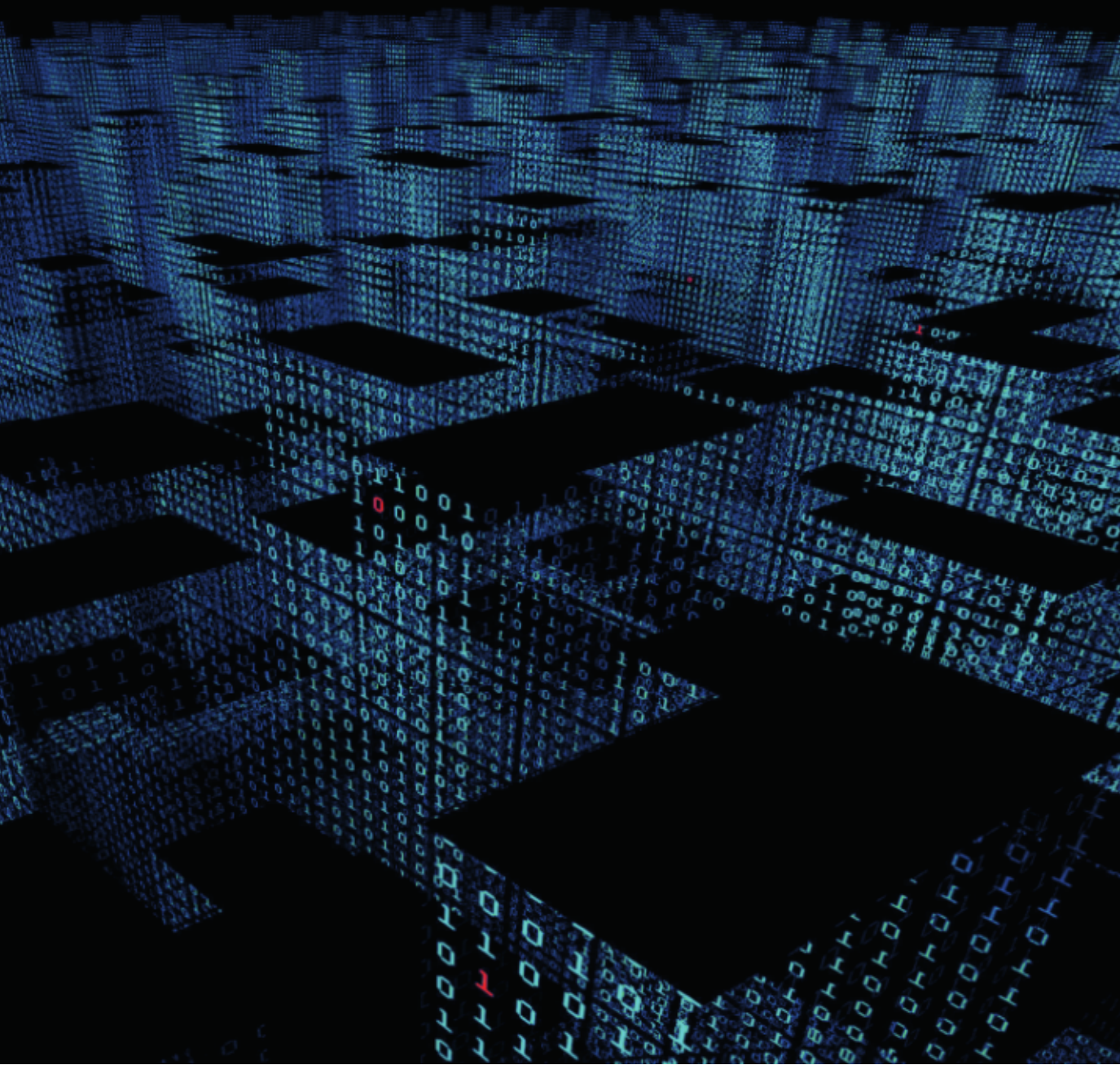
Space Efficient Indexes for the Big Data Era

Lefteris Sidirourgos



SPACE EFFICIENT INDEXES FOR THE BIG DATA ERA

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor

aan de Universiteit van Amsterdam

op gezag van de Rector Magnificus

prof. dr. D. C. van den Boom

ten overstaan van een door het college voor promoties ingestelde commissie, in het openbaar te verdedigen in de Aula

op Woensdag 14th Mei 2014, te 13:00 uur

door Eleftherios Sidirourgos

geboren te Athene, Griekenland

Promotor:	prof. dr. M.L. Kersten
Copromotor:	prof. dr. P.A. Boncz
Overige leden:	prof. dr. M. de Rijke
	prof. dr. J.A. Bergstra
	prof. dr. A. Ailamaki
	dr. P. Larson
Faculteit:	Natuurwetenschappen, Wiskunde en Informatica

The research reported in this thesis has been partially carried out at *CWI*, the Dutch National Research Laboratory for Mathematics and Computer Science, within the theme *Database Architectures*.

The research reported in this thesis has been partially carried out as part of the continuous research and development of the MonetDB open-source database management system.

SIKS Dissertation Series No 2014-23.
The research reported in this thesis has been carried out under the auspices of *SIKS*, the Dutch Research School for Information and Knowledge Systems.

The research reported in this thesis was partially funded by NWO.

The cover was designed by Elina Chatzichronoglou [1].

The printing and the binding of this disseration was carried out by CPI Wöhrmann print service.

ISBN 978-90-6196-567-1

[1] elixat.design@gmail.com

Contents

Chapter 1

Introduction

1.1 Scientific Discovery

Scientific discovery is the process of answering questions that have troubled mankind for centuries. In the early times, scientific discovery was based only on empirical observations. It was not until few centuries ago that we developed the necessary theories to explain our observations. The next breakthrough in scientific discovery was made possible the last few decades with the introduction of computers. Scientist were able to design complex mathematical models to simulate events, and with this newly acquired computational power, they were able through vast computations to examine "what if" scenarios and predict outcomes. However, the journey of scientific discover has not yet reached an end.

Nowadays, scientific discovery has *shifted from being an exercise of theory and computation, to become the exploration of an ocean of observational data*. This transformation was identified by Jim Gray as the *4th paradigm of scientific discovery* [35]. State-of-the-art observatories populated with astronomical data, digital sensors, and modern scientific instruments, produce every day petabytes of information. This scientific data is stored on massive datacenters for later analysis. But even from the data management viewpoint, the capture, curation, and analysis of data is not a computation-intensive process any more, but a data-intensive one. The explosion in the amount of scientific data presents a new "stress test" for database systems design. Meanwhile, the scientists are confronted with new questions: *how can relevant and compact information be found from such a flood of data?*

The predominant answer given by the data management community is the raw

power of big datacenter installations, complemented by new technologies focusing on scalable distribution of data and operations [2, 24]. When it comes to curating and analyzing vast amounts of data one can hardly argue that another approach is feasible. However, scientific discovery implies also a more delicate and refined task, that of forming the scientific question, a hypothesis to be later tested for correctness. Such formation of a scientific questions or hypothesis, as well as the initial proof-of-validity, constitutes an *ad-hoc exploratory scientific query workload*. During an iterative and interactive query session, the scientists pose queries in order to examine the nature of the stored data, discover interesting aspects of it, formulate their hypothesis, but also test the syntactical correctness of their queries. Scientists need *interactive* and *low-cost* means to make an initial exploration over the daily produced data. Facing this challenge calls for a database architecture exhibiting features different from contemporary ones.

1.2 E-commerce

Besides data-intensive scientific discovery, the *e-commerce surge* of the last decade has driven the need for managing huge amounts of data. The modern enterprise IT world is challenged with complex tasks aiming to get faster results and create more value out of their existing data loads and the readily available public data.

The e-commerce surge is primarily driven by the exponential evolution of the Web. Social networks, photography, videos, and blogging, as well as telecom data, web logs, even medical records, fuel the industry of decision-making applications. The collection and analysis of these data exceeds the capabilities of previous system deployments in enterprises. Data oriented frameworks are needed to provide real time services. Up until now, companies have used traditional analytic tools and data warehouses to analyze their own structured data, but more is to be discovered from the semi-structured web data. Enterprises seek to increase their profits by analyzing the impact of their services and the behavior of their users.

This is exactly where managing big amounts of data comes into play. Data analytics are needed in almost all of the big decision making capabilities for the enterprise community. New technologies are needed to overcome various technical obstacles, such as scalability, complexity of data, and speed (velocity) of the generation of data. For these reasons, we witness even more data centers and computer clusters put into use by enterprises. Big data holds great potential in the coming years to comes. Enterprises are not only looking for storing data but also want to get the best result out of it.

1.3 Big Data Analytics

The Big data term was coined following the e-commerce surge and the data-intensive scientific discovery. Big data refers to the collection of many data sets, but also, the term is used to describe the challenges posed to existing data management systems. The capture, curation, and analysis, but also the storage, transfer, and visualization of big data renders existing systems incapable of managing such loads. The challenges put forward are best described by the so called "3Vs", volume, velocity, and variety [46, 68].

Volume refers to the vast amount of data that is produced and needs to be managed. Special care is needed so one will not "get lost" in such flood of data. The techniques proposed to deal with the first V are tier storage [71], extract statistical valid samples [73, 74], and identify cold spots [48] to name a few.

Velocity describes the increased frequency in which queries are fired to the system, as well as the demand for real time interaction, e.g., web applications. Solutions in the literature include constant reorganization of data [38] and multilayered kernel processing [41].

Variety of the big data includes incompatible data formats, incomplete data structures and inconsistent semantics. XML-based data formats, or semantic frameworks such as the RDF, are used to map schemas, express semantic relationships and inference new statements.

Recently, a fourth "V" was added, namely *veracity*. In traditional business intelligence and analytics applications, data are well structured. Significant amount of time and money is invested to guarantee the correctness of the data that is stored. However, in the big data era, this is no longer possible, thus veracity of the data, i.e., the *uncertainty of data*, is now a necessary evil that has to be dealt with. The data is now imprecise, inaccurate and often erroneous.

1.3.1 Big Data System Landscape

The database community, motivated by these challenges, introduced systems with innovative architecture. For example, MapReduce [24] was proposed as a new paradigm for data processing on large clusters, based on two phase execution: namely map and reduce. In addition, many systems based on Hadoop, an open source implementation of MapReduce, have been proposed to bridge the gap between this distributed programming model and SQL-based data management [2]. System designers have build

database warehouses to work on top of distributed environments, such as Hive, Pig, Impala, and Shark/Spark.

The aforementioned systems harvest the raw data processing power by *scaling out*, i.e., when more processing power is needed, more machines are added. But, with the big data era, high-performance main memory systems have also flourished. These systems are designed to *scale up*. They take advantage of new and more powerful hardware by exploiting every last bit and cpu cycles available. Main memory transactional systems such as H-Store [39] and Hekaton [26], as well as read optimized systems such as MonetDB [58], Vectorwise, SAP Hana, and hybrid systems that support both transactional and analytical workloads, such as HyPer [40] belong to this category of highly optimized database systems.

1.4 Space Efficient Indexes

As mentioned before, the predominant answer of the database systems community to the big data challenge is the abundant use of computational and storage power. This is achieved with the deployment of large computer clusters and cloud computing. Even so, in most big data applications the access and the transfer of the data from slower storage units to faster memories still remains the bottleneck.

Main memory has become cheap enough to obtain large amounts, nevertheless all data can not possible fit in main memory, thus secondary (slower) storage is still in use, such as SSDs and HDDs. During the evaluation of a query, it is necessary to access the data stored in such slower media to ensure completeness of the answer. Also, the data has to be transferred from slower storage to main memory, and from main memory to L2 cache, L1 cache, and so on. Evidently, the access and transfer of the data through different layers of the memory hierarchy is the bottleneck for data intensive applications.

In this thesis we present a collection of storage efficient indexes to overcome the data transfer bottleneck. An index is typical used to restrict access to only the relevant to the query parts of the data. By space efficient we emphasize that the index has to be *significantly* smaller that the original data and typically has to reside in at least one level higher in the memory hierarchy than the indexed data. During query evaluation, before accessing and transferring data from a slower memory to a faster, we consult the space efficient index which resides in much faster memory. The index will reveal which parts, if any, of the ”slower” data are relevant to the query and should be transferred.

The main novelty of the indexes presented in this thesis compared to prior work is that they take advantage of the memory hierarchy to achieve optimal performance and to facilitate simple but efficient compression rates. The indexes are memory conscious

in the sense that they are designed to exactly fit in memory to optimize access. Moreover, they are *secondary indexes*, i.e., no data replication or reclustering is needed. These features make the indexes particularly useful for big data, both for systems that scale up, by exploiting the hardware, and for distributed systems that scaled out, since they can be used to reduce the amount of data transferred from one node to another.

The indexes we present here, except being space efficient, they also address different challenges of the 3Vs big data problems. Imprints is a cache-conscious index structure suitable for read intensive applications. It allows to identify which cachelines of data should be read to the cpu caches, thus reducing the time cost of bringing all data from main memory to the cpu. Imprints address the *volume* dimension of the big data challenge.

In the same line of thinking, the second index we introduce, named Split Bloom filters, is an index that restricts access to slower secondary storage. Besides addressing the *volume* problem, split Bloom filters also tackles the *velocity* of the big data. They allow for faster evaluation of frequent queries and more efficient updates of Bloom filters. The value indexes presented next in this thesis work for semi-structured data of any type, thus exploring solutions for the *variety* challenge posed by big data. Finally, the n-gram based index we present last allow for indexing of very large data sets on disks.

Space efficient indexes allow significant reduction of hardware investments. They make sure that all available power is harvest before moving to larger installations.

1.4.1 Contributions

The contributions of this thesis can be summarized as follows:

1. We introduce *column imprints*, a collection of many small bit vectors, each indexing the data points of a single cacheline. We introduce a compression schema for imprints, which is cpu friendly and exploits the empirical observation that data often exhibits local clustering or partial ordering as a side-effect of the construction process. We conducted an extensive experimental evaluation to assess the applicability and the performance impact of the column imprints.

2. We show how skew in access patterns can be exploited to improve Bloom filter efficiency (less space and/or lower false positive rate). We do this by splitting the filter into many smaller ones, where each filter covers only a small subset of records. We define a mathematical model and show how to optimally size the Bloom filters. We also describe how to adjust the sizing of the filters because of changes in access patterns or deterioration caused by updates. An extensive

experimental evaluation confirms that our algorithms construct better Bloom filters optimized for skewed access patterns, that they achieve a lower false positive rate or lower memory consumption depending on the settings, and that they can adapt gracefully to data and workload changes.

3. We describe a collection of indices for XML text, element, and attribute node values that *(i)* consume little storage, *(ii)* have low maintenance overhead, *(iii)* permit fast equi-lookup on string values, and *(iv)* support range-lookup on any XML typed value (e.g., double, dateTime). We evaluate the stability of the hash function, the storage overhead, and the indices creation and maintenance time in the context of MonetDB/XQuery.

4. We study methods to conserve the scalable creation time and efficient exact substring query properties of gram indices, while reducing storage space. We first propose a *partial gram* index based on a reduction from the problem of omitting indexed q-grams to the *set cover problem*. While this method is successful in reducing the size of the index, it generates false positives at query time, reducing efficiency. We then increase the accuracy of partial grams by splitting posting lists of frequent grams in a frequency-tuned set of *signatures* that take the bytes surrounding the grams into account. The resulting *qs-gram* scheme is tested on big data collections (up to 426GB) and is shown to achieve an almost 1:1 data to index size, and query performance even faster than normal gram methods.

1.4.2 Published Papers

The material in this thesis has been published in major international refereed database conferences.

1. **Column Imprints: A Secondary Index Structure**. Lefteris Sidirourgos and Martin Kersten. *Proceedings of the ACM SIGMOD Conference in New York, New York, USA, 2013.*

2. **Memory Efficient Bloom Filters for Skewed Access Patterns**. Lefteris Sidirourgos and Per-Åke Larson. *Submitted for publication at the moment of printing this thesis.*

3. **Generic and Updatable XML Value Indices Covering Equality and Range Lookups**. Lefteris Sidirourgos and Peter Boncz. *Fourth International workshop on Database Technologies for Handling XML Information on the Web (DataX'09), collocated with EDBT/ICDT 2009 joint conference, St. Petersburg, Russia, 2009.*

4. **Space-Economical Partial Gram Indices for Exact Substring Matching**. Nan Tang, Lefteris Sidirourgos and Peter Boncz. *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM), Hong Kong, China, 2009.*

1.4.3 Thesis Outline

Each chapter of this thesis is an independent work, each of which introduces a new index schema. We do not include in this thesis a single related work chapter, since each index has its own. Chapter 2 presents column imprints. Next, Chapter 3 presents split Bloom filters. Chapter 4 presents the XML value indexes. Finally, Chapter 5 presents the qs-grams. Chapter 6 concludes this thesis.

Chapter 2

Column Imprints: A Secondary Index Structure

Large scale data warehouses rely heavily on secondary indexes – such as bitmaps and b-trees – to limit access to slow IO devices. However, with the advent of large main memory systems, cache conscious secondary indexes are needed to improve also the transfer bandwidth between memory and cpu. In this chapter, we introduce *column imprint*, a novel and powerful, yet lightweight, cache conscious secondary index. A column imprint is a collection of many small bit vectors, each indexing the data points of a single cacheline. An imprint is used during query evaluation to limit data access and thus minimise memory traffic. The compression for imprints is cpu friendly and exploits the empirical observation that data often exhibits local clustering or partial ordering as a side-effect of the construction process. Most importantly, column imprint compression remains effective and robust even in the case of unclustered data, while other state-of-the-art solutions fail. We conducted an extensive experimental evaluation to assess the applicability and the performance impact of the column imprints. The storage overhead, when experimenting with real world datasets, is just a few percent over the size of the columns being indexed. The evaluation time for over 40000 range queries of varying selectivity revealed the efficiency of the proposed index compared to zonemaps and bitmaps with WAH compression.

2.1 Motivation

Indexes are a vital component of a database system. They allow the system to efficiently locate and retrieve data that is relevant to the users' queries. Despite the large body of research literature, just a few solutions have found their respective places in a database system [8, 33, 38, 62]. Nevertheless, the pursuit for more efficient and succinct indexing structures remains.

Indexes are divided into *primary* and *secondary* according to their ability to govern the placement of the data. Primary indexes combine navigational structures with physical data clustering to achieve fast access. The benefit is that relevant data is placed in adjacent pages and thus significantly improving the evaluation of range queries. However, each additional primary index on the same relation calls for a complete copy of the data, rendering the storage overhead prohibitive. Similarly, secondary indexes are auxiliary structures that speed up search, but they do not change the order of the data in the underlying physical storage. Secondary indexes are typically much smaller than the referenced data and, therefore, faster to access and query. However, retrieving the relevant data from disk can be a costly operation since it may be scattered over many pages. As long as the time to scan the secondary index is significantly less than accessing the data, and the selectivity of the query is high, secondary indexes can significantly improve the query evaluation time.

Most structures designed for primary indexing, such as B-tree and hash tables, can also be used for secondary indexing. However, they are not as lightweight as one would wish. Bitmaps, or variations of bitmaps, are more often used for this task [80]. Bitmaps work by mapping individual values to an array of bits. At query time, the bitmap is examined and whenever the bits that correspond to the query's predicates are set, the mapped data is retrieved for further processing. Bitmaps are traditionally used for attributes with low cardinality [60], although bit-binning techniques make them suitable for larger domains too [19, 75].

