Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia

Karolina Alexiou
Systems Group, ETH Zurich
Zurich, Switzerland
kalexiou@student.ethz.ch

Donald Kossmann*
Systems Group, ETH Zurich
Zurich, Switzerland
donaldk@ethz.ch

Per-Åke Larson
Microsoft Research
Redmond, USA
palarson@microsoft.com

ABSTRACT

Bloom filters are a great technique to test whether a key is *not* in a set of keys. This paper presents a novel data structure called ARF. In a nutshell, ARFs are for range queries what Bloom filters are for point queries. That is, an ARF can determine whether a set of keys does *not* contain any keys that are part of a specific range. This paper describes the principles and methods for efficient implementation of ARFs and presents the results of comprehensive experiments that assess the precision, space, and latency of ARFs. Furthermore, this paper shows how ARFs can be applied to a commercial database system that partitions data into hot and cold regions to optimize queries that involve only hot data.

1. INTRODUCTION

Bloom filters [5] are a powerful technique with many applications. They have been successfully deployed for processing joins in distributed systems, to detect duplicates in data archives, and to speed-up lookups in the Squid cache. Bloom filters have a number of advantages. They are compact and can be implemented efficiently both in space and time. Furthermore, they degrade gracefully; even small Bloom filters are useful and the effectiveness of a Bloom filter increases with its size.

One limitation of Bloom filters is that they only work for *point queries*. In some applications, this limitation is acceptable, but in many other applications support for *range queries* is important. Consider, for instance, a Map-Reduce job that analyzes customer behavior over a specific time period. To do this efficiently, it would be useful to have *range filters* that quickly detect files that potentially have relevant events for the specified time period.

The main contribution of this paper is the description of a new trie-based data structure called ARF which is short for *Adaptive Range Filter*. In a nutshell, an ARF is a Bloom filter for range queries. That is, an ARF can be used to index any ordered domain (e.g., dates, salaries, etc.) and it can be probed to find out whether there are any potential matches for any range query over that domain. Just like Bloom filters, ARFs are fast, compact (i.e., space efficient), and degrade gracefully.

*Work done while visiting Microsoft Research.

Proceedings of the VLDB Endowment, Vol. 6, No. 14

One particular feature of ARFs is that they are adaptive. That is, ARFs dynamically *learn* the query and data distribution and adjust their *shape* accordingly. It is possible to start with an empty ARF at the beginning. As more and more queries are processed, the ARFs self-tune in a fine-grained way based on the query feedback (false positives). Specifically, ARFs self-tune by indexing important regions of the domain in a fine-grained way (e.g., hot and choppy regions) and less important regions in a coarse-grained way. If the workload or data distribution changes, ARFs adjust, too.

This paper summarizes the results of experiments carried out with ARFs thereby varying the data distribution, the query distribution, and updates among others. The results show that ARFs are indeed robust and adaptive. Furthermore, the experiments show that ARFs are fast, compact, and degrade gracefully. To the best of our knowledge, no other comparable data structures for filtering range queries have been proposed in the literature. To get a feeling for the effectiveness of ARFs, we compare ARFs with Bloom filters even though ARFs and Bloom filters were designed for different purposes. It turns out that ARFs can outperform Bloom filters even for point queries if there is skew in the workload or data.

The remainder of this paper is organized as follows: Section 2 gives an overview of *Project Siberia* which is the project that motivated this work. Section 3 describes the ARF data structure. Section 4 explains how ARFs learn and adapt to the workload. Section 5 presents the results of experiments. Section 6 gives a qualitative analysis of ARFs. Section 7 discusses related work. Section 8 contains conclusions and possible avenues for future work.

2. APPLICATION EXAMPLE

2.1 Project Siberia

To illustrate the usefulness of ARFs, this section shows how we envision to use them as part of Project Siberia at Microsoft Research. Project Siberia is investigating techniques for managing cold data in Hekaton. Hekaton is a high-performance, main-memory database engine that is part of the Microsoft SQL Server product suite. Hekaton was specifically designed to manage and process *hot data* that is frequently accessed and has particularly low latency requirements. Obviously, not all data is *hot* and it is not economical to keep all data in main memory. The goal of Project Siberia is to have the system automatically and transparently migrate cold data to cheaper external storage, retain the hot data in main memory, and to provide the right mechanisms to process any kind of query. The expectation is that the great majority of queries involve hot data only. However, the few queries that involve cold data must be processed correctly, too.

This paper is not about Project Siberia or Hekaton. Both systems have been described in other papers; e.g., [17] describes how

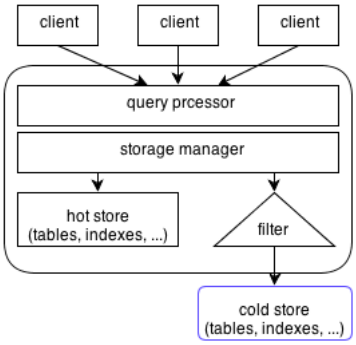


Figure 1: Query Processing in Project Siberia

Project Siberia determines which data is hot and cold; [10] gives an overview of Hekaton. Furthermore, there are many more use cases for ARFs such as quickly eliminating files and directories containing no relevant data when processing a Map Reduce job. We illustrate the use of ARFs in Project Siberia because it is a particularly intuitive use case.

Figure 1 shows how queries and updates are processed in Project Siberia. There is a hot store (i.e., Hekaton) for hot data and a cold store (e.g., standard SQL Server tables) for cold data. Clients are unaware of where data is stored and issue SQL queries and update statements just like in any other relational database system. Logically, every query must be computed over the *union* of the data stored in the hot and cold stores. For instance, a query that asks for the number of orders of a specific customer to be shipped within the next week, needs to retrieve all relevant orders from both stores and then compute the aggregate. Note that migrating data to the hot store is more effective than caching data in main memory because the hot store fully indexes all its data.

The focus of this paper is on the design of the *filter* component shown in Figure 1. This filter tries to avoid unnecessary accesses to the cold store. Access to the cold store is assumed to be expensive even if it is only to find out that the cold store contains no relevant information. Given a query, the filter returns *true* if the cold store possibly contains records that match the query. It returns *false* if the cold store is guaranteed to contain no relevant information for the query: In this case (the most frequent case), the query can be processed without accessing the cold store.

Again, Project Siberia includes several approaches to decide which data to place into the *hot store* and which data to place into the *cold store*. An important requirement is that the partitioning of the database is transparent to the application developers. That is, we cannot assume that the developer specifies whether the query involves only *hot* data. That is why we need *filters*. Furthermore, the partitioning can change at any point in time; there might be a continuous migration of records between the hot and cold stores. This observation makes it necessary to devise adaptive techniques.

2.2 Requirements

We now summarize the requirements that guided our design of ARFs. While these requirements are motivated by the use of ARFs in Project Siberia, we believe that they apply more generally to other applications of ARFs.

- *Correctness:* There must be no *false negatives*. That is, if the filter returns *false* for a query or update, then it is guaranteed that the cold store contains no relevant records.
- *Precision:* The number of *false positives* should be minimized. A false positive is a query or update for which the filter returns *true* even though the cold store contains no relevant records. False positives do not jeopardize the correctness of the system, but they hurt performance.
- *Space efficient:* The filter must live in main memory to guarantee efficient access to the filter. As main-memory space is expensive, minimizing the space occupied by the filter is critical to be cost effective.
- *Graceful degradation:* A direct consequence of the space-efficiency requirement is that the precision of a filter should grow and degrade with its space budget. Even a tiny filter should be useful and filter out the most common queries.
- *Fast:* Filtering must be much faster than access to the cold store. The filtering cost must be in the same order as processing a query in the hot store: Most queries are expected to be hot-only queries and almost all queries (except for primary key lookups in the hot store) involve a filtering step. That is why the filter lives in main memory.
- *Robustness:* Both the data and the query distribution are typically heavily skewed. The filter should be designed to work well in such situations. Furthermore, the filter must adapt whenever the workload changes and/or data is migrated back and forth from the cold to the hot store.
- *Generality:* The filter must not make any assumptions about the partitioning scheme used to classify records as hot or cold. It must support partitioning at the record level, the finest possible granularity. Furthermore, the filter should support both point and range queries.

3. ADAPTIVE RANGE FILTERS

This section describes the main ideas of the ARF technique. It shows how ARFs filter range queries, how they are implemented in a space-efficient way, and how an ARF can be integrated into the B-tree of a hot store. The next section then shows how ARFs adapt to skew in the data and query distribution.

