Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 82

Systems Group, Department of Computer Science, ETH Zurich

in collaboration with

Microsoft Research

Adaptive Range Filters for Query Optimization

by

Ariadni-Karolina Alexiou

Supervised by

Prof. Donald Kossmann

March 2013

inf | Informatik
Computer Science

Abstract

Bloom filters are a great technique to test whether a key is present or not in a set of keys. This thesis presents a novel data structure called ARF for Adaptive Range Filter. 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 thesis describes the principles and methods for efficient implementation of ARFs and presents the results of comprehensive experiments that assess the accuracy, space, and lookup time of ARFs. Furthermore, this thesis shows how ARFs can be applied to a commercial database system that partitions data into hot and cold regions in order to optimize queries that involve only hot data. Extensions of the ARF data structure onto the multi-dimensional space are also presented.

Preface

This thesis has been carried out with the collaboration of Microsoft Research. The duration of the thesis was six months, from 15^{th} of September 2012 to 15^{th} of March 2013. Part of this work has been submitted to the VLDB 2013 conference, hence parts of the text have been written by co-author and supervisor Donald Kossmann.

Acknowledgments

I would like to express my gratitude to Prof. Donald Kossmann for providing me with the opportunity to work in such an interesting field and his constant guidance, investment of time for meetings and encouragement during the thesis. I am also very grateful for the collaboration and support from Paul Larson from Microsoft Research, who very frequently, despite being overseas, attended our online meetings and gave his insights. Finally, I would also like to thank Matthias Röthlin for proofreading my thesis.

Contents

Chapter 1

Introduction

1.1 Motivation

A popular rule of thumb says that 80 percent of database accesses go to 20 percent of the data in OLTP databases. This skew in the access distribution becomes even more pronounced when considering time: What is hot today is likely to be cold tomorrow and at any point in time, only a small fraction of the data is needed. For instance, in the context of a classic OLTP database, orders that need to be shipped today are much more likely to be part of a database transaction than orders that have already been delivered a week ago.
There are many ways to exploit skew in the access distribution. Caching frequently accessed data in main memory is the most prominent example. Replication of data in a distributed system is another technique that can speed up access to hot data. Partially indexing the hot data is yet another technique that exploits access skew [1].
All of these techniques have two things in common. First, more resources and more expensive resources are invested to support access to hot data; e.g., hot data is kept in main memory or additional, fine-grained indexes are built to keep track of hot data. Cold data, on the other hand, is more expensive to access because no optimizations are applied and it may also physically reside in higher-latency storage. Second, to be fully effective, these techniques need to *know* whether a query involves only hot data or whether access to cold data is needed. Devising and studying techniques that detect whether a query needs to access cold data is the goal of this thesis. We will from now on refer to the region where hot data is stored as the "hot store" and the one where cold data reside as the "cold store".
Detecting *hot-only queries* is straightforward for point queries on the primary key of a relation: If the record can be found in the hot store, then there is no need to go to the cold store. Most caching systems exploit this observation. Furthermore, even if the primary key of a relation is not involved, this task is easy if the query involves an equality predicate (e.g., *find all orders that are due today*) and there is a Bloom filter [2] that indicates whether relevant data might be in the cold store. The more general case of a query with a range predicate (e.g., *find all orders that are due within the next week*), however, has not been studied so far and is the focus of this work.

A hot versus cold partitioning on the data is not the only set-up for which an access filter that handles range queries can be useful. A filter that quickly answers "Yes" or "No" to whether a specific location (in the broadest sense) contains data for a given query, can also be used for other purposes in the context of Big Data. 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 a range filter that quickly detects files or directories of files that potentially have relevant events for the specified time period as a method to optimize the execution of these queries. In the interest of simplicity and because hot versus cold partitioning was the initial motivation of this thesis, we will keep referring to cold and hot data/stores throughout the thesis, where the cold store is essentially the data location for which we wish to have access filters.

1.2 Problem Statement

This thesis aims to implement a data structure that is able to answer the question whether processing a range query requires looking at data in a specific location or not, that is, act as an access filter to that location. A quick "No" to this question avoids unnecessary accesses and lookups, therefore saving resources. We will initially focus on the case where this query involves one attribute (e.g., order date, age, salary) and will provide multi-dimensional extensions later.

For a range query to make sense, the domain queried must be have a natural ordering. Also, since a point query can be thought of as a degenerate range query, this data structure is expected to work with point queries as well. We aim to make this data structure as fast and as space-efficient as possible. We also want the filter to be able to adapt to changing workloads and datasets without a lot of overhead. Motivated by the use case within modern database systems with hot/cold partitioning schemes, we also aim to exploit the resulting skew in data and access patterns to improve the quality of the filter.

1.3 Contribution

This thesis presents the design, implementation and evaluation of a new trie-based data structure called ARF, which is short for *Adaptive Range Filter*, for the purpose of answering the question whether a query involves data that is stored in a specific location or not. In essence, the ARF is a Bloom filter for range queries.
In addition to being able to handle range and point queries, the ARF can adapt to the data and query distribution and adjust its shape accordingly, even in the presence of updates and/or drastic workload changes. We show that the ARF can be competitive with the plain Bloom filter in terms of accuracy even for point queries under certain conditions. More importantly, we will show that it constitutes the only current solution for filtering range queries, while at the same time being efficient with respect to querying time and storage space required. We look at a variety of scenarios which demonstrate the performance of the

ARF. We also extend the ideas of the one-dimensional ARF to implement and evaluate several multi-dimensional variants of the data structure and report the results.

1.4 Outline of the Thesis

This document is organized as follows: Chapter 2 gives an overview of Project Siberia, which is the project that motivated this work and discusses what skew means in the context of it. Chapter 3 discusses related work. Chapter 4 describes the ARF data structure in detail and chapter 5 describes the ways it can learn from and adapt to the data and the query workload. Chapter 6 presents experimental results that assess the accuracy, space, and query time of ARFs for a number of different workloads and usage scenarios. Chapter 7 presents the implementation and the performance evaluation of multi-dimensonal ARFs. Finally, chapter 8 contains conclusions and possible avenues for future work.

Chapter 2

Project Siberia

To illustrate the effectiveness of ARFs, it is useful to think of them in terms of a concrete use case. This chapter describes how we envision to use them as part of Project Siberia[3] that is investigating techniques for managing cold data in Microsoft's new commercial database engine, Hekaton[4], that is optimized for large main memories and many-core CPUs. In this chapter we give an overview of the project's architecture and current filtering solutions, motivate the need for adaptive access filters for range queries and also justify the expectation of skew within the project.

2.1 Architecture

The goal of Project Siberia is to have the system automatically and transparently migrate cold data to cheaper external storage. The hot part of the database is kept in main memory, thereby exploiting dedicated query and transaction processing techniques to guarantee the most efficient access to hot data. Since the hot data is assumed to be only a fraction of the entire database, it is affordable to invest the additional resources for it. The cold data is stored using more traditional and low-cost techniques. However, where data is stored is completely transparent to users and applications, except for possible performance differences.
Figure 1 shows how queries 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 issue SQL queries and update statements just like in any other relational database system. These queries are parsed, optimized, and executed; again, just like in any other database system. Logically, every query must be carried out on the *union* of the data stored in the hot and cold stores. For instance, a query that asks for the number of orders by a specific customer to be shipped within the next week, needs to retrieve all relevant orders from both stores and then compute the aggregate.
The focus of this master thesis is on the design of the *filter* component shown in figure 2.1. 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 for a query or update operation. Given a query, the filter returns *true* if the cold store possibly contains records

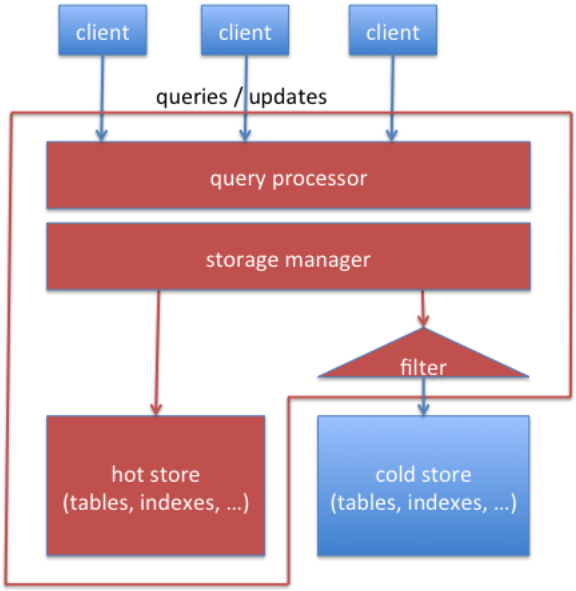

Figure 2.1: Query Processing in Project Siberia

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.

2.2 Partitioning of the data

As part of Project Siberia, a number of different partitioning techniques have been developed that decide which data to place in the *hot store* and which data to place in the *cold store*. Some of these techniques involve hints from system administrators (e.g., *place all orders with status = "shipped" into the cold store*). Other techniques are fully automatic and monitor the access frequency of records using log analysis. Describing and studying all these techniques is beyond the scope of this work and is described in detail in [3].

2.3 Use of Access Filters

Independent of the specific partitioning technique deployed, an important requirement of our system is that the partitioning of the database is transparent to the developer who writes queries and updates. So, we cannot assume that the developer specifies as part of the query whether the query involves only *hot* data, that is, there is no logical property or predicate that we can rely on to indicate this information. There is no hardcoded "threshold" value of an attribute that determines that the relevant record is cold. That is why we need

filters as shown in figure 2.1.

An access filter needs to be constructed without making assumptions on the partitioning scheme; in particular, a filter needs to be able to handle queries on the "customer id" tied to an order, even if the data is partitioned according to the "date" of the order. Furthermore, the partitioning can change at any point in time; there might be a continuous migration of records between the hot and cold stores. As will become clear, this observation makes it necessary to devise adaptive techniques which enable the filter to self-repair on-the-fly as the partitioning changes.

2.3.1 Filter Requirements

Overall, the filters for Project Siberia must fulfill the following requirements:

- *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 the hot store to guarantee efficient access to the filter. As space in the hot store 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 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 the hot store.
- *Robustness:* By design, most queries and updates are hot-only. Therefore, both the data and the query distribution is expected to be heavily skewed. We define skew in this context in section 2.4. Clearly, the filter should be designed to work well in the presence of skew. 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 any kind of query and update: that is, both equality and range predicates.

2.4 Skew in the context of Project Siberia

2.4.1 Data Skew

In databases, data skew is defined to mean that there are a lot of database records with an attribute that lies within a particular and sufficiently small range of values, leaving other ranges empty or sparsely populated. This can also mean that there are some specific values that are a lot more popular than others. This is especially relevant in the context of histograms [5]. The cardinality of a value is not relevant within the context of an access filter however, because we are only interested in whether the cold store contains relevant values within a given range, and not how many. Therefore, data skew in our use case can only be thought of in terms of locality. Thus, skewed data means data whose *distinct* values are clustered around one or possibly more regions.

Why can data skew be reasonably expected in the cold store in Project Siberia?

There is a possibility that data in the cold store is clustered around some values and not uniformly distributed over the domain of possible values for two reasons:

- Many database attributes, even though they are defined as having the capability to take on a wide range of values, do in fact constrain themselves to a smaller domain. For instance, order dates of SQL type DATE can theoretically run back thousands of years but an online store that has been running since 2005 will never deal with orders from the 20th century. Therefore, both the hot store and the cold store will be clustered around values from 2005 onwards. A counter-example where the attribute uses the whole domain would be the customer names of the store, which are generally uniformly scattered throughout the phone book from A to Z.

- The fact that the record was moved to the cold store may sometimes, but not always, imply that the value of one or more of its attributes had something to do with it - for instance, the timestamp is too old for the record to be frequently queried in the present. Therefore, the cold store will contain mostly records with older dates.

2.4.2 Query Skew

We've established that project Siberia migrates data that is not used frequently to the cold store. Real-life OLTP transactional workloads exhibit considerable access skew. This skew, as reported in [3], is either:

- *Value-correlated.* For example, when tracking shipments in an e-commerce site, only the most recent orders are accessed frequently and a given order gets accessed less and less the more it ages. In that case, the skew can be thought of as being correlated to the value of the timestamp.

- *Natural.* For instance, some items in online e-commerce sites such as Amazon, simply happen to be much more popular than the rest and are queried more often.

2.5 Current Solutions

The classic way to implement filters where false negatives are forbidden and false positives need to be minimized, is to use Bloom filters [2]. Bloom filters are correct, precise, space efficient, fast, and degrade gracefully and fulfill most of the requirements in section 2.3.1. Indeed, Project Siberia uses Bloom filters for attributes that are mostly accessed with equality predicates (i.e., point queries and updates). Unfortunately, Bloom filters are by definition not a good match for range queries, as we will show in the next chapter which considers related work. Thus, they violate the *Generality* requirement. Furthermore, they do not exploit skew in the data and workload and do not adapt to changes in the data and workload. Bloom filters, thus, also violate the *Robustness* requirement. To overcome these limitations, we have designed the ARF, a data structure to filter range queries that meets all the aforementioned requirements.

Chapter 3

Related Work

3.1 Bloom filters

The closest related work to ARFs are Bloom filters[2]. 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 the best known technique for such queries. There's also some interesting work on integrating them into an R-Tree for querying multi-dimensional data [6], although the Bloom filters themselves are not used for range queries within the proposed architecture. As we will demonstrate, two areas they fall short in are handling range queries and efficiently exploiting skew.

3.1.1 Range query handling

Theoretically, Bloom filters may also applicable to range queries - to handle a range query r we simply query every point in r and return the aggregated result, assuming it is possible to enumerate the points in the range. This has two disadvantages: Range queries take linear time to evaluate, with respect to the length of the range, and also, the false positive rate of the Bloom filter increases dramatically.
When using 8 bits per distinct value in the cold store, which is a setting often used in Project Siberia because it offers a good trade-off between accuracy and space, the error rate of the Bloom filter for a point query is approximately $e = 2\%$. However, for range queries with median length $\mu = 30$ the error rate becomes: $1 - (1 - e)^{30} = 45.5\%$

Clearly, the Bloom filter approach is not suited for handling range queries because of poor performance. However, one could argue that, when investing a larger number of bits per key, the false positive rate will become tolerable for small range queries, although not for larger ones. Indeed, when using 16 bits per key, the error rate for range queries with $\mu = 30$ is quite small. That still leaves a sweet spot for a different data structure to be used when we are constrained to less than 10 bits per key. Also, we must take in mind that, for certain data types used in modern databases, for instance, floating point data, determining and going through all points in a range does not constitute a realistic approach. And finally, we would like to be able to query abritrarily large ranges without

the error rate soaring. To this end, a data structure specifically designed to handle range queries, such as the ARF, is necessary.

3.1.2 Skew handling

Classic Bloom filters are skew-agnostic. There is no way to favor specific points that are queried often and somehow shield them from the probability of false positives. The probability of a false positive is the same, no matter which point is queried. However, more recent work deals with this exact issue. Weighted Bloom Filters have been shown to work well if the workload is skewed (i.e., several data points are queried more often than others)[7, 8]. We are not aware of any work on Bloom filters that take advantage of skew in the data distribution. Another feature of ARFs is its adaptivity with regard to updates. Counting Bloom Filters are a way to achieve this same feature with Bloom filters [9, 10]. Weighted and Counting Bloom Filters come at a significant cost of increased space requirements to keep track of the weights and counts, respectively. With the proposed ARF, the adaptivity comes naturally and is inherent to the data structure.

3.2 Other

There has been extensive work on histograms [5] and other summarization techniques in the database community [11]. There has also been work on self-adapting histograms and database statistics (e.g., [12]), 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 as a bucket of a histogram: Instead of a counter (or value distribution), an ARF keeps only a simple *occupied bit*. What makes ARFs (and Bloom filters) 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 [13, 14]. Recently, [15] proposed a related approach of recycling query results. As mentioned in the introduction, caching is an alternative approach to exploit skew in the access distribution and provide high performance to queries that access only hot data. Most caching schemes (e.g., in operating systems) only support cache lookups by the primary key of a collection. In contrast, the goal of predicate-based and semantic caching is to support any kind of query, including queries with range predicates. While [13, 14] 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., XFilter [16] or LeSubscribe [17]. 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 [18]. [18] proposes a variant of Skip lists to trade-off lookup performance and maintenance effort for context-aware stream processing systems.

Chapter 4

The Adaptive Range Filter

This chapter describes the main ideas of the ARF data structure. 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.