With the introduction of column stores and the shift of the memory bottleneck [54], the need for designing *hardware-conscious* secondary indexes becomes more evident. In a main memory DBMS, the problem of efficiently accessing disk blocks is replaced with the problem of minimizing cache misses. In addition, algorithms require a more careful implementation. There is much less design space to hide an inefficient implementation behind the latency of accessing a disk block.

A second paradigm shift concerns the volume and the nature of the data. Most notable of them all are scientific database applications that stress the limits of modern designs by including hundreds of attributes in a single relation. In addition, the value domains are often of double precision, rather than the traditional categorical ones encountered in business applications. Column stores are the prime candidates for provid-

ing solutions for such demanding applications. On high-end servers, with large main memories, it is even possible to keep many columns with billions of elements in memory over a long period of time. Nevertheless, fast access, supported by light-weight indexing structures, remains in demand to improve the interactive scientific exploration process.

We propose a simple but efficient secondary indexing structure, called *column imprints*. A column imprint is a cache conscious secondary indexing structure suitable for both low and high cardinality columns. Given a column with values from domain $\mathcal{D}$, we derive a small sample to approximate a histogram of a few (typically 64 or less) equal-height bins. The entire column is then scanned, and for every cacheline of data, a bit vector is created. The bits in each vector correspond to the bins of the histogram. A bit is set if at least one value in the cacheline falls into the corresponding bin. The resulting bit vector is an *imprint* of the current cacheline that describes which buckets of the approximated histogram the values of the cacheline fall into. The collection of all the resulting bit vectors form a unique *column imprint*. Consequently, by examining an imprint of a column, the execution engine can decide –in a cacheline granularity– which parts of the column data are relevant to the query predicates, and only then fetch them for further processing. A column imprint is particularly suited for evaluating both range and point queries on unsorted data. Contrary to existing work, a column imprint is a *non-dense* bit indexing scheme, i.e., only one bit is set for all equal values in a cacheline, instead of the traditional approach where each data point is always mapped to a different bit.

To reduce the memory footprint of a column imprint, we introduce a simple compression scheme based on a run-length encoding of imprints. Consecutive and identical bit vectors are compressed together and annotated with a counter. Paraphrasing, our compression schema can be characterized as row-wise, i.e., it compresses bit vectors horizontally, contrary to the more common column-wise approach that partitions a bitmap vertically and compress it per column [82]. The horizontal compression exploits our empirical observation that, in most data warehouses that we explored, data suitable for secondary indexing exhibits, in the cacheline level, some degree of clustering or partial ordering. These desirable properties stem either from the regular and canonical data insertion procedure, or from the production of the data itself, or even indirectly imposed by the other primary indexed attributes of the same relation. Column imprints are designed such that any clustering or partial ordering is naturally exploited without the need for extra pasteurization. In other words, they are less susceptible to the order in which individual values appear in a cacheline, while more opportunities for compression are presented. In addition, because of this immunity to value order within a cacheline, a column imprint remains robust even in the case of highly unclustered data. We experimentally demonstrate that imprints perform well and behave

as intended even in the presence of skewed data, where other state-of-the-art bitmap compression techniques, such as WAH [82], are less effective.

2.1.1 Contributions

The contributions of the work in this chapter can be summarized as follows:

- We introduce column imprints, a light-weight secondary index structure for main memory database systems.
- We detail the algorithms and the implementation details for constructing and compressing a column imprint.
- We present the algorithms to efficiently evaluate range queries with the use of column imprints.
- We study the effect on imprints when updating the values of a column.
- We quantify the amount of local clustering by introducing a metric called *column entropy*.
- We conduct an extensive comparative experimental evaluation of the imprint index structure using thousands of columns taken from several real-world datasets.

2.1.2 Outline

The remainder of this chapter is organized as follows. In Section 2.2 we detail the ideas and the algorithms for constructing a column imprint. In Section 2.3 we present the algorithms for querying the proposed index. Next, we study the different cases of updating column imprints in Section 2.4. Section 2.5 presents the related work. In Section 2.6 we present an extensive experimental evaluation for column imprints. We summarize this chapter in Section 2.7.

2.2 Secondary Index with Imprints

An imprint index is an efficient and concise secondary index for range and point queries. It is designed for columnar databases where multiple memory-resident or memory-mapped columns are repeatedly scanned. Imprints provide a coarse-grain filtering over the data, aimed at reducing expensive loading from memory to the cpu

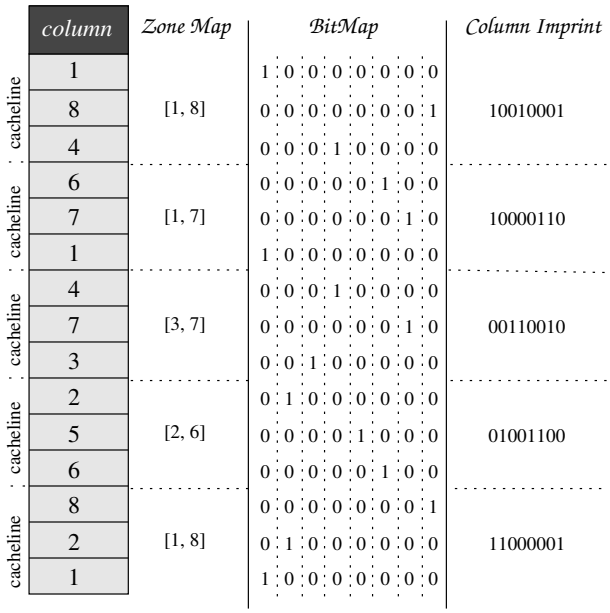


Figure 2.1: Example of zonemaps, bitmaps, and *imprints* indexes.

cache. Deployment of column imprints is suited for those cases where alternative properties do not hold. For example, if a column is already sorted, the proper use of binary search algorithms largely alleviates the overhead of accessing non-relevant memory pages. If the data is appended out of order, or the order is disturbed by updates, then column imprints can be considered as a fast access method to locate relevant data. An efficient column imprint maximizes the filtering capabilities with minimal storage overhead.

Columnar databases decompose a relation into its attributes and sequentially store the values of each column. This differs from the traditional approach of row-stores that place complete tuples in adjacent pages. To enable tuple reconstruction in a column store, an ordered list of *(id, value)* pairs is maintained, where *id*s are unique and increasing identifiers. Values from different columns, but with the same *id*, belong to the same tuple. Typically, a column is implemented by a single dense array, thus *id*s need not be materialized since they can be easily derived from the position of the values in the array.

Figure 2.1 shows a column with 15 integer values in the range of 1 to 8. The values

are unsorted because the column corresponds to one of the unordered attributes of a relation. In the absence of any secondary index, a complete scan is needed to locate all values that satisfy the predicates of a query. The result of such a scan is the positions in the array of the qualifying values. It is preferred to return the positions rather than the actual values because of the *late materialization* strategies usually used in column stores [1]. However, instead of scanning the entire column, secondary indexes can be used to avoid accessing data that is certain not to be part of the query result.

2.2.1 State of the Art in Secondary Indexes

Zonemaps is a common choice for indexing secondary attributes. A zonemap index notes the minimum and the maximum values found across a predefined number of consecutive values, called the zones. The zonemap index of Figure 2.1 partitions the column into 5 zones. In this example, each zone has the size of a cacheline that fits exactly 3 values. The first zone contains the values 1, 8, and 4. The minimum value is 1 and the maximum is 8. Similarly, for the second zone the minimum value is 1 and maximum 6, and so on for the remaining zones. To evaluate a query using zonemaps, the minimum and maximum values of each zone are compared with the predicates of the query. If the predicates' ranges overlap with the range of a zone, then the zone (i.e., the cacheline) is retrieved and the exact positions of only the *qualifying* values are returned. Note that the ranges of the predicates and the zone may overlap but not be strictly inclusive.

Bitmaps are another popular choice for secondary indexing. They work by mapping the column domain to bit vectors. Each vector has as many bits as the size of the column. For each value found in a specific position of the column, the corresponding bit in the mapping bitvector is set. The mapping can be $1 - 1$ if the cardinality of the column is low, or $N - 1$, with the help of binning strategies, if the cardinality is high. A bitmap index uses significantly less storage than the column, thus making it cheaper to scan. Deciding if a value satisfies a query involves first checking the corresponding bitmap, and returning only the position of the bits that are set. The checking is done with bitwise operators, making the process faster than the value comparison needed by zonemaps. Figure 2.1 details a bitmap index with 15 bits per bit vector, where each bit corresponds to one position of the column. There are 8 such bit vectors (drawn vertically in the figure), where the first one maps value 1, the second one value 2, and so on. Bits are set as follows: the 11th position of the column contains the value 5, therefore, in the 5th bit vector, the 11th bit is set. Similarly, the 3rd value of the column is 4, hence the 3rd bit of the 4th bitmap is set. In this example there is a 1-1 mapping between the eight unique values of the column and the eight vectors of the bitmap index.

2.2.2 Column Imprints

We propose *column imprints* as an alternative secondary index that best combines the benefits of the aforementioned state-of-the-art indexes. Column imprints map the values of a column to a vector of bits. However, instead of allocating one such vector per value, imprints allocate one vector per cacheline. We call the vectors of a column imprints index *imprint vectors* to distinguish them from the bitvectors of a bitmap index. An imprint vector does not have only one bit set per position, but as many bits as are needed to map all distinct values of a cacheline. To decide if a cacheline contains values that satisfy the predicates of a query, first the imprint vectors are checked. If at least one common bit between the bitvector that maps the query's predicates and the imprint vector is set, then the entire cacheline is fetched for further processing. The imprint is checked with the bitwise operator `AND` thus making the initial filtering very fast, while the number of imprint vectors to be checked is significantly reduced because of having one per cacheline instead of one per value. The rightmost index in Figure 2.1 depicts the imprint index of the example column. Each imprint vector uses 8 bits per cacheline, while three bits are set. The partitioning of the column is done per cacheline, same as the zones of the zonemap index. The imprint vector corresponding to the first cacheline has the 1st, 4th, and 8th bit set, since the first three values of the column are 1, 8 and 4. For the second cacheline the 1st, 6th, and 7th bits are set, and so on for the rest of the cachelines. There are in total five imprint vectors to index the column of Figure 2.1. The example is designed with the cardinality of the column to be small enough to allow a 1-1 mapping between values and bits. In the more common cases of large cardinality, imprints use approximated equi-width histograms to divide the domain into ranges and map one bit per range. We detail this technique in the following subsection along with all the construction algorithms for column imprints.

Column imprints inherit many of the good properties of both zonemaps and bitmaps, while avoiding their pitfalls. First, although imprints are defined per cacheline, they are resilient to skewed data distribution, where zonemaps typically fail. If each cacheline contains both the minimum and the maximum value of the domain and one random value in between, zonemaps are practically useless, but imprints will have a different bit set for each of these random values. In addition, checking imprints is faster than zonemaps because there is no value comparison. Compared to bitmaps, imprints need less space since they are defined per cacheline and not per value. Finally, as we will demonstrate, imprints compress significantly better than state-of-the-art compression scheme for bitmaps.

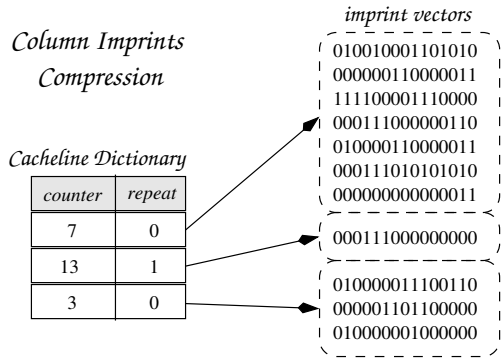


Figure 2.2: Column imprint index with compression (23 cachelines).

2.2.3 Imprints Compression

We develop a compression scheme similar to a run-length encoding but for imprint vectors. The compression scheme combined with bit-binning, makes column imprints an efficient solution for indexing very large columns with high cardinality of any type, such as doubles, floats, etc. The compression scheme benefits from our empirical observation that local clustering is a common phenomenon even for secondary attributes. In addition to that, the opportunities for compression also increase because of the non-dense nature of column imprints. Most importantly, even for cases where there is no clustering at all, column imprints remain space effective. The compression works by *i)* grouping together imprint vectors that are identical and consecutive, and *ii)* implying the *id* of the values with a concise numbering schema for the indexed cachelines. More specifically, we keep track of which imprints map to which cachelines by defining a *cacheline dictionary* with two entries, a *counter* and a *repeat* flag. By knowing the number of the cacheline we can easily compute the *id*'s of the values of the specific cacheline, since each cacheline contains a fixed number of values.

The cacheline dictionary contains two types of *counter* entries, distinguished by the *repeat* flag. Assume that the *counter* has the value x. If *repeat* is unset, then the next x cachelines have all different imprint vectors. If, however, *repeat* is *set*, then the next x cachelines all have the same imprint vector, thus only one vector needs to be stored. Figure 2.2 shows an example of the column imprints compression schema. Assume a column that can be partitioned to 23 cachelines and that each imprint vector has 15 bits. From the *cacheline dictionary* of Figure 2.2 we can deduce that the first 7 cachelines all contain random values, thus each of them map to a different imprint

vector. Therefore, the first 7 imprint vectors correspond to the first 7 cachelines. The next imprint vector, i.e., the 8th, corresponds to the next thirteen cachelines, which according to the cacheline dictionary all have an identical imprint since *repeat* is set. Finally, the last 3 cachelines are mapped by the last 3 imprints.

In the next subsection we demonstrate the technical details to create a column imprint. We build our ideas on top of the MonetDB architecture [58]. The choice of a specific columnar database architecture allows us to better present the details of our implementation, however, imprints can also be implemented with minor adjustments on other columnar architectures, such as C-Store [76] and MonetDB/X100 [14]. The most important design decision is how many values of a column an imprint vector covers. The decision is based on the size of the block managed by the specific database buffer pool. The *access granularity* of the underlying system design determines the number of values that each vector of an imprint covers. For example, if the execution model of the database engine is based on vectorization, then the size of the data vectors is used. In our scenario, where typically the database hot-set fits into main memory, our goal is to optimize the cpu cache access. For that reason, a column imprint consist of one vector per cacheline. The size of the cacheline is determined by the underlying hardware. In this work we assume the commonly used size of 64 bytes.

2.2.4 Imprints Construction Algorithm

The first step to create an imprint index for a column is to build a non-materialized histogram by sampling the values of that column. Then the imprint vectors are created with as many bits as the number of bins in the histogram, but never more than 64 bits. Each imprint covers a cacheline of 64 bytes. For all values in a cacheline, the bins of the histogram into which they fall is located, and the corresponding bits are set. The process is repeated such that all cachelines are mapped by imprints. If consecutive imprint vectors are identical they are compressed to one and the counters of the cacheline dictionary are updated.

The histogram serves as a way to divide the value domain $\mathcal{D}$ of the column into equal ranges. For this, only the bounds of each bin need to be stored in the imprint index structure. The histogram is created by sampling a small number of values from the column, not more than 2048 in our implementation. The first bin always has values between $-\infty$ (i.e., the minimum value of the domain $\mathcal{D}$), up until the smallest value found in the sample. Similarly, the last bin contains all values greater than the largest sampled value up to $+\infty$. We expect that future inserts in the column will retain the same distribution of the values, however, the left and right most bins serve as overflow bins for outlier values. If the sampling returns fewer than 62 unique values, then the imprint can be adjusted to have as many bits as needed to map the columns with low

cardinality. If the number of distinct sampled values is more than 62, the domain is divided into 62 ranges, where each range contains the same count of sampled values, including in the count the multiple occurrences of the same value. Based on these ranges the borders of the histogram are deduced. By counting also duplicate sampled values, it allows us to roughly approximate an equal-height histogram, since repeated values are more likely to be sampled, creating smaller ranges for their respective bins. The ranges of each bin are defined to be inclusive on the left, and exclusive on the right. For example, if $b[i]$ defines the border of the ith bin, then if $b[3] = 10$ and $b[4] = 13$, all values that are equal or greater than 10 but less than 13 fall into the 4th bin with borders $[10, 13)$, while value 13 falls into the 5th bin.

For each imprint, an index number is needed to point to the corresponding cacheline. In practice, these pointers need not be materialized since the sequence of the imprint vectors indirectly provide the numbering of the cachelines. However, since identical imprints tend to repeat multiple times, even if the data of the indexed column is not clustered or sorted, there is a great opportunity for compressing imprints together. With a 64-bit imprint vector one may encode hundreds, and in many cases thousands, of sequential cachelines. Therefore, the *cacheline dictionary* is needed to keep track of the count of the cachelines and imprints. We define the two structures to store and administer the column imprints index, namely *imp_idx* and *cache_dict* (see Algorithm 2.1). Structure *imp_idx* holds all the constructs needed to maintain the imprints index of one column. It consists of a pointer to the array of the cacheline dictionary (i.e., *cache_dict*), a pointer to the array of the imprint vectors, an array with 64 values that holds the bounds of the bins of the histogram, and the actual number of bins of the histogram. Recall that it may not be needed to have all 64 bins if the cardinality is small, e.g., an 8-bit imprint vector may be enough instead of a 64-bit vector. The dictionary structure *cache_dict* is a 4-byte value, split as follows: 24 bits are reserved for the counter *cnt*, 1 bit is to mark if the next imprint is repeated *cnt* times, or if the next *cnt* imprints correspond to one cacheline each. Finally, 7 bits of the cacheline dictionary structure are reserved for future use.