3.1 Overview

Figure 2 gives an example of an ARF. It shows the keys of records stored in the cold store (e.g., *shipping dates of orders*) and one possible ARF built for this bag of keys. In most workloads, several ARFs exist for a table: one for each attribute that is frequently involved in predicates of queries and updates. This paper focuses on one-dimensional ARFs; [2] studies multi-dimensional ARFs using space-filling curves and bounding rectangles in intermedediate nodes of the index.

An ARF is a binary tree whose leaves represent ranges and indicate whether the cold store contains any records whose keys are contained in the range. To this end, each leaf node keeps an *occupied bit*. The ARF of Figure 2, for instance, indicates that the cold store might have records with keys in the ranges [0,7] and [11,11]: the *occupied bits* of these two leaves are set to *true*. Furthermore, the ARF of Figure 2 indicates that the cold store has no records with keys in the ranges [8,9], [10,10], and [12,15].

The intermediate nodes of an ARF help to navigate the ARF in order to find the right leaves for a given query. Each intermediate node represents a range and has two children: the left child represents the left half of the range of its parent; the right child represents the right half of the range. The root represents the whole domain of the indexed attribute (e.g., [0,15] in the toy example of Figure 2).

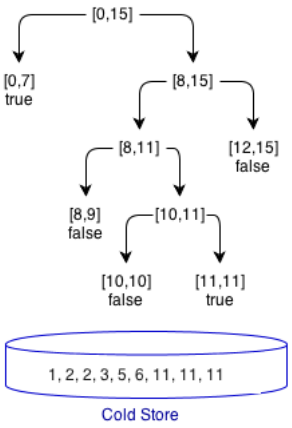


Figure 2: Example ARF

Using an ARF, a range query, $[l,r]$, is processed as follows: Starting at the root, the system navigates to the leaf node that contains l, the left boundary of the range query. If the *occupied bit* of that leaf is set to *true*, the filter returns *true*, thereby indicating that the cold store needs to be accessed for this query. If the *occupied bit* of that leaf is set to *false*, then the right sibling of that leaf is inspected until the leaf covering r has been inspected. For instance, the range Query [8,12] would navigate first to the Leaf [8,9], then visit the Leaf [10,10] and then [11,11]. At this point, it would return *true* indicating that the cold store needs to be visited.

Revisiting the requirements for filters listed in Section 2.2, the ARF structure is correct if the *occupied bit* of a leaf is set to *false* only if the cold store indeed contains no records whose keys match the range of that leaf. Therefore, these *occupied bits* need to be maintained whenever new records are migrated to the cold store or records are updated in the cold store (Section 4.4). False positives may arise in a number of situations. Most importantly, an ARF cannot precisely represent all the keys of the cold store if there is a limited space budget. For instance, the ARF of Figure 2 does not capture the fact that the cold store contains no record for 4 because doing so would require to grow the ARF much deeper. As a result, the ARF of Figure 2 results in a false positive for the query [4,4]. How to make best use of the space budget and adapt to a given data and query distribution is the subject of Section 4.

3.2 Encoding

One of the main advantages of the ARF structure is that it can be implemented in a space-efficient way. Because the children of every node always partition a region in half, the nodes need not store the delimiters of their ranges: these ranges are implicit. More precisely, an ARF is a trie and every level of the trie represents information for the next significant "bit" of the domain. So, all that is needed to represent an ARF is to encode the whole domain, the *shape* of the tree, and the *occupied bits* of the leaves.

To be concrete, every intermediate (non-leaf) node can be represented using two bits. These two bits encode whether the node has 0, 1, or two children. The following four situations can arise:

- *00*: Both of the children of the intermediate node are leaves; e.g., Node [10, 11] in Figure 2.
- *01*: The left child is a leaf; the right child is not a leaf; e.g., Nodes [0, 15] and [8, 11] in Figure 2.

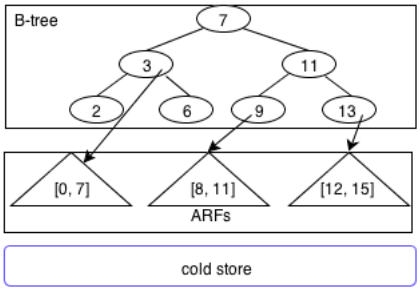


Figure 3: Integrating ARFs into a B-tree

- *10*: The right child is a leaf; the left child is not a leaf; e.g., Node [8, 15] in Figure 2.
- *11*: None of the children are leaves.

The shape of a whole ARF is encoded by serializing the intermediate nodes in a breadth-first traversal. For example, the shape of the ARF of Figure 2 is represented by the following bit sequence: 01 10 01 00. No pointers are needed! These 8 bits are all that is needed to determine the ranges of all four intermediate nodes.

In addition to the bit sequence that encodes the intermediate (non-leaf) nodes, an ARF maintains the *occupied bits* of the leaves. Continuing the example of Figure 2, the bit sequence that represents the five leaves is: 10010.

Putting shape and occupied bits together, the ARF of Figure 2 is represented using 13 bits. In general, an ARF with n nodes is encoded using $1.5 * n$ bits: $2 * n/2$ bits for the intermediate nodes plus $n/2 + 1$ bits for the leaves.

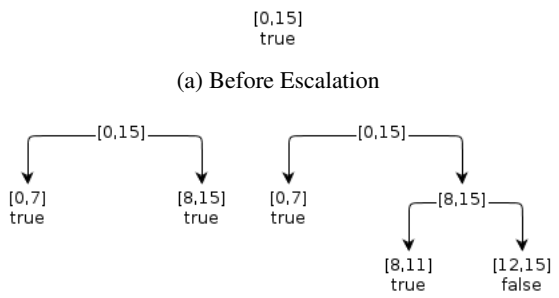
A breadth-first encoding (as opposed to depth-first) of the ARF is used because it supports a more efficient navigation through the ARF. The exact algorithms and optimizations that make probing and maintaining ARFs fast on modern hardware are given in the appendix of the electronic version of this paper. Since ARFs are not balanced, the running time complexity to probe an ARF is linear with the size of the ARF in the worst case. As shown in Section 5.5, ARFs are nevertheless fast up to a certain size because of their compact encoding and an efficient low-level implementation on modern hardware. To implement fast search in a very large ARF, we propose to partition the ARF and embed each partition into a balanced data structure. Project Siberia, for instance, embeds ARFs into B-trees as described in the next sub-section.

3.3 ARF Forests

We recommend using many small ARFs that each cover a specific sub-range instead of one big ARF that covers the whole domain. One reason is lookup performance. Another reason is space efficiency. A third reason is that ARFs can nicely be embedded into existing index structures such as B-trees.

Figure 3 shows how ARFs can be embedded into a B-tree in the hot store. The upper part of Figure 3 shows a normal, traditional B-tree which indexes the data in the hot store. For simplicity, this example B-tree is shown as having only one entry in each node. To find data in the hot store, our system navigates this B-tree just like any other B-tree. There is nothing special here.

What makes the B-tree of Figure 3 special is that any node (leaves and intermediate node) can have a reference to an ARF. For instance, the leaf node of the B-tree that contains the Key 9 has a reference to the ARF that corresponds to the Range [8,11]. Likewise, the intermediate node 3 of the B-tree points to the ARF that



(a) Before Escalation

(b) Split of Node [0,15]

(c) Complete Escalation

Figure 4: Escalating an ARF

corresponds to the Range [0,7]. Processing a query now involves navigating the B-tree to find all relevant hot records and following the pointers to ARFs in order to find out whether the system needs to access the cold store.

Comparing Figures 2 and 3, one way to interpret the forest of ARFs is to think of a big ARF with its top chopped off. For instance, the forest of ARFs depicted in Figure 3 emerged from the ARF of Figure 2 by removing nodes [0,15] and [8,15] and connecting the resulting three subtrees to the appropriate places in the B-tree. This approach saves space because the domain and the top-level nodes of the ARF are not stored. Furthermore, this approach saves time to navigate the first levels of the ARF: While traversing the B-tree, we implicitly also traverse the top levels of the ARF.

4. LEARNING AND ADAPTATION

One of the nice features of the ARF structure is that it can adapt to the data and query distribution. For instance, large regions that contain no data such as [12,15] in Figure 2 can be represented in a compact way. Likewise, large regions that are densely packed such as [0,7] in Figure 2 can be represented in a compact way. This way, an ARF can invest most of its bits into regions that are queried frequently and are modestly populated such as [8,11] in Figure 2. This section shows how to construct and evolve an ARF, thereby adapting to the data and query distribution and to data movement from and to the cold store. First, we describe the basic primitives of splitting and merging nodes of an ARF. Then, we present specific adaptation and learning techniques. Throughout this section, we will show examples for a "single big ARF" approach. All the techniques apply naturally to a forest of ARFs (Section 3.3).