4.1 Overview

Figure 4.1 gives an example of an ARF with keys in the domain [0,15]. It shows the values of a specific attribute of the records stored in the cold store and one possible ARF built for this bag of values. In most workloads, several ARFs need to be constructed for a table: one for each attribute that is frequently involved in predicates of queries and updates. For the case of queries involving two or more attributes refer to chapter 7 on multi-dimensional ARFs.
An ARF is a typically non-balanced 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*. This is visualized as either green(true) or red(false) in the figures. The intermediate nodes are white. The ARF of figure 4.1, for instance, indicates that the cold store has 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 4.1 indicates that the cold store has no records with keys in the ranges [8,9], [9,10], and [12,15]: the *occupied bits* of these three leaves are set to *false*.
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 example of figure 4.1).
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]

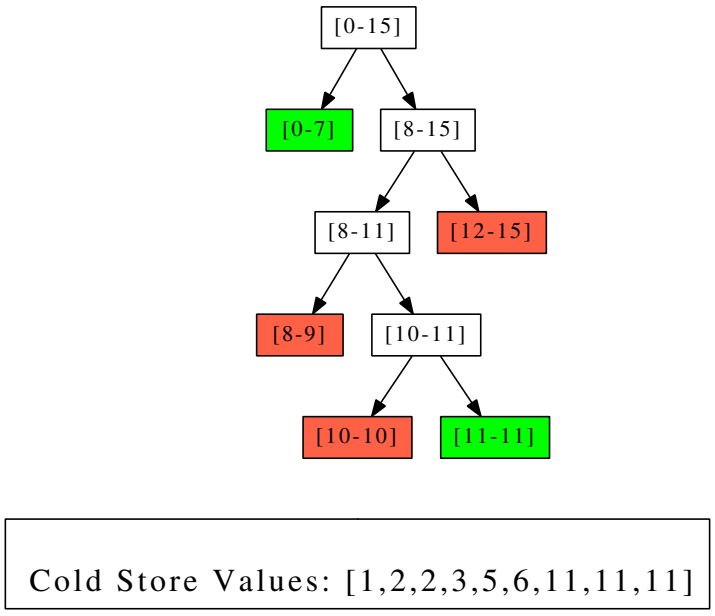

Figure 4.1: Example ARF

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 2.3.1, 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. 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 4.1 does not capture the fact that the cold store contains no records for 4 or 7 because doing so would require to grow the ARF much deeper. As a result, the ARF of figure 4.1 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 chapter 5.

4.2 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. Prefix compression for B-trees [19] applies the same idea. 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. In other words, the following four situations can arise:

- *00*: Both of the children of the intermediate node are leaves. For instance, Node [10, 11] in figure 4.1 is represented by this bit sequence.
- *01*: The left child is a leaf; the right child is not a leaf. For instance, Nodes [0, 15] and [8, 11] in figure 4.1 are encoded in this way.
- *10*: The right child is a leaf; the left child is not a leaf. For instance, Node [8, 15] in figure 4.1 is encoded in this way.
- *11*: None of the children are leaves. (This case is not shown in figure 4.1.)

A whole ARF can be serialized by serializing each node in a breadth-first traversal. For instance, the (shape of the) ARF of figure 4.1 could be represented by the following bit sequence (the first 01 encodes the root, the 10 encodes Node [8, 15], and so on):

01100100

Note that no pointers are needed: This bit sequence is all that is needed to determine both the shape and the ranges of all nodes of the ARF of figure 4.1. In addition to the bit sequence that encodes the intermediate nodes and, thus, the shape of the ARF, we must maintain the *occupied bits* of the leaves. Continuing the example of figure 4.1, the bit sequence that represents the five leaves is:

10001

The first 1 reprsents the *true* for the *occupied bit* of Leaf [0, 7], the following 0 represents the *false* for [8, 10], and so on.
Putting shape and occupied bits together, the ARF of figure 4.1 can be 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. In addition, we need to maintain the domain; i.e., the range covered by the root node. As shown in Section 4.3, this range is stored as part of a separate B-tree in the hot store which needs to store such ranges anyway. The underlying data structures holding these bit arrays also require some auxilliary storage.
Appendix A contains details of the implementation of BFS-encoded ARFs, the navigation algorithm, and some optimizations to speed up lookup times. It is also an option to use a DFS encoding for the ARF. For a comparison between the two and other alternative encodings, refer to section 4.4.

4.3 ARF Forests

In practice, 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 for this recommendation is that ARFs can be nicely embedded into existing index structures such as B-trees. Another reason is lookup performance and space efficiency. We will explain these reasons in the remainder of this section.
Figure 4.2 shows how ARFs can be embedded into a B-tree in the hot store. The upper part of figure 4.2 shows a normal, traditional B-tree which indexes

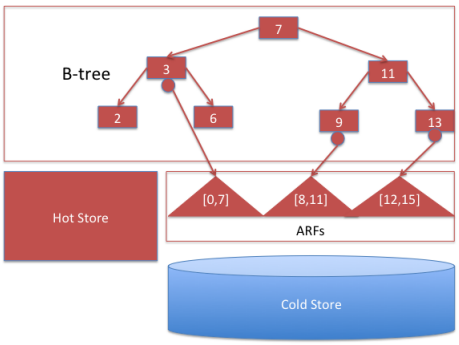

Figure 4.2: Integrating ARFs into a B-tree

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 through this B-tree just like any other B-tree.
What makes the B-tree of figure 4.2 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 of the B-tree with the marker 3 points to the ARF that 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 4.1 and 4.2, one way to interpret the forest of ARFs is to think of a big ARF with several nodes at its top chopped off. For instance, the forest of ARFs depicted in figure 4.2 emerged from the big ARF of figure 4.1 by removing nodes [0,15] and [8,15] and connecting the resulting three subtrees to the appropriate places in the B-tree. There are three reasons why doing so improves the space and time efficiency of the ARF:

- *Space:* The root node of the ARF need not store the boundaries of the range it represents. These boundaries are implicitly stored in the B-tree.

- *Space:* We save the bits to encode the upper levels of the big ARF. For instance, the four bits to represent nodes [0,15] and [8,15] in the big ARF of figure 4.1 are not needed and the three ARFs of figure 4.2 can be represented with a total of 9 bits (instead of 13 bits).

- *Time:* We save the cost to navigate the first levels of the ARF. In other words, while traversing the B-tree, we implicitly also navigate through the top levels of the ARF.

4.4 Alternative encodings

4.4.1 BFS versus DFS

A breadth-first encoding (as opposed to depth-first) of the ARF is used in our prototypes because it supports faster navigation through the ARF and is also easier to debug. In particular, navigating to the right sibling of a leaf, which is needed to filter large range queries that cover multiple leaves, is implemented by advancing to the next bit in the bit sequence that represents the ARF while a DFS encoding would require us to navigate through the entire left subtree first. Ultimately, the BFS encoding is simpler to write code for and has a speed advantage over the DFS encoding. However, it has the disadvantage of requiring auxillary space to store the length of each level, which may be negligible for big ARFs, but may be less so when using a forest of smaller ARFs. Therefore, the DFS-encoded ARF, which can be stored as one bit sequence, is a better candidate for integration within a B-tree. We implemented an ARF using DFS encoding to determine the exact speed disadvantage over the BFS encoding and found out that, for small ARFs in the order of a few hundred bits, the difference in access speed is negligible. These results are presented in section 6.5.1.

4.4.2 More than 2 children

It is also possible for inner nodes to have a fixed number of 3,4,5 or any number of children instead of 2. This has the result of some leaves at the lowest level being unused, unless the domain of the ARF is a power of the number of children used. This typically does not waste a lot of space, because the space budget is not enough to allow the ARF to grow to its maximum height in a lot of locations. The exact optimal number of children for a given ARF seems to be related to the particulars of the data distribution it describes. However,we observed that, in all of our benchmarks, going from 2 to 3 children always gave the ARF a small boost in accuracy. The boost was quite small, so we decided to use the plain ARF for the ARF prototypes within the context of this master thesis in the interest of simplicity.

Chapter 5

Learning and Adaptation

One of the powerful features of the ARF structure is that it can adapt to the data and query distribution, investing bits only where it is needed. For instance, large regions that contain no data such as [12,15] in figure 4.1 can be represented in a compact way. Likewise, large regions that are densely populated such as [0,7] in figure 4.1 can also 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 4.1. This chapter 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 primitive functions of splitting and merging nodes of an ARF. We also present specific adaptation and learning techniques. Throughout this chapter, we will show examples for a "single big ARF" approach. All the techniques apply naturally to a forest of ARFs (section 4.3).

5.1 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 5.1 shows how the ARF of figure 4.1 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 5.1a shows this minimal ARF for the example of figure 4.1. Figure 5.1b shows the next incarnation created by splitting this single node. Again, a split always partitions the range of its parents in half. Figure 5.1c 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. The sequence of splits shown in figure 5.1, for instance, could have been induced by processing the range query [12,15]. Concretely, this range query is processed as follows:

- The starting point is the ARF of figure 5.1a, which is an ARF in its most basic form. Probing the range query [12,15] results in *true*.

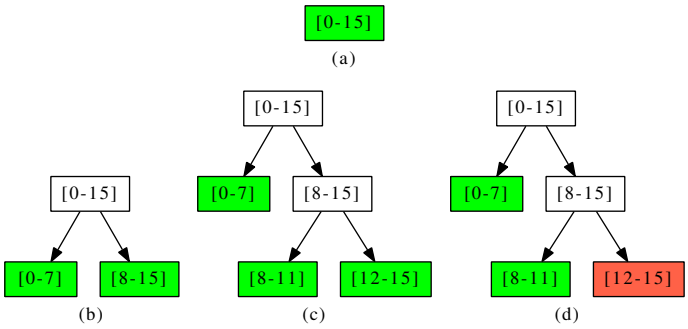

Figure 5.1: Example: Escalating an ARF for Query [12,15]

- The query is processed in the cold store. The result is empty.
- At this point, we know that [12,15] is a *false positive*. So, we know that the ARF of figure 5.1a is not accurate. To improve precision, we grow the ARF so that the resulting ARF reflects what we have learned; i.e. the end result as shown in figure 5.1d, correctly shows that the cold store contains no data for keys in the region [12,15]. We call this process *escalation*.
- Escalation is carried out recursively: We split the relevant leaf node until we have created leaves that are either fully contained or disjoint with the false positive query. In this example, we need to carry out two steps of this recursion: First, we split Node [0,15] (figure 5.1b), then we split Node [8,15] (figure 5.1c). At this point, we have only leaf nodes that are disjoint with the query (i.e., [0,7] and [8,11]) or are fully contained in the query (i.e., [12,15]).
- We set the *occupied bits* of all leaves that are contained by the query to *false*, indicating that the cold store has no data that matches these regions. In this example, we set the *occupied bit* of [12,15] to *false*. The *occupied bits* of the other leaves are initialized to *true* because we have no other information and we must be conservative and avoid *false negatives* for correctness. The end result (figure 5.1d) is an ARF that yields a true negative for all future queries with range contained in [12,15].

Again, the goal of this escalation process is to improve the precision of an ARF and reduce the number of false positives in the future. Putting it differently, we try to avoid making the same mistake twice. Escalation requires additional space. If space is limited, then we need to *de-escalate* the ARF, thereby gaining space and losing precision. Section 5.2 describes this process.

5.2 De-escalation (Merge)

The ultimate goal is to minimize the number of false positives with a given (small) space budget. If we shoot for perfection, then the ARF will grow and grow. In the worst case, the ARF can become as large as the cold store which

defeats their purpose as access filters. In this section, we present three approaches that can be used to make the ARF smaller whilst sacrificing as little accuracy and performance as possible.

5.2.1 Replacement policy that considers access patterns

We can often achieve very good results (few false positives) by exploiting skew in the workload. Recall that we expect the workload to be skewed in such a way that most queries and updates hit the same region or the same few records; e.g., orders that are due today or the names of customers that ordered today. The idea is to exploit this skew in the same way as a cache exploits skew: keeping statistics and using these statistics as part of a replacement policy.

Figure 5.2 demonstrates this principle using the running example and a *clock replacement policy*. We suggest using a clock policy as opposed to LRU or LRU-k because its statistics are small (i.e., a single bit as opposed to 64 bits for a pointer). Figure 5.2a shows the example ARF from figure 4.1; this time with a *used bit* for each leaf node. The *used bit* of a leaf is set whenever a query hits that leaf. As we will see, we generally only set this used bit when the leaf in question is empty. We call this process "learning from true negatives".

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 5.2a shows the *clock pointer* (visualized as a blue arrow) which happens to point 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 process goes as follows:

- Firstly, the clock strategy checks leaf [8,9]: Since its *used bit* is set, the clock strategy advances the clock pointer to the next leaf ([10,10] in this example) and unsets the *used bit* of [8,9], the leaf that it just visited.
- Next, the clock replacement policy selects [10,10] as a victim because its *used bit* is not set. Figure 5.2b 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 can be found.
- Finally, the clock pointer advances to the next leaf ([12,15]) as shown in figure 5.2b. The de-escalation can continue in the same manner and the clock pointer can be reset to the beginning of the ARF, if the ARF still exceeds the allocated space budget.

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 and can therefore be merged. In the example of figure 5.2b, nodes [8,9] and [10,11] have different values for their *occupied bits* so they cannot be merged at this point. If the *occupied bit* of Node [8,9] had been set to *true*, then these two nodes would have been able to be merged, too.

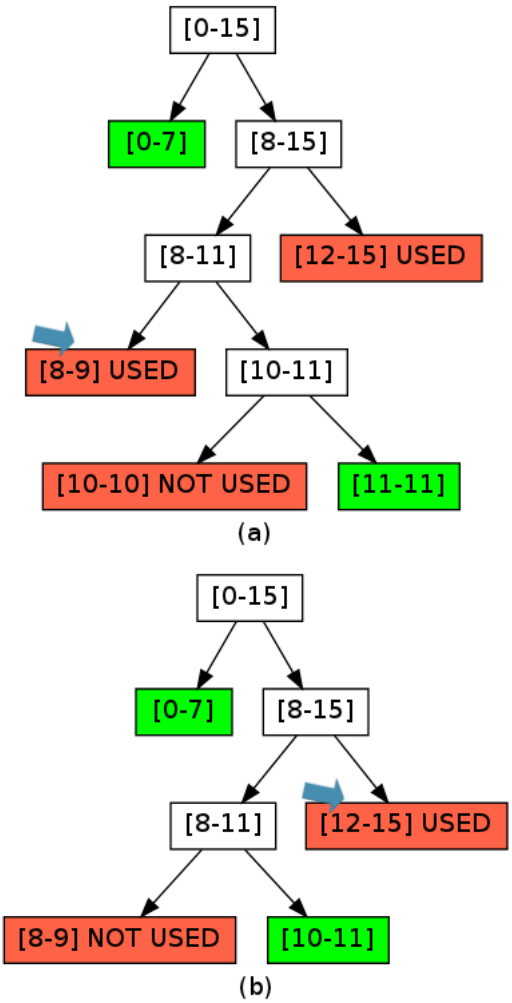

Figure 5.2: ARF Replacement

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

5.2.2 Round-robin replacement policy

A simple replacement policy which merely goes through the leaves in the ARF in navigation order and merges those with a sibling. Although not recommended for the training phase, because it effectively wastes all the hard-earned information about record access frequency, this policy is a good option during runtime for two reasons: Firstly, there is no need to invest bits for usage counters, which leaves the ARF more space to expand and more accurately capture the data. Secondly, the trimming down to fit the given budget takes less time that way.

5.2.3 Replacement policy that considers access patterns and locality

So far we've only taken record access frequency into account when designing a replacement policy. This, however, is only half of the picture. The ARF has been designed to not only take advantage of query skew, but also, data skew. If we want to design a replacement policy that considers data skew, we need to take into account the locality of each leaf. A leaf that has a small range is less important than a leaf higher up in the ARF, which covers a wider range of keys, because it contains less information. The strategy we designed works as follows:

- Step 1: Go through all the leaves that have a sibling and sort them according to their range in ascending order (use shuffle for leaves with equal ranges).
- Step 2: Go through the leaves in the sorted order and apply the rules of replacement described in section 5.2.1 to decide whether to merge a given leaf with its sibling. The only difference is that the clock pointer advances to the next leaf according to the sort order, and not their physical order in the ARF structure.
- Step 3: If the ARF now fits budget, stop. Otherwise, go back to Step 1 and recompute the necessary information on the new, trimmed down ARF.

One disadvantage of this policy is that it is more computationally intensive than its counterpart that only takes into account usage counters. The big advantage of this policy is that it mitigates the danger of evicting a leaf with information about a big range, only to end up storing information about a point query or a very small range. We experimented briefly with this approach and did notice some improvements in the replacement behavior, especially in the absence of workload skew, which, however were small. For this reason, we do not show these graphs in the next chapter, choosing to present the results of the simpler and faster approaches of sections 5.2.1 and 5.2.2.

5.3 Speeding-up Learning

There are a few techniques we used to aid with the escalation process, namely learning from the results of true positive queries and querying the cold store in a smarter way when we cannot avoid accessing it, in order to retrieve information

that is useful for the ARF. We implemented both these learning techniques for the prototype used in the experiments reported in chapter 6.

5.3.1 Learning from true positives

Section 5.1 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 it can be inferred that the gaps between two consecutive results are empty.
To given an example, let the result of Query [5,15] be: 7 and 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.

5.3.2 Learning from querying the cold store

Learning from such *true positives* speeds up learning dramatically and comes at virtually no cost. Another way to speed up learning whilst paying a small 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.

5.4 Handling 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 4.1, 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 or even escalating only up to a certain height in the trie. We did not experiment with

these variants and implemented the simple variant that sets the *occupied bit* to true because that variant worked reasonably well in all our experiments.
If a record is removed from the cold store, the ARF is not changed. In figure 4.1, 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 6.6, however, the escalation mechanism of section 5.1 that adapts an ARF after a *false positive* is quite effective. An ARF can adjust quickly to changes in the data distribution caused by excessive migration of records between the hot and cold stores. In fact, this fine-grained re-adaptation is one of the key advantages of ARFs as compared to Bloom filters. The only way a single large Bloom filter can adjust to such migrations is to be completely reconstructed, which involves a complete scan of the data in the cold store.