Algorithm 2.1 details the process of creating the column imprints index. Function `imprints()` receives as input a column *col* and its size *col_sz*. The function returns an imprints index structure *imp* containing an array of imprints and the cacheline dictionary. The algorithm works by first calling the `binning()` procedure, which is described in detail later on in the text. The result of the `binning()` procedure is the number of bins needed to partition the values of the columns, and the ranges of the bins. Next, for each value of the column, the `get_bin()` function is invoked in order to determine the bin the current value falls into. The corresponding bit in the imprint vector is then set. If the end of a cacheline has been reached, the current imprint vector must be stored and a new empty one must be created. However, in order to compress

Algorithm 2.1 Main function to create the column imprints index: `imprints()`

Input: column *col* of size *col_sz*
Output: imprints index structure *imp* for column *col*

```
typedef struct cache_dict {
    uint cnt:24;
    uint repeat:1;
    uint flags:7;
} cache_dict;

typedef struct imp_idx {
    cache_dict *cd;
    ulong *imprints;
    coltype b[64];
    uchar bins;
} imp_idx;

struct imp_idx imp;                          /* initialize the column imprints index structure */
char vpc;                                    /* constant values per cacheline */
ulong i_cnt = 0;                             /* imprints count */
ulong d_cnt = 0;                             /* dictionary count */
ulong imprint_v = 0;                         /* the imprint vector */
binning(imp);                                /* determine the histogram's size and bin borders */
for i = 0 → col_sz − 1 do                    /* for all values in col */
   bin = getbin(imp, col[i]);                /* locate bin */
   imprint_v = imprint_v | (1 ≪ bin);        /* set bit */
   if (i mod vpc-1 ≡ 0) then                 /* end of cacheline reached */
     if (imp.imprints[i_cnt] ≡ imprint_v ∧   /* same imprint */
         imp.cd[d_cnt].cnt < max_cnt − 1) then   /* cnt not full */
       if (imp.cd[d_cnt].repeat ≡ 0) then
         if (imp.cd[d_cnt].cnt ≠ 1) then
           imp.cd[d_cnt].cnt − = 1;          /* decrease count cnt */
           d_cnt + = 1;                      /* increase dictionary count d_cnt */
           imp.cd[d_cnt].cnt = 1;            /* set count to 1 */
         end if
         imp.cd[d_cnt].repeat = 1;           /* turn on flag repeat */
       end if
       imp.cd[d_cnt].cnt + = 1;              /* increase cnt by 1 */
```

cont. to next page

```
cont. from previous page
    else                                          /* different imprint than previous */
        imp.imprints[i_cnt] = imprint_v;
        i_cnt += 1;
        if (imp.cd[d_cnt].repeat ≡ 0 ∧
            imp.cd[d_cnt].cnt < max_cnt − 1) then
          imp.cd[d_cnt].cnt += 1;                 /* increase cnt by 1 */
        else
          d_cnt += 1;                             /* increase dictionary count d_cnt */
          imp.cd[d_cnt].cnt = 1;                  /* set count to 1 */
          imp.cd[d_cnt].repeat = 0;               /* set flag repeat off */
        end if
    end if
    imprint_v = 0;                                /* reset imprint for next cacheline */
  end if
end for
```

consecutive imprints, the algorithm checks if the imprint vector is equal to the previous one. If so, the count *cnt* field of the cacheline dictionary and the *repeat* flag is updated as follows. If the *repeat* of the previous entry in the cacheline dictionary is not set and the count *cnt* is greater than 1, a new entry is created. If the *repeat* of the previous entry is not set but the count *cnt* is 1, then the *repeat* is set and the count *cnt* is incremented to 2. If the imprint vector of the current cacheline is not equal to the previous one, then a slightly different procedure is followed to update the entries in the cacheline dictionary. If the current entry does not have the *repeat* flag set, then the counter *cnt* is simply increased. Otherwise, a new entry is created with count $cnt = 1$ and the *repeat* unset. After this, the cacheline dictionary is correctly updated and the imprint stored. Finally, a new imprint vector is created with all the bits off, the next value of the column is fetched, and the process is repeated.

2.2.5 Binning and Efficient Binary Search

Algorithm 2.2 describes the implementation of the `binning()` procedure. Given a column *col*, a uniform sample of 2048 values is created. Afterwards, the sample is sorted and all duplicates are removed. At this point the size of the sample *smp_sz* might be smaller than 2048. If *smp_sz* is less than 64, the cardinality of the column can be approximated to be equal to the number of unique values found in the sample. Therefore,

Algorithm 2.2 Define the number of bins and the ranges of the bins of the histogram: `binning()`

Input: imprints index structure *imp*, column *col*
Output: number of bins *imp.bins* and the ranges *imp.b*

```
coltype *sample = uni_sample(col,2048);          /* sample 2048 values */
sort(sample);                                    /* sort the sample */
smp_sz = duplicate_elimination(sample);          /* remove duplicates */
if (smp_sz < 64) then                            /* less than 64 unique values */
  for i = 0 → smp_sz − 1 do
     imp.b[i] = sample[i];                       /* populate b with the unique values */
  end for
  if (i < 8) then imp.bins = 8;                  /* determine the number of bins */
  else if (i < 16) then imp.bins = 16;
  else if (i < 32) then imp.bins = 32;
  else imp.bins = 64;
  end if
  for i = i → 63 do
     imp.b[i] = coltype_MAX;                     /* default value */
  end for
else                                             /* more then 64 unique values */
  double y = 0, ystep = smp_sz/62;
  for i = 0 → 62 do
     imp.b[i] = sample[(int)y];                  /* set ranges for all bins */
     y+ = ystep;
  end for
  imp.b[63] = coltype_MAX;
end if
```

each bin of the histogram can contain exactly one value. Even if this approximation is not precise, there is an extremely slim possibility to be much off. In such a case, simply more than one value will fall into the same bin. The next step of the algorithm is to fill the *b* array with the unique values of the sample, and to set the number of the *bins* to the next larger power of 2. Moreover, the remaining empty bins are assigned the maximum value of the domain. This is needed in order for the `get_bin()` procedure to work properly. If the total number of unique values of the sample is 64 or more, we need to divide the bins into larger ranges. This is done by dividing the *smp_sz* by 62 and assigning the result of the division to *ystep*. Notice that *ystep* is a double. This is necessary in order to guarantee an even spread of the ranges of the bins. For example, if the result of the division is 1.2, then the 5th bin should contain the 6th value of the sample, but if we kept the result as an integer, i.e., *ystep* $= 1$, the 5th value of the sample would be assigned to the 5th bin. Each bin *b* is assigned to be equal to the next *ystep* sampled value, until all bins are set. When done, `binning()` returns control to the `imprints()` function.

In order to determine the bin a value falls into, `get_bin()` is invoked. Algorithm 2.3 details the implementation. The approach is to implement a cache-conscious binary search over the 64 bins. For this, we use nested if-statements instead of a for-loop. We noticed during our experimentation that by explicitly unfolding the code for the binary search and by using if-statements without any else-branching, the search can become three times faster, or even more. This is because each if-statement is independent allowing the cpu to execute the branches in parallel. For this, three macros are defined. The macro `middle()`, checks if a value falls inside a range, and two others, called `left` and `right`, check if a value is smaller or larger than a range boundary. The algorithm then is constructed by repeatedly dividing the search space into half, and invoking the `right`, `middle` and `left` macros, in that order. Since there are no else-statements, many if-statement may evaluate to be true, but only the last assignment of the return variable *res* will hold. For this reason the search is performed by starting from the 63rd bin and decreasing.

2.2.6 Complexity Analysis

The algorithms to construct the column imprint index are short and optimized to be cpu friendly. The complexity of `imprints()` function is linear to the size of the column. Assume that a column has n values, and each cacheline contains c values. The most costly part is the call to the `get_bin()` function which performs 3 comparisons before dividing the search space in half, thus it needs $3 \times \log 64 = 18$ comparisons for each value. Therefore, for creating the entire imprint index we need $18 \times n$ comparisons. The call to `binning()` also involves one scan of the n values of the column but the

Algorithm 2.3 Binary search with nested if-statements to locate the bin which a value falls into: `getbin()`

Input: imprints index structure *imp*, value *v*
Output: the bin where value *v* falls into

```
middle(v, p):  if (v ≥ imp.b[p − 1] ∧ v < imp.b[p]) res = p;
left(v, p):    if (v < imp.b[p])
right(v, p):   if (v ≥ imp.b[p − 1])

right(v, 32)
      right(v, 48)
            right(v, 56)
                  right(v, 60)
                        right(v, 62)
                              res = 62;
                              right(v, 63)
                                    res = 63;
                        middle(v, 61)
                        left(v, 60)
                              res = 60;
                  middle(v, 59)
                  left(v, 58)
                        res = 58;
      ⋮
middle(v, 31)
left(v, 30)
      right(v, 16)
            right(v, 24)
      ⋮           ...
```

rest of the operations are independent of the input. Finally, the update of the cacheline dictionary is only performed $\frac{n}{c}$ times, and the cost is negligible (5 comparisons in the worst case) compared to `get_bin()`. During our experimentation we thoroughly studied the effects of different design and implementation choices. Here, we presented the one that performed the best.

2.3 Imprints Query Evaluation

In this section we present the algorithms for evaluating range queries over the column imprints index. Given a range query $\mathcal{Q} = [low, high]$, all values v in column *col* that satisfy $low \leq v \leq high$ need to be located. Since our setting is a columnar database, it suffices to return the *id* list of the qualifying values v.

Evaluating range queries over column imprints is a straightforward procedure. The first step is to create an empty bit-vector and set the bits that correspond to the bins that are included in the range of query $\mathcal{Q}$. There might be more than one bits set, since the query range can span multiple bins. The query bit-vector is then checked against the imprint vectors, and if bitwise intersection indicates common bits set for both the query and the imprint vector, the corresponding cacheline is accessed for further processing. However, if all bits set correspond to bins that are fully included in the query range $[low, high]$ the cacheline need not be checked at all. Otherwise, the algorithm examines all values in the cacheline to weed out false positives. Finally, because of our compression schema, some administrative overhead to keep the cachelines and the imprint vectors aligned is needed.

Algorithm 2.4 presents the details for evaluating a range query using imprints. The constant *vpc* is set equal to the number of values that fit in a cacheline. This is needed to align *id*s with the cachelines. In addition, counters *i_cnt* and *cache_cnt* are maintained to align imprints and cachelines, respectively. Next, two bit-vectors are produced, namely *mask* and *innermask*. The *mask* is a bit-vector that sets all bits that fall into the range $[low, high]$. The *innermask* is a bit-vector with only the bits that fall entirely inside the query range set. More precisely, if a bin range contains one of the borders of the query range, the corresponding bit is not set. Therefore, if an imprint vector has only the bits from the *innermask* set, then all values in the corresponding cacheline fall into the query range and no further check for false-positives is needed. The algorithm runs by iterating over all entries in the cacheline dictionary. If the *repeat* flag is not set, then the next *cnt* imprint vectors correspond to *cnt* distinct cachelines. For any of these imprints, if there is at least one bit set in the same position as the ones in the *mask* bit-vector, the cacheline contains values that satisfy the query range. If in addition, there are no bits set different than the bits of the *innermask*, then all the

Algorithm 2.4 Evaluate range queries over the column imprints index: `query()`

Input: imprints index structure *imp*, column *col*, query $\mathcal{Q} = [low, high]$
Output: array *res* of *id*s

```
char vpc;                                          /* constant values per cacheline */
ulong i_cnt = 0;                                   /* imprints count */
ulong cache_cnt = 0;                               /* cacheline count */
ulong id = 0;                                      /* ids counter */
ulong *res;                                        /* large enough array to hold the result */
(mask, innermask) = make_masks(imp,[low,high]);
for i = 0 → d_cnt − 1 do                           /* iterate over the cacheline dictionary */
  if (imp.cd[i].repeat ≡ 0) then                   /* if repeat is not set */
    for j = i_cnt → i_cnt + imp.cd[i].cnt − 1 do
      if (imp.imprints[j]&mask) then               /* if imprint vector matches mask */
        if ((imp.imprints[j]& ~innermask) ≡ 0) then
          for id = cache_cnt × vpc → (cache_cnt × (vpc + 1)) − 1 do
            res = res ← id;                        /* add id to the result set res */
          end for
        else                                       /* need to check for false-positives */
          for id = cache_cnt × vpc → (cache_cnt × (vpc + 1)) − 1 do
            if (col[id] < high ∧ col[id] ≥ low) then
              res = res ← id;                      /* add id to the result set res */
            end if
          end for
        end if
      end if
      cache_cnt + = 1;                             /* increase cache count by 1*/
    end for
    i_cnt + = imp.cd[i].cnt;                       /* increase imprint count */
  else                                             /* repeat is set */
    if (imp.imprints[i_cnt]&mask) then             /* if imprint vector match mask */
      if ((imp.imprints[f_count]& ~innermask) ≡ 0) then
        for id = cache_cnt×vpc →
                          (cache_cnt × vpc) + vpc × imp.cd[i].cnt − 1 do
          res = res ← id;                          /* add id to the result set res */
        end for
```

cont. to next page

cont. from previous page

```
      else                                        /* need to check for false-positives */
        for id = cache_cnt×vpc →
                              (cache_cnt × vpc) + vpc × imp.cd[i].cnt − 1 do
          if (col[id] < high ∧ col[id] ≥ low) then
            res = res ← id;                       /* add id to the result set res */
          end if
        end for
      endif
    end if
    i_cnt + = 1;                                  /* increase imprints count by 1 */
    cache_cnt + = imp.cd[i].cnt;                  /* increase cache count */
  end if
end for
```

values of the cacheline satisfy the query. In any other case, we need to check each value of the cacheline individually. For all qualifying values, the corresponding *id*s are materialized in the result array. If however the *repeat* flag is set, then by checking only one imprint vector we can determine if the next *cnt* cachelines contain values that fall into the range of the query. As before, an extra check with the *innermask* bit-vector may result in avoiding the check of each individual value for false-positives.

Algorithm 2.4 returns a materialized list of the *id*s that satisfy the range query. This list is then passed to the next operator of the query evaluation engine. However, it might be the case that a user's query contains many predicates for more than one attribute of the same relation. In this case, the `query()` procedure of Algorithm 2.4 is invoked multiple times, one for each attribute, with possible different $[low, high]$ values. The most expensive part of Algorithm 2.4 is the check for false-positives and the materialization of the *id*s. But in the case of multiple range queries over many columns of the same table, both of these expensive operations can be postponed. This technique is known in the literature as late materialization. To achieve this, instead of producing the materialized *id* lists, Algorithm 2.4 has to return the list of the qualifying cachelines. After every range query is evaluated over the respective columns, the lists of cachelines are merge-joined, resulting in a smaller set of qualifying *id*s. This is based on the general expectation that the combination of many range queries will increase the selectivity of the final result set. After the merge-join, the qualifying *id*s that were common to all cachelines can be checked for false-positives. Note that the alternative indexing schemes used in the evaluation of Section 2.6 have been coded with the same

rigidity.

2.4 Updating Column Imprints

Column imprints are designed to support read intensive database applications. In such scenarios, updates are a relatively rare event, and when they occur, are performed in batches. The most common type of updates is appending new rows of data to the end of a table. Column imprints can easily cope with such updates. However, we can not exclude from our study updates that change an arbitrary value of a column, or insert/delete a row in the middle of a table.

2.4.1 Data Append

During data appends, any index that is based on bit vectors and bit-binning techniques has to perform two operations. The first one is to readjust, if necessary, the borders of the bins. Such a readjustment should be avoided since it calls for a complete rebuild of the index. For column imprints, this is very rare, since *i)* the first and last bins are used for overflow values, and *ii)* the bins were determined by sampling the active domain of the column. Any new data appended, need to have dramatically different value distribution to render the initial binning inefficient. The second operation is to update the bit vectors. For bitmap indexes this is a costly operation, since all bit vectors have to be updated, even those that are not mapping the new values [18]. For column imprints this is not necessary. The imprint vectors are horizontally compressed, thus data appends simply cause new imprint vectors to be appended to the end of the existing ones, without the need of accessing any of the previous imprint vectors.

2.4.2 Imprints and Delta Structures

In place updates are never performed in columnar databases because of the prohibitive cost they entail. Instead, a delta structure is used that keeps track of the updates, and merges them at query time. A delta structure can be as simple as two tables with insertions and deletions that need to be union-ed and difference-ed, respectively, with the base table, or it can be a more complex structure, such as positional update trees [34].

Column imprints can cope with inter-column operations, such as unions and differences, by first applying them to the cacheline dictionaries, such that a candidate list of qualifying cachelines is created for both operands. The details of inter-column operations are out of the scope of this paper, and are left to be presented in the future. Nevertheless, even without such a functionality, column imprints can be used to access

the base table to create a candidate list of qualifying cachelines. The underlying delta structure may then hold in addition the cacheline counter where an update has been performed in order to merge to the final result.

Moreover, imprints can produce false positives, thus a deletion can be ignored by the corresponding imprint vector. An insertion however, will call for additional bits to be set to the imprint corresponding to the affected cachelines. Such an approach will eventually saturate the imprint index. In these cases, it is not uncommon to disregard the entire secondary index and rebuild it during the next query scan. The overhead for rebuilding an imprint index during a regular scan in minimal, such that it will go undetected by the user.

2.5 Related work on Bitmap Indexes

Column imprints can be viewed as a new member of the big family of bitmapped based indexes. Bitmapped indexes have become the prime solution to deal with the dimensionality curse of traditional index structures such as B-trees and R-trees. Their contribution to speed up processing has been credited to Patrick O'Neil through the work on the Model 204 Data Management System [60, 61]. Since then, database engines include bitmapped indexes for both fast access over persistent data and as intermediate storage scheme during query processing, e.g. Sybase IQ, Postgresql, IBM DB2, Oracle. Besides traditional bitmaps, Bloom filters [12] have been used to decide if a record can be found in a relation, and thus postponing bringing the data into memory. However, Bloom filters are not suited for range queries, the target of column imprints.

Bitmap indexing relies on three orthogonal techniques [84]: binning, encoding and compression. Binning concerns the decision of how many bit vectors to define. For low cardinality domains, a single bit vector for each distinct value is used. High cardinality domains are dealt with each bit vector representing a set of values. The common strategy is to use a data value histogram to derive a number of equally sized bins. Although binning reduces the number of bit vectors to manage, it also requires a post analysis over the underlying table to filter out false positives during query evaluation. Column imprints use similar binning techniques.

Since each record turns on a single bit in one bit vector of the index only, the bitmaps become amendable to compression. Variations of run-length encoded compression have been proposed. The state-of-the art approach is the Word-Aligned Hybrid (WAH) [82, 85] storage scheme. WAH forms the heart of the open-source package FastBit[1], which is a mature collection of independent tools and a C++ library for index-

[1] http://crd-legacy.lbl.gov/~kewu/fastbit/

ing file repositories. Consequently, column imprints use another variation of run-length encoded but for identical cacheline mappings instead of consecutive equal values.

Bitmap indexing has been used in large scientific database applications, such as high-energy physics, network traffic analysis, lasers, and earth sciences. However, deployment of bitmap indexing over large-scale scientific databases is disputed. [75] claims that based on information theoretic constructs, the length of a compressed interval encoded bitmap it too large when high cardinality attributes are indexed. The storage size may become orders of magnitude larger than the base data. Instead, a multi-level indexing scheme is proposed to aid in the design of an optimal binning strategy. They extend the work on bit binning [28, 83]. Alternatively, the data distribution in combination with query workload can be used to refine the binning strategy [45, 19].

With the advent of multi-core and gpu processors it becomes attractive to exploit data parallel algorithms to speed up processing. Bit vectors carry the nice property of being small enough to fit in the limited gpu memory, while most bit operations nicely fall in the SIMD algorithm space. Promising results have been reported in [27]. Similar, re-engineering the algorithms to work well in a flash storage architecture have shown significant improvements [81].

2.6 Experimental Evaluation

We performed an extensive experimental study to gain insights into the applicability of the imprints index, the storage overhead and creation time, as well as the query performance. We compare our index with two state-of-the-art commonly used secondary index solutions, namely *zonemaps* and *bit-binning* with *WAH encoding*. We also provide, for a baseline comparison, the time measurements for sequential scan. In order to study the impacts of different value types, different column sizes, and different value distributions, we used real world datasets gathered from various test cases. These datasets are either publicly available or part of in-house projects.

Column imprints, zonemaps, and WAH are all implemented in C, and the code is available for download from the MonetDB software repository. The implementation of zonemaps and WAH follow the same coding style and rules as imprints to ensure fairness of comparison. Each experimental run is done by first copying a column into main memory, and then creating the zonemap, imprints and WAH indexes. The timer is always started during the snippets of code that implement each index, thus avoiding measuring administrative overhead, which may not be common for all indexes. We report the wall-clock time as returned by the timing facilities of the `time.h` library of C. All code has been compiled with the *clang* compiler with optimization level 3.

