Batched Processing for Information Filters

Peter M. Fischer, Donald Kossmann
Swiss Federal Institute of Technology (ETH) Zürich, Switzerland
{peter.fischer,kossmann}@inf.ethz.ch

Abstract

This paper describes batching, a novel technique in order to improve the throughput of an information filter (e.g. message broker or publish & subscribe system). Rather than processing each message individually, incoming messages are reordered, grouped and a whole group of similar messages is processed. This paper presents alternative strategies to do batching. Extensive performance experiments are conducted on those strategies in order to compare their tradeoffs.

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 [3], or the Message Broker and bus from BEA [4] 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. 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 [13, 7, 14]. The indexing schemes differ in the kind of messages (XML or structured tuples) and profiles (keywords, predicates, complex queries) they support. In any case, the goal is 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 paper is to study 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 instead this set of messages. 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 is particularly effective during peak times in which temporarily more messages arrive than the system can handle. On the negative side, 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 paper always restore this order.

1.2 Contributions

In order to study batching, this paper makes the following contributions:

1. Introduce the concept and show how it can be applied to various existing indexing schemes.
2. Propose alternative batching strategies. The batching strategies differ in the way they group messages and in the way they index groups of messages.
3. Develop a cost model that helps to identify the critical parameters that affect the performance of the alternative batching strategies and, thus, helps to study performance trade-offs.

4. Present the results of performance experiments with different workloads and indexing schemes.

5. Devise a model that allows to define an upper bound for the latency of a message (in non-overload situations).

1.3 Structure of the paper

The remainder of this paper is structured as follows: Section 2 describes the architecture of information filtering systems into which batching can be incorporated. Sections 3 and 4 describe the batching framework, strategies to batch and implementation details. Section 5 gives a qualitative cost analysis, while Section 6 presents the results of performance experiments. Section 7 shows how the latency of messages can be controlled and Section 8 discusses related work. Section 9 concludes this work and gives avenues for future work.

2 Information filter architecture

The purpose of an information filter is to match messages to profiles. In a relational world, such messages are attribute/value pairs and profiles are conjunctions of predicates such as equality, range or set containment [14]. 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 consisting of the predicates (x=3) $\wedge$ (z $>$ 2), whereas it does not match the profile (z $\leq$ 0). In an XML world, messages are XML documents, profiles are XPath [13] or XQuery statements, and matches are defined accordingly.
Figure 1 gives an overview of the architecture of an information filter, following 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 messages while the filter is busy. 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 can involve several predicates, the sets of profiles returned by each index need to be merged. Logically, the merging step carries out conjunctions and disjunctions. 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,

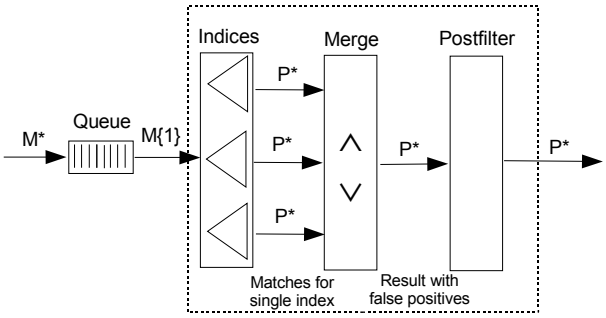


Figure 1. Information filter architecture

cache awareness). Since a profile can involve additional predicates that are not indexed, a postfilter step is necessary in order to evaluate those predicates. We will now look more closely at options to implement the index stage. Details on the other stages will be discussed in Sections 5 and 6.

Indexing to speed up finding matching predicates has been used in traditional information filters. These have been subject to extensive studies in the literature [14, 13]. There is a number of alternative ways to implement such indexes:

- **index type:** Hash table, B-Tree, R-Tree [16, 8], Spatial/Moving Object Indexes [18], Interval Indexes [17], custom indexes, etc.