5.5 Training Phase

In addition to its ability to adapt to changes in the data distribution in a fine-grained way, one main advantage of ARFs over Bloom filters 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 5.1a 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 is also beneficial for ARFs. This way, the ARFs can perform well from the very beginning. Specifically, we propose to train ARFs in the following way if such a training phase is affordable:

- *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. Such a perfect ARF, for instance, would escalate Node [0,7] in figure 4.1 to capture that the cold store contains no keys that match 4 and 7. This ARF is likely to exceed the space budget, but that is tolerable in the training phase. (The ARF is trimmed to fit its space budget in the last step of this algorithm.)

- *Explore the Workload:* Run a series of example queries which are representative for the query workload. The purpose of this step is to make sure that the ARF learns the query/update distribution. As part of this step of the training phase, keep *usage counters* at all the leaves of the perfect ARF, indicating how often each leaf was involved in a query.

- *Meet Space Constraints:* Trim the ARF by iteratively selecting the leaf with the lowest *usage counter* as victim for replacement (as part of a de-escalation as described in section 5.2) until the ARF fits into the space budget. At this point, discard all the *usage counters* and keep more space-efficient usage statistics such as *used bits* for a clock strategy as proposed in section 5.2 or even no statistics at all.

Note that this training algorithm can be applied simultaneously to all ARFs of a table (e.g., ARFs that index the *status*, *ship_date*, etc. fields of an *Order*

table). Also, when such a training algorithm is used, the importance of learning from true positives, at least in the training phase is reduced, because the ARF starts off with perfect knowledge of the data distribution and the gaps between consecutive distinct values.

5.6 Summary and Variants

This chapter 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. Again, 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 several gaps.
Given the basic techniques presented in this chapter, 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 chapter 6. Here, we want to briefly sketch and summarize the most important dimensions of this design space:

- *To adapt or not to adapt?* Sometimes adaptation is 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 rarely needed if the ARF was trained using the algorithm of section 5.5 and there are no further updates or significant workload changes.

- *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 5.2.1 because it is space-efficient and easy to implement. In the next chapter, we will show that a *round-robin* replacement policy as described in 5.2.2 that maintains no statistics can do slightly better if the workload is not skewed.

- *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 training algorithm of section 5.5 whenever the statistics indicate that it is worth doing so.

Chapter 6

Experiments and Results

This chapter presents the results of extensive experiments that we conducted with a prototype implementation of ARFs and a synthetic benchmark which models a hot/cold data partitioning as the one described in chapter 2 on Project Siberia. The experiments study the precision, space and time efficiency, robustness, and graceful degradation of several ARF variants for various different workloads and database configurations. We use a simple but reasonably optimized implementation of the Bloom filter to use as a baseline for our results. Specifically, this section presents the results of the following three ARF variants:

- *ARF no-adapt:* The ARF is trained once (as described in section 5.5) and then never changed.
- *ARF adapt-1bit:* The ARF is trained at the beginning (as described in section 5.5) and then adapted with every false positive as described in section 5.1. After every escalation, the ARF is de-escalated to fit its space budget as described in section 5.2, thereby using a clock replacement policy with one *used bit*.
- *ARF adapt-0bit:* Same as "ARF adapt-1bit" with the only difference that we use a clock strategy without any *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 4.3, 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 6.5.

6.1 Benchmark Environment

To study ARFs, we ran a series of synthetic workloads using the three ARF variants and Bloom filters. Table 6.1 lists the parameters of our benchmark database and query/update workload.

Data generation

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 around the beginning of the domain. 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 varied the number of distinct keys of records in the cold store from 1000 to 10,000. If not stated otherwise, this value was set to 1000 distinct keys; for shipping dates, 1000 distinct keys represent roughly three years worth of orders. We also varied the sizes of the filters from 1000 to 10000 bits, because this is the space investment that we are interested in (1 to 10 bits per distinct key in the cold store).

Workload generation

We studied a number of different 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, a *random Zipf* distribution or a *Uniform* distribution. With a *Zipf* distribution, most of the queries were at a similar *location* (e.g., asking for orders which are due today or tomorrow). This distribution aimed to model *value-correlated skew* as defined in Section 2.4.2. With a *random Zipf* distribution, most of the queries targeted a specific set of regions which did not, in total, cover the whole domain, but they were also not clustered around one single location. This distribution aimed to model *natural skew* as defined in section 2.4.2. Experiments that use a *random Zipf* distribution for the queries are clearly marked.
With a *Uniform* distribution, the queries were evenly distributed. With a *Zipf* database and a *Zipf* query distribution, the query location was different from the location of the data. We expect this situation to be the typical scenario for Project Siberia: Using the "order" example again, most of the orders in the cold store would be old orders and most of the queries ask for present or future orders. The Zipf factor for both data and queries, unless otherwise specified is 1.2.
The second parameter that determined the query workload is the mean range size of range of queries; i.e., $r - l$. This parameter is referred to as μ in Table 6.1. 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 in the range of $\mu = 1$ to $\mu = 300$. The standard deviation was set to $\sigma = \mu/3$.

Update generation

We also studied how quickly ARFs adapt to migrations between the hot and cold store. In those 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

Database Parameters	
Distribution	Zipf or Uniform
Distinct Keys	Vary
Domain Size	2^{20}
Query Workload	
Location (l)	Zipf, Random Zipf or Uniform
μ, Mean Range Size ($r - l$)	Point ($\mu = 0$) or Range ($\mu = 1 \dots 300$)
Updates	
Deletes	Uniform
Inserts	Zipf or Uniform
Number of Updates	Vary
Range Filter Parameters	
Size	1000 to 10000 bits

Table 6.1: Benchmark Parameters

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 6.6. Furthermore, we studied how ARFs adapt to changes in the workload in section 6.7.

Other parameters

All experiments were carried out in the same way. First, the ARFs were trained using the algorithm of section 5.5. After that, we ran 20,000 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.

6.2 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. In the interest of the integrity of timing measurements we ran the benchmarks 20 times and we also ensured that the program did not switch processors mid-execution.
We implemented ARFs as described in Chapters 4 and 5. We implemented Bloom filters using multiplicative hash functions, as proposed in [20]. 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 common results on classic Bloom filters.
There has been some previous work on special kinds of Bloom filters for range queries [21]. Unfortunately, that work is not competitive because it requires a large amount of space per key stored in the cold store and also, it does not avoid checking multiple points for membership. So, there really is no good way to compare ARFs with Bloom filters for range queries. To have nevertheless a

baseline and get an impression for the effectiveness of ARFs for range queries, we used Bloom filters for range queries, too, thereby probing each value of a query range individually with the Bloom filter. For Query [10,13], for instance, we would probe keys 10, 11, 12, and 13 separately using the Bloom filter, thereby making use of short circuiting.

6.3 Experiment 1: False Positives

6.3.1 Point Queries

Figures 6.1 to 6.4 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 1 KBits (i.e., 1 bit per distinct key in the cold store) to 10 KBits (i.e., 10 bits per distinct key in the cold store. As a rule of thumb, Bloom filters are at their best for 8 bits per key and figures 6.1 to 6.4 confirm this result.
The results presented in figures 6.1 to 6.4 give insight into three important requirements of filters: 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 a positive sign if the false-positive rate decreases for a Zipf data or query distribution, because that signifies that the ARF can benefit from skew.

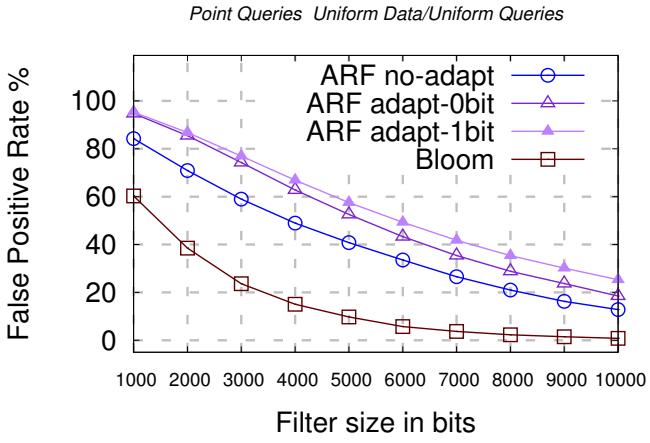

Figure 6.1: Precision: Uniform DB, Uniform Workload
Point Queries, No Updates, Vary Size, 1000 Distinct Keys

Turning to figure 6.1, 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 for and have been proven successfully in the forty years since their invention. The precision of the ARF

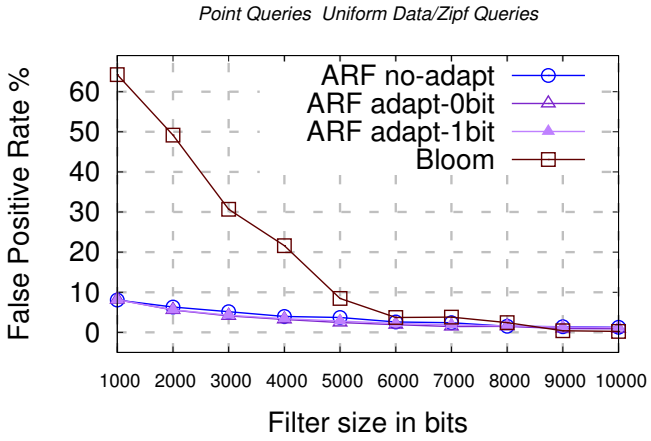

Figure 6.2: Precision: Uniform DB, Zipf Workload
Point Queries, No Updates, Vary Size, 1000 Distinct Keys

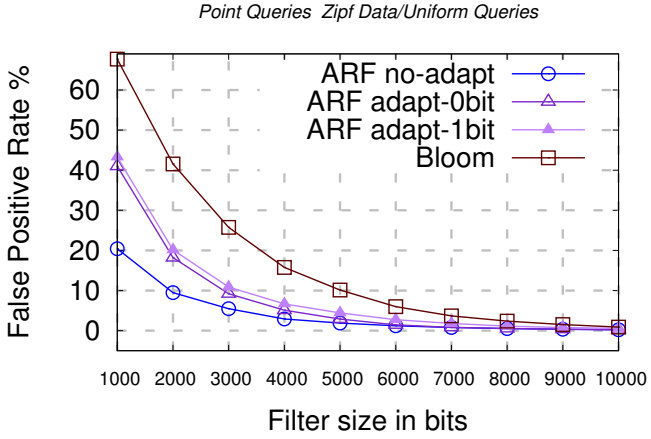

Figure 6.3: Precision: Zipf DB, Uniform Workload
Point Queries, No Updates, Vary Size, 1000 Distinct Keys

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 6.1, 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 the worst variant in this experiment. However, the differences are not big. 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 can actually be harmful here with the replacement policy used (See section

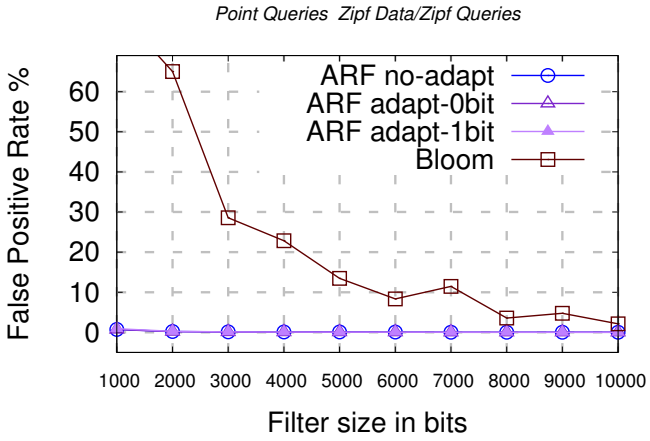

Figure 6.4: Precision: Zipf DB, Zipf Workload
Point Queries, No Updates, Vary Size, 1000 Distinct Keys

5.2.1): Escalating the ARF to handle one false positive in a small region might result in the de-escalation of a large region. However, the effects are fairly small and all three ARF variants perform equally bad. The conclusion is that ARFs are not good for workloads with only point queries and no skew: For such workloads, Bloom filters are clearly the best choice.
Turning to figures 6.2 to 6.4, it can be seen that all three ARF variants benefit significantly from skew in the data or query distribution. In contrast, the results for Bloom filters are independent of any kind of skew. Again, we would like to reiterate that we expect heavy skew in Project Siberia: That is what the architecture of Project Siberia was designed for in the first place. A somewhat surprising result is that ARFs show almost perfect precision for small filter sizes of 3 bits per key. If both the data and the workload are skewed, ARFs work extremely well even with filter sizes around 1 bit per key. Again, space efficiency is critical for Project Siberia because the filters must be maintained in the (expensive) hot store.

6.3.2 Range Queries

Figures 6.5, 6.6, 6.7 and 6.8 show the results of the four combinations of data and query distributions for range queries with a mean range size of $\mu = 30$ ($\sigma = 10$). In our shipping data / 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 positive result or until all keys of the range had been probed (section 6.2).
It is perhaps apparent by comparing the graphs for point queries and range queries that the ARF results for range queries show no significant difference with the ones for point queries. The observations in the previous section about skew exploitation and about the relative accuracy of the three ARF variants apply here as well. Even if we vary the mean range size of μ from 1 up to 300, as shown in figure 6.9, the ARFs sustain their accuracy whereas the Bloom

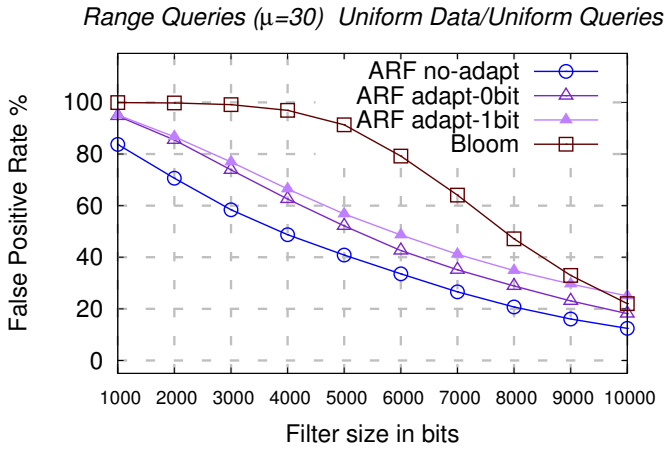

Figure 6.5: Precision: Uniform DB, Uniform Workload
Range Queries, No Updates, Vary Size, 1000 Distinct Keys

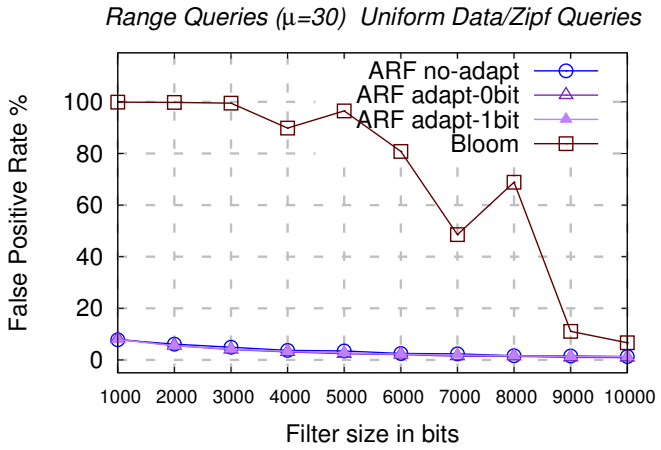

Figure 6.6: Precision: Uniform DB, Zipf Workload
Range Queries, No Updates, Vary Size, 1000 Distinct Keys

filters deteriorate in the expected way.

6.3.3 Random Zipf workload

The Zipf distribution we've used so far where only a specific location in the domain is queried often does not cover all the cases of skewed access to data. Some times, some specific regions are queried more often than others: In an online store, some products are more popular than others, while being unrelated to each other. In our example order table, maybe specific days scattered throughout the year (holidays) or specific timeframes (e.g., Christmas and Easter) are of interest because of special buyer patterns during those days (although a query pattern like this would more likely be part of a OLAP workload). In order to

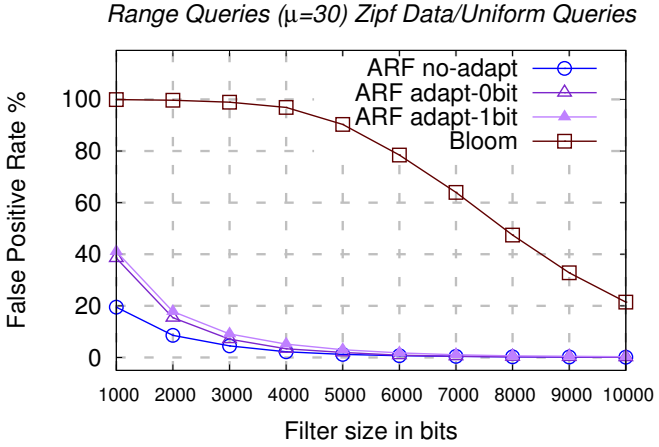

Figure 6.7: Precision: Zipf DB, Uniform Workload
Range Queries, No Updates, Vary Size, 1000 Distinct Keys

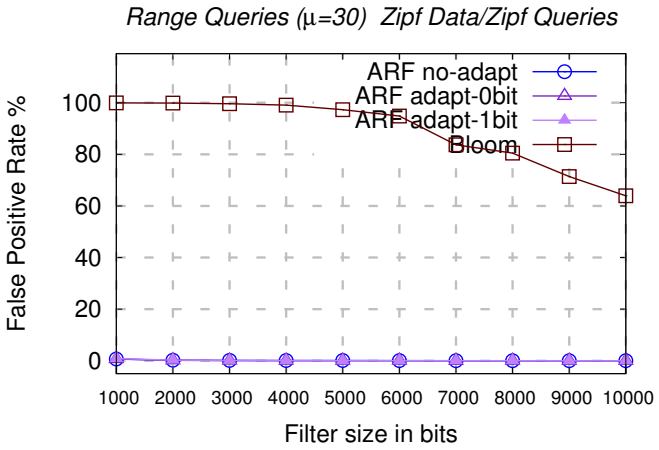

Figure 6.8: Precision: Zipf DB, Zipf Workload
Range Queries, No Updates, Vary Size, 1000 Distinct Keys

model situations like this, we used a *random Zipf* distribution.