Zonemaps are implemented as two arrays containing the min and max values of

Dataset	Size	♯Col	Value types	Max rows
Routing	5.4G	4	int, long	240M
SDSS	6.2G	4008	real, double, long	47M
Cnet	12G	2991	int, char	1M
Airtraffic	29G	93	int, short, char, str	126M
TPC-H 100	168G	61	int, date, str	600M

Table 2.1: Dataset statistics.

each zone. The size of the zones is chosen to be equal to the size that each imprint vector covers, i.e., the size of the cacheline. The min and max arrays are aligned with the zone numbering, i.e., the first min and max values correspond to the minimum and maximum values found in the first zone, and so on. For the bit-binning approach of bitmaps, the bins used are identical to those used for the imprints index, as described in the `binning()` procedure of Algorithm 2.2. Using this binning scheme, each value of the column sets the appropriate bit on a vector large enough to hold all records. To compress the resulting bit-vectors we apply WAH compression with word size 32 bits, as described in [82].

All experiments were conducted on an Intel^{R} Core^{TM} i7-2600 cpu @ 3.40GHz machine with 8 cores and 8192 KB cache size. The available main memory was 16 GB, while the secondary storage was provided by a Seagate^{R} Constellation^{TM} SATA 1-TB hard drive and capable of reading data with a rate of 140MB/sec.

2.6.1 Data Analysis

We start the presentation of our experimental analysis by first describing in detail the datasets used. We then introduce a novel metric, called *column entropy*, to quantify the clustering property of the values of a column.

Table 2.1 lists the name, the size in gigabytes, the total number of columns, the column types, and the maximum number of rows of the datasets used for our experimentation. The first dataset, denoted as Routing, is a collection of over 240 million geographical records (i.e., longitude, latitude, trip-id, and timestamp) of "trips" as logged by gps devices. The next dataset, SDSS, is a 6.2 GB sample of the astronomy database SkyServer. This database contains scientific data, with many double precision and floating point columns following a uniform distribution, thus stressing compression techniques to their limits. Cnet is a categorical dataset describing the properties of technological products. All data are stored on a single but very wide table, where each column is very sparse, thus presenting ample opportunities for compression. The dataset was re-

created based on the study of J.Beckham [7]. The Airtraffic delay database represents an ever growing data warehouse with statistics about flight delays, landing times, and other flight statistics. The data are updated per month, leading to many time-ordered clustered sequences. Lastly, we used the TPC-H benchmark dataset with scale factor 100, in order to compare against a well recognizable dataset.

2.6.2 Column Entropy

We wish to better study the properties of the columns that are typically not ordered, part of very wide tables, and eligible for secondary indexing. Our initial motivation was based on the observation that "secondary data" exhibit some degree of clustering, either inherited during the creation process of the data, or indirectly imposed by the few columns that are ordered because of primary indexing. Column imprints are designed such that this clustering is naturally exploited without the need of explicit configuration. This is why imprints are built per block and compressed row-wise per imprint vector, instead of vertically per bin. To better understand and quantify the degree of clustering found in data, we define a new metric, called *column entropy*. Column entropy measures how close a column is to being ordered, or, in other words, the amount of clustering found in a column when the values are partitioned into bins. More formally, column entropy $\mathcal{E}$ is defined to be

$$\mathcal{E} = \frac{\sum_{i=2}^{n} d(i, i-1)}{2 \times \sum_{i=1}^{n} b(i)}$$

where $d(i, i-1)$ is the *edit distance* between bit-vector i and $i-1$, and $b(i)$ is the number of bits that are set in bit-vector i. We define the edit distance between two bit-vectors to be the number of bits that need to be set and unset in the first bit-vector in order to become the second. Column entropy $\mathcal{E}$ takes values between 0.0 and 1.0. The higher the entropy $\mathcal{E}$ the more random the data is and the less clustered it appears to be.

SDSS	Routing	Airtraffic	Cnet	TPC-H 100
profmean	*latitude*	*AirlineID*	*attr18*	*p_retailprice*
$\mathcal{E} = 0.794$	$\mathcal{E} = 0.312$	$\mathcal{E} = 0.351$	$\mathcal{E} = 0.200$	$\mathcal{E} = 0.228$

Figure 2.3: Prints of column imprint indexes ('x' = bit set, '.' = bit unset) and the respective column entropy $\mathcal{E}$.

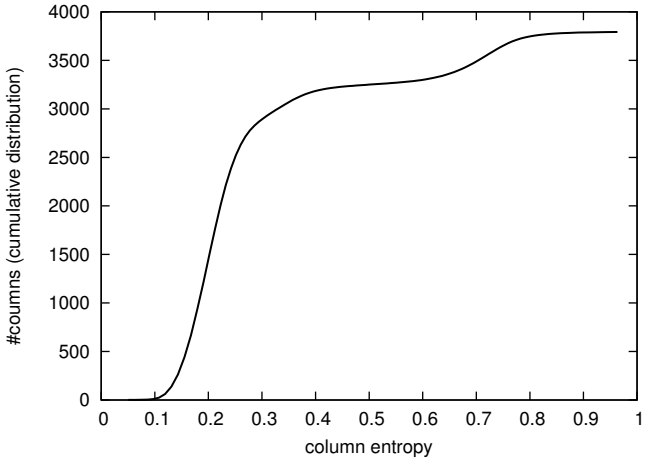


Figure 2.4: Cumulative distribution of the columns' entropy $\mathcal{E}$.

To give a more intuitive view of *column entropy*, we print a small portion of the column imprint index of five columns, one from each dataset, and list them in Figure 2.3, together with their respective entropy value $\mathcal{E}$. The prints in Figure 2.3 correspond to the actual imprint indexes as constructed in our code for the experiments. If a bit is set then an `'x'` is printed, otherwise an `'.'`. The first column imprint of Figure 2.3 corresponds to a column from the SkyServer dataset. It is of type real and has a high entropy value of almost 0.8 which implies that each next imprint vector is significantly different from the previous one. Such columns with high entropy, as demonstrated in the next section, are harder to compress. The next imprint is the latitude attribute of the Routing dataset. It exhibits nice clustering properties, something to be expected since the dataset is taken from real observations, and thus trips are continuous without any jumps, unless the trip-id changes. The next two imprints are taken from the Airtraffic and Cnet dataset. These are categorical datasets, with low cardinality – hence the smaller bit-vectors – and with low entropy value. The last imprint index is the *retail_price* attribute of table *part* of TPC-H. This dataset is created to contain a sequence of prices that are not ordered, but they are still the same repeated permutation of an order. Such an organization of data resembles closely an ordered column, and thus also has a low entropy value.

Figure 2.4 depicts the cumulative distribution of the entropy $\mathcal{E}$ for all columns of all datasets that we used in our experiments. We exclude all columns that are less

than 1 megabyte in size, since they are of minimal interest and introduce outliers in our measurements. More than 3000 columns have entropy smaller than 0.4, thus supporting our claim that data often tend to exhibit good local clustering and ordering properties. Nevertheless, there are almost a thousand columns that have high entropy values, up to almost 1.0. Those columns are not to be ignored since they sum up to over 20% of the total data. A secondary index should be immune to such high entropy, and still be able to take advantage of any opportunities for compression. In the next section we study the storage overhead of imprints and other state-of-the-art secondary indexes, while giving emphasis to their behavior on columns with high entropy. We show that imprints are robust against columns with high entropy, while bitmaps with WAH fail to achieve a good compression rate.

2.6.3 Index Size and Creation Time

We analyze the storage overhead introduced by the column imprints index and compare it with that of zonemaps and WAH. The upper row of the graphs in Figure 2.5 depict the sizes of the indexes over all columns and all datasets. Each graph corresponds to a different value type. For presentation reasons, we divide the types according to their size in bytes. For example, char is 1-byte, short is 2-byte, int and date are 4-byte, and long and double 8-byte types. The y-axis depicts the size of the indexes measured in megabytes, starting from a few bytes for the smaller columns to almost one gigabyte for the large ones. Notice that y-axis is *log-scaled*. The x-axis enumerates the columns according to their size (in increasing order). Because many columns have exactly the same size, since they originate from the same tables, we distinguish them by placing them next to each other. As a result, the flat horizontal patterns appearing in the graphs correspond to different columns of the same size, while the "stepping" effect corresponds to the next group of larger columns.

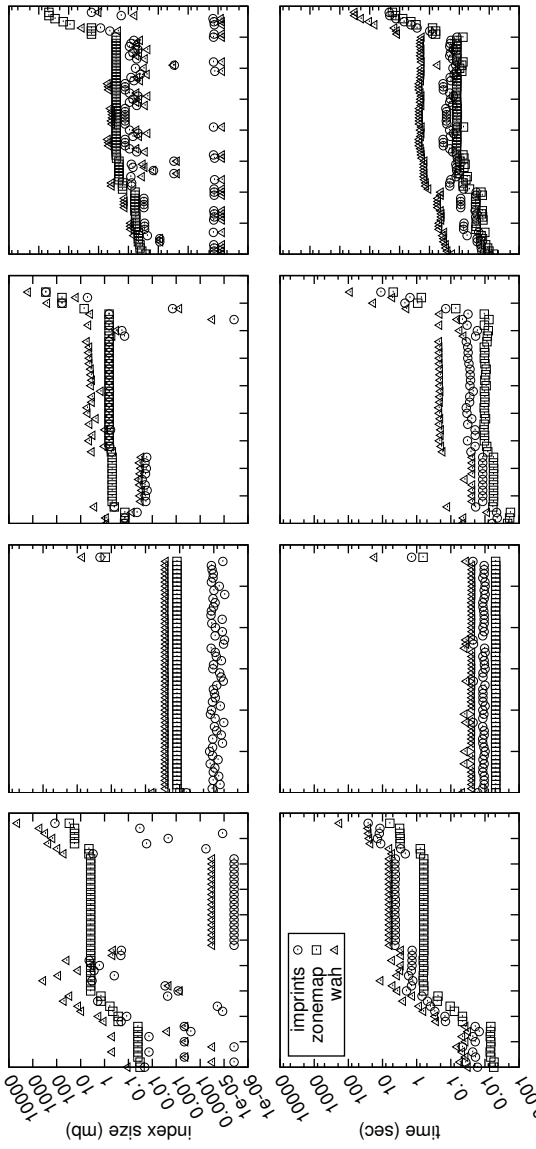


Figure 2.5: Index size and creation time for different types of columns (x-axis enumerates the columns, ordered by size).

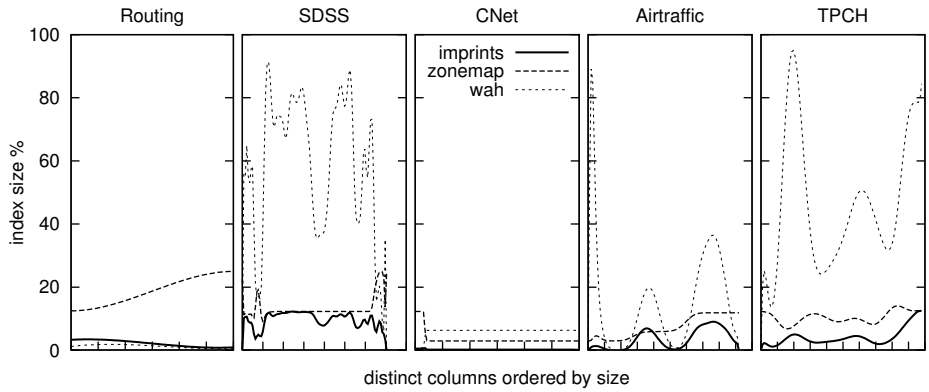


Figure 2.6: Index size overhead % over the size of the columns.

The triangle points of the plots in Figure 2.5 mark the size of the bit-binning index with WAH compression, the squares mark the size of zonemaps, and the circles mark the size of the column imprints index. The general picture drawn for all types is that WAH index entails the largest storage overhead, zonemaps come second, while imprints have the least requirements of storage space. More specifically, the general trend is that imprints are between one and two orders of magnitude more space efficient than zonemaps and WAH. However, there are exceptions to that rule, especially for WAH indexing, which depicts the biggest fluctuation in storage needs. For 1-byte types, there are cases where WAH achieves better compression and reaches that of imprints. By examining the data closer we noticed that this is true for columns that although they have more than 126 million rows (taken from the Airtraffic dataset), they only contain two distinct values, thus allowing both WAH and imprints to fully compress their bit-vectors. Another point of interest is found in the case of 8-byte types, where WAH can become slightly more space efficient than imprints. This is true for those columns that contain primary keys (e.g., bigint identifiers) and in addition are ordered. Although we are studying secondary indexes that typically apply to unordered columns, we did not exclude any ordered columns from our experimental datasets for completeness.

Since it is impractical and hard to explicitly show the size of each individual column, we compute the percentage of the size of the indexes over the size of the column. Figure 2.6 shows such a graph. In addition, instead of grouping on value type, we group columns from the same datasets together, such that more insights about the different applications, and hence different value distributions can be gained. The categorical

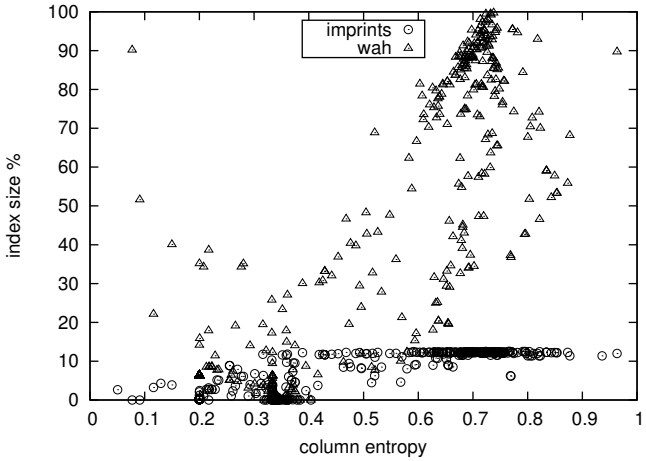


Figure 2.7: Index size overhead % over column entropy $\mathcal{E}$.

dataset Cnet which has columns with low cardinality, as well as the nicely clustered routing dataset, achieve the best compression for both imprints and WAH, thus requiring – in many cases – less than 10% space overhead. However, the same can not be said for broader value domains with uniform distributions. Specifically, the scientific dataset of SkyServer, consisting of many columns with real and double values, with high cardinality and no apparent clustering, makes the WAH index very unstable and induces high storage overhead. Imprints perform fairly stably and much better than WAH, with space overhead closer to zonemaps. The failure of WAH is expected due to the increasing random values in SkyServer, which allows for very few compression opportunities. However, imprints do not suffer from the same problem. Since one imprint vector is constructed for each cacheline, the space requirements are less than bitmaps, while the chance of consecutive imprint vectors to be identical, and thus compressible, is increased.

Figure 2.7 depicts the index size overhead of both imprints and WAH as percentage of the size of the column, ordered over the entropy $\mathcal{E}$. Imprints achieve storage overhead less than 10% for columns with low entropy, i.e., up to 0.4. The same observation holds with few exceptions for WAH indexing. However, the picture changes for columns with entropy of 0.5 and higher. Imprints exhibit a steady storage overhead that does not exceeds 12%. WAH indexing suffers more, with up to almost 100% of storage overhead on the size of the column. Imprints on the one hand need at most 64

bits per cacheline unit, making them immune to high entropy, while benefiting from low entropy. On the other hand, WAH can potentially become very inefficient. If there are very few opportunities for compression, most 32-bit words will be aligned with 31-bit literals, i.e., no big long sequences of same bits will be found in the bit-vectors. In addition, since we use a 64 bit-binning approach, there will potentially be 64 uncompressed bits per value. All in all, WAH is more suitable for low entropy data, while imprints are more stable and with better compression for the entire range of entropy values, i.e., they work even if data are not locally clustered.

Another concern is the time spent to create each secondary index. The bottom row of graphs in Figure 2.5 depicts the creation time for WAH, zonemaps, and imprints. As expected the zonemaps are the fastest to create. For each row only two comparisons have to be made to determine the minimum and the maximum values for the current zone. The slowest is the WAH index, since there is significantly more work to be done in order to compress the bit-vectors. Imprints on the other hand, always perform between zonemaps and WAH. The overall differences of the construction time between the three indexes is steady and to be expected since each of them require a different amount of work per value. Most importantly, the time for all indexes increases linearly to the size of the columns, thus making them a cost-effective solution for secondary indexing.

2.6.4 Query Performance

Next, we turn our attention to the performance analysis of evaluating range queries. The execution scenario for this set of experiments is as follows. For each column, ten different range queries with varying selectivity are created. The selectivity starts from less than 0.1 and increases each time by 0.1, until it surpasses 0.9. These 10 queries are then fired against the three indexes (i.e., zonemaps, WAH, and imprints) defined over the column, and also evaluated with a complete scan over that column. The result set of each query is a materialized ordered list of *id*'s. The ordering of *id*'s is guaranteed by the sequential scan, the zonemap index, and the imprints index. However, this is not true for WAH, since each pass over the different bit-vectors will produce a new set of *id*'s which needs to be merged. The merging is done by defining another bit-vector aligned with the *id*'s. The bits that are set in this *id* bit-vector correspond to the *id*'s that satisfy the range query. In this way no final merge is needed, just the materialization of the *id*'s. This implementation only adds a small, but necessary for fairness, overhead to WAH compared to the other indexes.

Figure 2.8 plots the query times of over 40,000 queries evaluated over each index. The queries are ordered on the x-axis according to their selectivity. If the selectivity is 0.1, the query returns 10% of the total values in the column, while 0.9 returns 90%

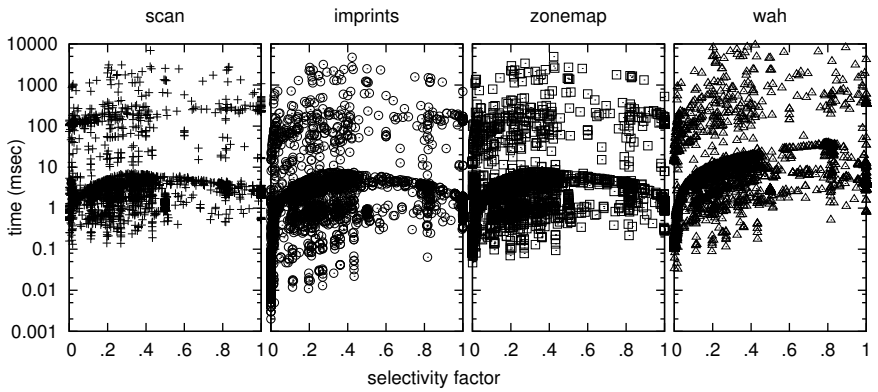


Figure 2.8: Query time for decreasing selectivity.

of the total values. All three indexes and the sequential scan produce the same graph patterns for query times. However, these patterns are shifted along the y-axis. Imprints is the fastest index overall since the points in the graph are shifted the most to the bottom. As expected, if the selectivity of the query is low and thus more data are returned, the smaller the differences that are observed between indexes. In fact, sequential scans then also become competitive. This is due to the fact that the overhead of decompressing the data, and materializing almost all of the *id*'s, surpasses the time needed to sequentially scan the entire column and check each value. In addition, zonemaps exhibits query times similar to that of sequential scan for low selectivity queries, since zonemaps require the least administration overhead compared to imprints and bitmaps with WAH.