- **index target:** Value indexes index the values of predicates; all the indexes listed in the above examples are typical representatives of value indexes. On the other hand, structure indexes index the structure of the profile; i.e. the attributes that are used in the index or in a more general XML context, the XPath expressions used in the profiles. Examples for such structure indexes are YFilter [13] or Data Guide [6]

- **index scope** It is possible that an index covers all predicates of all profiles. Alternatively, there could be several indexes, each index covering all predicates that involve a certain set of (message) attributes. Furthermore, indexing can be limited to certain ranges, as partial indexes [19]. Typically, only the most selective attributes are indexed for performance reasons.

3 Batching strategies

3.1 General idea

Considering the architecture laid out in Section 2, Figure 1, we can apply the following changes to enable batching, as shown in Figure 2: The *input queue* is now an integral part of the filter, as it needs to be controlled by a new

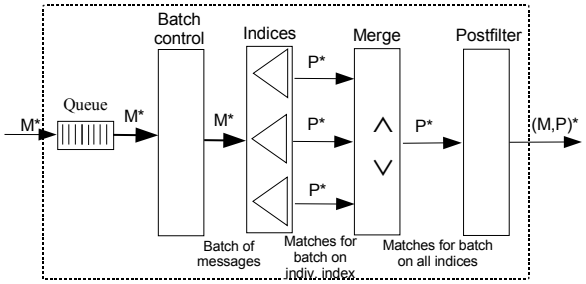


Figure 2. Batch-enhanced IF Architecture

component, the *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 can reorder and group the messages. In contrast to the traditional approach, the *predicate 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 [21].

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

There are two other possible improvements from batching. One, indexing the messages to speed up postfiltering, as messages fulfilling a certain predicate can be found faster. Two, the delivery of messages can be improved. These two advantages will be explained in more detail in Sections 3.3 and 3.4.

3.2 Grouping messages

Our goal is to save work on batched profile index accesses and merge while having low number of profiles in the union set. Just naïvely handling the largest possible batch is not a good strategy to achieve this goal. Instead, we have to break up the batch into smaller subsets that have a greater amount of similarity. The need for this improvement becomes clear if one considers 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 batchsizes, on the other hand, severely limit the room 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 is 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 batchsizes
- Taking advantage of skew in message values
- Working for different profile workloads

There are many ways to group. Here are some alternatives. We will study their tradeoffs in Section 6:

fixed-size: The complete batch is ordered and split into minibatches of a fixed size (e.g. 2 messages), as shown in Figure 3a). This method will most likely work well if the overall batchsize does not change much, and the values of the messages have a uniform distribution. For a varying overall batchsize, 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 3, b). This method takes into consideration that the size of

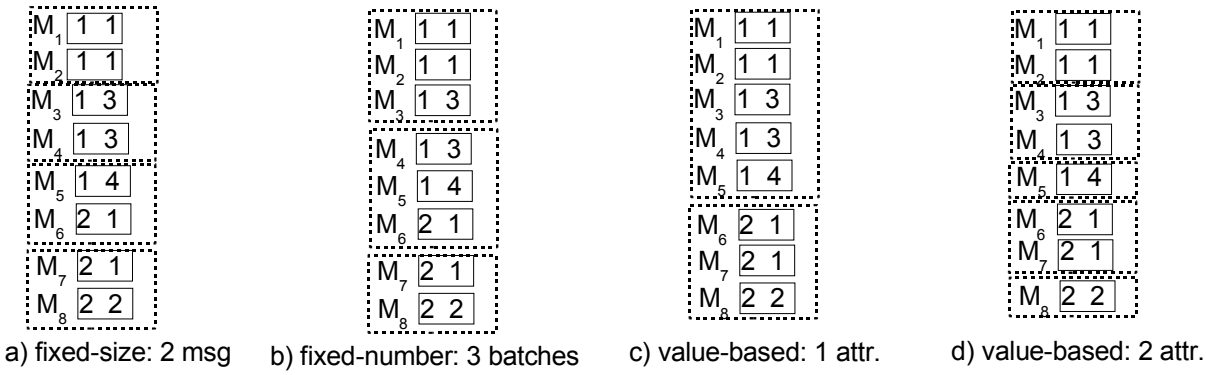


Figure 3. Minibatching strategies

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, but a large number of variants makes it difficult to choose the right approach. The two examples in Figure 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:

- Split the batch into 50 minibatches, but make the minibatches no smaller than 10 messages and no larger than 500 messages.
- Combine all messages differing not more than 50 from the designated group value, but ensure that a minibatch has at least 20 messages in it, otherwise merge it with the closest group

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

A common drawback of all these methods is that they have tuning parameters. We will study the sensitivity of those tuning parameters in Section 6.5 and present an approach in order to determine these parameters automatically.

3.3 Indexing messages

In Figure 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 equipredicates, hashing can be used. For range predicates, we propose 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.

3.4 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 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 b). This is useful in a unicast situation with direct

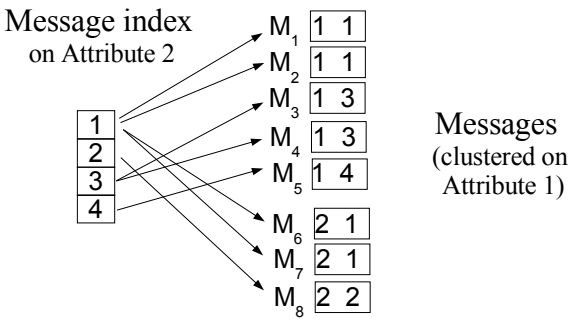


Figure 4. Message index on second attribute

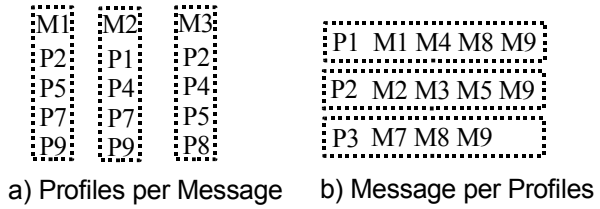


Figure 5. Delivery options for batching

delivery, as we send each client all relevant messages in a single transfer.

3.5 Summary of batching strategies

Considering the ideas outlined in this ection, 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 3.2)

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

3. Is a message index (MIx) used in the postfilter stage? The alternatives are yes and no. (Section 3.3).

Not all combinations within these three dimensions are 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 is detrimental, 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, detailled approaches in the rest of the paper

Unbatched/PIx/-: baseline, representing the current state of the art

Full/PIx/-: Apply a full batch at the profile indexes

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

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

MiniB/PIx/MIx: Generate message indexes on the individual minibatches to speed up postfiltering

4 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. We 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 is quite fast, 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), we 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, we can reduce the search cost by continuing at the last probed value and also build the result set incrementally.
A good example of such improvements can be shown on the interval skip list [17], which is generally used to index range predicates in main memory: As its name implies, it is a list carrying the indexed values at each node, but instead of having a single forward pointer at each node, it has a number of additional forward pointers that ”skip” over a range of values. The additional forward pointers are ordered in

"levels", the higher their level is the farther they reach out. The number of levels on a node is randomly determined at insertion time. Lower levels occur more often than higher levels.
In order to retrieve all intervals covering a probed value, the index is searched following the forward pointers. 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.
A batched access can continue its search by starting the search at the lowest possible forward pointer that had not been covered before and the intermediate result of the higher levels can be reused.

5 Qualitative analysis

5.1 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 batchsizes
- response time/latency, as batching introduces "waiting times".

This chapter will focus on a 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 7, since they depend more on the traffic characteristics than on one of the different batching strategies.