Figures 6.10 and 6.11 show the results for a Uniform database and random Zipf queries for point queries and range queries($\mu = 30$) respectively. In the case of point queries, the ARF data structure is a bit worse than the Bloom filters, even though there is skew in the query distribution. Therefore we can deduce that the plain Zipf workload where the queries target a specific region is more favorable towards the ARF than the random Zipf workload which targets specific points scattered throughout the domain, although in both cases the ARF exploits the skew to some extent. If we increase the Zipf factor s from 1.2 to 1.4 (figure 6.12), effectively decreasing the number of distinct regions/points that

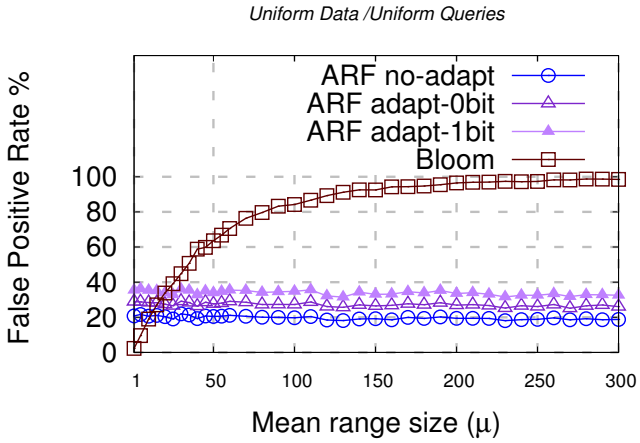

Figure 6.9: Accuracy: Uniform DB, Uniform Workload
Vary μ, No Updates, 8KBits, 1000 Distinct Keys

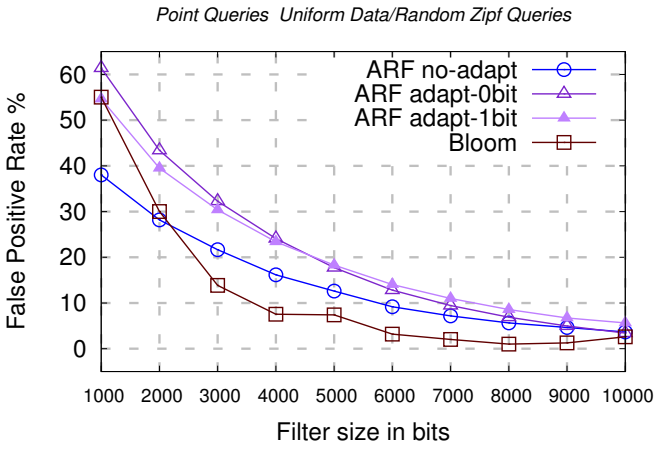

Figure 6.10: Precision: Uniform DB, Random Zipf Workload ($s = 1.2$)
Point Queries, No Updates, Vary Size, 1000 Distinct Keys

are queried often, we can see that the ARF becomes a more competitive choice for point queries, because the space budget can accomodate a more accurate storing of information about those regions. Also, we can observe that the adaptive ARF which stores usage bits has a small advantage over the one that uses round-robin replacement, which means that the space investment for statistics pays off in this case.

We do not show results for a Zipf database, random Zipf workload set-up because there is no significant difference to the Zipf database/Zipf queries set-up.

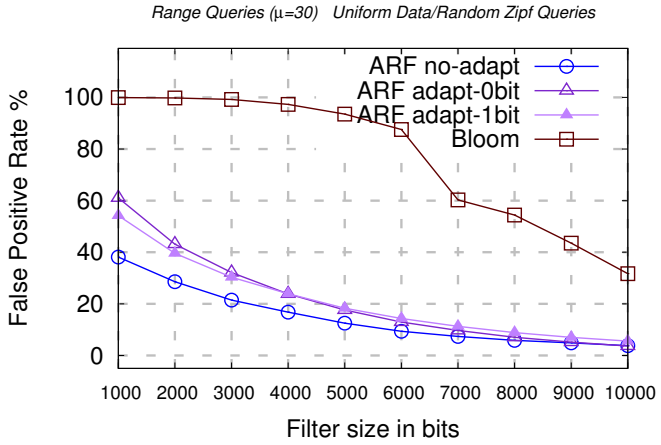

Figure 6.11: Precision: Uniform DB, Random Zipf Workload ($s = 1.2$)
Range Queries, No Updates, Vary Size, 1000 Distinct Keys

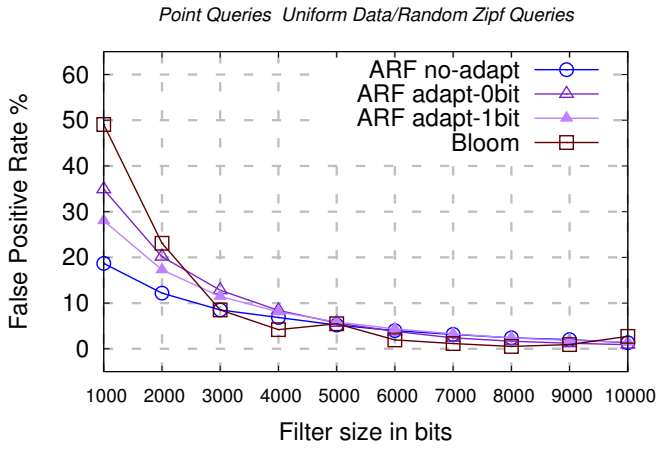

Figure 6.12: Precision: Uniform DB, Random Zipf Workload ($s = 1.4$)
Point Queries, No Updates, Vary Size, 1000 Distinct Keys

6.3.4 Summary of results

From this first round of experiments we can infer that the ARF data structure can be competitive with classic Bloom filters in the presence of skew in the data and/or query workload, including *value-correlated skew* and *natural skew* in queries. It can sustain the same level of accuracy for range queries, which is the point where Bloom filters deteriorate significantly.

6.4 Experiment 2: Scale Database

So far we've used databases with 1000 distinct keys. One could argue that this is quite a small-scale set-up, so, for this reason, we've run experiments to demonstrate that the number of the distinct keys does not matter, as long as the "bits per key" ratio is kept constant and also, show what happens while this ratio changes. We increase the size of the database, while keeping either the filter size or the "bits per key" ratio constant.

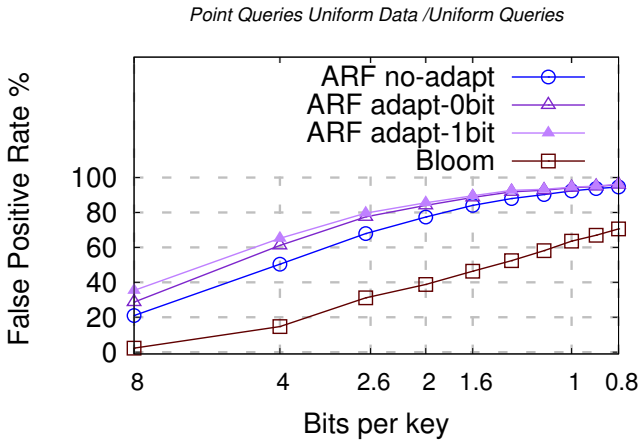

Figure 6.13: Accuracy: Uniform DB, Uniform Workload
Point Queries, No Updates, 8 KBits, Vary Distinct Keys

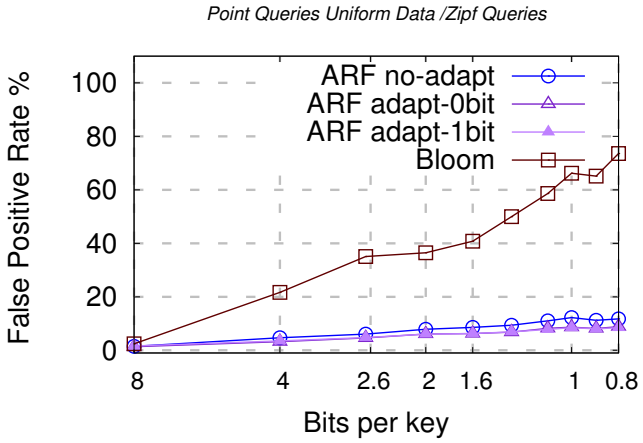

Figure 6.14: Accuracy: Uniform DB, Zipf Workload
Point Queries, No Updates, 8 KBits, Vary Distinct Keys

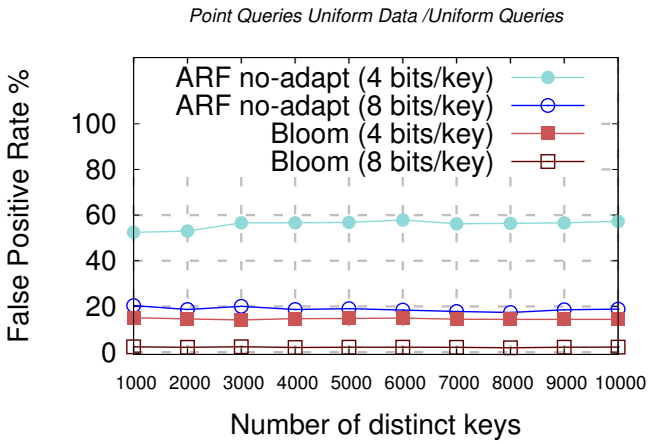

Figure 6.15: Accuracy: Uniform DB, Uniform Workload
Point Queries, No Updates, Vary Filter Size, Vary Distinct Keys

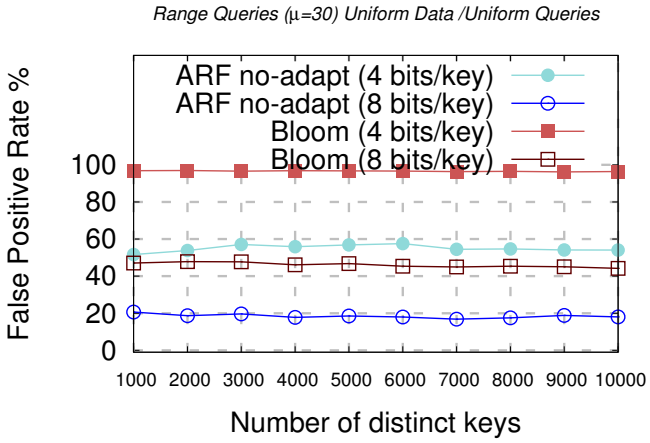

Figure 6.16: Accuracy: Uniform DB, Uniform Workload
Range Queries, No Updates, Vary Filter Size, Vary Distinct Keys

6.4.1 Constant filter size

Figures 6.13 and 6.14 show the accuracy of ARFs and Bloom filters with a constant size of 8000 bits while varying the number of distinct keys from 1000 to 10,000 and therefore changing the "bits per key" ratio from 8 to 0.8. Only point queries are shown - the results for the respective range queries differ in that the ARFs have the same performance whereas Bloom filters become significantly worse.
Not surprisingly and confirming previous studies with Bloom filters, the accuracy got worse (i.e., increasing false-positive rate) with a growing number of distinct keys in the cold store as the "bits per key" ratio dropped. Figure 6.13 shows the results for a Uniform database and Uniform workload; figure 6.14

shows the results for a Uniform database and a Zipf workload. The results for Bloom filters were almost the same because the accuracy of Bloom filters does not depend on skew.
The ARFs also decline in performance when less bits are used per key. However, in the presence of skew, the decline is less significant and good accuracy can be achieved even with around less than 1 bit per key.

6.4.2 Constant bits per key

Figure 6.15 shows the accuracy of Bloom filters when varying the number of distinct keys in the cold store (for point queries). That is, we scaled the filter size with the number of distinct keys. Again, confirming all known results of Bloom filters with respect to point queries, the false-positive rate is almost constant in this scenario. If the "bits per key" ratio was set to 8, the Bloom filter was almost perfect; with a "bits per key" ratio of 4, Bloom filters had a false-positive rate of about 20 percent.
In the same figure, all ARF variants had constant accuracy if the "bits per key" ratio did not change; this observation was true for this scenario and all other scenarios (i.e., range queries and/or skew in the data and workload distribution). We only show the Uniform/Uniform case for simplicity, since it also constitues a worst-case for the ARF. Figure 6.16 shows the results for range queries with $\mu = 30$, where 8 bits/key for the Bloom filters are not enough to ensure "almost perfect" accuracy.
We also ran some experiments (not shown) where we scaled the cold store up to $50,000$ keys. The ARFs and Bloom filters continue to have the same performance. After the $50,000$ keys mark, the cold store becomes quite densely populated, most range queries become true positives and the importance of filtering is reduced.

6.4.3 Summary of results

Predictability and robustness is one of the strengths of Bloom filters. It turns out that ARFs have the same features, except they also work with range queries. Just like Bloom filters, ARFs get worse if the "bits per key" ratio drops. This effect is shown in figures 6.13 and 6.14.
Figures 6.15 and 6.16 show that, when the database increases in size, increasing the filter size so as to keep the "bits per key" ratio constant preserves almost identical levels of accuracy in the ARF, which is the same behavior as Bloom filters.

6.5 Experiment 3: Latency

In addition to *accuracy, 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.

6.5.1 Lookup Times

Table 6.2 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 4.1 and as used in

	Point Queries ($\mu = 0$)		*Range Queries ($\mu = 30$)*	
	1Kbits	*3Kbits*	*1Kbits*	*3Kbits*
ARF-1	900 (± 362)	2536 (± 1004)	1500 (± 1360)	3798 (± 3605)
ARF-64	137 (± 180)	200 (± 100)	478 (± 603)	670 (± 900)
Bloom	178 (± 13)	219 (± 52)	273 (± 300)	696 (± 300)

Table 6.2: Lookup Time (Cycles): Unif. DB, Unif. Workl.
Vary Queries, No Updates, Vary Size, 1000 Distinct Keys

all other experiments reported in this chapter (referred to as ARF-1), and (b) a forest of 64 ARFs such as the one shown in figure 4.2 (referred to as ARF-64). Lookup times do not vary significantly between the different ARF variants, so we only present the results for "ARF no-adapt" here. All ARFs are encoded in BFS-order unless otherwise specified. Table 6.2 reports on the mean lookup times (and standard deviation) in cycles measured over $20,000$ queries for a Uniform database and a Uniform workload. On a machine with cores clocked at 2.66 GHz, 2666 cycles take 1 μsec.
Looking at Table 6.2, the following observations can be made:

- 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 did not linearly increase with the size of the range because of short circuiting (ie, a false or true hit was found relatively fast when looking at all the points in a range).

- ARFs were more expensive and had a higher standard deviation because they are not balanced trees. Queries that hit the leftmost part of the filter were much quicker to answer than queries that hit the rightmost part. Furthermore, the ARFs were more sensitive to the filter sizes, although they also scaled sublinearly with the length of the range. With the given filter sizes, they were competitive and it never took more than a few μsecs to carry out a lookup.

- 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. In other measurements we carried out with larger filter sizes (because of more distinct keys), the difference between the Bloom filter lookup time and the ARF lookup time, predictably, grew larger. Therefore, in the context of a practical implementation, B-tree integration is necessary for good performance.

Theoretically, the complexity of a Bloom filter lookup for a point query is $O(k)$ where k is the number of hash functions whereas the complexity of a ARF lookup is $O(N)$ where N is the size of the filter. Even though a lookup is basically a trie navigation, the lack of pointers prevents it from being $O(logN)$. The reason the differences in lookup time for small filters are not as big as theoretically

expected, has to do with access patterns and the fact that not all queries need to go to the most far right part of the trie to be answered. The ARF lookup is done serially throuh the data structure, whereas the Bloom filter requires random access.

Lookup Times for DFS encoding

As mentioned in chapter 4, a DFS-encoding of the ARF is possible, which may be more suited for B-tree integration. Figure 6.17 shows the clock cycles needed for lookups in a BFS- and DFS-encoded ARF. As we've already mentioned in section 4.4.1, the lookup time differences for small filters (smaller than 500 bits) are not significant, making the DFS-encoded ARF a good candidate for B-tree integration.