To better understand the behavior of zonemap, WAH, and imprints, for queries with low selectivity, and compare them with sequential scans, we plot in Figure 2.9 the cumulative distribution of the queries over time. More precisely, we count the queries that finish execution at each time frame, and cumulatively sum them up. The steeper the graph in Figure 2.9 the more queries finish in a shorter time, thus the more efficient the index is overall. Figure 2.9 shows that almost $15,000$ queries need each of them less than 0.1 milliseconds to be evaluated with imprints index. Zonemaps, which is the second best, manage to evaluate just over 7,500 queries in the same time frame. However, as the evaluation time increases the time gap between the different approaches is reduced.

Further, we are interested in the factor of improvement that is achieved by the

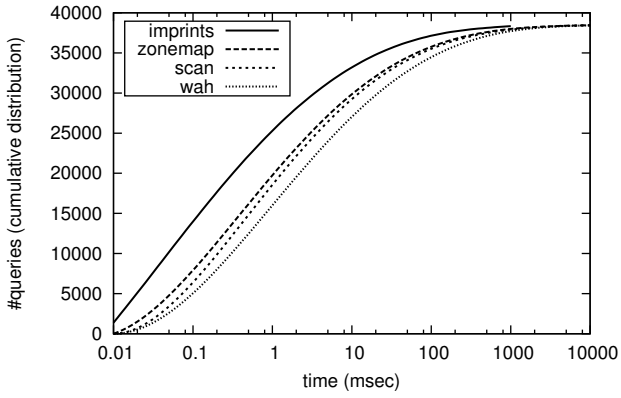


Figure 2.9: Cumulative distribution of query times.

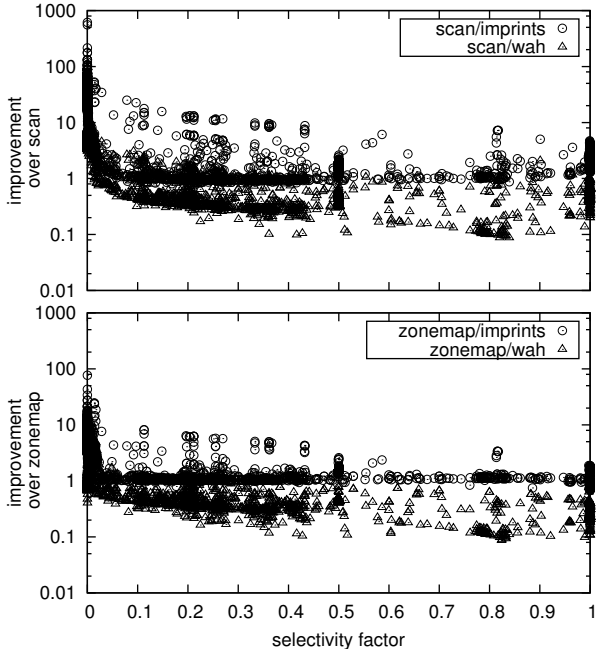


Figure 2.10: Factor of improvement over scan and zonemap.

imprints index over the sequential scan baseline and the competitive zonemap indexing. Figure 2.10 depicts the factor of improvement achieved for each query. A point above 1 is translated as a factor of improvement over the baseline, while a point below 1 shows how many times an approach is slower than the baseline. The upper graph of Figure 2.10 shows with circle points the improvement of imprints over sequential scans, while the triangles, the corresponding improvement of bitmaps with WAH over sequential scans. Both imprints and WAH, show a significant improvement for queries with high selectivity, i.e., when less than 20% of the tuples are returned. For imprints that improvement is in some cases almost a 1000 times faster, and for WAH over 10. However, for queries with low selectivity, imprints become less competitive, while WAH can become significantly slower than scans. This observation is aligned with the strategy of most modern database systems, where, if the cost model of the query optimizer detects a select with low selectivity, a sequential scan is preferred over any index probing. Moreover, WAH is punished in a main memory setting. The processing overhead of the WAH compression outweighs the throughput of data that is achieved from main memory to the cpu cache. Therefore, WAH is more suitable for cases where data do not reside in memory, but need to be fetched from disk. Similarly, the bottom graph of Figure 2.10 depicts the same comparison, but with zonemap indexing being the baseline, instead of sequential scans. The same trend can be seen here, although zonemaps is more competitive and thus the improvement factor for imprints is closer to 100 times. However, in a few cases of low selectivity zonemaps can become faster than imprints due to less computation needs.

Finally, we compare the number of index probes and data comparisons performed (originating from testing for false positives) normalized over the number of records in a column. This experiment reveals implementation-independent statistics for column imprints in comparison with zonemaps and WAH. The top graph of Figure 2.11 shows the number of index probes, while the bottom the number of comparisons, for all queries with selectivity between 0.4 and 0.5. The number of index probes for WAH is the highest of all indexes, almost always more than the number of total records. This is true since for each record many bit vectors have to be probed. However, WAH achieves the best filtering since the number of data comparisons is usually very low. On the other hand, zonemaps have a steady number of index probes, i.e., exactly the number of cachelines of the column. The number of comparisons for zonemaps depends on the data skew and can vary. Column imprints achieve a balance between index probes and data comparisons. Columns with high entropy entail more index probes but less data comparisons. On the other hand, columns with low entropy will need less index probes but more data comparisons.

In conclusion, for high selectivity queries column imprints index can achieve a factor of 1000 improvement over sequential scans, and a factor of 100 over zonemap.

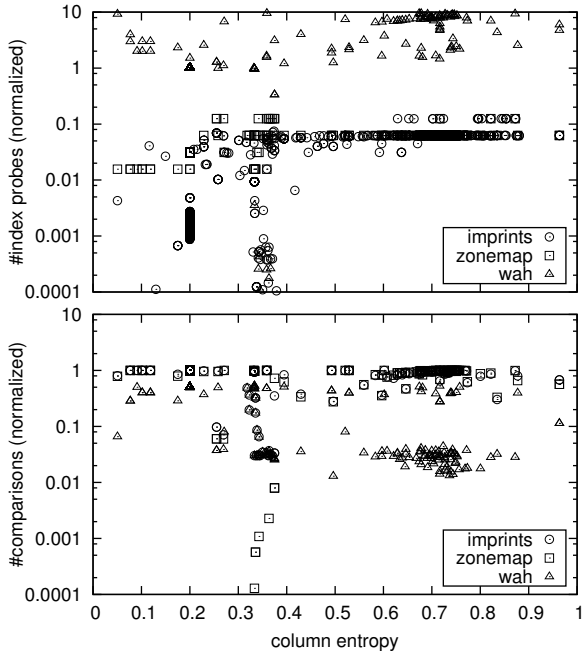


Figure 2.11: Number of index probes and value comparisons for queries with selectivity between 0.4 and 0.5.

Further experimentation, revealed that there is a correlation between the query evaluation time and the sizes of the column, or the size of the index, which in turn is correlated with the column entropy. We do not show these graphs since they do not reveal any new insights into the performance of imprints compared to zonemap or WAH index.

2.7 Summary

Column imprints is a light-weight secondary index with a small memory footprint suited for a main-memory setting. It belongs to the class of bitvector indexes, which has a proven track record of improving access in large-scale data warehouses. Our extensive experimental evaluation shows significant query evaluation speed-up against pure scans and the established indexing approaches of zonemaps and bitmaps with bit-binning and WAH compression. The storage overhead of column imprints is just a few percent, with a max of 12% over the base column.

Column imprints can be extended to exploit multi-core platforms during the construction phase and during multi-attribute query processing. Akin to prevailing techniques, such as [75, 81], judicious choice of the binning scheme, and a multi-level imprints organization, may lead to further improvements in specific application domains.

Chapter 3

Memory Efficient Bloom Filters for Skewed Access Patterns

Main memories are becoming sufficiently large that most OLTP databases can fit entirely in memory. Nevertheless, OLTP workloads often exhibit skewed access patterns where some records are hot (frequently accessed) and many records are cold (infrequently or never accessed). It is still more economical to store the cold records on secondary storage. To minimize unnecessary disk accesses an in-memory Bloom filter can be used to quickly check whether a given record exists in cold storage. In this chapter, we show how skew in access patterns can be exploited to improve Bloom filter efficiency (less space and/or lower false positive rate). We do this by splitting the filter into many smaller ones, where each filter covers only a small subset of records. The challenge is, given the (skewed) access frequencies per subset of records, to determine the optimal number of bits per Bloom filter. We define a mathematical model and show how to optimally size the Bloom filters. We also describe how to adjust the sizing of the filters because of changes in access patterns or deterioration caused by updates. Furthermore, since with this split the Bloom filters are smaller, it is faster to create and query them because of fewer cache misses. An extensive experimental evaluation confirms that our algorithms construct better Bloom filters optimized for skewed access patterns, that they achieve a lower false positive rate or lower memory consumption depending on the settings, and that they can adapt gracefully to data and workload changes.

3.1 Motivation

Database systems have traditionally been designed assuming that data is disk resident. However, the drop in memory prices over the past 30 years is invalidating this assumption. Several database engines have emerged that optimize for the case when most data fits in memory [30, 37, 63, 77, 78]. This architectural change necessitates a rethink of all layers of the system, from concurrency control [47] and access methods [66, 72] to query processing [11].

In OLTP workloads record accesses tend to be skewed. Some records are "hot" and accessed frequently while others are "cold" and accessed infrequently. Clearly, good performance depends on the hot records residing in memory but cold records can be moved to cheaper external storage such as flash with little effect on overall system performance.

Consider, for example, a table containing customer orders. The most recent, active orders are stored in an in-memory table – the hot table – with two hash indexes, one on order number and one on customer number. Older, closed orders are accessed rarely and, to reduce memory requirements and overall system cost, are moved to another table – the cold table – stored on a flash SSD. The cold table also has hash indexes on order number and on customer number. Suppose orders that have been closed for more than 30 days are automatically moved to the cold table.

To retrieve all orders for a customer we must check both the hot table and the cold table. For customers that have no orders older than 30 days, accessing the cold table is wasted effort. A typical way to avoid such unnecessary accesses is to use an in-memory Bloom filter, in this case, one on customer number that covers customers with orders in the cold table. Before accessing the cold table, we do a quick check against the Bloom filter and if the result is negative, we have avoided an expensive lookup in the cold table.

3.1.1 Contributions

Instead of a single large Bloom filter per index, we propose to use a collection of smaller Bloom filters, called hereafter *Split Bloom filters*. Each smaller filter covers a fixed set of hash buckets (in the case of a hash index) or a fixed set of pages (in the case of a B+tree index). Each filter may, for example, cover two consecutive buckets (pages). The split Bloom filters are of variable size, where more bits are allocated to buckets (pages) that contain many distinct keys and/or are frequently accessed. Using a collection of smaller Bloom filters is a simple idea but it offers several advantages.

- We can exploit skew by allocating more bits to filters that are frequently accessed

and/or cover larger subsets of records. We have developed an algorithm (section 3.4) to determine the optimal allocation of bits to filters. Our experiments confirm that the split Bloom filters are more efficient (less space and/or lower false positive rate) than a single large Bloom filter.

- It is faster to build a collection of split filters than a single large filter and checking a small filter is faster because of fewer cache misses. This is confirmed by our experiments.

- When the set of records covered by a filter changes due to insertions and deletions, the filter no longer performs optimally (too many bits set). When this occur we can remedy the situation simply be rebuilding the filter. This is cheaper than rebuilding a single large Bloom filter, because each split filter covers only a small subset of records.

- Split Bloom filters also make it possible to adapt to changes in access frequencies. If a split filter becomes more frequently accessed and/or produces more false positives than expected, we can rebuild the filter and increase its size. Vice versa, if a filter is hardly ever accessed we can rebuild it and reduce its size, thereby saving memory space. How to detect and adapt to changing access frequencies is discussed in section 3.5.

This work is part of a research project investigating how to manage cold data in Hekaton, an OLTP database engine optimized for main-memory and integrated into SQL Server [17, 26]. The goal is to enable Hekaton to automatically migrate cold records to cheaper secondary storage while still providing access to such records transparently. Needless to say, it is important to avoid unnecessary accesses to secondary storage. We considered using classical Bloom filters but found them too inflexible. We also considered counting Bloom filters but deemed them too space inefficient and not able to exploit skew.

3.1.2 Outline

The rest of this chapter is organized as follows. Next section summarizes related work on Bloom filters. Section 3.3 presents the system infrastructure. Section 3.4 details the process of splitting Bloom filters. In Section 3.5 we study the different cases that call for a total or partial rebuild of Split Bloom filters. The experimental evaluation of the proposed algorithms is presented in Section 3.6. We conclude our work with Section 3.7.

3.2 Related Work on Bloom Filters

Bloom filters where introduced in 1970 by Burton H. Bloom [12]. Since then, numerous extensions to the original design have been proposed. One of the most notable extensions are spectral Bloom filters [23] which use counters instead of single bits and can support multi-sets. Instead of only setting a bit whenever an element is hashed to a position, the spectral Bloom filters increase a counter associated with that position, thus allowing correct support of both count queries and deletion of elements from the filter.

The Bloomier filter [21] is an extension of Bloom filters that supports membership checks over static functions that divide the domain into several subsets. Given an element, it decides if it belongs to set $A, B, C, \ldots$ or none of them. To achieve this, multiple Bloom filters are created for each subset, hence it performs better if the number of subsets is small. Compressed Bloom filters [57] study the problem of optimizing the compression rate of a Bloom filter instead of only optimizing the false positive ratio. The author demonstrates that it is better to use fewer hash functions to achieve better compression rates since there will be more unset bits.

Several other variants of Bloom filters aimed at making them more space and time efficient have been proposed in the literature [64, 69]. In [69] the authors propose a cache efficient variant that uses blocks of bits and precomputed bit patterns to be set. In [43] only two hash functions are used to produce the k needed, resulting in less cpu time. The false positive ratio can be significantly reduced by using partitioned hashing [31]. This is done by first defining many different families of hash functions, and then hashing an element to determine to which family it belongs. These functions are then used as in the traditional Bloom filters.

Scalable Bloom filters [3] address the problem that arises if the number of elements to be inserted in a Bloom filter is unknown, yet a strict false positive ratio needs to be enforced. The authors proposed a scalable growth of the Bloom filter by appending bits as needed.

Weighted Bloom filters [16] and Conscious Bloom filters [89] have both been proposed to deal with skewed workloads. In Weighted Bloom filters, the authors propose to use different number of hash functions for each key depending on the frequency it is looked up and the likelihood to be a member, instead of decreasing or increasing the number of bits used, as per our solution. Similarly, the authors of Conscious Bloom filters present a $(2 + \epsilon)$-Approximation algorithm to decide how many hashes to use. Weighted and Conscious Bloom filter do not facilitate updates or adjusting to changes in the access skew, since they use a single large filter instead of many smaller ones that are easier to maintain.

3.3 Avoiding Trips to Cold Storage

For each table that resides in memory, there is a corresponding table in the cold store for records that are infrequently accessed. Records are never duplicated – a record is either in the hot store or in the cold store. A cold table may have any number of indexes, each defined for a different search key of the records. A record may be accessed through any index so we need multiple filters, one for each index.

A straightforward approach is to build a single large Bloom filter that covers an entire index of the cold table. The filter would use a predefined number m of bits, called hereafter the *memory budget*, and k hash functions to identify which bits to be set. A Bloom filter performs optimally when the number of false positives is minimized. It has been shown [12] that the optimal balance between the number of hash functions k, the number of bits m and the number of distinct keys n is achieved when $k = \ln 2 \frac{m}{n}$.

Here, instead of maintaining a single large Bloom filter per index, we split it into a collection of smaller Bloom filters that add up to the same memory budget m. Each of the smaller Bloom filters may be of different size. The number of bits for a filter is determined based on *i)* how frequently the records covered by the filter are accessed, and *ii)* the number of (distinct) keys hashed. The intuition is to assign more bits to Bloom filters that contain more keys and/or are queried more often, which is where the bits yield the most benefits. There is little benefit in using a large filter for a small set of keys or for keys that are hardly ever queried, because even if it returns a false positive half of the time, the overall impact on the false positive rate of the entire index is negligible.

Obviously, our approach benefits from skewed access patterns; access skew provides opportunities to redistribute the memory budget so as to achieve a lower false positive rate than a single large Bloom filter. Aging data, where the older a record becomes the less likely it is to be queried, exhibits such skew and this is a fairly common pattern.

Before presenting our mathematical model and the algorithms for optimal sizing each Split Bloom filter, we detail the over all system design for two different types of indexes, namely *hash indexes* and *B+trees*.

3.3.1 Split Bloom Filters for Hash Indexes

Consider an in-memory table with n records of customer orders where the column *order_id* is a unique key and the column *customer* identifies the customer that placed the order, as illustrated in Figure 3.1. Older orders are moved to the cold store and indexed on *customer* using a hash index. Given a query that retrieves all orders placed by a customer, we first search the in-memory hot table, possibly using a hash index on

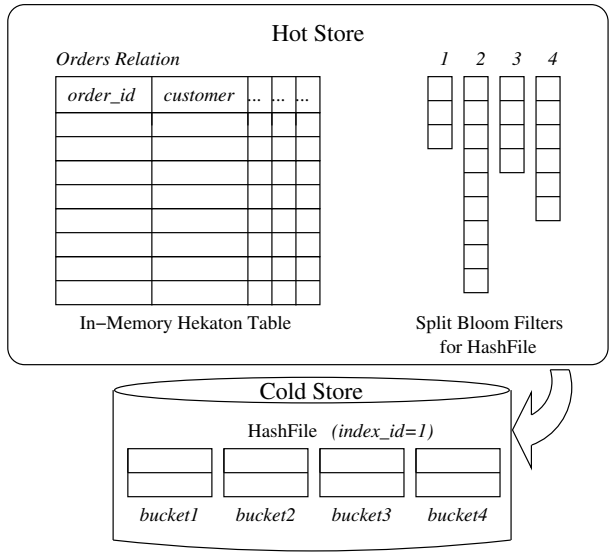


Figure 3.1: In-memory Split Bloom filters for a cold store resident hashfile index.

customer. Next, we scan the appropriate bucket of the hash index in the cold store to find the customer's older orders.

To avoid an unnecessary access to the cold store, we cover a predefined number of hash buckets in the cold store with a smaller Bloom filter. The choice of how many hash buckets to cover per filter is a space versus efficiency trade-off. Smaller filters covering fewer buckets are easier to maintain, while fewer larger filters reduce management overhead. In Figure 3.1 there are four different hash buckets in the cold store. Suppose we split the Bloom filter into four smaller ones each covering one hash bucket. Given the *customer*, we first compute the hash value of the key to determine to which hash bucket it corresponds to in the cold store. But, before accessing the cold store, we probe the corresponding Bloom filter, i.e., the one that covers the target hash bucket, to determine whether a trip to the cold store can be avoided.

3.3.2 Split Bloom Filters for B+Tree Indexes

Consider an index that supports ranges, such as BW-trees [49], a novel lock-free version of in-memory B+trees. Such a B+tree variation index over the *customer* column of the orders table is shown in Figure 3.2. A query about all orders of a customer will

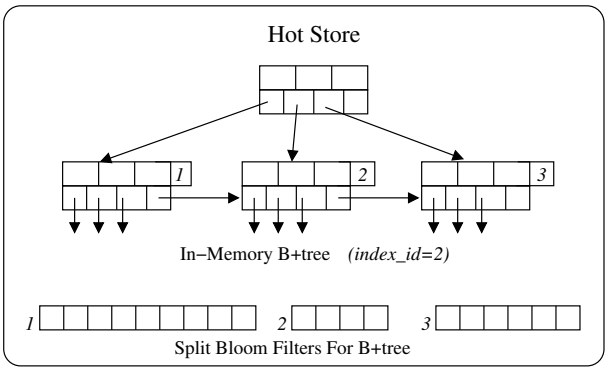


Figure 3.2: In-memory Split Bloom filters for a B+tree index.

traverse the B+tree until it reaches the leaf nodes that point to the records in the orders table. The next step is to retrieve the corresponding pages in the cold store with the older orders of the same customer. To make use of Split Bloom filters and avoid unnecessary access of pages in the cold store, we add to the leaf nodes a field, *filter_id*, that identifies which Split Bloom Filter covers the node's key range. Multiple adjacent leaf nodes may share the same filter, in particular, leaf nodes covering key ranges with few records in the cold store. For leaf nodes with no records in the cold store *filter_id* is set to null. If *filter_id* is set, we locate the corresponding Bloom filter and check whether it is necessary to access the cold store.