4.1 Escalation (Split)

Technically, an ARF grows and shrinks just like any other tree: It grows by splitting leaves and it shrinks by merging leaves. Figure 4 shows how the ARF of Figure 2 could have evolved. At the beginning, an ARF just contains a single node that represents the whole range and indicates that the cold store needs to be accessed for every query or update operation. Figure 4a shows this minimal ARF for the example of Figure 2. Figure 4b shows the next incarnation created by splitting this single node. Again, a split always partitions the range of its parents in half. Figure 4c shows the resulting ARF after splitting Node [8,15].

Unlike most other trees, splitting (and merging) of nodes is not initiated by updates to the database. B-tree nodes, for instance, are split as a result of inserting a new key. In contrast, an ARF grows (and shrinks) as a result of *learning* and adapting to the data and query distribution. Algorithm 1 gives the pseudo-code of the escalation process. When a query, q, is processed, it is checked whether the query is a false positive. If it was, then the ARF escalates so that processing the same query again does not result in a false positive again. As shown in Algorithm 1, this escalation is done iteratively by splitting the leaf that contains the left and right bounds of the query (denoted by q.left and q.right in Algorithm 1) until the left bound of the query is the left boundary of a leaf and the right bound of the query is the right boundary of a leaf. This splitting is only done if the query bounds are contained in leaves whose *occupied bit* is set to true. As a final step, the `MarkEmpty` method is called in order to make sure that the *occupied bits* of all the leaves that are contained by the query are set to *false*.

Algorithm 1 ARF Escalation

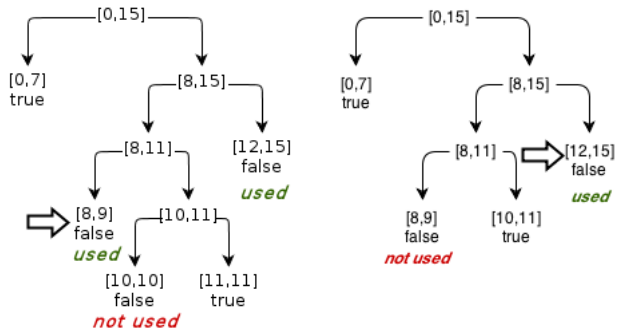
```
procedure PROCESS(Query q)
    ...
    if (q is false positive) then
        node ← NAVIGATE(q.left)
        while (node.left <> q.left) & node.occupied do
            SPLIT(node)
            node ← NAVIGATE(q.left)
        end while
        node ← NAVIGATE(q.right)
        while (node.right <> q.right) & node.occupied do
            SPLIT(node)
            node ← NAVIGATE(q.right)
        end while
        MARKEMPTY(q.left, q.right)
    end if
    ...
end procedure
```

The post condition of executing the code snippet of Algorithm 1 is that the ARF accurately captures that the cold store contains no value in the range, q. Furthermore, Algorithm 1 does not change the status of any other values so that if the ARF was correct and did not allow any *false negatives* before executing the code snippet of Algorithm 1, then the ARF is also correct after the execution of Algorithm 1. As a result of the splits, the size of the ARF grew. If space is limited, then the ARF *de-escalates*, thereby gaining space and losing precision. The next sub-section describes this process.

4.2 De-escalation (Merge)

The ultimate goal is to minimize the number of false positives with a given (small) space budget. As a result of the escalation that improves precision as described in the previous sub-section, it is possible that the size of the ARF grows beyond its space limits. In such a situation, we need to shrink the ARF so that it fits within the space budget. Shrinking is done by merging nodes in the ARF. We call this process *de-escalation*.

In the same way as escalation increases the precision of an ARF, de-escalation reduces the precision of an ARF. So, it is critical to decide which nodes of the ARF to merge; the goal is to merge nodes that are not likely to overlap future queries in order to make sure that this de-escalation does not result in false positives in the immediate future. To decide which nodes of an ARF to merge, we propose to make use of a *replacement policy*. In principle, any replacement policy known from buffer management can be used (e.g., LFU, LRU, or LRU-k). Section 6 gives a better intuition of the analogy of replacement of nodes in an ARF and replacment of objects in a cache. We propose to use a *clock replacement policy*



(a) Before De-escalation (b) After De-escalation

Figure 5: ARF Replacement

for ARFs because its statistics are small (i.e., a single bit) and space efficiency is particularly important for ARFs.

Figure 5 demonstrates how de-escalation with a clock policy works using the running example. Figure 5a shows the example ARF from Figure 2; this time with a *used bit* for each leaf node. The *used bit* of a leaf is set whenever a query hits that leaf. For instance, the processing of Query [13,14] sets the *used bit* of Leaf [12,15], thereby indicating that this leaf is useful and not a good candidate for replacement. Furthermore, Figure 5a shows the *clock pointer* which points to Leaf [8,9] in this example. If the ARF has grown beyond its space budget, then the replacement policy looks for a victim. In this example, the clock strategy first checks Leaf [8,9]: Since its *used bit* is set, the clock strategy advances the clock pointer to the next leaf and unsets the *used bit* of [8,9]. In this example, the clock replacement policy selects [10,10] as a victim because its *used bit* is not set.

Figure 5b shows the ARF after *replacing* [10,10]. Technically, such a de-escalation works by merging the victim with its sibling—[11,11] in this example. Actually, it is only possible to de-escalate if the sibling is also a leaf. If not, the clock strategy continues to look for victims until two leaves are found.

Merges cascade if the *occupied bits* of two leaves have the same value. In general, two sibling leaves that have their *occupied bits* set to the same value do not carry any information, thus they can be merged. In the example of Figure 5b, Nodes [8,9] and [10,11] have different values for their *occupied bits* so that the merge cascade stops at this point. If the *occupied bit* of Node [8,9] had been set to *true*, then these two nodes would have been merged, too. Algorithm 2 shows pseudocode for the whole de-escalation process. This pseudocode is executed immediately after possible escalations. The `FindVictim` function implicitly advances the clock and unsets used bits. The `Sibling` function determines the sibling of a leaf node, and the `Merge` function implicitly cascades if the *occupied bits* of two siblings have the same value.

An interesting observation is that *used bits* (or more generally, usage statistics for a replacement policy) are only needed for leaf nodes whose *occupied bits* are set to *false*. Only these leaves carry useful information and only these leaves need to be protected from replacement. Putting it differently, it never hurts to merge a leaf with *occupied bit* set to *true* with its sibling if that sibling is not useful for any queries. By applying the replacement policy only to leaves whose *occupied bit* is set to *false*, half of the space for keeping usage statistics is saved. Again, space economy is also the reason why we suggest using a clock strategy as a replacement

Algorithm 2 ARF De-escalation

```
procedure PROCESS(Query q)
    ...                                  ▷ Escalation as in Algorithm 1
    while (size > budget) do
        node1 ← FINDVICTIM
        node2 ← SIBLING(node1)
        MERGE(node1, node2)
    end while
end procedure
```

policy. Overall, on average, only half a bit per leaf node is required.

4.3 Speeding-up Learning

Section 4.1 showed how an ARF learns from *false positive* queries: It escalates, thereby making sure that it does not make the same mistake twice. In fact, the ARF can also learn from *true positive* queries. True positives are queries for which the ARF indicated that the cold store needs to be accessed (the *occupied bit* of one of the relevant leaves was set to *true*) and for which indeed the cold store returned a non-empty result. The results of these queries are useful to train the ARF because the gaps between two results are empty.

To give an example, let the result of Query [5,15] be $\{7, 9\}$. If the query involves no other predicates on other dimensions, we can now infer that the cold store has no records that match keys in the ranges [5,6], [8,8] and [10,15]. Correspondingly, we could set the *occupied bit* of a leaf node, say, [12,15] to *false* if it is still set to *true* from its initialization because no other information was available at that point in time. We use such information from *true positive* queries to set the *occupied bits* of existing bits to the right value; however, we do not suggest to escalate the ARF and only trigger escalation for *false positives*. The reason is that escalation always comes at a cost (de-escalation of the ARF at another region) and a *true positive* query does not justify this cost.

Learning from such *true positives* speeds up learning dramatically and comes at virtually no cost. Another way to speed up learning at little incremental cost is to ask slightly bigger queries to the cold store whenever the cold store is visited. For instance, if the query is [13,14] and the relevant leaf node of the ARF is [12,15] and its *occupied bit* is set to *true* from its initialization, then we could forward the Query [12,15] to the cold store and filter the results returned by the cold store. The reason is that if [13,14] is a false positive and, actually, the whole range [12,15] is empty, then we do not need to escalate the ARF to process this *false positive*. And, even if [12,15] is not entirely empty, then we possibly would not have to escalate the sub-tree rooted in [12,15] as deeply as we would have to if we only knew that [13,14] was empty.