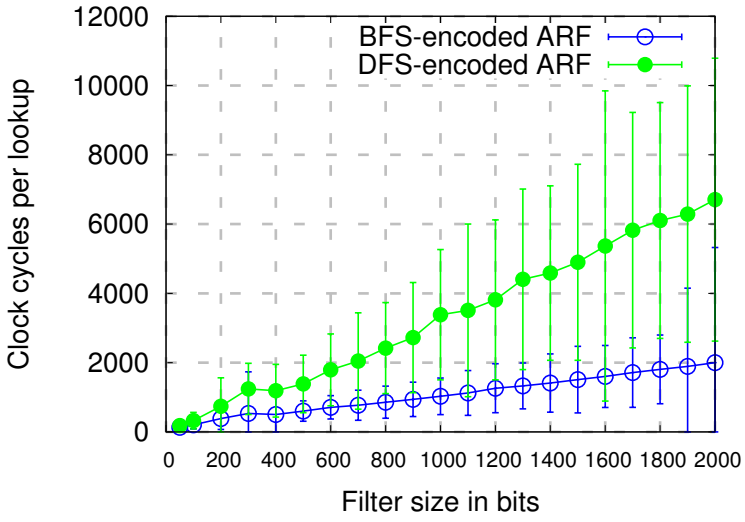

Figure 6.17: Comparison of lookup times of different ARF encodings

Scaling with the length of range

With respect to scaling with the length of the range, both the Bloom filter and the ARF scale sublinearly. This has to do with the following:

- *Short-circuiting* in both data structures, where a hit is found before the range is exhausted. This hit may be a real one or may be a false positive.
- In the case of the ARF, we don't need to look at as many leaves as the length of the range in the general case and also, re-navigating the data structure is optimized as described in Appendix A.3.

6.5.2 Maintenance Times

Table 6.3 shows the mean clock cycles (and standard deviation) needed to escalate and de-escalate an ARF for the "ARF adapt-0bit" variant. These operations

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

Table 6.3: Maintenance Time (Cycles): Unif. DB, Unif. Workl.
Point Queries, No Updates, Vary Size, 1000 Distinct Keys

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.

6.5.3 Summary of results

The ARF is measured to be in the same ballpark as the Bloom filter for looking up point queries when the filters are reasonably small, and scales sublinearly with the length of the range in range queries. Maintenance takes significantly more time to be carried out. In the context of a practical implementation it is advisable to integrate the ARF in a B-Tree to ensure that only filters in the magnitude of a few hundred of bits are used. In that case, a DFS encoding is also a practical alternative.

6.6 Experiment 4: Updates

This next set of experiments studies how the alternative ARF variants adapt to changes in the data distribution; e.g., the migration of records between the hot and cold store. We carried out these experiments in the following way:

1. Create an initial cold database and train the initial 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, depending on the ARF variant. Again, this step is just as in all previous experiments.
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.

Figures 6.18 and 6.19 show the false-positive rate as a function of the number of updates (i.e., iterations of the loop above) with a Uniform workload, for point queries and range queries respectively. In this case, the original database containing 1000 keys was created using a Uniform distribution and the keys of all *Inserts* in Step 4 were also generated using a Uniform distribution. Likewise,

figures 6.20 and 6.21 show the false-positive rate for point and range queries if the data and update distribution is Zipf and the query workload is Uniform. All 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 *Deletes*.

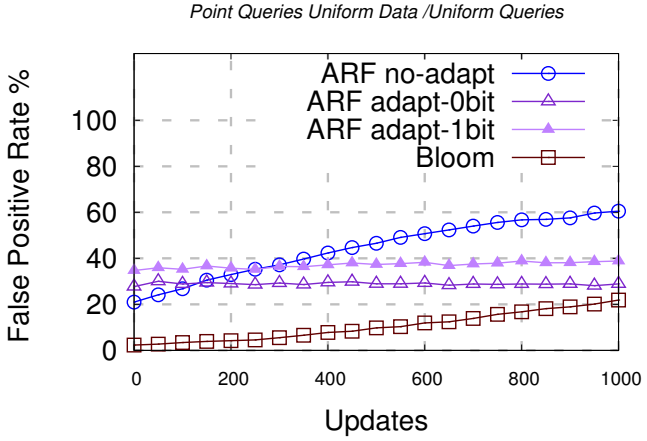

Figure 6.18: Precision: Uniform DB, Uniform Workload
Point Queries, Vary Updates, 8 Bits/Key, 1000 Distinct Keys

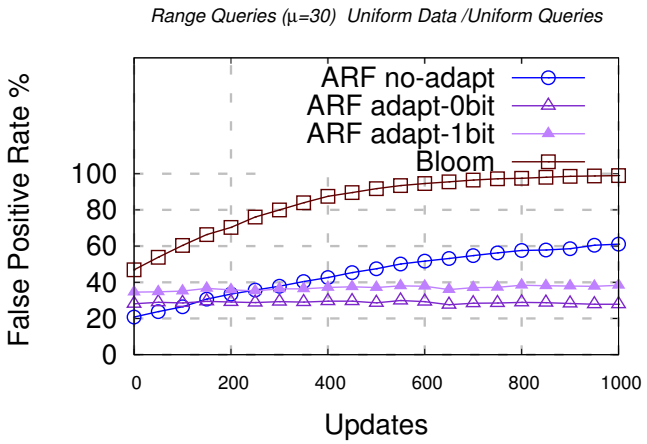

Figure 6.19: Precision: Uniform DB, Uniform Workload
Range Queries, Vary Updates, 8 Bits/Key, 1000 Distinct Keys

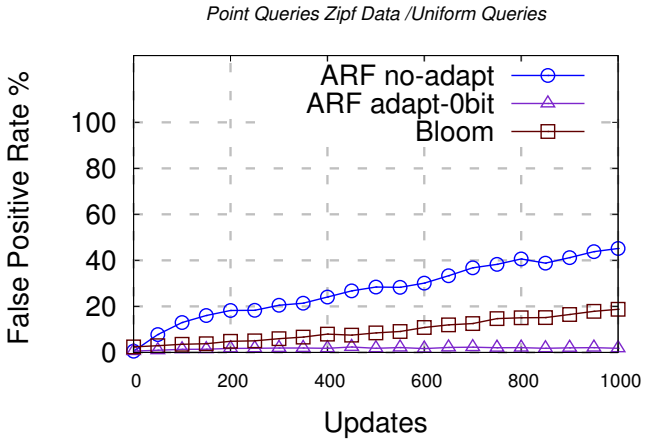

Figure 6.20: Precision: Zipf DB, Uniform Workload
Point Queries, Vary Updates, 8 Bits/Key, 1000 Distinct Keys

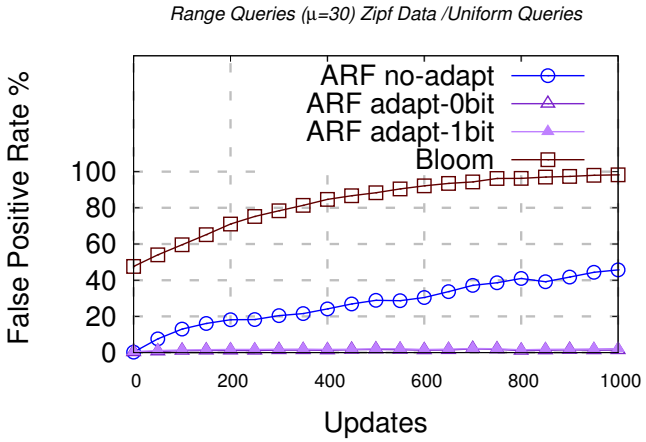

Figure 6.21: Precision: Zipf DB, Uniform Workload
Range Queries, Vary Updates, 8 Bits/Key, 1000 Distinct Keys

Comparing figures 6.18 and 6.20, we can again see that Bloom filters won if there was no skew (neither in the data nor workload) and only point queries were carried out. Even more updates would be necessary for the Bloom filter to deteriorate to the point of being worse than the ARF. In all other scenarios (range queries or skew in either the data or query distribution), ARFs are competitive in terms of accuracy. As shown in figure 6.20, however, it is possible that a non-adaptive ARF deteriorates faster than a Bloom filter with the number of updates. Handling this issue with a more involved way to mark inserts than the one currently used (see section 5.4) is left for future work.

6.7 Experiment 5: Workload changes

We also investigated the behavior of the ARF when the workload changes. A meaningful scenario is one where the data distribution is Uniform and the query workload is skewed and, at some point in time, the center of the skew changes so that different parts of the data are being accessed. We did experiments both with a Zipf query workload and a random Zipf in order to show that the adaptive ARF variants can handle a workload change very well without requiring re-training.

6.7.1 Zipf workload

In this experiment, we used a Zipf workload. The workload in the beginning targeted the area towards the end of the domain and, after the change, it targeted the area towards the beginning. Figures 6.22 and 6.23 show the accuracy results as a function of time for point queries and range queries respectively. The workload switch happens after 20000 queries.
The non-adaptive ARF, having been trained with a different workload in mind, has less accurate information about the area that starts getting queried after the workload change and therefore deteriorates in performance. On the other hand, the adaptive variants acquire information about the new area queried relatively quickly, and as can be seen more clearly in figure 6.22, return to their previous accuracy percentages after less than 3000 queries. The Bloom filter is unaffected by the workload change.

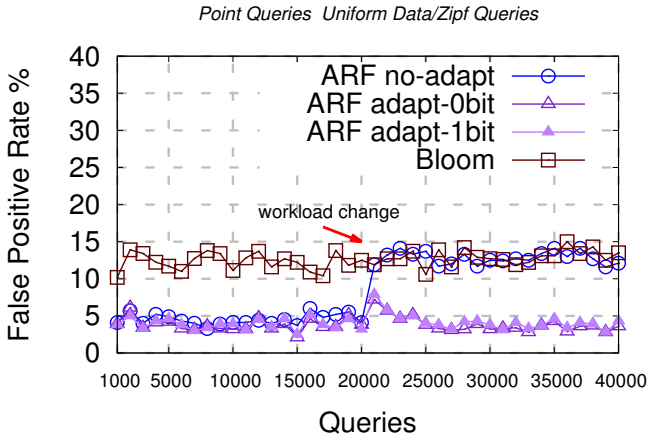

Figure 6.22: Precision: Uniform DB, Zipf Workload
Point Queries, Vary Workload, 3 Bits/Key, 1000 Distinct Keys

6.7.2 Random Zipf workload

In this experiment, we used a random Zipf workload. The workload change consisted of completely changing the set of frequently queried points/regions after 20000 queries. Again, as shown in figures 6.24 and 6.25, the non-adaptive

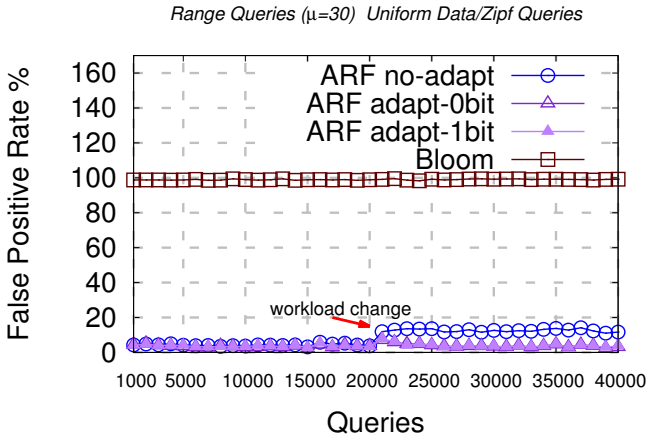

Figure 6.23: Precision: Uniform DB, Zipf Workload
Range Queries, Vary Workload, 3 Bits/Key, 1000 Distinct Keys

ARF is unable to recover, whereas the adaptive variants recover quickly and the Bloom filter is unaffected. It is worth noting that, in this case, the effects of the workload change are more drastic than when using a Zipf workload.

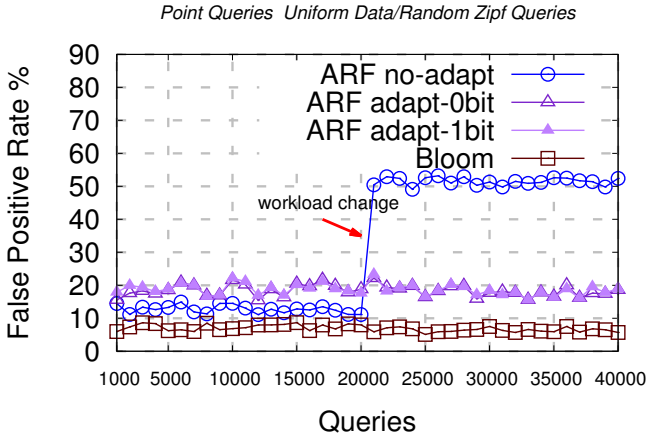

Figure 6.24: Precision: Uniform DB, Random Zipf Workload
Point Queries, Vary Workload, 5 Bits/Key, 1000 Distinct Keys

6.8 Summary of results

In this chapter we showed how the ARF performs within a variety of scenarios. It can be competitive with classic Bloom filters for point queries in the presence of skew, and is the only good solution so far when it comes to range queries within the defined space requirements. The ARF variants that continue adapt-

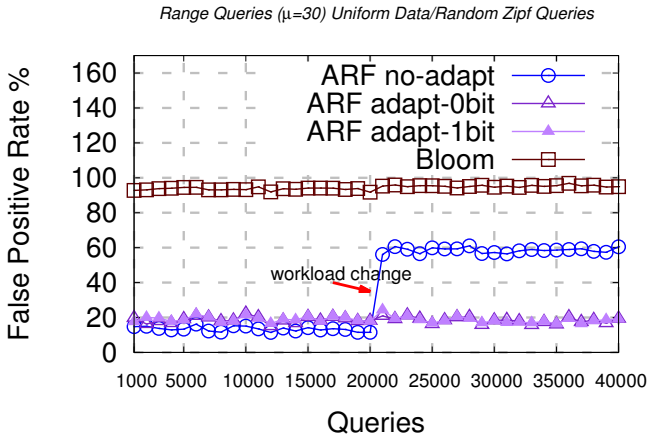

Figure 6.25: Precision: Uniform DB, Random Zipf Workload
Range Queries, Vary Workload, 5 Bits/Key, 1000 Distinct Keys

ing after training do not seem to have an advantage over the "frozen" ARF when no data or workload change is taking place.
Additionally, we showed that the ARF is a robust data structure that scales perfectly with the number of distinct keys, as long as the "bits per key" ratio stays constant. The lookup time it requires, although generally more than the Bloom filter, can still be affordable in practical implementations. A B-tree integration is very promising with respect to keeping down the lookup and maintenance time as well.
We also demonstrated that the ARF can learn from updates on-the-fly, something that Bloom filters can't do without being recomputed, and finally, that it can adapt to a change in the query workload without any re-training. Overall, keeping statistics after the training phase (ARF adapt-1bit) typically does not pay off - the ARF adapt-0bit that uses a round-robin replacement policy performs either slightly better or just as well in most of the usage scenarios and has the added advantage of being simpler and supporting faster merging.

Chapter 7

Multi-dimensional ARFs

The ARF data structure presented in the previous chapters covers the case of filtering range queries on one attribute of a relation. If we wish to use the filter as a guide in even more types of queries, that is, queries involving two or more attributes, we need to extend the ideas of the ARF to be able to support multi-dimensional data. The main approaches of handling multi-dimensional data according to the survey section of [22] are:

- Methods that use *space filling curves*, to map a k-d space onto a l-d space. It was quite straightforward to use this method in the context of ARFs.

- Methods that partition the original k-d space in pre-determined ways. These approaches include k-d trees and quadtrees, the ideas of which we used.

- Methods that divide the original k-d space into appropriate sub-regions (overlapping or dis- joint) by employing some custom algorithm to find the optimal partitioning. This approach, which is the core concept behind the well-studied R-trees[23] and R+-trees[22], is however incompatible with the concepts of the ARF. The space-efficiency of the ARF lies in the fact that the division into sub-regions is always pre-defined in some way. Using an approach like this would require explicit storage of the bounds of each sub-region, so, in the interest of maintaining the space-efficiency property of our data structure, we did not pursue this approach further.

- There is also the trivial solution of having several one-dimensional indexes (one for each attribute of interest) and returning their aggregate result (not a solution mentioned in [22]). We briefly examined this approach and found out that it's very inaccurate, even if a lot of space is invested for the one-dimensional indexes, so we did not consider this approach further. However, if one does not wish to invest space for a multi-dimensional index and the relevant one-dimensional indexes already exist, the trivial solution is better than nothing.

In this chapter, we present the data structures we developed using the approaches above and the results that emerged from their performance evaluation.

7.1 Space-filling curves

A "space-filling curve" is formally defined as a continuous, surjective mapping from the one-dimensional space onto a higher-dimensional space, more formally, from the unit interval $[0, 1]$ to the k-dimensional unit hypercube $[0, 1]^k$. The space-filling curve in the pure sense, which actually passes from every point in the unit hypercube, is in fact the limit of approximating curves whose length is bounded but continues to grow with each approximation. The curves start off with a very basic shape and then grow with each subsequent approximation. Figure 7.1 shows the first 6 approximations of the Hilbert curve in 2D, which we will discuss in more detail later. Because the mapping cannot be both continuous and bijective (as proven in[24]), the curve at the limit is always self-intersecting, even if the approximating curves are not. We are not interested in limit conditions and are happy to use approximate curves that can offer a bijective mapping between points with integer coordinates in k-d to 1-d, so that there is no ambiguity. That is, every point in the high-dimensional corresponds to a sequence number on the (approximate) curve and vice versa. We will not distinguish between the space-filling curve at the limit and its chosen approximation from now on.