5.2 CPU cost

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 higher the skew, the higher the similarity of the messages in a batch, and thus the more effective batching becomes.
For postfiltering, the cost model does not change significantly, but the higher number of match candidates in the union set can have a negative impact. Using a message index on one attribute reduces the cost of evaluating this attribute from $O(num\_msg)$ to $O(log(num\_msg))$.
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.

5.3 Memory requirements

To handle batches of messages, we need additional memory over what a traditional information 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 consists 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.

6 Performance experiments and results

Following the analysis in the last section, our goal is to measure the troughput 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.

6.1 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 [17], and slightly modified to fit into our architecture. For point queries, we use a hash table implementation from the GHT library [5]. Both were extended as described in Section 4. For sorting values, we used the built-in `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.

6.2 Workloads

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

Parameter	Description	Values
P	No of profiles	500K
AR	No of attr. RQ	8
AttPR	No of attr./profile RQ	8
AttMR	No attr. per msg (RQ)	8
AP	No of attr. PQ	32
AttMQ	No attr. per msg (PQ)	32
AttPP	No of attr./profile PQ	4
CO	No of indexed attr.	2
PD	distribution of profile pred.	uniform
VRR	Values range (RQ)	[0,10000]
VRP	Value range (PQ)	[1,35]
MD	Dist. of msg values	Zipf, uniform, gauss
BS	Batchsize	1 - 100K
MBS	Minibatchsize (number)	1 - $\frac{BS}{2}$
MBS	Minibatchsize (range)	1 - 1250

Table 1. Workload Parameters

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

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 [17]. 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.

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 [14]. 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 * 10^6, corresponding to 1.3 matching profiles per message or a profile being match by every 380'000th document.

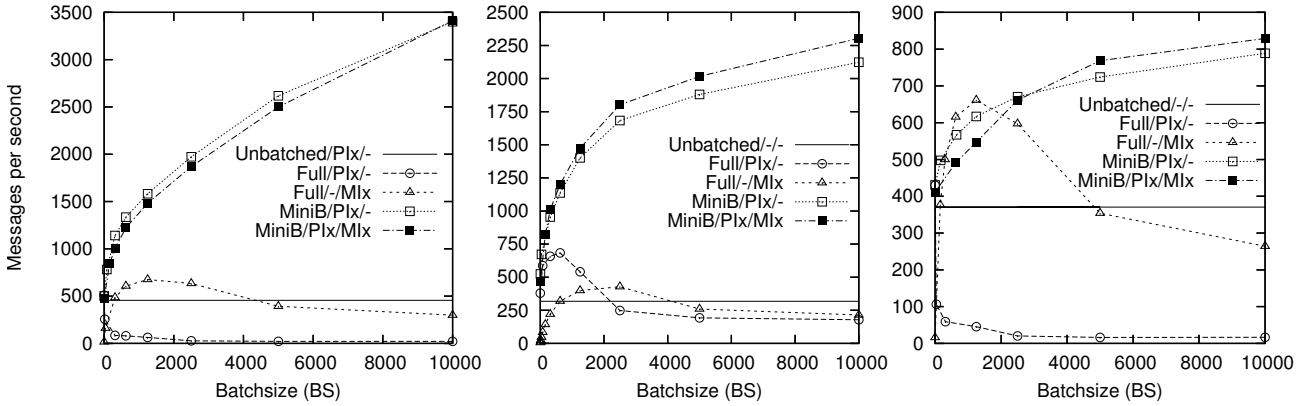
All workloads consisted of 500K profiles(P). In both cases, indexing the two most selective attributes yielded the best performance for the unbatched case(CO). All profile values had a uniform value distribution(PD).
Batchsizes were varied from 1 to 100K messages per batch(BS). For all experiments, the minibatches within a single batch had either an equal number of messages or covered the same range of values(MBS). For minibatchsizes based on the number of messages, we varied the number of messages in minibatch between 1 and half the actual batchsize. For minibatchsizes based on values, we changed the value range from 1 to 1250, as bigger values did not yield any benefits.