3.3.3 Split Bloom Filters Construction

We store Split Bloom filters in a regular in-memory table to take advantage of the system's transactional guarantees. This ensures that the filters will be durable and consistent with the data in the cold store. We use one Split Bloom filter table per cold table, thus storing in this table all filters for all indexes of that particular cold table. Therefore, for each cold table, there is one Split Bloom filter table in memory. One could also use one Bloom filter table per index, per database, or even a single system-wide table.

An example of a Split Bloom table is shown in Figure 3.3. Each Bloom filter is identified by a unique combination of the identifier of the index, which is given by the catalog information of the system, and the id of the Split Bloom filter. For example, the first record with identifier 1.1 of the Split Bloom filter table in Figure 3.3 corresponds

index.filter	n	k	m	f	cnts	Bloom filter
1.1	$n_{1.1}$	$k_{1.1}$	3	$f_{1.1}$	...	110
1.2	$n_{1.2}$	$k_{1.2}$	9	$f_{1.2}$	...	100111010
1.3	$n_{1.3}$	$k_{1.3}$	4	$f_{1.3}$	...	1010
1.4	$n_{1.4}$	$k_{1.4}$	6	$f_{1.4}$	...	100110
2.1	$n_{2.1}$	$k_{2.1}$	9	$f_{2.1}$	...	100101110
2.2	$n_{2.2}$	$k_{2.2}$	4	$f_{2.2}$	...	1100
2.3	$n_{2.3}$	$k_{2.3}$	6	$f_{2.3}$	...	101010

Figure 3.3: In-memory Split Bloom Filters table.

to the hash index with id=1 and the Split Bloom filter with id=1. The id of the Split Bloom filter is identified as follows. For a hash index, each Split Bloom filter covers a fixed number of hash buckets, say w buckets, so the Split Bloom filter id of a bucket b is simply $\lfloor b/w \rfloor$. In the case of a B+tree, the id is given by the leaf node. The next three attributes of the Split Bloom Table in Figure 3.3 are the number n_i of distinct records in the group of buckets (or group of pages) covered by filter i, k_i the number of hash functions used by that Split Bloom filter, and m_i the number of bits used. The counter f_i is incremented whenever the Split Bloom filter is probed and enables us to estimate the filter's access frequency. Next, there are a number of attributes, denoted as cnts in Figure 3.3. The details of these counters are given in Section 3.5 where we discuss Split Bloom filters maintenance strategies. Finally, the last attribute of the Split Bloom filter table is a variable length bit vector that stores the actual Bloom filter.

The main function for creating the Split Bloom filters over an index of the cold table is detailed in Algorithm **??**. It simply scans the buckets or pages covered by each Split Bloom filter, hashes the keys and sets the selected bits in the filter, and finally creates and inserts a record with the filter in the Split Bloom filter table. In the next section we determine the parameters to optimally size each Split Bloom filter according to the frequency and the number of distinct keys it covers.

3.4 Optimal Sizing of Split Bloom Filters

We wish to split a memory budget of m bits into l Bloom filters such that the total false positive ratio is *minimized*. Denote the size (in bits) of Bloom filter i by m_i. The sum

Algorithm 3.1 Split Bloom filters Construction

```
l = totalBuckets/bucketsPerGroup;
(m[], k[]) = Algorithm 3.2(l, m, n, n[], f[]);
for i ← 1 to l
    /* allocate a new Bloom filter of size m[i]*/
    Bloomfilter = allocate_bits(m[i]);
    for key in buckets i×bucketsPerGroup until
                        (i + 1)×bucketsPerGroup
        for h ← 1 to k[i]
            hash_h(key)→Bloomfilter;
        end for
    end for
    id= (IndexId << 32) | i;
    new record(id, n[i], k[i], m[i], f[i], Bloomfilter);
    insert(record)→SplitBloomTable;
end for
```

of bits assigned to the filters should equal the memory budget m, that is,

$$m = \sum_{i=1}^{l} m_i.$$

The false positive ratio of the ith Bloom filter can be approximated by

$$p_i = (1 - e^{-k_i n_i / m_i})^{k_i} \tag{3.1}$$

where k_i is the number of hash functions used, and n_i the number of distinct keys included in the ith Bloom filter.

To be able to estimate the access frequencies of the filters, each filter has a counter f_i that is incremented on every access to the filter. Let f denote the sum over all access counters,

$$f = \sum_{i=1}^{l} f_i.$$

The total or *overall* false positive ratio P for the l Bloom filters covering an index is equal to

$$P = \sum_{i=1}^{l} \frac{f_i}{f} p_i. \tag{3.2}$$

That is, the overall false positive rate of a collection of Bloom filters is a weighted average of the false positive ratios of the individual Bloom filters, weighted with their respective fraction of total accesses.

To minimize P, optimal values for all $m_i, i = 1, \ldots, l$ must be determined. The problem amounts to a non-linear minimization problem that can be solved using Lagrange multipliers. Our derivation of the solution is included at the end of this chapter. We derive the following formula for finding the optimal filter size m_i for $i = 1, \ldots, l$

$$m_i = (\ln^{-2} 2\gamma_i + \frac{m}{n})n_i \tag{3.3}$$

where

$$\gamma_i = \ln f_i - \ln n_i + \sigma \tag{3.4}$$

$$\sigma = \frac{1}{n} \sum_{i=1}^{l} n_i(\ln n_i - \ln f_i). \tag{3.5}$$

The optimal size of a Bloom filter is a function of the number of keys n_i and a factor γ_i. γ_i is equal to the difference between the natural logarithm of the frequency f_i and the keys n_i plus a constant σ. σ is the same for all Bloom filters, thus it needs to be computed only once. Once m_i is determined, then k_i is set to $k_i = \ln 2\frac{m_i}{n_i}$.

Algorithm 3.2 implements the mathematical formulas derived here. The input to the algorithm consists of the number of filters l, the memory budget in bits m, the total number n of keys, as well as two arrays, namely $n[]$ with the number of distinct keys per group of hash buckets or group of B+tree pages, and $f[]$ containing the access frequencies.

For the initial build of the Bloom filters the number of distinct keys, $n[]$, can be computed by a scan over the index. If estimates of the frequencies $f[]$ are not available we assume uniform access frequencies.

Algorithm 3.2 starts by first computing σ. For this, l multiplications and subtractions are needed, $2 * l$ calls to function log, and finally one division. Then, for each Bloom filter the parameter γ is computed. The cost for computing γ is $2 * l$ calls to function log, one subtraction and one addition. If γ is zero or negative, then the access frequency is so rare, or the number of unique keys are so few, that no bits need to be used. In this case, a query to this filter will always return true, and a trip to the cold table will not be avoided. If parameter γ is a positive number, the number of bits and hash functions per Bloom filter are computed.

Our model is a continues approximation and thus produces non-integer (i.e., floating point) values instead of integer values for $m[i]$. Therefore, we use function `ceil`

Algorithm 3.2 Sizing Split Bloom filters

Input: number of filters l, total bits to be used m, total keys n, keys per filter $n[]$, access frequencies per filter $f[]$
Output: number of bits per filter $m[]$, number of hash functions per filter $k[]$

```
// first compute σ
float σ = 0;
for i ← 1 to l
    σ = σ + n[i] * (log(n[i]) − log(f[i]));
end for
σ = σ/n;

// next calculate the number of bits
log2 = 1.0/(log(2.0) * log(2.0));
for i ← 1 to l
    γ = log(f[i]) − log(n[i]) + σ;
    if (γ < 0)
      m[i] = 0;
      k[i] = 0;
    else
      m[i] = ceil((log2 * γ + (m/n)) * n[i]);
      k1 = floor(log(2.0) * m[i]/n[i]);
      k2 = ceil(log(2.0) * m[i]/n[i]);
      k[i] = min(fp(k1), fp(k2));
    end if
end for
```

to take the closest integer that is larger than $m[i]$. However, to decide the number of hash function, both options of $\lfloor k[i] \rfloor$ and $\lceil k[i] \rceil$ are checked and the one that produces the smaller false positive ratio p is used. The theoretical false positive ratio can never be exactly achieved, but only approximated by any Bloom filter. This is because of the natural limitation of not being able to use a non-integer number of bits or a fraction of hash functions as the formulas indicate.

3.5 Split Bloom Filter Maintenance

To maintain a low false positive rate it may be necessary to rebuild a few or all filters. In this section we describe our approach to monitoring filter performance and triggering filter maintenance.

3.5.1 Incremental Build of Individual Filters

The algorithms presented in Section 3.4 derive the optimal size for each Bloom filter given the number of keys, the number of accesses, and the available memory budget. However, for reasons of efficiency it is preferred to rebuild the Bloom filter that needs to, instead of rebuilding the entire collection of Split Bloom filters. The incremental build procedure described in this section ensures that the target false positive ratio initially set by Algorithm 3.2 is achieved, even if the access frequency or the number of unique keys have changed for the specific filter to be rebuild. Experiments detailed in Section 3.6 show that by rebuilding filters one at a time we still approach the global minimum as computed by Equation 3.3.

Each filter contributes to the overall false positive by a fraction $\frac{f_i}{f}$ of its own false positive. If that fraction changes because of access skew, or the false positive ratio p_i changes because of updates, then an incremental rebuild must be performed. However, the following equation must hold in order for the overall contribution to remain the same

$$\frac{f_i}{f} p_i = \frac{f'_i}{f'} p'_i \tag{3.6}$$

where f'_i, f' are the new values of access frequency that triggered the incremental rebuild. This observation leads to the following equation for computing the new size of a single Bloom filter

$$m'_i = -n'_i \ln \big(\frac{f'}{f'_i} \frac{f_i}{f} p_i \big) \ln^{-2} 2. \tag{3.7}$$

Algorithm 3.3 Incremental sizing of a Bloom filter

Input: record(n_i, k_i, m_i, f_i) of filter i to be build, FPt_i current false positive target of filter i, Fr_i frequency ratio of filter i

Output: a new record with updated Bloom filter, updates FPt_i and Fr_i

log2 $= 1.0/(\log(2.0) * \log(2.0))$;
// calculate the new false positive target
new_fp = $(f/f_i) \times Fr_i \times FPt_i$;
if (new_fp> 1)
  new_fp$= 1$;
  $m_i = 0$;
  $k_i = 0$;
else
  $m_i = -n_i \times$ log2 $\times \log($new_fp$)$;
  $k1 =$ floor$(\log(2.0) * m[i]/n[i])$;
  $k2 =$ ceil$(\log(2.0) * m[i]/n[i])$;
  $k_i =$ min$($fp$(k1),$ fp$(k2))$;
endif
Bloomfilter = rebuildFilter(i, m_i, k_i);
$FPt_i =$ new_fp;
$Fr_i = f_i/f$;
return new record$(n_i, k_i, m_i, f_i,$ Bloomfilter$)$, FPt_i, Fr_i;

The details of the derivation of this formula are described at the end of this Chapter. Note that n_i' is the updated number of unique keys, thus taking into account any insertion or deletions.

Algorithm 3.3 implements the above formula to incrementally rebuild a filter. The input is the record with the counters and the Bloom filter to be rebuild, as well as counters FPt_i and Fr_i. FPt_i is exactly the p_i of Equation 3.7, and Fr_i is the fraction $\frac{f_i}{f}$. While the current f_i' is the counter f_i stored in the record of the Bloom filter. The first step for Algorithm 3.3 is to compute the new false positive ratio p_i'. Next, if it is smaller than 1.0, the number of bits m_i are computed from the formula. If the new false positive is greater than 1.0, the access frequency is so low that no filter is needed. Finally, the algorithm will rebuild the filter with the newly computed parameters and replace the old record in the Bloom table.

During the incremental rebuild, we only scan the hash buckets or B+tree pages that are covered by the specific filter, and not the entire index. Therefore it is more efficient than a total rebuild. In the experiments we will show that a series of incremental rebuilds approach the same overall false positive ratio that would have been achieved, if we performed a total rebuild of the collection of Bloom filters.

3.5.2 How Bloom Filters Deteriorate

Inserts

When new records are added to the cold store, some Bloom filters may have to be updated to prevent false negatives. False negatives occur when a Bloom filter incorrectly indicates that a requested key does not exist in the cold store even though it does. This produces an incorrect query result which, of course, is unacceptable. The filter update has to be done in a manner that ensures that false negatives cannot occur at any point. We store Bloom filters as a variable length attribute of an in-memory table and take advantage of the transactional infrastructure of the database engine. The filter update is done as part of the same transaction that updates the cold store so the Bloom filter update commits with the cold store update.

Deletes

When a record r_1 is deleted, it is not safe to turn off the bits set by r_1 in a Bloom filter because some other record r_2 still in the cold store may have set one or more of the same bits. Turning off the bits would "hide" r_2 and cause a false negative. To ensure correctness we do not update Bloom filters when a record is deleted. This may, however, increase the false positive rate because the Bloom filter still indicates that r_1

is present. Deletes, same as inserts, will cause Bloom filters to deteriorate and produce a higher false positive rate than necessary. Either too many bits will be set because of insertions or too many bits will remain set despite records having been deleted. At some point the Bloom filters have to be rebuilt in order to ensure optimal performance.

Changes in Access Frequency

Changes in access frequencies may also cause suboptimal performance and call for a rebuild. Suppose the access frequency of a hash bucket or a B+tree page decreases dramatically. If so, the contribution of the Bloom filter covering these buckets (pages) to the total false positive rate is also reduced. The Bloom filter may now be unnecessarily large; it may be better to shrink it and invest the freed up bits elsewhere. Similarly, if the access frequency for a bucket (page) increases, its covering Bloom filter may be too small for optimal performance.

Skewed Access Patterns

There is an additional situation that may call for a rebuild. The formulas for estimating false positive rates and sizing Bloom filters assume that incoming requests select bits at random. In practice that may not be the case. Suppose we have sized a Bloom filter for an expected false positive rate of 5%. Now consider a particular key value K that results in a false positive. If all incoming requests are for key value K, the actual false positive rate will be 100%. This is an unlikely case but it shows that the actual distribution of requests may result in false positive rate that is higher (or lower) than theoretically expected. If the Bloom filter is rebuild with a different size, a different set of bits will be set and chances are that K will no longer cause a false positive.

When a set of Bloom filters have deteriorated sufficiently we rebuild them, possibly also changing their size. A rebuild requires scanning the buckets (pages) covered by the filters, recomputing the filters, and updating the Bloom table with the new filters. This has to be done in a transactional manner to ensure that false negatives cannot occur. Apart from rebuilding a set of filters, one may also incrementally build individual Bloom filters such that the over all false positive target of the collection remains fixed. The process of incrementally building filters, instead of total rebuild, is described later in this section. First we detail the counters maintained and the policies used for triggering a rebuild.

3.5.3 Counters

To be able to monitor Bloom filter performance and react to changes we maintain several counters at different levels. There are six counters for each Bloom filter.

- $I_{idx,i}$, the number of new distinct keys inserted into buckets (pages) covered by filter i of index idx,
- $D_{idx,i}$, the number of distinct keys deleted from buckets (pages) covered by filter i of index idx,
- $Nl_{idx,i}$, the number of lookups in filter i of index idx,
- $Nf_{idx,i}$, the number of false positives produced by filter i of index idx,
- $FPt_{idx,i}$, the theoretical – or *target* – false positive ratio of filter i as computed during the last rebuild of the entire collection of Bloom filters for index idx, or during the last incremental build of filter i, and
- $Fr_{idx,i}$ the access frequency fraction $\frac{f_i}{f}$ of filter i as noted during the last rebuild or incremental build.

Counters $I_{idx,i}$ and $D_{idx,i}$ are used to determine if a filter rebuild is needed because of too many updates. $Nl_{idx,i}, Nf_{idx,i}$ can be used to continuously monitor the real false positive ratio and compare it with the target ratio $FPt_{idx,i}$ so as to be able to detect large deviations between observed and theoretical false positive. In addition, $FPt_{idx,i}$ and $Fr_{idx,i}$ are used for incremental rebuilds.

There are four counters for the collection of Bloom filters covering an index.

- Nl_{idx}, the number of lookups in index idx,
- Nf_{idx}, the number of false positives produced by the filters of index idx,
- E_{idx}, the difference between memory used and the memory budget for the filters of index idx, and
- FPt_{idx}, the theoretical – or *target* – total false positive ratio as computed during the last rebuilt of the entire collection of the Bloom filters for index idx.

When an entire collection of filters for an index is rebuilt we allow their total size to change if this is necessary to maintain the filters' target total false positive ratio FPt_{idx}. Therefore, we also maintain a counter E_{idx} that counts by how many bits the total actual size is over or under the budget. We will later use E_{idx} to assess whether to do a complete rebuild of the index' collection of Bloom filters. In addition, Nl_{idx} and Nf_{idx} are used to monitor the current false positive ratio and how much it deviates from the target ratio.

3.5.4 Triggering Filter Rebuild

The filters covering an index and each filter individually was assigned a target false positive ratio and a memory budget when built. We monitor the actual false positive ratio and trigger a rebuild when it falls outside a predefined range. When rebuilding, we size the filters for the same target ratio but allow memory usage to increase (or decrease) to achieve that target. Excessive memory usage triggers a global rebuild.

Individual Bloom Filters

The goal for individual filters is to keep the false positive ratio close to the target while keeping the actual filter size smaller than the initial.

By maintaining the access counters $Nl_{idx,i}$ and $Nf_{idx,i}$ and the two counters for new distinct keys inserted $I_{idx,i}$ and deleted $D_{idx,i}$, we can monitor the performance of a filter and take action when the performance deviates from the target. We set upper B_u and lower B_l bounds on the deviation from the target ratio and trigger a rebuild when the deviation exceeds the bounds. That is, a rebuild of a filter is triggered whenever

$$Nf_{idx,i}/Nl_{idx,i} > FPt_{idx,i} + B_u \text{ or}$$
$$Nf_{idx,i}/Nl_{idx,i} < FPt_{idx,i} - B_l.$$

As mentioned earlier, access skew may cause the actual false positive ratio of a Bloom filter to be higher than what is theoretically expected. If this is the case, and we rebuild the Bloom filter with the same size, the existing keys will set the same bits as before and the bias may remain. However, if we rebuild the filter with a different size, then a different set of bits will be selected by the hash functions and, most likely, the effects of access skew will be remedied.

We can detect if enough updates have been done to disrupt the balance between set and unset bits by comparing the $I_{idx,i}$ and $D_{idx,i}$ counters with the total number of unique keys in the buckets or pages of index idx. If either $I_{idx,i}$ or $D_{idx,i}$ exceed a given percentage of the number of unique keys then a rebuild of the Bloom filter is triggered. In that case, the size of the filter may need to change too. The incremental algorithm described in the next paragraph achieves these restrictions.