We implemented both these learning techniques for the prototype used in the experiments reported in Section 5.

4.4 Updates

An important requirement of Project Siberia is that records can freely move at any point in time between the hot and cold stores. If a record is moved from the hot store into the cold store, then all ARFs constructed for the table of that record need to be updated in order to avoid *false negatives*. In Figure 2, for instance, the *occupied bit* for Node [12,15] must be set to *true*, if, say, a record with Key 14 is inserted into the cold store. Actually, there are several alternative approaches to avoid false negatives in this situation; e.g., escalating Node [12,15] or conditionally escalating Node [12,15] if its *used bit* is set. We did not experiment with these variants and implemented the simple variant that sets the *occupied bit* to true because that variant worked well in all our experiments.

If a record is removed from the cold store, the ARF is not changed. In Figure 2, for instance, we cannot set the *occupied bit* of Node [11,11] to *false*, even if we knew that one or several records with Key 11 have been removed from the cold store. Fortunately, unlike insertions, deletions in the cold store cannot result in *incorrect* ARFs: the worst that can happen is that they result in inefficiencies caused by *false positives*. As shown in Section 5.6, however, the escalation mechanism of Section 4.1 that adapts an ARF after a *false positive* is quite effective so that an ARF adjusts quickly to changes in the data distribution caused by excessive migration of records between the hot and cold stores.

4.5 Training ARFs

In addition to its ability to adapt to changes in the data distribution in a fine-grained way, one main feature of ARFs is that they learn on the fly: It is always safe to start with a trivial ARF with only one node such as the ARF of Figure 4a and build it up in a *pay-as-you-go* manner as part of processing queries and updates. If a dedicated training phase is affordable, such a training phase can be used to initialize ARFs so that the ARFs perform well from the very beginning. We propose to train ARFs in the following way:

- *Explore the Data:* Scan the table in the cold store. From the results, construct a *perfect* ARF which accurately models all gaps in the cold store. That is, we escalate the ARF as proposed in Algorithm 1, but we would *not* de-escalate the ARF as shown in Algorithm 2. This way, the ARF is likely to exceed the space budget, but we tolerate that in the training phase.

- *Explore the Workload:* Run a series of example queries and updates which are representative for the query and update workload. The purpose of this step is to make sure that the ARF learns the query/update distribution. As part of this step, the *used bits* of leaves are set. In fact, we propose to use *usage counters* (instead of *used bits*) to distinguish between leaves that are used more frequently than others.

- *Meet Space Constraints:* Trim the ARF by running the code snippet of Algorithm 2 so that the space budget is met. If *usage counters* were used in Step 2, discard those and keep *used bits* only for the regular operation of the ARF.

This training algorithm can be applied simultaneously to all ARFs of a table.

4.6 Summary and Variants

This section presented a series of techniques to *train* and *adapt* an ARF. The ultimate goal is to converge to an ARF that is small and whose shape reflects both the data and query distribution and has, thus, high precision with few false positives. The rule of thumb is that an ARF should be coarse-grained (i.e., have leaves that cover large ranges) for empty and densely-populated regions or regions that are infrequently queried; in contrast, an ARF should invest its bits and be fine-grained for regions that are frequently queried and have many gaps.

Given the basic techniques presented in this section, there is a large design space of possible variants. We studied many of these variants as part of our experiments and present the most interesting results in Section 5. Here, we want to briefly sketch and summarize the most important dimensions of this design space:

To adapt or not to adapt? Adaptation can be harmful. For instance, an ARF might escalate as part of processing a query that will never be asked again. After this escalation, the ARF might have to de-escalate to fit the space budget, thereby removing leaves which are needed for the next query. In particular, adaptation is often not needed if the ARF was trained using the algorithm of Section 4.5.

When to adapt? If the answer to the first question is *yes*, then it is worth asking whether escalation should be carried out for every false positive or whether we should selectively escalate as a result of a false positive query. For instance, we could maintain *used bits* for leaves with their *occupied bit* set to *true*, too, and only escalate those leaves if their *used bit* is set. That is, rather than avoiding making the same mistake twice, we would avoid making the same mistake three times.

What is the best replacement policy? We chose the clock strategy to find victims in Section 4.2 because it is space-efficient and easy to implement. In Section 5, we will show that a *random* replacement policy that maintains no statistics does even better if the workload is not skewed. Section 6 reiterates this question.

When to adapt from a systems perspective? A more systems-related question involves decoupling the decision *when to adapt* from the actual process of implementing the escalation and de-escalation processes. For instance, we could find out during query processing that we need to escalate the ARF, but then carry out the actual escalation asynchronously, thereby retrieving more precise information from the cold store. An extreme variant would be to keep statistics about false positives for the whole ARF and to completely reconstruct the ARF using the algorithm of Section 4.5 whenever the statistics indicate that it is worth doing so.

5. EXPERIMENTS AND RESULTS

This section presents the results of extensive experiments that we conducted with a prototype implementation of ARFs, a synthetic benchmark and a real workload from an Xbox Live server. The experiments study the precision, space and time efficiency, robustness, and graceful degradation of several ARF variants for various different workloads and database configurations. Specifically, this section presents the results of the following three ARF variants:

- *ARF no-adapt:* The ARF is trained once (as described in Section 4.5) and then never changed.

- *ARF adapt-1bit:* The ARF is trained at the beginning (as described in Section 4.5) and then adapted with every false positive as described in Section 4.1. After every escalation, the ARF is de-escalated to fit its space budget as described in Section 4.2, thereby using a clock replacement policy.

- *ARF adapt-0bit:* Same as "ARF adapt-1bit" with the only difference that we use a clock strategy without *used bits*. This way, the replacements are somewhat random, but the additional space can be used to construct ARFs that more accurately model the data.

If not stated otherwise, this section presents the results obtained using a single "big ARF" rather than a forest of ARFs. As stated in Section 3.3, forests of ARFs have a number of advantages. Indeed, forests of ARFs showed higher precision in all experiments for being more space efficient, but the effects were small so we only show the precision results for a single, big ARF. In terms of latency, forests of ARFs showed significant improvements so we included these results in Section 5.5.

We are not aware of any comparable technique for filtering range queries. To get a feeling for the effectiveness of ARFs, we use Bloom filters as a baseline in all experiments, even though Bloom filters were designed to filter point queries only. We apply Bloom

Database Parameters	
Distribution	Uniform or Zipf
Distinct Keys	Vary
Domain Size	2^{24}
Query Workload	
Location (l)	Uniform, Zipf1, or Zipf2
μ, Mean Range Size ($r - l$)	Point ($\mu = 0$) or Range ($\mu = 1 \ldots 300$)
Updates	
Deletes	Uniform
Inserts	Zipf or Uniform
Number of Updates	Vary

Table 1: Benchmark Parameters

filters to range queries by probing each value of the range and applying short-circuiting. We also include experiments with *range-based Bloom filters* for which every entry of a Bloom filter covers a range of values. In the literature, a number of Bloom filter variants have been proposed (Section 7). We implemented the classic variant [5] because it is typically used in practice. A comparison of alternative Bloom filter variants for point queries is not the purpose of this paper and can be found in the literature (Section 7).

5.1 Benchmark Environment

To study ARFs, we ran a series of synthetic workloads using the three ARF variants and Bloom filters. Table 1 lists the parameters of our benchmark database and workload. We studied two different kinds of benchmark databases: *Zipf* and *Uniform*. Using a *Zipf* distribution, the keys of the records stored in the cold store were highly skewed. Using a *Uniform*, the keys were evenly distributed in the domain. In practice, we expect a *Zipf* distribution; for instance, the cold store may contain all orders with a past shipping date and only a few orders with a shipping date in the future. We also studied an Xbox gaming workload; that workload is also skewed.

We varied the number of distinct keys of records in the cold store from 1000 to 1 Million. If not stated otherwise, this value was set to 100,000 distinct keys.

We studied a number of different synthetic query workloads with point and range queries. All queries were executed sequentially and generated using a random query generator. Each query has the form $[l, r]$. The left boundary, l, was generated using either a *Zipf* distribution or a *Uniform* distribution. We studied two different kinds of Zipf distributions. In the first variant, called *Zipf1*, some regions of the domain were queried more frequently than other regions and these *hot* regions were sprinkled randomly (using a Uniform distribution) across the whole domain. In the second variant, called *Zipf2*, the *hot* regions were clustered. With a Zipf database and a Zipf2 query workload, the queries were clustered so that the query location was different from the location of the data. We expect this situation to be a typical scenario for Project Siberia. Using the "order" example again, most of the orders in the cold store are old orders and most of the queries ask for present or future orders.