6.3 Comparison of batching strategies

6.3.1 Impact of batchsize

The purpose of our first set of experiments is to compare the individual batching methods and determine the respective benefits. For each of these experiments we measured the maximum throughput the filter could handle for various batchsizes between 1 and 10K. The unbatched method (Unbatched/PIx/-) represents the current state of the art and was compared against the batching methods presented in Section 3.5.

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 6 and 7 show the results. On RQ-Z (Figure 6a), unbatched reaches a throughput of about 450 msg/sec. Full batching with PIx is not able to achieve a comparable performance, it quickly degrades when increasing the batchsize, reaching just 20 msg/sec when the batchsize 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 performace on very small batchsizes (15 msg/sec at BS 1), but performs better at higher batchsizes. At a batchsize of



(a) RQ-Z (zipf) (b) RQ-G (gaussian) (c) RQ-U (uniform)

Figure 6. Comparing batching strategies - range profiles - varying message distributions

around 1250, the throughput is 50 percent higher than in the unbatched case (about 675 msg/sec). When further increasing the batchsize, the performance begins to drop slightly, and falls below the unbatched performance at a batchsize of 5000. The low performance at smaller batches in the increase 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` [20].

Minibatching with a profile index (MiniB/PIx/-) shows a significant performance improvement over all the other approaches. At a batchsize of 150, the performance is already twice as high as for unbatched. The gap grows with an increasing batchsize: at 10000 messages the speedup is about 7.5. Bigger batchsizes result in a higher number of similar messages in the batch, which can be grouped together, and thus to higher troughput. 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/PIX/MIx) leads to relatively similar results: On smaller batchsizes, the performance is slightly worse, for larger batchsizes, this approach catches up and matches (MiniB/PIx/-). 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 therefore not shown.

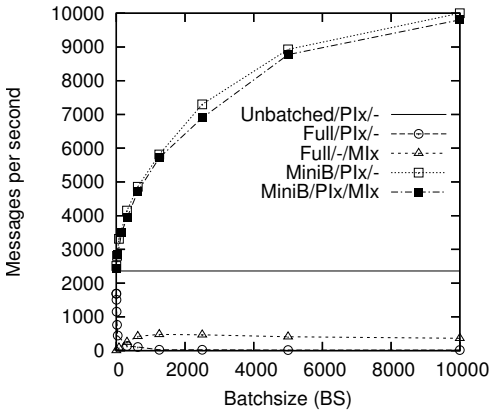


Figure 7. Comparing batching strategies - PQ

The results for RQ-G (Figure 6b) and RQ-U (Figure 6c) 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 7) 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 then a factor of 4.

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

6.3.2 Cost breakdown

The performance numbers of the respective strategies are verified when looking at the individual cost factors con-

	Ub./P/	Full/P/-	Full/-/M	Mini/P/-	Mini/P/M
SortMB	0	0	0	0.01	0.01
SortPIdx	0	0.01	0	0.01	0.01
PIx	14.28	1.97	0	5.51	7.87
SortMIx	0	0	0.01	0	0.01
MIx	0	0	71.58	0	12.98
PostF	6.54	77.58	28.76	8.27	6.08

Table 2. Cost breakdown of 6a) BS 75

	Full/P/-	Full/-/M/	MiniB/P/-	MiniB/P/M
SortMB	0	0	0.03	0.03
SortPIx	0.01	0	0.01	0.01
PIx	0.05	0	0.73	0.95
SortMIx	0	0.01	0	0.01
MIx	0	15.91	0	2.83
PostF	462.79	14.9	3.29	2.45

Table 3. Cost breakdown of 6a) BS 5K

tributing 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 2 shows the cost factors in seconds for a batchsize of 75, while table 3 shows them for a batchsize of 5000. As 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/PIx/-), 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 batchsizes by looking at the time it takes to do the index probes. As the batchsize increases, the performance also improves. For (MiniB/PIx/-), 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 batchsizes, using a message index incurs an additional cost, and does not help in speeding up the overall performance. For higher batchsizes, however, the investment into the message index pays off.

