Structures. In *VLDB*, 2002.

[20] M. Weiser. Some Computer Science Issues in Ubiquitous Computing. *CACM*, pages 74–84, 1993.

[21] T. W. Yan and H. Garcia-Molina. The SIFT Information Dissemination System. *TODS*, 24(4):529–565, 1999.

[22] Y. Yao and J. Gehrke. Query Processing in Sensor Networks. In *CIDR*, 2003.