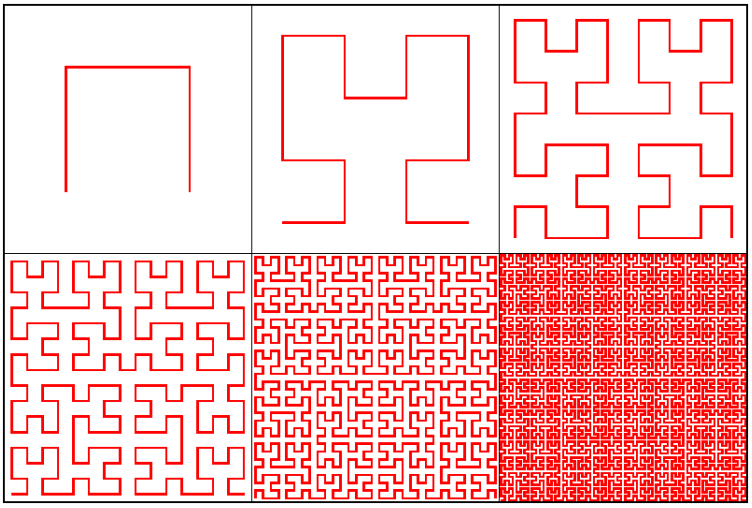

Figure 7.1: Approximations of the Hilbert Curve in two dimensions. Source: *Wikipedia*

Generally, while a multi-dimensional point maps to one sequence number on the space-filling curve, a multi-dimensional range maps to one or more ranges of sequence numbers on the curve. In the context of the ARF, that means that, in order to answer a multi-dimensional range query, we need to look at possibly several ranges on the one-dimensional space-filling curve.

Our work with space-filling curves consists of:

- implementing the mappings from multi-dimensional points to sequence

numbers and vice versa

- implementing the mappings from multi-dimensional ranges to a list of ranges of sequence numbers
- querying an underlying regular, one-dimensional ARF which contains existence information about the sequence numbers of the space-filling curve. The domain d of the ARF is the domain of the composite key on the attributes we want to filter. The domain is generally a power of 2: $d = 2^m$
- this curve is actually the $(m-1)^{\text{th}}$ order approximation of the space-filling curve at the limit. This ensures that there is a bijective mapping for all points in k-d to 1-d.

We used three types of space-filling curves.

- The **C Curve**, or more plainly, composite keys. While its mappings are very simple to compute by merely concatenating the values of each dimension, the C Curve does not cluster data well, i.e. data points that are adjacent in the multi-dimensional space may be very far apart in their respective C Curve sequence numbers.
- The **Z Curve** or Morton codes. A multi-dimesional point can be converted to a Z Curve sequence number by interleaving the bits of each dimension. The efficient mapping of a multi-dimensional range to ranges on the Z Curve is more involved. We did our own implementation of this mapping according to the description of the relevant algorithms in [25]. While the Z Curve does not offer perfect clustering (mostly because of the jumps that are evident in its shape in figure 7.2), it can still be an adequate choice for most indexing purposes - for instance, the Z Curve is the underlying space-filling curve in UB-Trees[26].

  Figure 7.2 shows a very small example of an ARF using a Z Curve to filter two-dimensional data. The cold store of the database contains points $[2,1]$,$[2,4]$,$[3,1]$, $[4,0]$,$[4,5]$,$[5,0]$ which are visualized as green points. Both dimensions can take values from 0 to 7. The underlying ARF has domain $[0,63]$. The leaves that are green in the underlying ARF are the ones containing the values $(6,7,16,17,36,50)$, i.e. the values the two-dimensional points map to.
- The **Hilbert Curve** (as illustrated in figure 7.1). The Hilbert Curve, first described by the German mathematician David Hilbert in 1891, clusters data better than the Z Curve[27], but its methods are quite complex and computationally intensive. We used an existing implementation[28] for the mappings.

There are some more types of space-filling curves that have been used in the context of research for indexing multi-dimensional data, such as several variants of the Gray-Curve, the Snake Curve and others[29]. However, according to the literature[27], the Hilbert curve is regarded as the curve with the best clustering properties and the Z Curve is a simpler and computationally less intensive yet almost as good alternative, so we saw no significant benefit in inspecting more cases in between. We used the C Curve as a baseline, in order to show that good clustering properties matter in the context of our use case, i.e range queries.

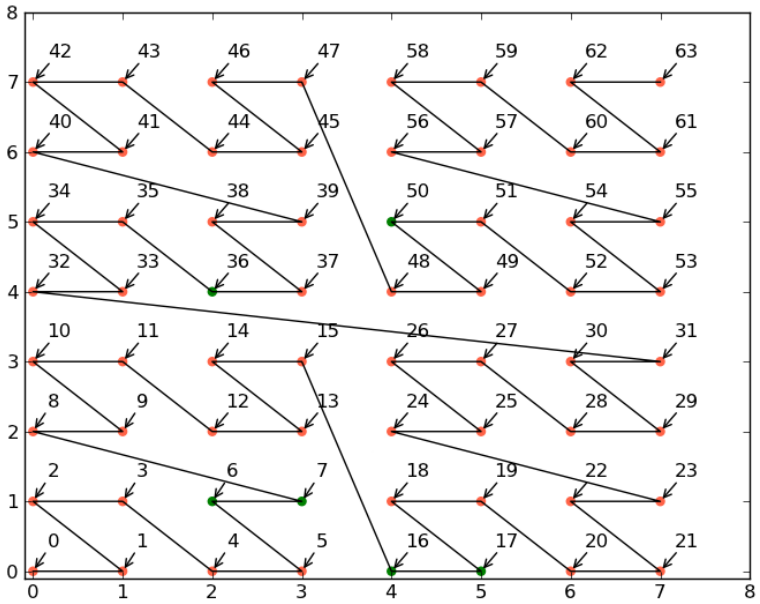

Figure 7.2: A 3rd order approximation of the Z Curve showing the sequence numbers and the mapping to 2D points. The area covered is the unit square, it is however divided into 8 parts in each dimension.

7.2 Space partitioning

The second approach used was to encode the multi-dimensionality of the data within the access filter and partition the space accordingly. In order to do that, it was necessary to start over and implement slightly differentiated data structures from the original ARF. The one-dimensional ARF splits the range in half in every descent. In two-dimensional space, the straightforward choices are: either split in half only in one dimension and have two children or split both dimensions in half and have four children (2^n children for n dimensions).

7.2.1 K-d Trees

The first idea is inspired from k-d trees. The k-d tree is a binary tree in which every node is a k-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. Points to the left of this hyperplane represent the left subtree of that node and points right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the k-dimensions, with the hyperplane perpendicular to that dimension's axis. So, for example, if for a particular split the "x" axis is chosen, all points in the subtree with a smaller "x" value than

the node will appear in the left subtree and all points with larger "x" value will be in the right subtree. We will call "x" the "split dimension".

We use the idea of splitting only in one dimension per descent in our multi-dimensional ARF. With every descent we keep the bounds the same in all other dimensions, and split the selected dimension in half. We define two variants:

- *K-d alternating:* Every dimension is used as a split dimension in a sequential way. If the dimensions have different domains, this is not a problem - after some point when one dimension has been exhausted, the navigation and splitting algorithms simply stop considering it as a split dimension. This variant is referred to in the graphs as, simply, "KD".
- *K-d with extra dimension bits:* With every descent, we specify which dimension was used for splitting within the ARF, using a binary representation of the split dimension's id. The split dimension chosen is the one that better serves the escalation process when a node is first split. This variant is referred to as "KD dimension-bit".

7.2.2 Quad Trees

The second idea is inspired from quad trees in the two-dimensional space and octrees in the three-dimensional space. With every descent, we split the two-dimensional space in four rectangles with the same area, the three-dimensional space in eight cubes with the same volume, and into 2^n hypercubes for n dimensions. During merging, we also merge 2^n nodes together. This variant is referred to as "QUAD" in the graphs.

7.2.3 Examples

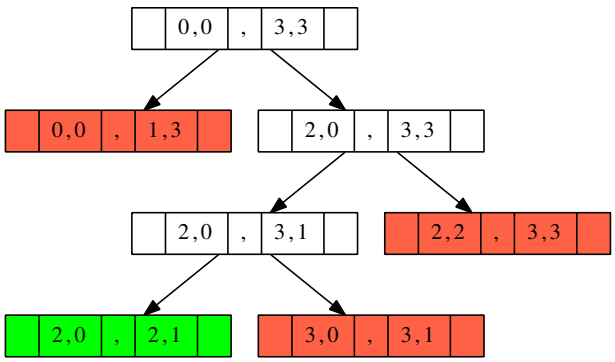

Figure 7.3: An example KD ARF

Figures 7.3 and 7.4 show an example KD ARF (alternating) and an example QUAD ARF, respectively. The examples are minimal, the cold store only contains values $[2, 0]$ and $[2, 1]$. The QUAD ARF tends to be less tall than the KD ARF because of the bigger node fanout. It is also evident how the KD ARF alternates between splitting in the first and second dimension at each level.

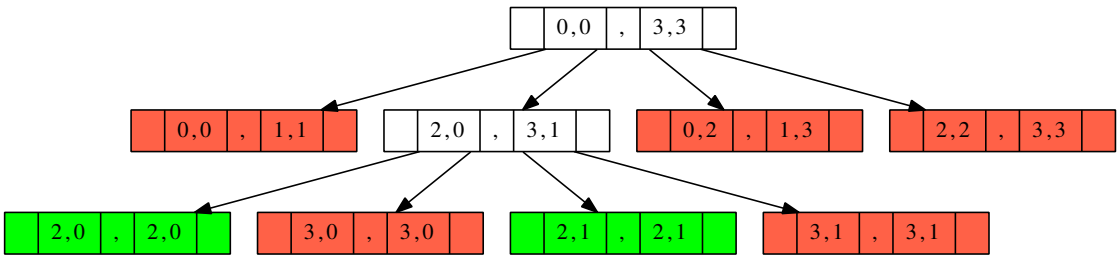

Figure 7.4: An example QUAD ARF

7.3 Adaptation

The basic concepts about adaptation remain the same as the ones described in chapter 5. It is the implementation that differs:

- The ARF variants that use a space-filling curve, handle a false positive query by converting it into a set of one-dimensionales and subsequently marking those as empty in the underlying one-dimensional ARF.
- The other variants need to "create" the range that caused the false positive within the ARF structure and then mark it as empty. In order to do that, we generalized the escalation algorithm described in 5.1. While in the one-dimensonal case, it suffices to escalate to the lower and upper bound of the range and mark everything in between as empty, in the multi-dimensional case we need to do more work because the ranges are no longer necessarily continuous.

7.4 Performance Evaluation

This section presents the results of experiments that we conducted with all the variants of multi-dimensional ARFs presented above. We compared those variants to each other and to a Bloom filter over a composite key. The benchmark environment and ARF training policy used were the same as the ones used for the experiments carried out for one-dimensional ARFs (chapter 6). We limited our experiments to only looking into the case of ARFs that become "frozen" after training. Our goal was to determine whether multi-dimensional ARFs can be useful for multi-dimensional range and point queries, which variants are the best and how they scale with the number of dimensions.

7.4.1 Data and Query Generation

For these benchmarks we used three types of data/workloads to model the cold store and the query workload:

- All combinations of Zipf and uniform data and workloads. We used an independent data generator for each dimension.
- All combinations of correlated/anti-correlated/uniform data and workloads. We used the same data generator as [30]. As an example, figure

7.6(left) shows two-dimensional correlated data (in blue) being queried with respect to rectangles which are generated by two-dimensional anti-correlated data (in red). This case means that not all the data is being queried, effectively introducing query skew.

- A real-life dataset containing locations of earthquakes on the Fiji Islands (courtesy of the R Project for Statistical Computing). For the query workload we used a either Zipf, uniform, correlated or anti-correlated data generator. We used the dataset as two-dimensional and as three-dimensional, adding the frequency of earthquakes at given points as the third dimension. As shown in figure 7.5, the data set is mostly clustered around two particular locations in the two-dimensional space.

  We also used independent normal distributions in each dimension to determine the size of each range query.

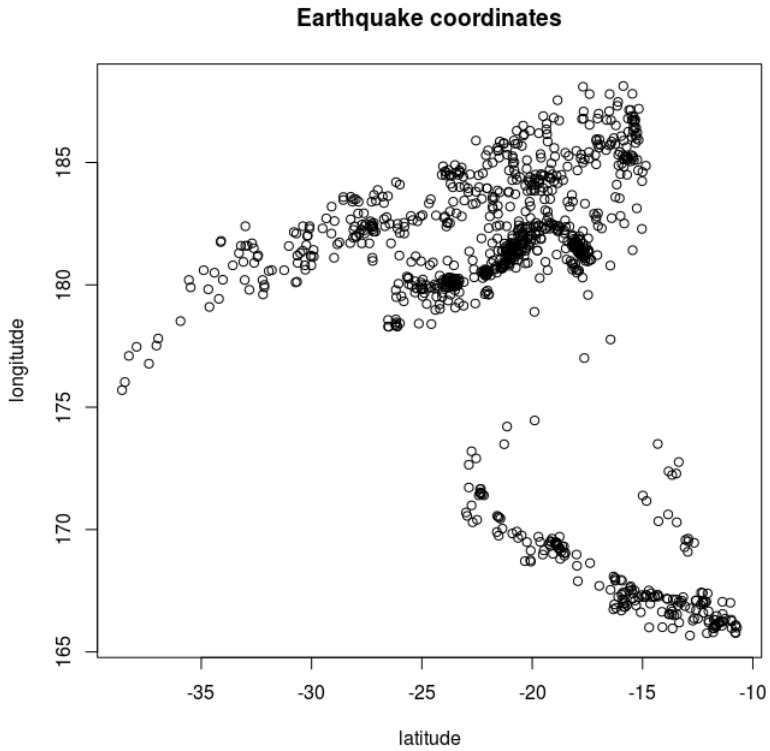

Figure 7.5: Locations of earthquakes on Fiji Islands

7.4.2 Experiment 1: Accuracy

We present a selection of our experimental results for two and three dimensions. In the 2D space, we ran point queries and range queries with area $a = 36$ and in the 3D space, point queries and range queries with volume $v = 27$. The

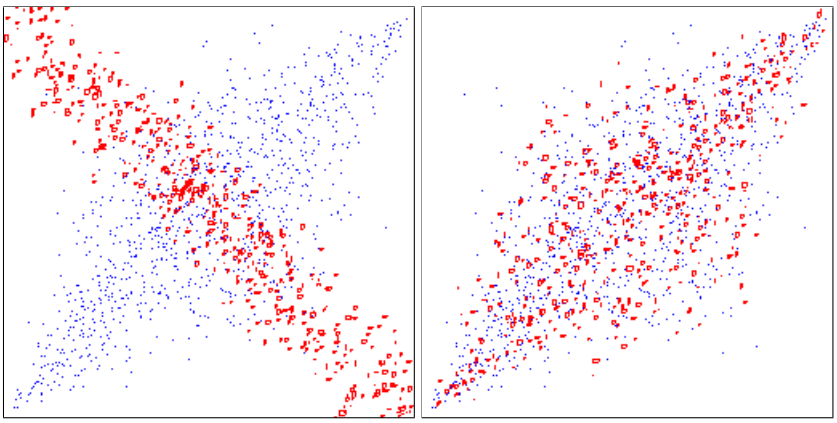

Figure 7.6: Data points (blue) and query rectangles (red) in 2D. Left: Correlated data, anti-correlated queries. Right: Correlated data, correlated queries

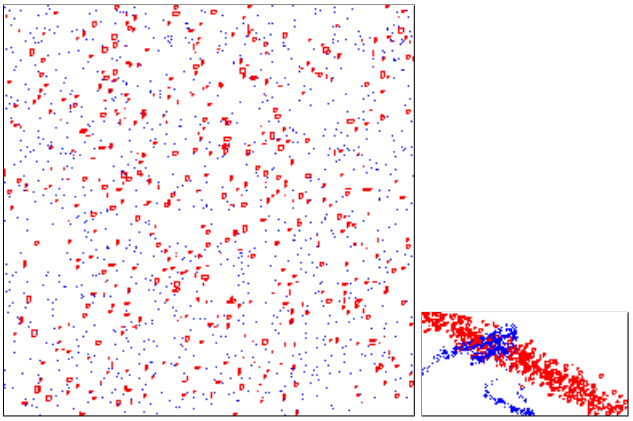

Figure 7.7: Data points (blue) and query rectangles (red) in 2D. Left: Uniform data, uniform queries. Right: Geographical data (earthquake locations), anti-correlated queries. Figures are to scale.

database was populated with 1000 points and we varied the filter sizes from 1 to 20 KBits. We assumed the high-dimensionality of the data justified examining what happens with a bigger space investment than the one-dimensional case. All other parameters (e.g., number of training queries, number of measured queries) were the same as in the experiments in chapter 6.