6.3.3 Scaleup for larger BS

In this experiment, we further increased the batchsize up to 100K messages (table 4), again using RQ-Z. We compared

	1K	5 K	10 K	25 K	50 K	100 K
MiniB/P/-	3,01	5,82	7,48	9,59	11,35	13,07
MiniB/P/M	2,86	4,7	7,58	10,47	12,37	15

Table 4. Speedup for bigger BS vs unbatched

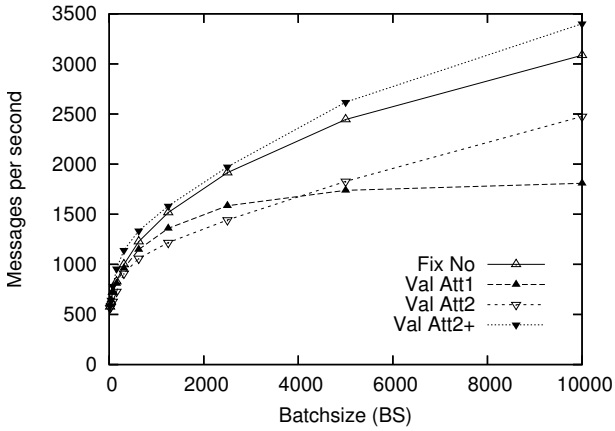


Figure 8. Competing grouping approaches for Minibatching

the methods based on minibatching (which show the best performance) to Unbatched/PIx/- an observered that the speedup increased with a larger batchsize. Furthermore, the additional message index is becoming more important with a growing batchsize. In the extreme case, MiniB/PIx/MIx is a factor of 15 faster than traditional unbatched processing and 15 percent faster than MiniB/PIx/-.

6.4 Comparing minibatching methods

Following the discussion in Section 3.2, 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 4. Figure 8 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 batchsizes. Based on these observations,

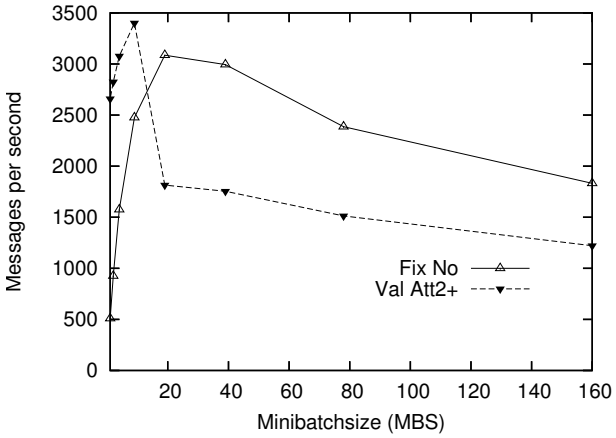


Figure 9. Tuning MBS

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.

6.5 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 9 shows the effects of varying those on an overall batchsize 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 minibatchsize. 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 minibatchsize into one directions (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.

6.6 Summary

Batching provides significant throughput enhancements, even under circumstances where messages have a limited amount of similarity. Increasing similarity and batchsize 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.

7 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.

7.1 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 ($Avg_Queuing$)

2. matching profiles for an individual message can only be determined when all messages have been processed ($Matching$)

For a steady-state system, the average time of these two factors can be estimated as follows:

$$Avg\_Queuing = \frac{\frac{Batchsize}{2}}{Arrival\_Rate}$$

$$Matching = \frac{Batchsize * matching\_single}{speedup}$$

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 batchsize. 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.

Latency (ms)	10	100	1000	10000
Speedup	1.1	1.3	2.28	9.2

Table 5. Max. speedup for given latency

The average latency is then $Avg\_Queuing + Matching$, the maximum latency $2 * Avg\_Queuing + Matching$. 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 $Matching$.