Bloom Filters for an Index

Even if each filter in the collection retains its false positive ratio close to its target, the distribution of accesses may change causing the false positive ratio for the entire collection to deviate from its target.

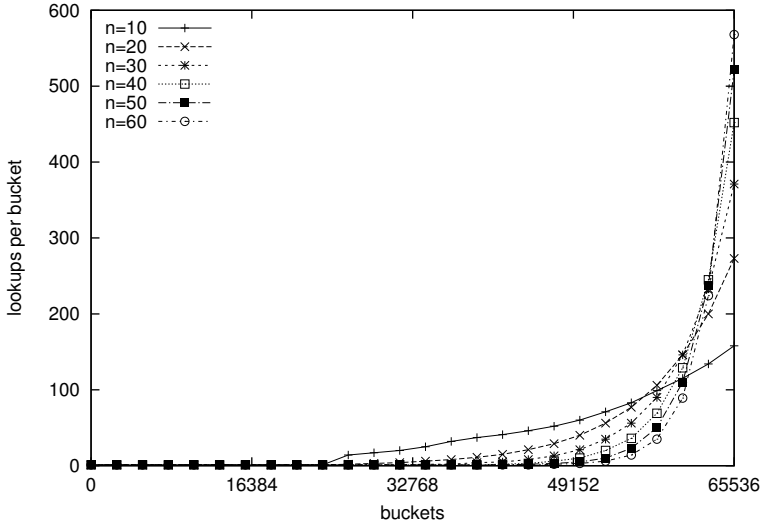


Figure 3.4: Variations of the workload skew used in experiments.

In the same way as for individual filters, we can establish upper and lower bounds on the deviation from the target and trigger a rebuild of all filters for the index. To do so we need to recompute the optimal size. Since rebuilding an entire collection of filters can be costly, it should only be done if we wish to either change the overall false positive ratio or assign a new memory budget. In other cases it is better to rebuild only those Bloom filters that is needed.

3.6 Experimental Evaluation

We performed an extensive experimental study to gain insights into the behavior of Split Bloom filters. We focus on experiments that show the differences between our solution and classical Bloom filters. Since our solution is affected in the same way as a classical filter when varying the number of bits per key, or the number of hash functions, we do not study these parameters in depth. We rather focus on the effects of parameters such as the number of hash buckets per Bloom filter, or the influence of the skew of the workload, which are all specific to our approach.

We implemented our algorithms, classical Bloom filters, and an underlying storage

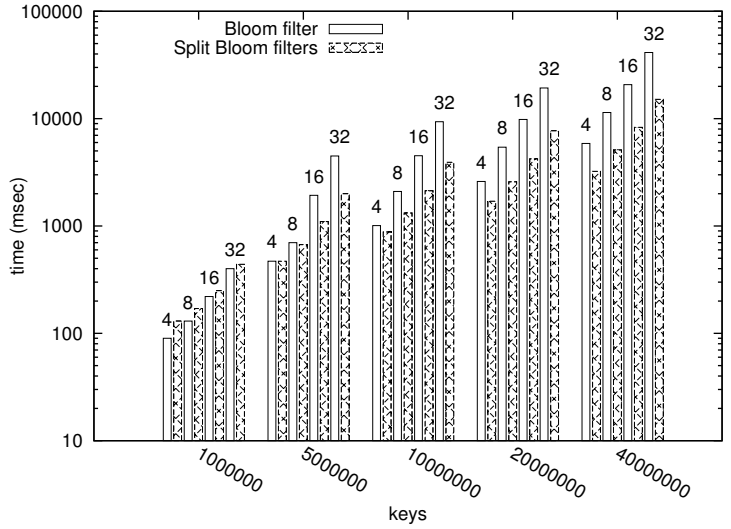


Figure 3.5: Creation time for different number of keys and bits per key.

based on hash files all in C. The timing results reported here are all averaged over three runs, although there were hardly any differences between each run. The data keys were randomly created according to a uniform distribution. We emulate skewed access with a power-law random generator using distribution $P(x) = x^n$, as described in [44]. If not otherwise stated, we used $n = 10$. For reference, Figure 3.4 plots the workload skew output by the random generator for different values of n. Which buckets are to become more popular than others are randomly chosen for each experiment. All experiments were conducted on an Intel$^{\text{R}}$ Core$^{\text{TM}}$ i7-2600 CPU @ 3.40GHz machine with 8 cores and 8192 KB cache size. The available main memory was 16 GB, while the secondary storage was provided by a Seagate$^{\text{R}}$ Constellation$^{\text{TM}}$ SATA 1-TB hard drive and capable of reading data with a rate of 140MB/sec.

3.6.1 Creation

The first experiment measures the creation time for Split Bloom filters and compares it with that of classical Bloom filters. Figure 3.5 plots the results. The figure shows five histogram clusters, each cluster corresponding to a different number of data keys: 1M, 5M, 10M, 20M and 40M. The bars within each cluster represent different filter

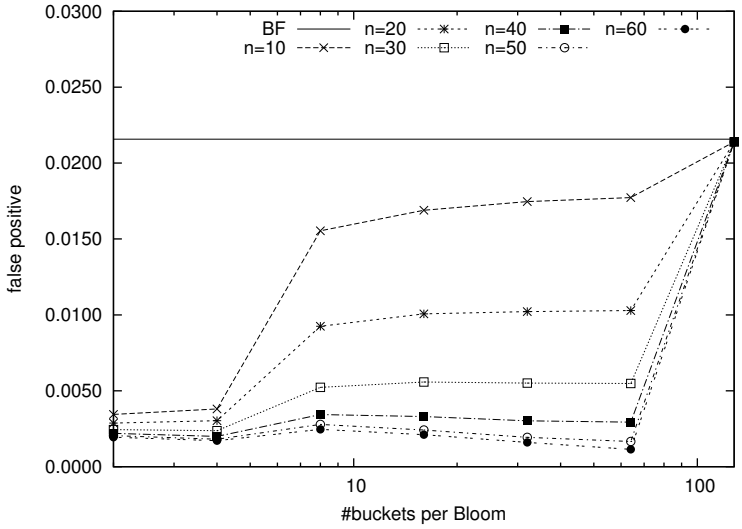


Figure 3.6: False positive ratio for different number of buckets per Bloom filter.

sizes: 4, 8, 16, and 32 bits per key. We plot the creation time (in milliseconds) for both classical Bloom filters and Split Bloom filters. Note that the y-axis is in log scale. Figure 3.5 shows that Split Bloom filters are faster to create than a large single filter. The difference in time increases with the number of keys and the number of bits per key. This is explained by the improved cache locality for Split Bloom filters. Each key added to a Bloom filter needs to set k different bits. The smaller a Bloom filter is, the fewer cache lines it occupies, thus increasing the chance of a hit in the L1 or L2 cache. When the number of keys or the number of bits per key increase, the classical Bloom filters grow and the cache miss rate increases. For the same reasons of cache locality, querying and updating Split Bloom filters is faster than for a single large filter.

The number of hash buckets (or pages in a B+tree) each filter covers influences its ability to adjust to data and access skew, but also affects the total false positive. Figure 3.6 illustrates this effect. The y-axis of shows the overall false positive of a collection of Split Bloom filters while the exponent of the power law distribution of the workload varies between $n = 10$ until $n = 60$. The x-axis plots the number of hash buckets that each Bloom filter covers. In all cases, we used the same number of data keys (10 million) and the same number of total hash buckets $(65,536)$. The solid line plots the false positive ratio of a single Bloom filter that covers all buckets and keys.

The first observation is that Split Bloom filters are able to exploit the skew in access frequencies to reduce the false positive ratio compared to the solid line. Second, the optimal number of buckets that each Bloom filter should cover increases with the exponent of the power law distribution. For example, if $n = 10$, then 2 or 4 buckets per Bloom filter is sufficient to improve the false positive ratio. If more buckets are used per filter, then the overall false positive ratio quickly increases to that of a single Bloom filter. The reason is that the more buckets are used, the more the workload skew is "blurred out". In the case of $n = 60$, where the access skew is high, the overall false positive ratio decreases continually until 64 buckets per Bloom filter. Therefore, the more skewed the workload is, the more buckets can be covered per Bloom filter, which in addition reduces the administration overhead.

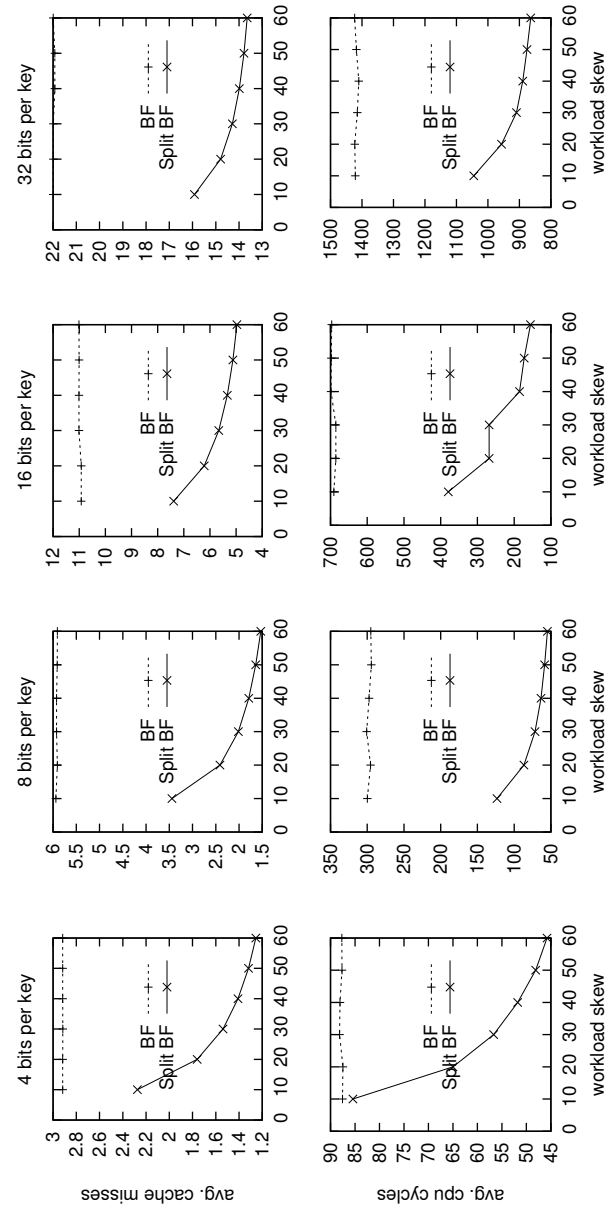


Figure 3.7: Average cache misses and cpu cycles over 10 million queries.

3.6.2 Queries

A disadvantage of an in memory Bloom filter is that it produces a lot of cache misses because of the random assignment of bits per key. One way to overcome this is to create hash functions that for a given key they produce values in the same cacheline range. However, this approach increases the false positive ratio since bits are not assigned truly random any more. Split Bloom filters on the other hand, are smaller by design since they cover less keys, and a smaller filter occupies less cachelines. Therefore, the probability of two or more hash values pointing at bits of the same cacheline is higher.

The next experiment demonstrates these benefits. We performed 10 million queries over a classical Bloom filter and a collection of Split Bloom filters. All queries are point lookups for keys that do exist in the filters, such that we force all hash functions to be computed each time. We report the results of our experiments in Figure 3.7. The upper row of plots show the average cache misses per key lookup. The lower row of plots show the average cpu cycles per key lookup. We repeated the experiments for different number of bits per key, from 4 bits up to 32 bits. An increase on the bits per key, also means an increase of the number of hash functions used per key. The x-axis signifies the skew of the keys stored in the buckets, according to the workloads of Figure 3.4. The goal is to demonstrate the benefits of our approach when skew increases. Figure 3.7 plots the behavior of one large Bloom filter. As expected both the cache misses and the cpu cycles remain steady per plot. However, the Split Bloom filter have a smaller number of cache misses (even for low skew) while the difference with classical Bloom filters increases for larger skewed workloads. This is also in accordance with the results of measuring the cpu cycles per key. Split filters are faster since there are fewer cache misses, but also, for less frequently accessed buckets, there are fewer hash functions to compute resulting too in reduced cpu time.

3.6.3 Access skew

Filters become less effective because of changes in access frequencies, data distribution, or inserts and deletes, and thus they need to be rebuilt. An incremental strategy, rebuilding a few filters at a time, is clearly preferable to a strategy of complete rebuilding that uses Algorithm 3.2 to compute optimal sizes. However, this raises two questions about the incremental rebuild strategy that we proposed in Section 3.5. Does it converge to a solution that is close to the optimal solution? And if it does, how rapidly does it adapt? The experiments discussed in this section are designed to shed some light on these two issues.

The first experiment investigates whether the incremental approach converges to an optimal solution. We first create a hash index with 10 million unique keys and 16,384

Bloom filters using 8 bits per key. The stored keys are uniform distributed over the buckets. We then ran 1 million lookups selecting the target buckets according to a skewed workload with $n = 10$. During the lookups we apply the incremental rebuild strategy of Algorithm 3.3. Once the lookups complete we also compute the optimal filter sizes using Algorithm 3.2. The results are plotted in Figure 3.8. The x-axis enumerates the 16,384 Bloom filters, sorted from the least frequently accessed to the most frequently accessed. The y-axis of the upper graph shows the false positive ratio per filter and of the lower graph shows the filter's size in bits. The plot line marked *Init* shows the initial situation when access frequencies are uniform. The line *IB* shows the false positive ratio and the filter size distribution after 1 million lookups and using the incremental rebuild Algorithm 3.3 to resize filters. The least accessed filters on the left of the graph are virtually never accessed so they end up with no bits and thus a false positive ratio of 1.0. However, while the access frequency of filters increase (towards the right side of the x-axis), more bits are allocated to the filters to keep their false positive ratio close to the target. The line *RB* shows the distribution of the false positive ratio and filter size when we rebuild the entire collection of Bloom filters using Algorithm 3.2. The *IB* and *RB* lines are very close, which demonstrates that the incremental rebuild Algorithm 3.3 converges towards the optimal solution provided by Algorithm 3.2.

Figure 3.9 shows the evolution of the total false positive ratio and the total size of the Bloom filters when the filters are periodically rebuilt using three different strategies. A rebuild is initiated every $10,000$ lookups. The x-axis indicates the number of lookups, up to $300,000$ in total. The *Init* line shows what happens if the collection of Bloom filters is rebuilt completely without changing the memory budget. The *IB* line shows the results when Bloom filters are rebuilt incrementally, keeping the target false positive ratio for each filter unchanged. The *RB* line corresponds to doing a complete rebuild every $10,000$ lookups but by using the same total memory space as the incremental build. The lower graph of Figure 3.9 plots the total size of the filters for all three settings. The upper graph plots the corresponding overall false positive ratio.

The filters corresponding to the *Init* always use the same amount of memory as shown by the straight line. Meanwhile, the overall false positive ratio drops as shown by the *Init* line in the upper graph of Figure 3.9. This is caused by bits being removed from rarely accessed filters and given to more popular ones. Consequently, the total size remains the same but the overall false positive ratio is reduced and eventually converges to a minimum. The Bloom filters maintained with incremental rebuild keep the overall false positive ratio stable and close to the target, while freeing bits that are no longer needed. The overall false positive ration is steady as shown in the upper graph of Figure 3.9, while the total size converges to a minimum as illustrated by the lower graph. The *RB* line shows the results if completely rebuilding the Bloom filters

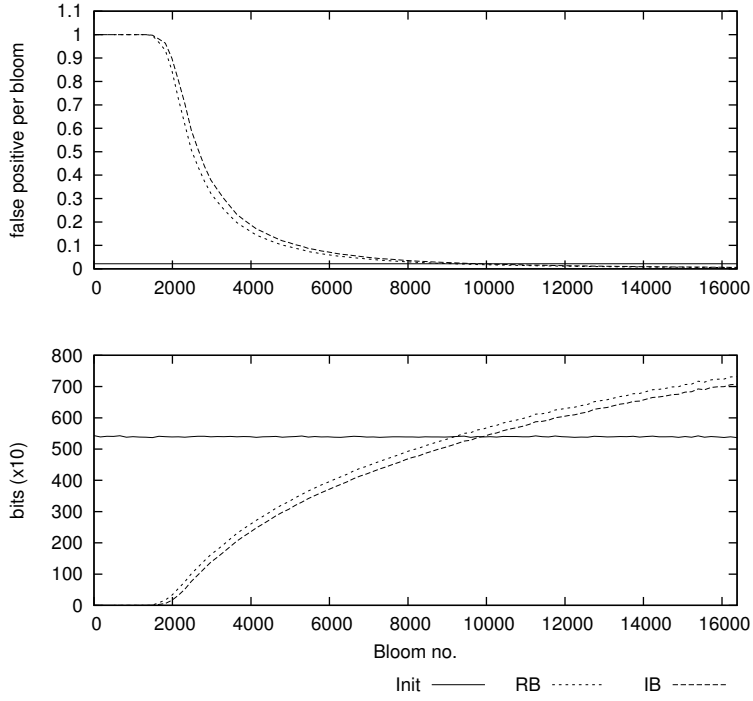


Figure 3.8: Distribution of false positive ratio and filter size across the Bloom filters of one index.

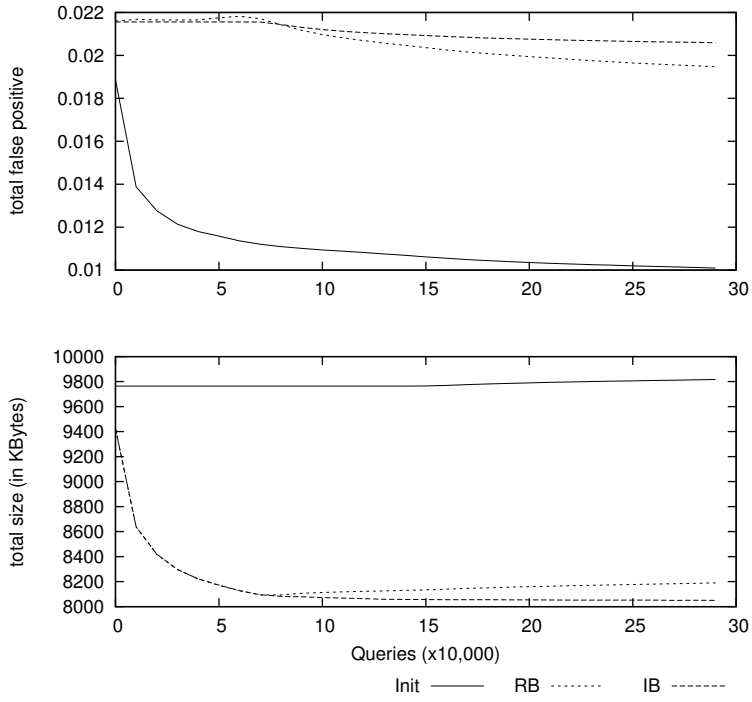


Figure 3.9: Evolution over time for a collection of Bloom filters without changing the access pattern.

after every $10,000$ lookups using the same memory space as the incrementally rebuilt filter. The *IB* and *RB* lines are very close, which confirms that the incremental approach produces filters that are close to optimal.