The second parameter that determined the query workload is the mean range size of range queries; i.e., $r - l$. This parameter is referred to as μ in Table 1. We studied two kinds of queries: *point queries* and *range queries*. For point queries, $l = r$ (i.e., $\mu = 0$). For range queries, $r - l$ was generated using a normal distribution with mean μ. We varied the mean of this normal distribution from $\mu = 1$ to $\mu = 300$. The standard deviation was set to $\sigma = \mu/3$.

We also studied how quickly ARFs adapt to migrations between the hot and cold store. In these experiments, we ran a workload mix of queries, inserts (inserting new records into the cold store) and deletes (moving records from the cold store to the hot store). All deletes removed all records with a certain key (e.g., all orders with a certain shipping date) from the cold store; the key was selected using a Uniform distribution. All inserts inserted records with a key that did not yet exist in the cold store: The value of the new key was selected using either a Uniform or a Zipf distribution. We report on these results in Section 5.6. Furthermore, we studied how ARFs adapt to changes in the workload (Section 5.7).

All experiments were carried out in the same way. First, the ARFs and Bloom filters were trained; the ARFs were trained using the algorithm of Section 4.5. After that, we ran 300,000 randomly generated queries and measured the *false-positive rate*, lookup latency, and the time it took for escalate and de-escalate operations. The false-positive rate was defined as the number of false positives divided by the total number of hot-only queries.

5.2 Software and Hardware Used

We ran all experiments on a machine with four dual-core Intel processors (2.67 GHz) and 4 GB of main memory. The machine ran Ubuntu Version 11.10. We implemented ARFs and Bloom filters using C++. We measured running times using the *gettimeofday()* function and CPU cycles using the RDTSCP instruction of the processor.

We implemented ARFs as described in Sections 3 and 4. We implemented Bloom filters using multiplicative hash functions, as proposed in [16]. Furthermore, we optimized the number of hash functions of the Bloom filters according to the size of the Bloom filter. For instance, we used five hash functions if there were 8 bits per key available. This way, we were able to reproduce all common results on Bloom filters.

5.3 Experiment 1: False Positives

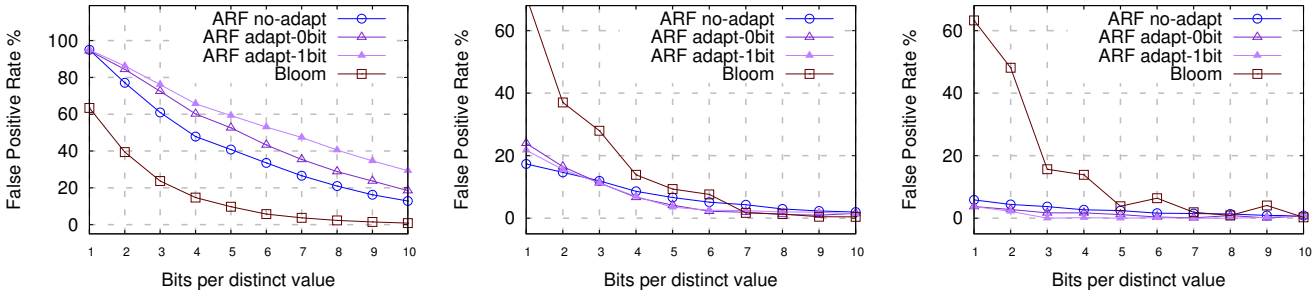
5.3.1 Point Queries: Synthetic Workload

Figures 6a to 6f study the *false positive rate* of the three alternative ARF variants and Bloom filters for point queries and for the different query and database distributions. In these experiments, we varied the size of the filters from 100 KBits (i.e., 1 bit per distinct key) to 1 MBits (i.e., 10 bits per distinct key). As a rule of thumb, Bloom filters are at their best for 8 bits per key and Figures 6a to 6f confirm this result.

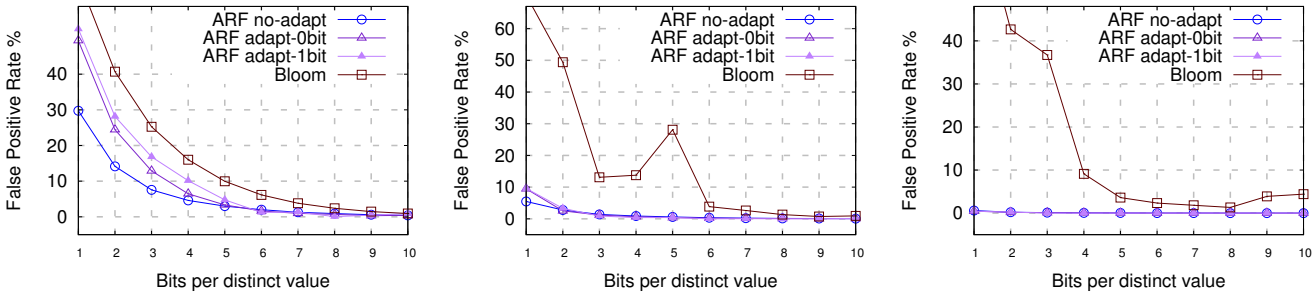
The results presented in Figures 6a to 6f give insight into three important requirements of filters (Section 2): precision, graceful degradation, and exploitation of skew in the data and query workload. Precision is directly reflected in the *false-positive rate* metric; the lower the false-positive rate, the higher the precision. Graceful degradation is reflected in the shape of the graphs; the flatter the curve, the more resilient the filter is towards space constraints. Exploitation of skew can be observed by comparing the results of the different figures: It is good if the false-positive rate improves for a Zipf data or query distribution.

Turning to Figure 6a, it can be seen that Bloom filters are the clear winner for point queries and no skew in the data and query distribution (i.e., Uniform data and query distribution). This result is no surprise because that is exactly the scenario for which Bloom filters were designed and have proven successful. The precision of the ARF variants improves with a growing size, but even for a fairly large size of 10 bits per distinct key, none of the ARF variants are competitive with Bloom filters which are almost perfect at this point.

Comparing the three ARF variants in Figure 6a, it can be seen that the "ARF no-adapt" variant is the winner, the "ARF adapt-0bit" variant comes in second place, and the "ARF adapt-1bit" is



(a) Uniform DB / Uniform Workload (b) Uniform DB / Zipf1 Workload (c) Uniform DB / Zipf2 Workload



(d) Zipf DB / Uniform Workload (e) Zipf DB / Zipf1 Workload (f) Zipf DB / Zipf2 Workload

Figure 6: Exp. 1 (Point Queries), Precision: Vary DB, Vary Workload, No Updates, Vary Size, 100K Distinct Keys

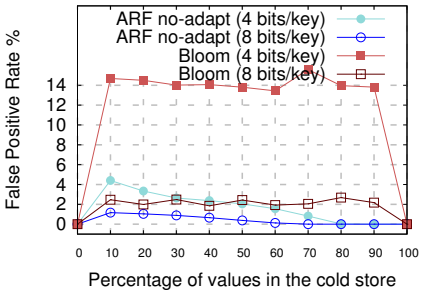


Figure 7: Xbox Workload

the worst variant in this experiment. However, the differences are small. The "ARF adapt-1bit" is worst because investing space for *used bits* is not worthwhile if the query workload is Uniform: There is no skew and the space is better invested to create a *deeper* ARF. The "ARF no-adapt" variant beats the "ARF adapt-0bit" variant because adaptation is actually harmful here: Escalating the ARF to handle one false positive in a small region might result in the de-escalation of a large region. But, again, the effects are small. The conclusion is that ARFs are not good for workloads with only point queries and no skew. For such workloads, Bloom filters continue to be the best choice.

Turning to Figures 6b to 6f, it can be seen that all three ARF variants benefit significantly from skew in the data or query distribution. In the presence of skew, all three ARF variants outperform Bloom filters. These results are encouraging because a data structure that does not do well for point queries is unlikely to do well for range queries. Again, the differences between the three ARF variants are small.

5.3.2 Xbox Workload

Figure 7 shows the results of experiments with an Xbox gaming workload. This workload is composed of 27 billion requests from approximately 250,000 Xbox Live users who access different objects of games. This workload is also skewed as some players play more often than other players and, thus, access their objects more frequently. Figure 7 varies along the x-axis the percentage of objects that are kept in the hot store. Furthermore, it shows the results for Bloom filters and ARFs with 4 bits per key and 8 bits per key. In the extreme, all approaches behave in the same way and have no false positives. If all objects are stored in the hot store ($x = 0\%$), there are no false positives because the cold store is empty. Likewise, if the hot store is empty ($x = 100\%$), there are no false positives because all queries are true positives. In the middle, however, ARFs clearly outperform Bloom filters because the ARFs take advantage of the skew in this workload. This experiment resembles the "Uniform DB / Zipf1 Workload" experiment of Figure 6b, albeit with a different kind of skew which was imposed by the behavior of gamers.