7.2 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 batchsize, and thus, the time a message is queued. Using the formulae from the previous Section, the latency can be constrained to $acceptable\_latency$ by choosing a batchsize according to the following formula:

$$batchsize < \frac{acceptable\_latency}{(\frac{1}{Arrival\_Rate} + \frac{matching\_single}{speedup})}$$

Obviously, constraining the batchsize also limits the throughput benefits that can be achieved by batching. Table 5 shows the possible speedups when a certain latency is tolerable. The setting from RQ-Z in Section 6 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.

7.3 Latency improvements for bursty traffic

So far, our discussion was 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 ressources 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 he 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 ressources, 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 6 shows, those savings can be significant. We took RQ-Z workload from the first experiment in Section 6 and computed the average and maximum latencies for temporary bursts with a rate of 3000 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 then ten-times lower.

Burst Duration		10	100	1000	10000
unbatched	avg	30.8	282.7	2802.8	28002.7
	max	59.4	563.2	5603.4	56003.3
batched	avg	29.7	196.3	985.4	2668
	max	47	301.9	1592.3	4451.3

Table 6. Latencies for bursty traffic (in ms)

8 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 [14], Siena [7], XML filters such as YFilter [13], and work on interval skip lists for predicate indexing [17]. Furthermore, techniques to optimize indexes for moving object indexes (e.g., [18]) 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 [21]; 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 [12, 15], bulk loading of (multi-dimensional) indexes in [9], and bulk join processing has been studied in [10].

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 [11].

9 Conclusion

This paper 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 study batching for a larger class of pub/sub techniques (e.g., YFilter). Furthermore, we plan to study 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. Another point of interest is the batching of update operations to the profiles; such updates arise in context-sensitive filters if boundaries of predicates (e.g., predicates on the location of a user) are updated and these updates need to be propagated to the index. Finally, we plan to incorporate our techniques into existing publish & subscribe and message broker products.

Acknowledgements: We thank Nadine Schmidt for the discussions on the concept of batching and her help when implementing and benchmarking the methods.

References

[1] `http://www.tibco.com`.
[2] `http://www.sap.de`.
[3] `http://www.microsoft.com/biztalk/`.
[4] `http://www.bea.com`.
[5] `http://www.ipd.bth.se/ska/sim_home/libghthash.html`.
[6] S. Abiteboul. Querying Semi-Structured Data. In *ICDT*, 1997.
[7] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra. Matching Events in a Content-Based Subscription System. In *Symposium on Principles of Distributed Computing*, pages 53–61, 1999.
[8] 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.
[9] J. Bercken and B. Seeger. An Evaluation of Generic Bulk Loading Techniques. In *VLDB*, 2001.
[10] J. Bercken, B. Seeger, and P. Widmayer. The bulk index join: A generic approach to processing non-equijoins. In *ICDE*, 1999.
[11] S. Chandresakaran and M. Franklin. Streaming Queries over Streaming Data. In *VLDB*, 2002.
[12] R. Choubey, L. Chen, and E. Rundensteiner. GBI: A Generalized R-Tree Bulk-Insertion Strategy. In *SSD*, 1999.
[13] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *TODS*, 2003.
[14] 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.
[15] A. Gärtner, A. Kemper, D. Kossmann, and B. Zeller. Efficient Bulk Deletes in Relational Databases. In *ICDE*, 2001.
[16] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984.
[17] E. Hanson and T. Johnson. Selection Predicate Indexing for Active Databases Using Interval Skip Lists. *Information Systems*, 21(3):269–298, 1996.
[18] A. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In *ICDE*, 1997.
[19] M. Stonebraker. The Case for Partial Indexes. *SIGMOD Record*, 18(4):4–11, 1989.
[20] `http://valgrind.kde.org/`.
[21] J. Zhou and K. A. Ross. Buffering Accesses to Memory-Resident Index Structures. In *VLDB*, 2003.