2D

We have selected to show the following graphs:

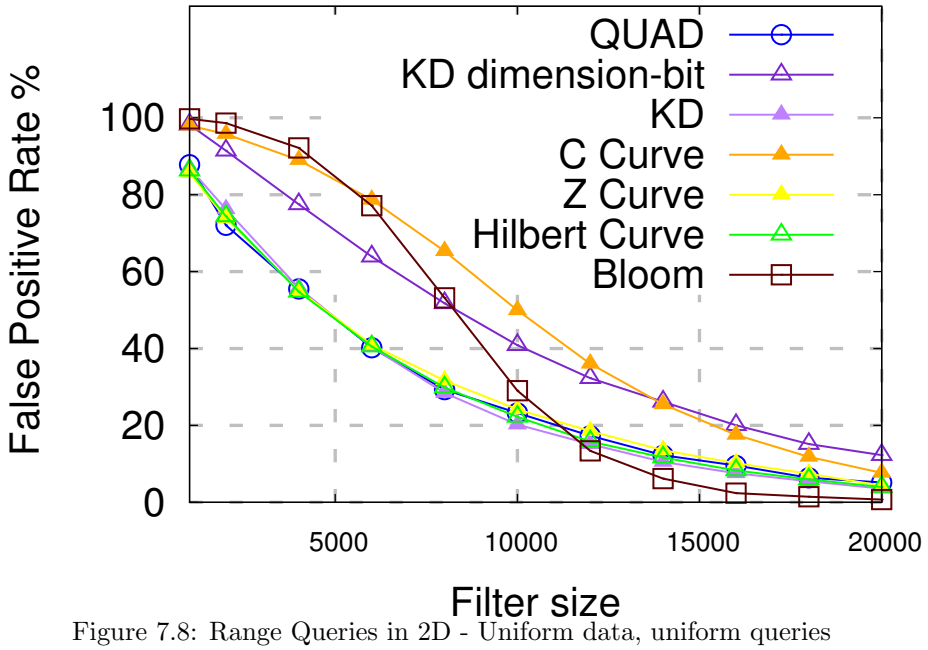

Figure 7.8: Range Queries in 2D - Uniform data, uniform queries

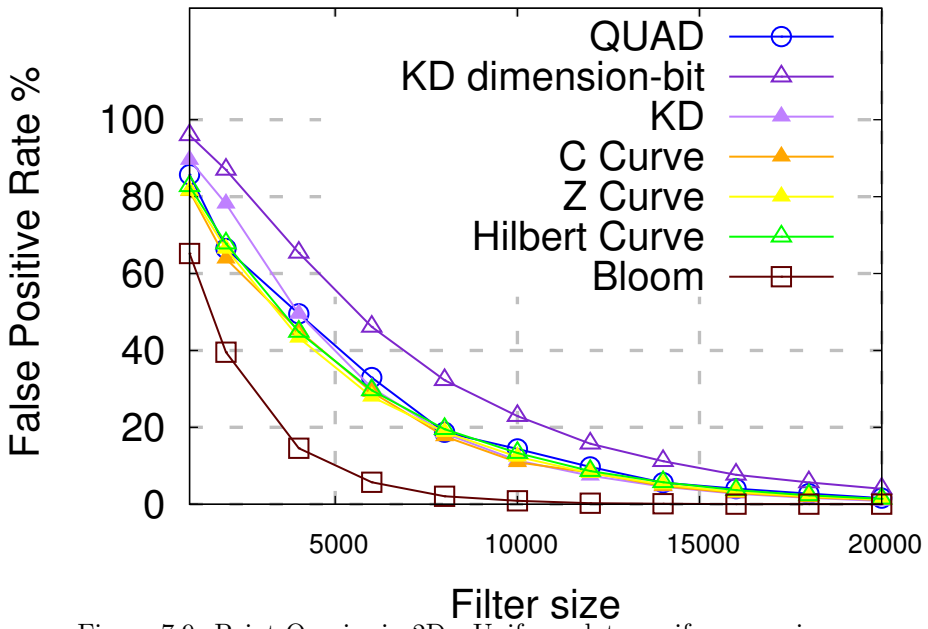

Figure 7.9: Point Queries in 2D - Uniform data, uniform queries

- Figure 7.8 shows the false positive rate for the data and query distribution described in figure 7.7(left), that is, uniform data/uniform queries. Figure 7.9 shows the results for point queries. Values in both dimensions have range from 0 to 1024.

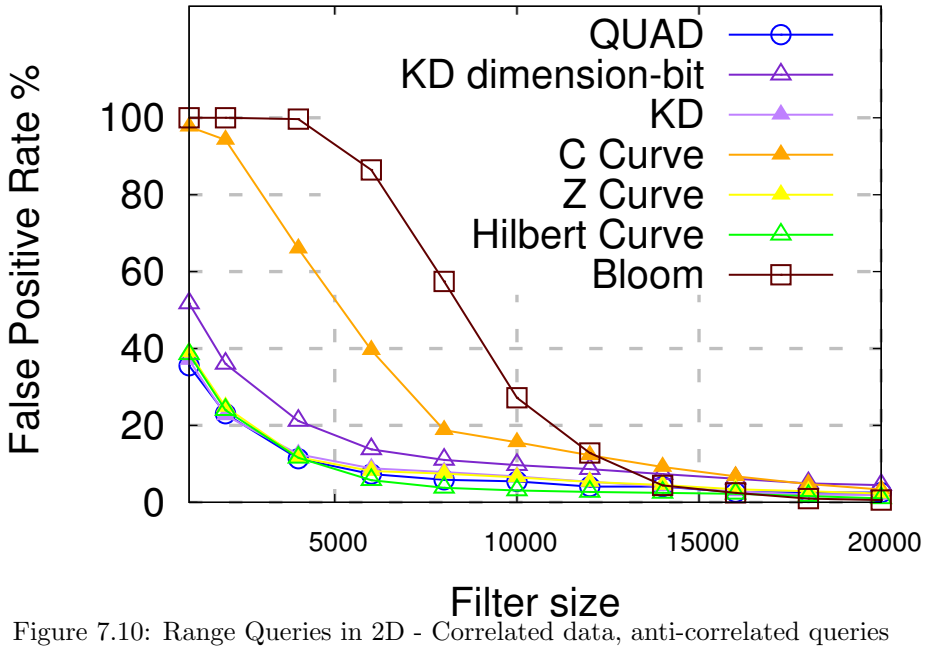

Figure 7.10: Range Queries in 2D - Correlated data, anti-correlated queries

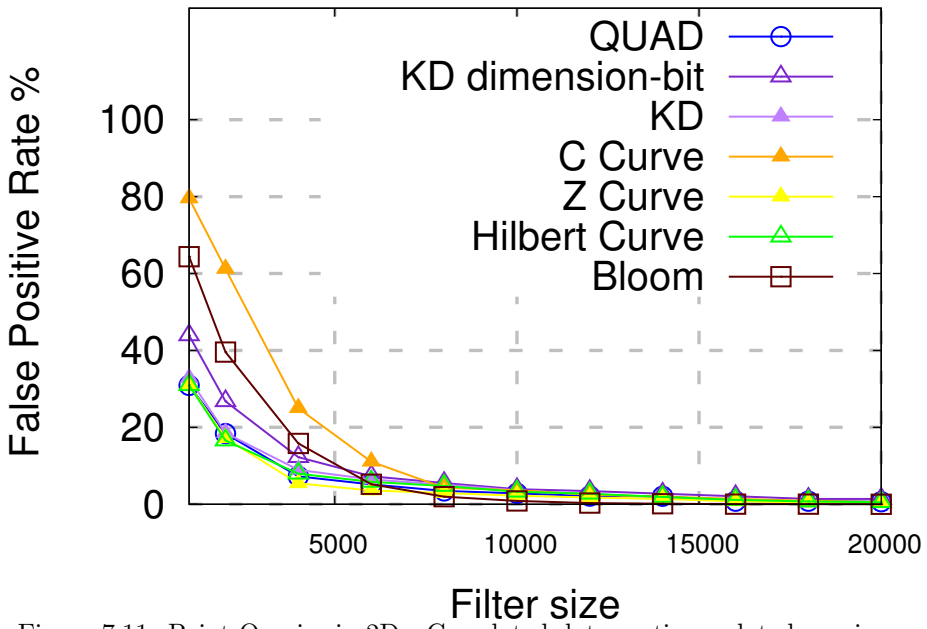

Figure 7.11: Point Queries in 2D - Correlated data, anti-correlated queries

- Figure 7.10 shows the false positive rate for the data and query distribution described in figure 7.6(left) (Correlated data/anti-correlated queries). Figure 7.11 shows the results for point queries. Values in both dimensions have range from 0 to 1024.

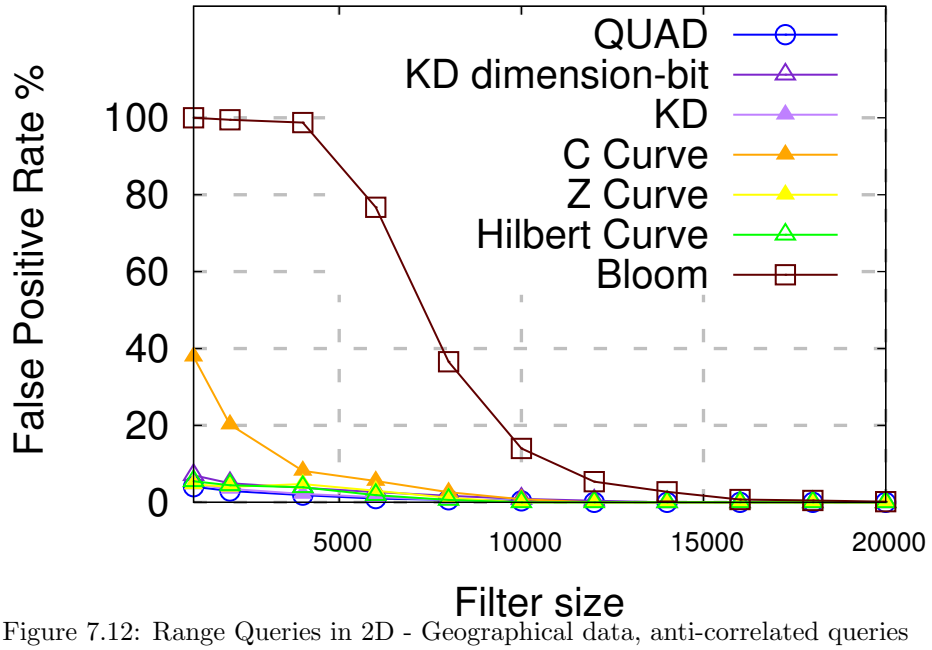

Figure 7.12: Range Queries in 2D - Geographical data, anti-correlated queries

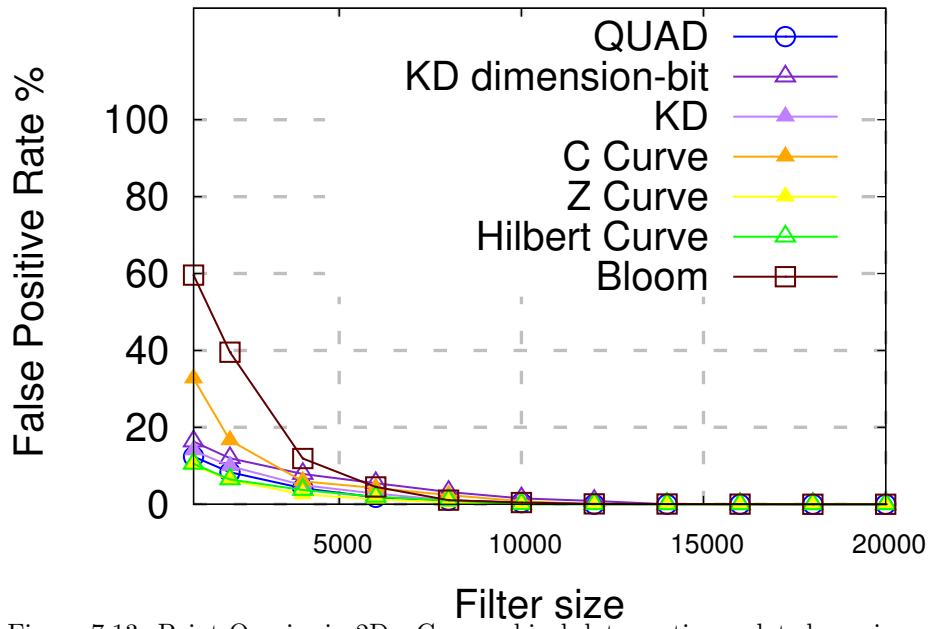

Figure 7.13: Point Queries in 2D - Geographical data, anti-correlated queries

- Figure 7.12 shows the false positive rate for the data and query distribution described in figure 7.7(right). The data distribution is the real-life dataset described in 7.4.1 and the queries are Anti-correlated (so as to have a small yet significant overlap with the data). Figure 7.13 shows the results

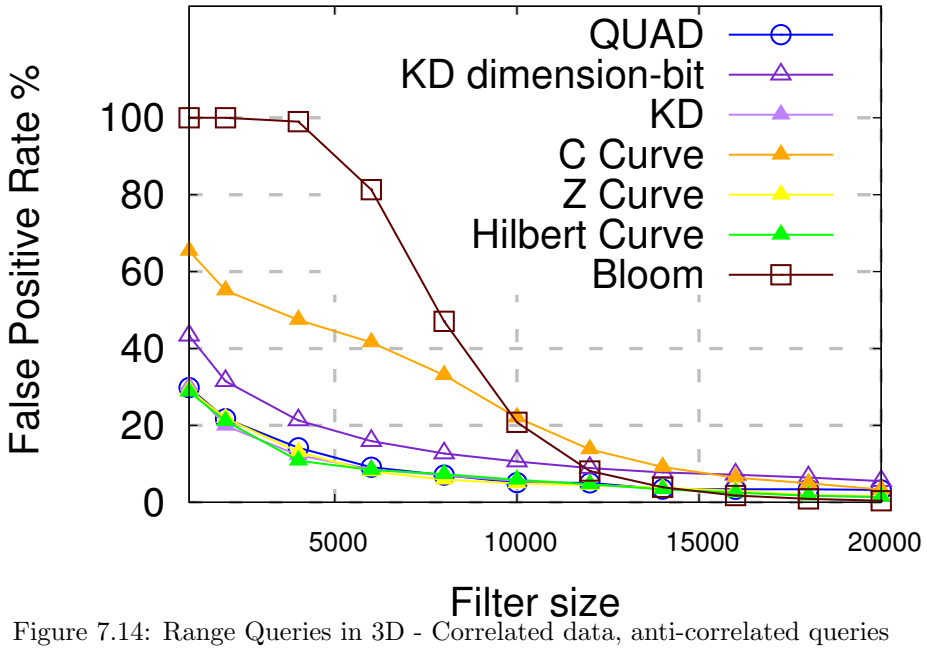

Figure 7.14: Range Queries in 3D - Correlated data, anti-correlated queries

for point queries. The values of the first dimension range from 0 to 512 and the values in the second dimension range fro 0 to 128.

It is evident from the graphs (in particular, figures 7.8 and 7.10) that the C Curve variant is not a good approach. The fact that it doesn't cluster data well makes it perform bad for range queries, because it cannot exploit the fact that the data might be close together, like the one-dimensional ARF does. Another obvious "bad" variant is the KD ARF which stores the "split dimension". The space investment does not pay off and would be better invested in creating a deeper ARF. The remaining four variants do not appear to have significant differences overall. The Z Curve and the QUAD variants store data in precisely the same order, although the QUAD variant has four children per node, and the KD ARF stores it in a very similar order, so this behavior is justified. The Hilbert Curve does display slightly better accuracy than the Z Curve on some cases, which can be attributed to better clustering, however, only in the order of 1 to 3 per cent.
The graphs also show, that, when the data and queries are uniformly distributed, the Bloom filters are the best choice. When there is some sort of skew, the "good" ARF variants gain an edge, at least in the beginning. When around 16 bits are invested, the Bloom filters are accurate enough to answer small range queries well, but this doesn't scale with the length of the range.
Overall, with skew present, we can say that the ARFs become reasonably good (less than 5 percent false positive rate) a lot faster (before the 10 Kbits mark) than the Bloom filters for range queries.

3D

We used three-dimensional data whose values ranged from 0 to 128 in each dimenson. Results in the 3D space are similar to the ones in the 2D space. Figure 7.14 (Correlated data/anti-correlated queries) shows that, with the exception of the variants we've identified as "bad", the ARFs can perform well (less than 5 percent false positive rate) even with a space investment around 10 Kbits.

7.4.3 Experiment 2: Accuracy with increasingly bigger query ranges

We varied the area a of the query boxes in the 2D space to see how the accuracy of the filters becomes affected. The filters used were all 8000 bits in size. Figure 7.15 shows the results for a correlated data/anti-correlated queries set-up and figure 7.16 shows the results of a correlated data/correlated queries set-up. The data and query distributions are visualized in 7.6(left) and 7.6(right) respectively.