5.3.3 Range Queries

Figure 8a shows the results of the three ARF variants and Bloom filters for range queries with a mean range size of $\mu = 30$ ($\sigma = 10$) with a Uniform database and workload distribution. In the shipping date / order example, a query with a key range of 30 corresponds to asking for the orders of a specific month. In our adoption of Bloom filters for range queries, we probed each day of the range until we found a match or until all keys of the range had been probed.

Comparing Figures 6a and 8a, it can be seen that all three ARF variants performed almost identically for point and range queries. Figure 8b shows the results for range queries with a Uniform database and a Zipf1 query workload. Comparing Figures 6b and Figures 8b, we can again observe that ARFs show almost the same performance for range as for point queries. The same is true for all other combinations of database and workload distributions (not

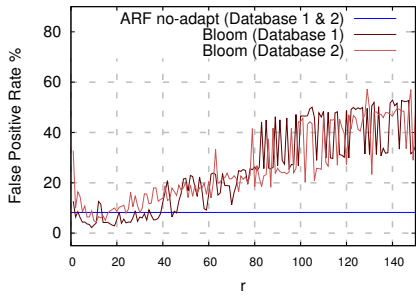


Figure 9: Range-based Bloom Filters: Vary Block Size
Uniform Database, Zipf1 Workload, 100K Keys, 8 Bits/Key

shown for brevity). Figure 8c shows the results with a varying mean query range size, μ, and a Uniform database and Uniform workload distribution. Again, the curves for all three ARF variants are flat which demonstrates the robustness of ARFs: ARFs are good for any kind of range query, independent of the size of the range.

In all these experiments, Bloom filters performed poorly, even for fairly small μ. This is no surprise because Bloom filters were not designed for range queries.

5.3.4 *Range-based Bloom Filters*

Figure 9 shows the false positive rate of a range-based Bloom filter for a Uniform database and Zipf1 query workload with range queries ($\mu = 30$). In such a range-based Bloom filter, each cell represents a whole range of values, rather than a single value in a classic Bloom filter. Range-based Bloom filters, thus, take an additional parameter, r, which determines the granularity (or size of the range). For $r = 1$, range-based and classic Bloom filters are the same. Figure 9 shows the results for two databases that were randomly generated with different seeds.

As shown in Figure 9, it is virtually impossible to set the r parameter correctly. Depending on the query workload and the database, a different setting of r is needed and even minimal changes to the optimum setting can have dramatic impact on the performance.

In general, setting the r parameter of such a coarse-grained, range-based Bloom filters exhibits a fundamental trade-off that is impossible to predict. With a growing r, the number of false positives caused by collisions of the hash functions decrease. However, with a growing r, the number of false positives caused by a the coarse granularity increases. With a "Uniform database / Uniform workload", the graph is less bumpy, albeit still with a high sensitivity to the right setting of r. As soon as the database or workload exhibits any kind of skew, we observed the bumpiness shown in Figure 9 in all our experiments.

5.4 Experiment 2: Scale Database

Experiment 1 studied the precision of ARFs and Bloom filters for point queries, range queries and various different scenarios of skew in the distribution of keys in the cold store and in the workload. We now turn to presenting the results of experiments that assessed the quality of filters with a varying number of distinct keys in the cold store. We know from experience with Bloom filters that Bloom filters are at their best if the filter has about 8 bits per key: More bits per key do not significantly improve precision and fewer bits per key hurt precision.

Figures 10a and 10b show the precision of the three ARF variants and Bloom filters with a constant size of 8000 bits while varying the number of distinct keys from 100,000 to 1 million, thereby changing the "bits per key" ratio from 8 to 0.8. Not surprisingly

	Point Queries ($\mu = 0$)		*Range Queries* ($\mu = 30$)	
	1Kbits	*3Kbits*	*1Kbits*	*3Kbits*
ARF-1	900 ($\pm$ 362)	2536 ($\pm$ 1004)	1500 ($\pm$ 1360)	3798 ($\pm$ 3605)
ARF-64	137 ($\pm$ 180)	200 ($\pm$ 100)	478 ($\pm$ 603)	670 ($\pm$ 900)
Bloom	178 ($\pm$ 178)	219 ($\pm$ 52)	273 ($\pm$ 300)	696 ($\pm$ 300)

Table 2: Lookup Time (Cycles): Unif. DB, Unif. Workl.
Vary Queries, No Updates, Vary Size, 1000 Distinct Keys

and confirming all previous studies with Bloom filters, the precision of Bloom filters got worse (i.e., increasing false-positive rate) as the "bits per key" ratio dropped. Figure 10a shows the results for a Uniform database and Uniform workload; Figure 10b shows the results for a Uniform database and a Zipf1 workload, but the results for Bloom filters were almost the same because the precision of Bloom filters does not depend on skew. In these experiments and all other experiments, we manually tuned the Bloom filters so that the optimal number of hash functions was used for each "bits per key" ratio. Figure 10c shows that the precision of Bloom filters is constant if the "bits per key" ratio is constant. In this experiment, we increased the size of the Bloom filter with the size of the database to achieve a constant "bits per key" ratio. Figure 10c shows the results for 4 and 8 bits per key. In both cases, the false positive rate is constant, independ of the number of distinct keys.

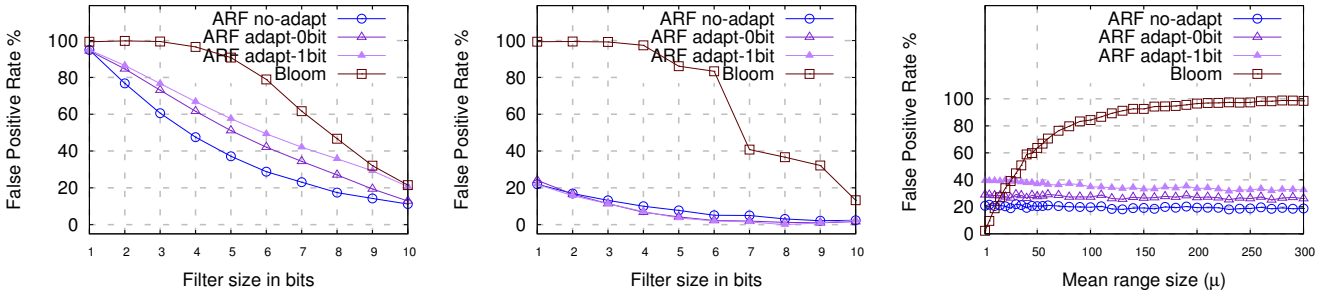
Predictability and robustness is one of the strengths of Bloom filters. It turns out that ARFs have the same features. Just like Bloom filters, ARFs get worse if the "bits per key" ratio drops. This effect is shown in Figures 10a and 10b. Likewise, ARFs have constant precision if the "bits per key" ratio is constant. This effect is shown in Figure 10c. In this figure, the ARF lines are just as flat as the lines for the Bloom filters. Figures 10a to 10c show these effects for range queries and two specific database / workload distributions, but the same effects can be observed for point queries and all other distributions.

5.5 Experiment 3: Latency

In addition to *precision, graceful-degradation, space efficiency, robustness,* and *generality* which were all studied as part of Experiments 1 and 2, *speed* is another important requirement for a filter in Project Siberia. Table 2 presents the lookup times for Bloom filters and two different kinds of ARFs: (a) a single, big ARF such as the one shown in Figure 2 and as used in all other experiments reported in this paper (referred to as ARF-1), and (b) a forest of 64 ARFs such as the one shown in Figure 3 (referred to as ARF-64). Lookup times do not vary significantly between the different ARF variants so that we only present the results for "ARF no-adapt" here. Table 2 reports on the mean lookup times (and standard deviation) in cycles measured over 20,000 queries. On a machine with cores clocked at 2.66 GHz, 2666 cycles take 1 μsec.