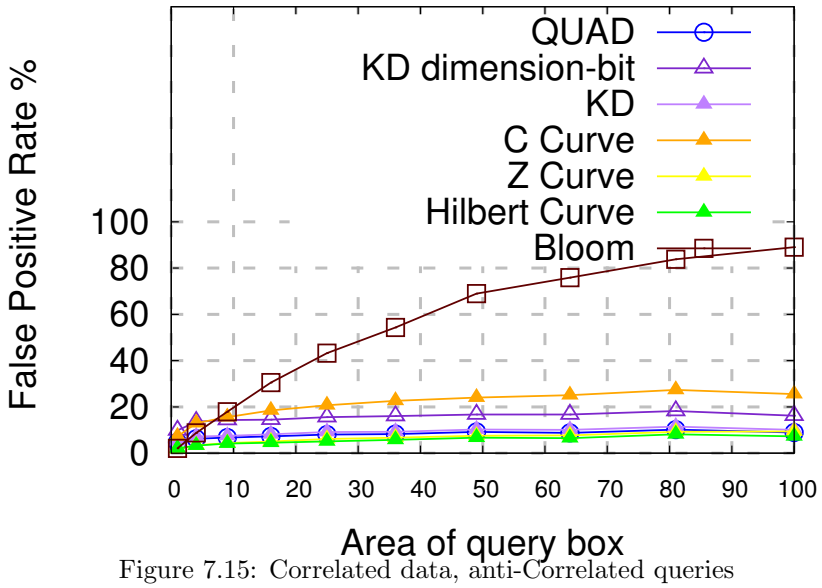

Figure 7.15: Correlated data, anti-Correlated queries

The most important finding of this experiment is, that, unlike one-dimensional ARFs which sustain the same accuracy regardless of the length of the range, multi-dimensional ARFs progressively deteriorate in accuracy as the ranges become bigger, although not as rapidly as the Bloom filters for this filter size. This does not have to do with the ARF per se, but with the lack of locality in range queries in the multi-dimensional space. The bigger the range is, the higher the possibility that several leaves far away from each other in the ARF need to be probed. The "bad" variants we identified in the previous section also scale worse than the rest.

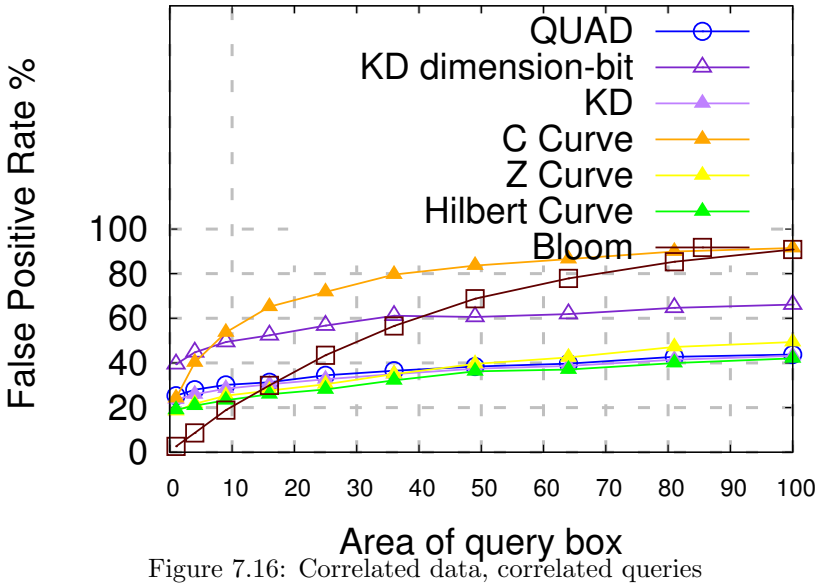

Figure 7.16: Correlated data, correlated queries

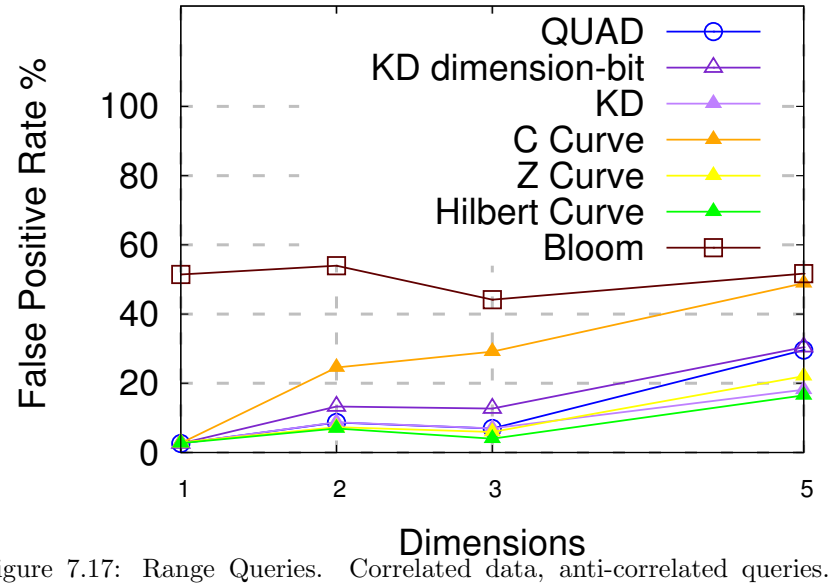

Figure 7.17: Range Queries. Correlated data, anti-correlated queries. See 7.6(left) for a visualization of the data and query distributions.

7.4.4 Experiment 3: Accuracy in higher dimensions

For this experiment, we looked at the behavior of the ARF with 1,2,3 and 5-dimensional data. We arranged the benchmark parameters so that the domain of the composite key would be between 2^{20} and 2^{21} and the hypercube queried each time contained roughly 30 multi-dimensional points. We used filters with size 8000 bits.

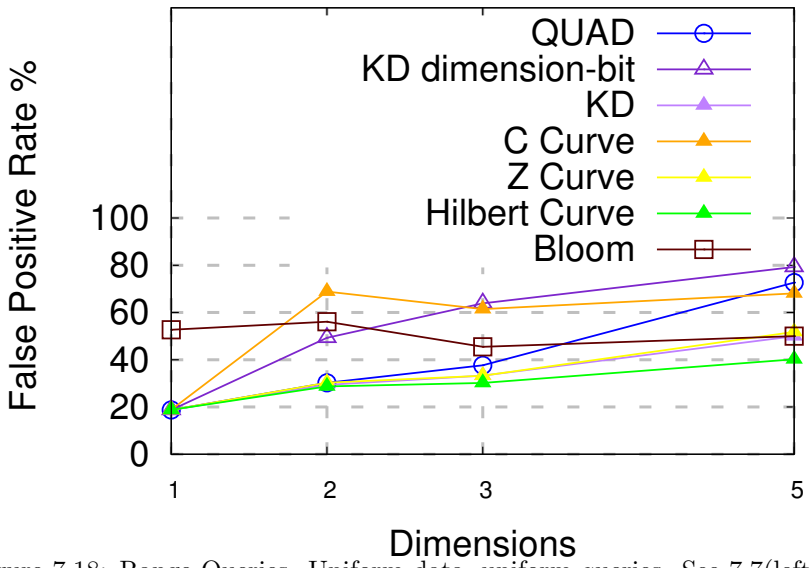

Figure 7.18: Range Queries. Uniform data, uniform queries. See 7.7(left) for a visualization of the data and query distributions.

With skew present (figure 7.17) or without (figure 7.18), the ARFs become worse as the number of dimension rises. That happens because when we increase the number of dimensions, it is harder to group neighbouring points together. An interesting observation is that, while in all other experiments the KD ARF and the QUAD ARF showed very similar accuracy, the increasingly large fanout of the QUAD ARF in higher dimensions, makes it become less and less accurate, as a result of very coarse-grained merging of leaf nodes. On the other hand, the Bloom filter is unaffected by the number of dimensions. This makes sense, because the Bloom filter maintains information about points only and is therefore not affected by locality in any way.

7.5 Conclusions

In this chapter, we looked into approaches for expanding the concepts of the ARF into the multi-dimensional space and evaluated these approaches in terms of accuracy. We determined the C Curve and the KD dimension-bit approaches to be inferior to all the others. Regarding the remaining four variants, the QUAD ARF scales badly with the number of dimensions and the Hilbert Curve is hard to compute, especially when it comes to range queries. Thus, the "winning" variants are the Z Curve and the KD ARF. We also showed that in the multi-dimensional space, skew is exploited by the ARFs as well.
It is important to note that range queries are not the same problem in the one-dimensional space as they are in higher dimensions. The more dimensions there are and the bigger the range queries, the less the data can be clustered together in a meaningful way. While multi-dimensional ARFs can be a good and space-efficient choice for range queries in 2 or 3 dimensions in the presence of skew, they are not a general solution for filtering multi-dimensional data in a robust

way, even when we're mostly interested in range queries, because range queries are increasingly difficult to handle in higher dimensions. This is consistent with the state of the industry, where no general-purpose data structure for indexing multi-dimensional data exists, and, in practical applications, people tend to use specialized data structures that suit their goals.

Chapter 8

Conclusions

This thesis presented ARFs, a versatile data structure designed to filter range queries. We discussed its application to a commercial database system that partitions data into a *hot* and *cold* region and its possible integration in an existing B-tree. ARFs have a number of advantages: 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. It was also shown possible to extend the ideas of the ARF into the multi-dimensional space, although range queries are inherently more difficult to handle in higher dimensions. Overall, we believe that ARFs are for range queries what Bloom filters are for point queries.

8.1 Future Work

One area to look into is to develop a mathematical framework for optimizing ARFs along the lines of the framework that has been developed for Bloom filters. As ARFs exploit skew, developing such a framework that meaningfully models skew in data and queries as discussed in 2.4, is much more challenging than for Bloom filters. Also, further investigation into the complexities and performance of multi-dimensional ARFs with respect to query time is needed to have a clear picture.

Another area for expansion is that of developing and comparing different policies for replacement, training and learning from updates. There are several approaches that have already been explored and others that have been proposed but we have by no means explored the whole space of possibilities, though we have given a sketch of the design space in 5.6 and discussed some ramifications of the choices on training for multi-dimensional ARFs in chapter 7.

Finally, we would also like to benchmark the ARF with more real world datasets or synthetic datasets with multiple clusters and, in the case of multi-dimensional data, different skew factors/correlations between dimensions.

Appendix A

Implementation Details

A.1 Data Structures

In our C++ implementation, we used a vector of vector<bool> to store the shape of the trie level by level. We used another vector of vector<bool> to store the leaf values for every leaf node in each level. To check if a node at a given index in a given level is a leaf or not, we only need to check if the corresponding bit in the shape vector is zero. For example, the shape of the example ARF in figure 4.1 (page 16) is stored in our implementation as the vector $[[0,1],[1,0],[0,1],[0,0]]$ and the leaves are stored as $[[1],[0],[0],[0,1]]$.

We chose to use the vector<bool> data structure as opposed to an array of booleans or a bitset because maximum space-efficiency and the ability to dynamically grow and shrink at runtime were required.

A.2 Navigation Algorithm

Algorithm 1 gives the pseudocode of the navigation procedure. The principle is the same as navigation through any other binary tree. The first difference is that we need to keep track of the ranges as we go. The second difference is the special way to navigate to children and leaves. Since no pointers are used for space efficiency, these tasks are a challenge. Algorithms 3 and 2 give the pseudo-code of these tasks in our implementation.

A.3 Optimizations

For point queries, we can simply return the value of
$navigate(key, 0, 0, 0, domain)$. That is, we start at $level = 0$,
$leftIdx = 0$ with starting range $[0, domain]$. For range queries, the strategy is to do a point query for the lower bound of the range and if it is empty and if the range of the leaf returned does not exceed the range of the query, continue doing point queries on the lowest not yet covered value in the range until it is exhausted or a hit is found. We optimized these "re-navigation" steps by storing the already calculated children indexes in a temporary data structure and using

them as a starting point to calculate children indexes of nodes further on the right (not shown in the pseudocode of Algorithm 3 for simplicity). Another optimization strategy we implemented and that is not shown in Algorithm 1 is to keep a history of the nodes visited and go back up and to the right through the data structure to do the re-navigation.

Algorithm 1 Navigation to leaf which contains key value

```
procedure NAVIGATE(key, level, leftIdx, low, high)
    rightIdx ← leftIdx + 1
    middle ← low + (high − low)/2
    if key <= middle then
        if shape[level][leftIdx] == 0 then
            return LEAFVALUE(level, leftIdx)
        else
            newIdx ← GETLEFTCHILD(level, leftIdx)
            return NAVIGATE(key, level + 1, newIdx, low, middle)
        end if
    else
        if shape[level][rightIdx] == 0 then
            return LEAFVALUE(level, rightIdx)
        else
            newIdx ← GETLEFTCHILD(level, rightIdx)
            return NAVIGATE(key, level + 1, newIdx, middle + 1, high)
        end if
    end if
end procedure
```

Algorithm 2 Find the value of a leaf node

```
procedure LEAFVALUE(level, idx)
    offset ← 0
    for all i such that 0 ≤ i ≤ idx − 1 do
        if shape[level][i] == 0 then                ▷ leaf node
            offset ← offset + 1                     ▷ skip its leaf value
        end if
    end for
    return leaves[level][offset]
end procedure
```

Algorithm 3 Find the index of the left child of a non-leaf node

```
procedure GETLEFTCHILD(level, idx)
    offset ← 0
    for all i such that 0 ≤ i ≤ idx − 1 do
        if shape[level][i] == 1 then                ▷ non-leaf node
            offset ← offset + 2                     ▷ skip its two children
        end if
    end for
    return offset
end procedure
```

Bibliography

[1] M. Stonebraker, "The case for partial indexes," *SIGMOD Rec.*, vol. 18, pp. 4–11, Dec. 1989.

[2] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[3] J. J. Levandoski, P.-A. Larson, and R. Stoica, "Identifying hot and cold data in main-memory databases," in *Proceedings of the 2013 ICDE international conference on Data engineering*, ICDE '13, 2013. to appear.

[4] 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 *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '13, 2013. to appear.

[5] Y. Ioannidis, "The history of histograms (abridged)," in *PROC. OF VLDB CONFERENCE*, 2003.

[6] Y. Hua, B. Xiao, and J. Wang, "Br-tree: A scalable prototype for supporting multiple queries of multidimensional data," *IEEE Trans. Comput.*, vol. 58, pp. 1585–1598, Dec. 2009.

[7] J. Bruck, J. Gao, and A. Jiang, "Weighted bloom filter," in *2006 IEEE International Symposium on Information Theory (ISIT'06)*, July 2006.

[8] M. Zhong, P. Lu, K. Shen, and J. Seiferas, "Optimizing data popularity conscious bloom filters," in *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, PODC '08, (New York, NY, USA), pp. 355–364, ACM, 2008.

[9] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *14th Annual European Symposium on Algorithms, LNCS 4168*, pp. 684–695, 2006.

[10] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, pp. 281–293, June 2000.

[11] D. Barbara, W. duMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. Ioannidis, H. V. Jagadish, T. Johnson, R. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik, "The new jersey data reduction report," *IEEE DATA ENG. BULL*, pp. 3–45, 1997.

[12] C. M. Chen and N. Roussopoulos, "Adaptive selectivity estimation using query feedback," 1993.

[13] A. M. Keller and J. Basu, "A predicate-based caching scheme for client-server database architectures," *The VLDB Journal*, vol. 5, pp. 35–47, 1996.

[14] S. D. Michael, M. J. Franklin, B. T. Jonsson, D. Srivastava, and M. Tan, "Semantic data caching and replacement," 1996.

[15] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves, "An architecture for recycling intermediates in a column-store," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, (New York, NY, USA), pp. 309–320, ACM, 2009.

[16] M. Altinel, "Efficient filtering of xml documents for selective dissemination of information," pp. 53–64, 2000.

[17] F. Fabret, A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha, "Filtering algorithms and implementation for very fast publish/subscribe systems," in *In SIGMOD*, pp. 115–126, 2001.

[18] J.-P. Dittrich, P. M. Fischer, and D. Kossmann, "Agile: adaptive indexing for context-aware information filters," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, (New York, NY, USA), pp. 215–226, ACM, 2005.

[19] R. Bayer and K. Unterauer, "Prefix b-trees," *ACM Trans. Database Syst.*, vol. 2, pp. 11–26, Mar. 1977.

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

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

[22] T. K. Sellis, N. Roussopoulos, and C. Faloutsos, "The r+-tree: A dynamic index for multi-dimensional objects," in *VLDB*, pp. 507–518, 1987.

[23] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA*, pp. 47–57, ACM, 1984.

[24] E. Netto, *Beitrag zur Mannigfaltigkeitslehre*. 1879.

[25] F. Ramsak, *Towards a general-purpose, multidimensional index: Integration, Optimization, and Enhancement of UB-Trees*. PhD thesis, Institut fur Informatik der Technischen Universitat Munchen, 2002.

[26] R. Bayer, "The universal b-tree for multidimensional indexing: general concepts," in *WWCA* (T. Masuda, Y. Masunaga, and M. Tsukamoto, eds.), vol. 1274 of *Lecture Notes in Computer Science*, pp. 198–209, Springer, 1997.

[27] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, "Analysis of the clustering properties of the hilbert space-filling curve," *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, p. 2001, 2001.

[28] D. Moore, "Fast hilbert curve generation, sorting, and range queries." http://www.tiac.net/ sw/2008/10/Hilbert/moore/index.html.

[29] J. Lawder, *The Application of Space-filling Curves to the Storage and Retrieval of Multi-dimensional Data*. PhD thesis, University of London, 2000.

[30] D. Kossmann, F. Ramsak, and S. Rost, "Shooting stars in the sky: An online algorithm for skyline queries," in *VLDB*, pp. 275–286, Morgan Kaufmann, 2002.