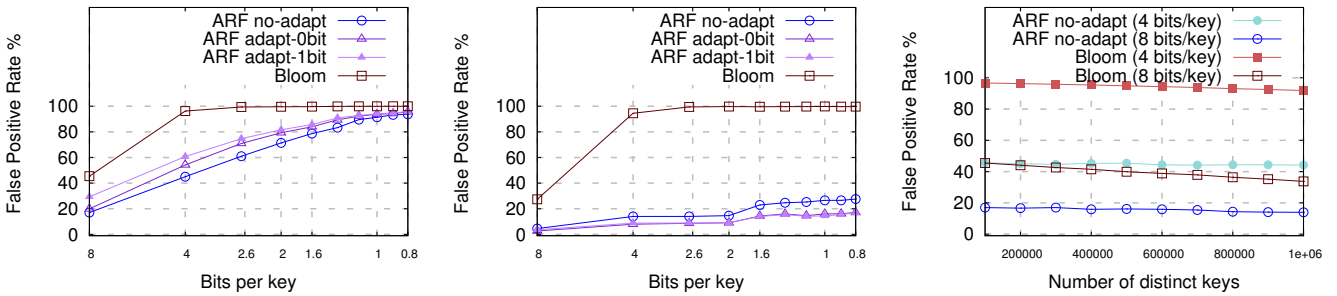
Looking at Table 2, the first observations is that Bloom filters were faster than ARFs. Bloom filters involve executing a few hash functions and lookups in a Boolean vector. These operations are cheaper than navigating through an ARF. The complexity of Bloom filters only mildly increased with the size of the filter (one hash function for 1KBit filters and two hashes for 3KBit filters). Also, it only mildly increased with the size of the range because of short circuiting. Furthermore, Bloom filters have a low deviation.

ARFs were more expensive and had a higher standard deviation because they are not balanced. Queries that hit the leftmost part of the filter were much quicker to answer than queries that hit the rightmost part due to the breadth-first traversal. Furthermore, the ARFs were more sensitive to the filter and range sizes. In theory, all operations on an ARF have $\mathcal{O}(n)$ complexity with n the size of the



(a) Uniform / Uniform ($\mu = 30$) (b) Uniform / Zipf1 ($\mu = 30$) (c) Uniform / Uniform (8KBits)

Figure 8: Exp. 1 (Range Queries), Precision: Uniform DB, Vary Workload, No Updates, Vary Size, Vary μ, 100K Distinct Keys



(a) Uniform / Uniform, Vary Bits per Key (b) Uniform / Zipf1, Vary Bits per Key (c) Uniform / Uniform, Vary Keys (4/8 Bits per Key)

Figure 10: Exp. 2 (Range Queries), Precision: Uniform DB, Vary Workload, No Updates, Vary Size, $\mu = 30$, Vary Distinct Keys

	Escalate		*De-escalate*	
	1Kbits	*3Kbits*	*1Kbits*	*3Kbits*
ARF-1	32788 ($\pm$ 30330)	91847 ($\pm$ 55905)	59974 ($\pm$ 322307)	166972 ($\pm$ 247141)
ARF-64	6449 ($\pm$ 2937)	7609 ($\pm$ 3207)	24357 ($\pm$ 6710)	25134 ($\pm$ 6697)

Table 3: Maintenance Time (Cycles): Unif. DB, Unif. Workl.
Point Queries, No Updates, Vary Size, 1000 Distinct Keys

ARF. Nevertheless, ARFs were competitive in all our experiments with ARFs of a few thousand bits and it never took more than a few μsecs to carry out a lookup. Large ARFs must be partitioned and embedded into a balanced search structure; either into an existing B-tree or into one that is dedicated for this purpose (Section 3).

Comparing the ARF-1 and ARF-64 results, embedding an ARF into a B-tree and thereby saving the cost of navigating the first layers of an ARF improved performance dramatically. In the best cases, a (small) ARF can even outperform a Bloom filter in terms of lookup times.

Table 3 shows the mean clock cycles (and standard deviation) needed to escalate and de-escalate an ARF for the "ARF adapt-0bit" variant. These operations are significantly more expensive in our implementation, mainly because modification of the shape of the ARF is needed, but they are still below 100 μsecs in all cases. Our implementation of ARFs is not tuned for escalation/de-escalation latency because such operations are rare. They only occur in the event of false positives and in such cases latency is high anyway because of accesses to the cold store. Again, embedding the ARF into a B-tree helped significantly to improve performance.

5.6 Experiment 4: Adapt to Data Changes

One important requirement of Project Siberia is that data can move from the hot to the cold store and vice versa at any point in time. We believe that a similar requirement to maintain a filter in the presence of updates is also important for other applications of ARFs. To study the effectiveness of ARFs in such scenarios, we carried out the following experiment:

1. Create an initial cold database and train the ARFs and Bloom filters (just as in all previous experiments).
2. Run queries, thereby measuring the false-positive rate and possibly adapting the ARFs for every false positive.
3. Delete the records of a random key from the cold store.
4. Insert the records of a random key into the cold store.
5. Goto Step 2.

Figure 11a shows the false-positive rate as a function of the number of updates (i.e., iterations of the loop above) of point queries with a Uniform workload. In this case, the original database containing 100,000 distinct keys was created using a Uniform distribution and the keys of all *Inserts* in Step 4 were also generated using a Uniform distribution. Figure 11b shows the false-positive rate for point queries if the data and update distribution is Zipf.

Both figures show that the false-positive rate of Bloom filters and "ARF no-adapt" increased over time as more and more updates were carried out. If queries repeatedly asked for keys and the corresponding records had been deleted from the cold store (as part of Step 3), then these approaches repeatedly resulted in a false positive for these queries. In other words, once constructed, these approaches are not able to adapt and therefore deteriorated over time. In contrast, the adaptive ARF variants did not deteriorate: They *self-repair* with every false positive and, therefore, adjust to changes in the database.

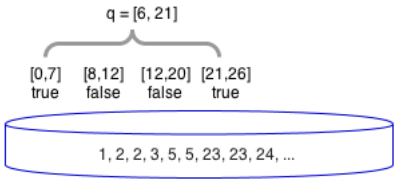


Figure 12: Example False Positive

5.7 Experiment 5: Adapt to Workload Changes

Figure 11c shows how ARFs adapt to changes in the workload. In this experiment, the ARFs were trained according to a Zipf1 workload. Then, we ran 300,000 queries according to the same Zipf1 workload. After that, we changed the Zipf1 workload so that other regions became hot and previously hot regions cooled down. As shown in Figure 11c, the adaptive ARF variants were able to quickly adjust to the workload change and performed equally well throughout the whole spectrum of queries. The performance of the non-adaptive ARF variant, in contrast, got significantly worse after the workload change. Bloom filters are also not able to adapt to workload changes and we are not aware of any dynamic Bloom filter variants. Figure 11c does not show this deficiency of Bloom filters because in this experiment, Bloom filters had poor precision even without a change in the workload. (This effect did show in experiments with point queries.)

6. QUALITATIVE ANALYSIS

Section 5 showed empirically that ARFs are precise, fast, space-efficient, and degrade gracefully for a wide range of possible databases and workloads. This section analyzes more formally the effectiveness of ARFs. Specifically, this section answers the following question: Given a database, workload, and fixed space budget, do ARFs minimize the number of false positives?

Unfortunately, the answer to this question is *no*, independent of which ARF variant is used. The reason is that finding an optimum range filter is NP-hard. What ARFs do is they approximate a solution. The remainder of this section discusses these two observations. For space reasons, we only sketch the proofs.

6.1 Optimal Range Filters

The first step in showing that finding optimal range filters is NP-hard is to show that an optimal range filter must maximize the region that is covered by leaves of the ARFs with their *occupied bit* set to *false*.

Figure 12 depicts how a false positive can occur in an ARF for a query that spans multiple leaves of an ARF. The figure shows that false positives occur if the query overlaps with one or two leaves whose *occupied bit* is *true*, yet the cold store does not contain any data for the overlapping region(s). Figure 12 shows that this overlap, the *region of uncertainty*, is at most the size of the ranges of two leaves with which the query overlaps. In general, the probability of a false positive is in the order of the average size of a leaf whose *occupied bit* is set to *true*. The important observation is that *the probability of a false positive is independent of the size of the query!* This observation allows us to reason about optimal ARFs without considering the size of queries.

For optimality, the query distribution does play a role. However, given the observation of the last paragraph, it is easy to show that finding an optimal ARF is NP-hard (in fact, NP-complete) for a Uniform workload. If it is NP-hard for a Uniform workload, it is NP-hard in the general case, too. The proof goes by reduction to the knapsack problem which is known to be NP-complete: The size of the knapsack is mapped to the space budget. The weights of items to be packed into the knapsack are mapped to the size of gaps between two distinct values in the cold store.

6.2 Practical Range Filters

What ARFs do is to approximate the knapsack problem. They do so by trying to incrementally improve the allocation of leaves whenever a false positive occurs. The best strategy depends on the data and workload distribution. If these are not known, any practical system needs to rely on estimates and heuristics.

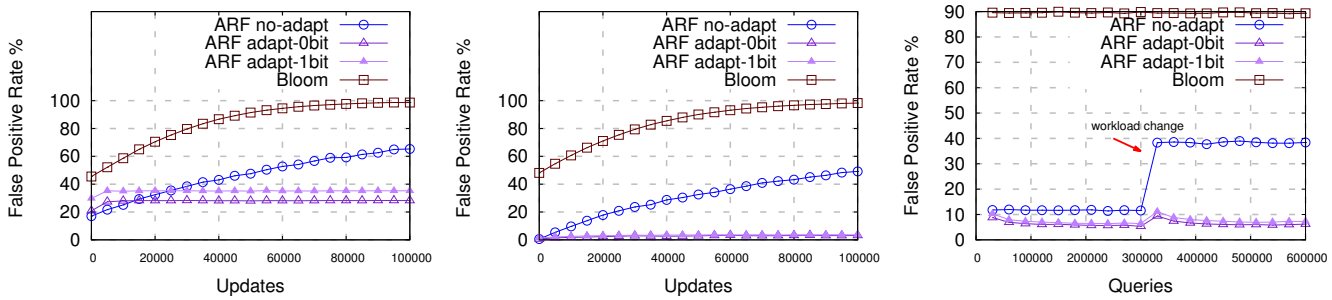
Incidently, buffer replacement policies try to solve exactly the same problem as ARFs: They also try to approximate the knapsack problem by keeping those objects in a cache that are likely to be used with the highest probability next. That is why all the work on replacement policies for buffer management (e.g., [1]) is directly applicable to the design of adaptive range filters like ARFs. Using a clock replacement policy as proposed in Section 4, for instance, approximates a "least frequently used" policy (LFU) which in turn tries to simulate the A_0 algorithm which was shown to be optimal in cases in which all objects have the same size and replacement cost [1]. Just as for buffer management, there are many scenarios in which the clock strategy is not optimal for ARFs. As explained in Section 4.2, we use the clock strategy in the absence of any further knowledge because it is space-efficient. The results presented in Section 5 show that it indeed works well for a wide range of data and workload distributions; in particular, it worked well for real applications such as the Xbox workload. Studying more sophisticated policies is left for future work, but it is encouraging to see that good results can already be achieved with a simple policy like *clock*.

7. RELATED WORK

The closest related work to ARFs are Bloom filters [5]. Bloom filters were designed for the same purpose of quickly testing whether a query matches a set of keys. Bloom filters were designed for queries with equality predicates and are still the best known technique for such queries. Theoretically, Bloom filters are also applicable to range queries by rolling out a range query into a series of point queries, but our experiments showed that this approach is not competitive even for fairly small ranges. There has been some previous work on special kinds of Bloom filters for range queries [3]. Unfortunately, that work is not competitive because it requires a considerable amount of space per key.

One of the main features of ARFs is that they take advantage of skew in the data and query distributions. *Weighted Bloom Filters* have been developed to take advantage of workload skew [7, 18]. Weighted Bloom filters, however, come at a high cost in both space and time to keep track of the *hot* keys. Furthermore, we are not aware of any Bloom filter variant that exploits skew in the data distribution. Another feature of ARFs is its adaptivity with regard to updates and workload changes. *Counting Bloom Filters* are a way to adapt to updates [6, 13]. Just like Weighted Bloom Filters, Counting Bloom Filters come at a high price in terms of space. Furthermore, we are not aware of any work on Bloom filters that adapt to workload changes. With ARFs, the adaptivity comes naturally and is inherent to the data structure.

There has been extensive work on histograms [14] and other summarization techniques in the database community [4]. There has also been work on self-adapting histograms and database statistics (e.g., [8]), one of the prominent features of ARFs. Like ARFs (and Bloom filters), all these techniques serve the purpose to provide a compact data structure that allows to get approximate answers to queries. Indeed, each leaf node of an ARF can be regarded



(a) Exp. 4: Uniform / Uniform, Vary Updates
$\mu = 30$, 100K Distinct Keys, 8 Bits/Key

(b) Exp. 4: Zipf / Uniform, Vary Updates
$\mu = 30$, 100K Distinct Keys, 8 Bits/Key

(c) Exp. 5: Uniform / Zipf1, Change Workload
$\mu = 30$, No Updates, 100K Distinct Keys, 5 Bits/Key

Figure 11: Exp. 4 & 5: Adaptation to Changes in the Database (Exp. 4) and Workload (Exp. 5)

as a bucket of a histogram: Instead of a counter (or value distribution), an ARF keeps only a simple *occupied bit*. What makes ARFs special is that they are allowed to err on the *high side* (i.e., false positives) but never on the *low side* (i.e., false negatives).

An interesting piece of related work is work carried out in the Mid 1990s on predicate-based caching and semantic caching [9, 15]. While [9, 15] propose index structures to describe the contents of such a query cache, their approaches are not as highly optimized and tuned as ARFs. We believe that ARFs are applicable to improve the performance of semantic caching techniques and studying this application of ARFs is an interesting avenue for future work.

Another interesting area in which ARFs may be applicable are data stream processing systems; e.g., LeSubscribe [12]. These systems involve filters to find out for which continuous queries an event might be relevant. Again, studying how ARFs can be applied to such systems is left for future work.

We adopted the idea of escalating and de-escalating hierarchical data structures to improve / degrade the precision of an index from [11]. [11] proposes a variant of Skip lists to trade lookup performance for maintenance effort in context-aware stream processing systems.

8. CONCLUSION

This paper presented ARFs, a versatile data structure to filter range queries. ARFs have a number of features. Most importantly, they are self-adaptive and exploit skew in the data and query/update distribution. Furthermore, they are space-efficient and have affordable latency. Extensive experiments demonstrated the effectiveness of ARFs for different data distributions and workloads. Overall, we believe that ARFs are for range queries what Bloom filters are for point queries.

There are a number of areas for future work. Most importantly, we plan to expand the analysis (Section 6) and develop a mathematical framework for optimizing ARFs along the lines of the framework that has been developed for Bloom filters. As ARFs exploit skew and adapt dynamically to the workload and data distribution, developing such a framework is more challenging than for Bloom filters.

Acknowledgments. We would like to thank the reviewers for their many constructive comments to improve this work. Several experiments were specifically proposed by the reviewers.

9. REFERENCES

[1] A. V. Aho, P. J. Denning, and J. D. Ullman. Principles of optimal page replacement. *J. ACM*, 18(1):80–93, 1971.

[2] A.-K. Alexiou. Adaptive range filters for query optimization. Master Thesis 84, Systems Group, ETH Zurich, 2013.

[3] B. Baig, H. Chan, S. McCauley, and A. Wong. Range queries using Bloom filters. http://www.scribd.com/doc/92463819/Ranged-Queries-Using-Bloom-Filters-Final.

[4] D. Barbará, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, T. Johnson, R. T. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik. The new jersey data reduction report. *IEEE Data Eng. Bull.*, 20(4):3–45, 1997.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[6] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting Bloom filters. In Y. Azar and T. Erlebach, editors, *ESA*, volume 4168 of *Lecture Notes in Computer Science*, pages 684–695. Springer, 2006.

[7] J. Bruck, J. Gao, and A. Jiang. Weighted Bloom filters. 2006.

[8] C.-M. Chen and N. Roussopoulos. Adaptive selectivity estimation using query feedback. In R. T. Snodgrass and M. Winslett, editors, *SIGMOD Conference*, pages 161–172. ACM Press, 1994.

[9] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB*, pages 330–341. Morgan Kaufmann, 1996.

[10] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *SIGMOD Conference*, 2013.

[11] J.-P. Dittrich, P. M. Fischer, and D. Kossmann. Agile: Adaptive indexing for context-aware information filters. In F. Özcan, editor, *SIGMOD Conference*, pages 215–226. ACM, 2005.

[12] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In S. Mehrotra and T. K. Sellis, editors, *SIGMOD Conference*, pages 115–126. ACM, 2001.

[13] L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *SIGCOMM*, pages 254–265, 1998.

[14] Y. E. Ioannidis. The history of histograms (abridged). In *VLDB*, pages 19–30, 2003.

[15] A. M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. In *PDIS*, pages 229–238. IEEE Computer Society, 1994.

[16] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[17] J. Levandoski, P.-A. Larson, and R. Stoica. Identifying hot and cold data in main-memory databases. In *IEEE Conf. on Data Engineering*, 2013.

[18] M. Zhong, P. Lu, K. Shen, and J. I. Seiferas. Optimizing data popularity conscious Bloom filters. In R. A. Bazzi and B. Patt-Shamir, editors, *PODC*, pages 355–364. ACM, 2008.