The next experiment demonstrates how the incremental rebuild process adjusts to frequent changes in access patterns. The results plotted in Figure 3.9 showed that the incremental rebuild strategy responds well when there is a sudden shift in access frequencies but also that it takes time to adjust. In this experiment we simulate "moving hotspots", that is, the skew in access frequencies remains the same but we change which buckets are popular. We vary how frequently the hotspot shifts and study the effects.

To be able to adapt to changes in access patterns, we need a mechanism to "forget history". We have defined a simple rule for when to erase the history of access frequency. *If the total space used by the Bloom filters increases as a result of an incremental rebuild, then erase the previous history, that is, zero the counters associated with the filters.* By doing so the incremental build algorithm only takes into account recent popular Bloom filters without the picture being blurred by filters that were popular in the past.

Figure 3.10 plots the results of this experiment. The solid lines show the total size of the Bloom filters in KB while the dashed lines show the overall false positive ratio. The false positive ratio remains stable while memory consumption fluctuates because of the shifting focus. The uppermost graph of Figure 3.10 is for the case when the focus changes every $1,000$ lookups. Because change is so rapid compared with the rebuild interval, memory consumption remains high, in the range $9,400$ KB to $9,500$ KB. In the middle graph of Figure 3.10 the focus change occurs every $10,000$ lookups so the incremental build algorithm has more time to adjust. In this case, the total memory consumption varies between $9,450$ KB and $8,950$ KB. Finally, the lowermost graph of Figure 3.10 shows the case when the focus changes every $100,000$ lookups. The incremental rebuild algorithm now has even more time to rebuild filters and reduce the total memory consumption before the focus changes again. The total memory consumption goes down to $8,700$ KB.

The experiments in this section show that as long as the focus remains steady the Bloom filters will reduce their memory consumption, while still being able to adjust to more frequent shifts but at the cost of additional memory space.

3.6.4 Updates

As described in Section 3.5 Bloom filters may deteriorate over time due to updates. In this set of experiments we are interested in studying the effects of updates, and whether our solution can react and adjust to them. To investigate this we ran an experiment with a stream of 3 million lookups. The lookups followed a uniform access distribution, i.e.,

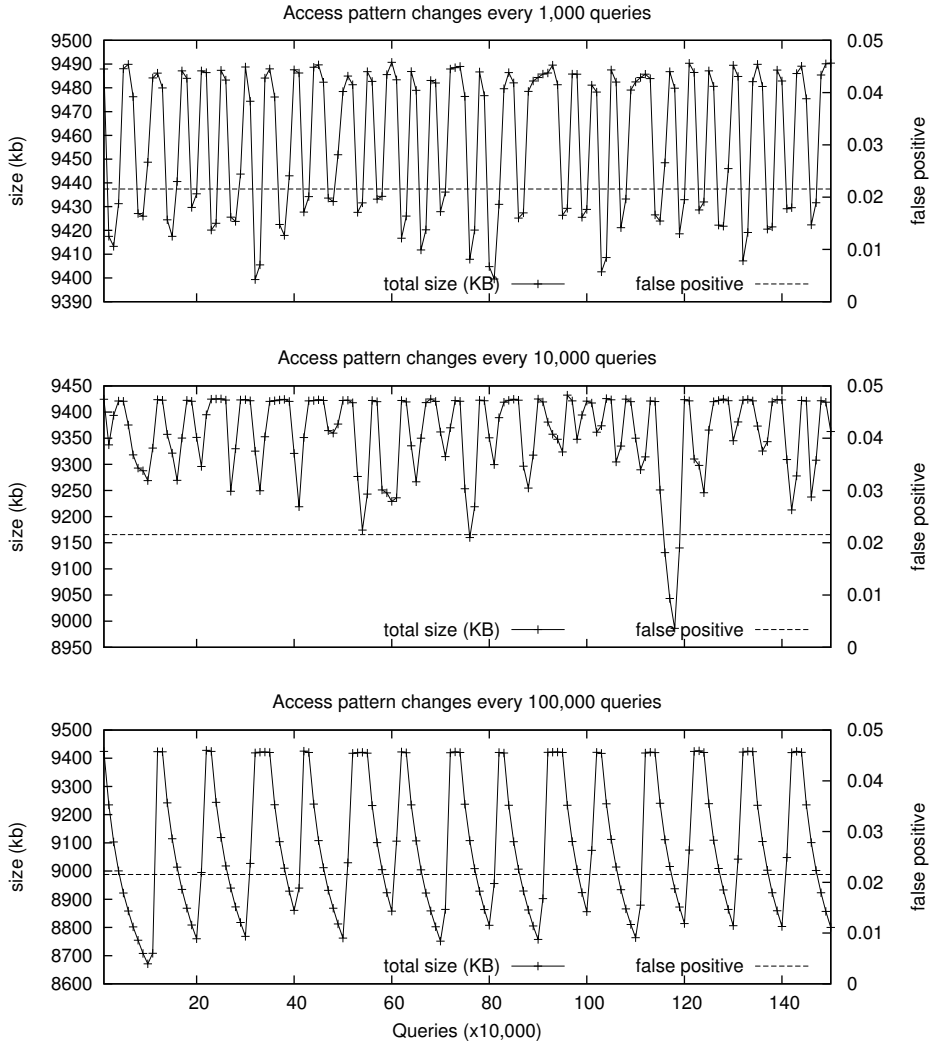


Figure 3.10: Incremental build every 10,000 lookups with access patterns that shift focus.

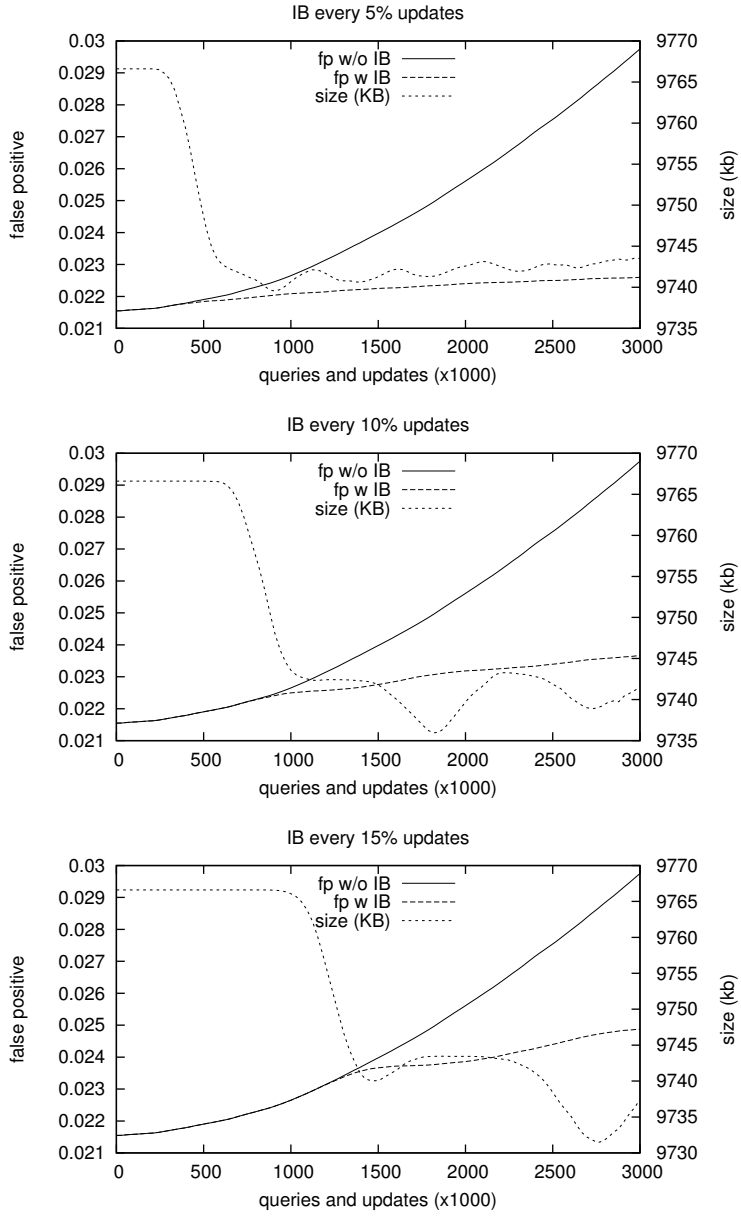


Figure 3.11: Incremental build of deteriorated Bloom filters because of updates.

all filters were equally likely to be accessed. One random insert and one random delete was done after each lookup. All lookups were performed on keys that are not present. We report the *observed* false positive ratio, which is computed by dividing the number of lookups that returned true (therefor are false positives) by the number of lookups processed until then.

Figure 3.11 illustrates the overall false positive ratio and the fluctuation of the total memory consumption for different rates of incremental rebuilds. We followed the following simple rule: *if a Bloom filter has had more than X% of its values updated, then rebuild the filter.* For example, if the limit is set to 5%, and if the total number of keys covered by a filter is 200, then after 10 updates the filter will be a candidate for rebuild. The solid line in Figure 3.11 plots the increase in the false positive ratio when no incremental rebuilds are done. The doted line shows the total KB used by the Bloom filters when the are incrementally rebuilt because of updates. The dashed line shows the resulting observed false positive ratio. The top graph of Figure 3.11 shows the results when the update limit is at 5%. In this case the false positive ratio remains low and there are little changes in the memory space used. The middle graph of Figure 3.11 shows the case when the limit is at 10%, while the bottom graph of Figure 3.11 is for 15%. When the rebuild triggering limit is increased, incremental rebuilds occur less frequently and, as expected, causes higher fluctuation of memory space used and a worse false positive ratio.

3.7 Summary

Databases often contain both hot data and cold data. Storing cold records in memory is wasteful; it is more economical to store them on cheaper external storage. However, *trips to the cold store* are expensive so it is important to avoid unnecessary ones. We maintain in memory a collection of variable-size Bloom filters. Each filter covers a subset of the key values (records). This approach makes it possible to adjust the size of the filters and use more bits for subsets that are larger and/or more frequently accessed. Rebuilding one or a few small filters in response to inserts, deletes or changes in access frequencies is also faster than rebuilding a single large filter.

We derived a mathematical model and methods to optimally size the filters in a collection, taking into account the number of distinct values covered by a filter and how frequently the filter is accessed. We also showed how and when to rebuild filters in response to changes in data or access frequencies. We performed an extensive experimental evaluation and found that our approach has several advantages compared with using a single large filter. They are faster to build; they take advantage of skew to achieve a lower false positive rate; and they can adapt to changes in both data and

access patterns.

3.8 Appendix A. Optimal Sizing of Split Bloom Filters

Consider a Bloom filter for the i-th group of hash buckets or B+tree pages that has m_i bits and uses k_i hash functions. The expected proportion of bits still set to 0 after n_i records have been added equals

$$\varphi_i = (1 - \frac{1}{m_i})^{k_i n_i} \tag{3.8}$$

For large enough n_i the following approximation holds

$$\varphi_i = (1 - \frac{1}{m_i})^{k_i n_i} \simeq e^{-k_i n_i / m_i} \tag{3.9}$$

It follows that the expected fraction of false positives for group i is

$$p_i = (1 - \varphi_i)^{k_i} \simeq (1 - e^{-k_i n_i / m_i})^{k_i} \tag{3.10}$$

It can be shown that Equation 3.10 is minimized when $k_i = \ln 2 \frac{m_i}{n_i}$. Therefore, if m_i and n_i are given, it is possible to choose an optimal value for k_i. Using the optimal k_i value we have

$$p_i \simeq 2^{-\ln 2 \frac{m_i}{n_i}} \tag{3.11}$$

Given *i)* l groups, *ii)* the number n_i of (distinct) records of each group such that $\sum_{i=1}^{l} n_i = n$ is the total number of records, and *iii)* the number f_i of access frequenies of a group such that $\sum_{i=1}^{l} f_i = f$, we wish to distribute m total bits over the l groups such that $\sum_{i=1}^{l} m_i = m$ and the total expected fraction of false positives of the corresponding l Bloom filters is minimized. The total expected false positives P is given by the following Equation

$$P = \sum_{i=1}^{l} \frac{f_i}{f} p_i \tag{3.12}$$

We solve the optimization problem of minimizing the sum of Equation 3.12 using Lagrange multipliers. The Lagrange function is defined as follows.

$$\Lambda(\vec{m}, \lambda) = P(\vec{m}) + \lambda(g(\vec{m}) - m) \tag{3.13}$$

where $\vec{m} = m_1, \ldots, m_l$, and

$$P(\vec{m}) = \sum_{i=1}^{l} \frac{f_i}{f} 2^{-\ln 2 \frac{m_i}{n_i}} \tag{3.14}$$

and

$$g(\vec{m}) = \sum_{i=1}^{l} m_i \tag{3.15}$$

We wish to solve the following system

$$\nabla_{\vec{m},\lambda} \Lambda(\vec{m}, \lambda) = 0 \tag{3.16}$$

The partial derivative $\frac{\partial \Lambda}{\partial m_i}$ for all $m_i, i = 1, \ldots, l$ is

$$\frac{\partial \Lambda}{\partial m_i} = \frac{f_i}{f} \Big(- \frac{2^{-\ln 2 \frac{m_i}{n_i}} \ln^2 2}{n_i} \Big) + \lambda \tag{3.17}$$

and for λ

$$\frac{\partial \Lambda}{\partial \lambda} = m_1 + m_2 + \ldots + m_l - m \tag{3.18}$$

Substituting Equation 3.17 into 3.16

$$\frac{f_i}{f} \Big(- \frac{2^{-\ln 2 \frac{m_i}{n_i}} \ln^2 2}{n_i} \Big) + \lambda = 0 \tag{3.19}$$

$$m_i = - \frac{n_i \ln \left(\frac{f \lambda n_i}{f_i \ln^2 2} \right)}{\ln^2 2} \tag{3.20}$$

similarly, substituting Equation 3.18

$$\sum_{i=1}^{l} m_i - m = m_1 + m_2 + \ldots + m_l - m = 0 \tag{3.21}$$

Next, we combine Equation 3.20 and 3.21 to solve for λ

$$\sum_{i=1}^{l} \frac{n_i \ln\left(\frac{f_i \ln^2 2}{f\lambda n_i}\right)}{\ln^2 2} = m \tag{3.22}$$

$$\sum_{i=1}^{l} \Big(n_i\big(\ln\left(f_i \ln^2 2\right) - \ln\left(f\lambda n_i\right)\big)\Big) = m \ln^2 2 \tag{3.23}$$

$$\sum_{i=1}^{l} n_i \ln \frac{f_i}{n_i} + \big(\ln \ln^2 2 - \ln f - \ln \lambda\big) \sum_{i=1}^{l} n_i = m \ln^2 2 \tag{3.24}$$

$$\ln \lambda = \frac{\sum_{i=1}^{l} n_i \ln \frac{f_i}{n_i}}{n} + \ln \ln^2 2 - \ln f - \frac{m}{n} \ln^2 2 \tag{3.25}$$

$$\lambda = e^{\frac{\sum_{i=1}^{l} n_i \ln \frac{f_i}{n_i}}{n}} e^{-\frac{m}{n} \ln^2 2} \ln^2 2 f^{-1} \tag{3.26}$$

Substituting λ from Equation 3.26 into 3.20 gives

$$m_i = \frac{n_i \ln \frac{f_i \ln^2 2}{f e^{\frac{\sum_{i=1}^{l} n_i \ln \frac{f_i}{n_i}}{n}} e^{-\frac{m}{n} \ln^2 2} \ln^2 2 f^{-1} n_i}}{\ln^2 2} \tag{3.27}$$

Therefore

$$m_i = \frac{n_i \ln \frac{f_i}{n_i \varpi}}{\ln^2 2} \tag{3.28}$$

where

$$\varpi = e^{-\frac{m}{n} \ln^2 2} e^{\frac{\sum_{i=1}^{l} n_i \ln \frac{f_i}{n_i}}{n}} \tag{3.29}$$

To verify the correctness of our calculations notice that, if $f_i = f_j$ and $n_i = n_j$, for all $i, j \in 1, \ldots, l$, then

$$\varpi = e^{-\frac{m}{n} \ln^2 2} e^{\frac{n \ln \frac{f_i}{n_i}}{n}} = e^{-\frac{m}{n} \ln^2 2} \frac{f_i}{n_i} \tag{3.30}$$

and

$$m_i = n_i \ln e^{\frac{m}{n} \ln^2 2} \ln^{-2} 2 = \frac{n_i}{n} m = \frac{m}{l} \tag{3.31}$$

which yields the same number of bits per group, since all groups have the same frequency and the same number of (distinct) records. Similarly, if we assume that $f_k =$

$1, n_k = n$ and $f_i = n_i = 0$ for all $i \neq k$ then

$$\varpi = e^{-\frac{m}{n}\ln^2 2} e^{\frac{n\ln\frac{1}{n}}{n}} = e^{-\frac{m}{n}\ln^2 2}\frac{1}{n} \tag{3.32}$$

$$m_k = n_k \ln e^{\frac{m}{n}\ln^2 2}\ln^{-2} 2 = n\frac{m}{n} = m \tag{3.33}$$

which correctly yields a solution where all bits are assigned to the group that contains all the records and gets all the queries.

In order to ease the numerical calculation of m_i, we rewrite Equation 3.28 as

$$m_i = (\ln^{-2} 2\gamma + \frac{m}{n})n_i \tag{3.34}$$

where

$$\gamma = \ln f_i - \ln n_i + \sigma \tag{3.35}$$

$$\sigma = \frac{1}{n}\sum_{i=1}^{l} n_i(\ln n_i - \ln f_i) \tag{3.36}$$

while σ is constant for all pages.

Moreover, in order for 3.34 to give correct results, that is not negative values, the the following must hold:

$$\ln^{-2} 2\gamma + \frac{m}{n} > 0 \tag{3.37}$$

$$\gamma > -\frac{m}{n}\ln^2 2 \tag{3.38}$$

$$\ln n_i - \ln f_i < \sigma + \frac{m}{n}\ln^2 2 \tag{3.39}$$

which holds if $f_i \not\ll f$ or if f_i is reseanable close to or greater than n_i. If f_i is very small then we assign $m_i = 0$ and assume that no Bloom filter is needed since these group of hash buckets or B+tree pages are reraly if ever queried.

3.9 Appendix B. Incremental Sizing of Split Bloom Filters

A Bloom filter that covers the i-th group of hash buckets or B+tree nodes *contributes* to the overall false positive ratio by $\frac{f_i}{f}p_i$ where $p_i \simeq 2^{-\ln 2\frac{m_i}{n_i}}$. Assume that the access

frequency of this Bloom filter has changed to f'_i, or the number of unique records to n'_i, or both. We need to readjust m_i to a new value m'_i such that the new contribution to the over all false positive remains the same. More formally, the following equality should hold

$$\frac{f_i}{f} p_i = \frac{f'_i}{f'} p'_i \tag{3.40}$$

where now

$$p'_i \simeq 2^{-\ln 2 \frac{m'_i}{n'_i}} \tag{3.41}$$

and

$$p'_i = \frac{f'}{f'_i} \frac{f_i}{f} p_i \tag{3.42}$$

We solve Equations 3.41 and 3.42 for m'_i

$$m'_i = -n'_i \ln \left(\frac{f'}{f'_i} \frac{f_i}{f} p_i \right) \ln^{-2} 2 \tag{3.43}$$

By keeping for each Bloom filter the targeted false positive ratio p_i and the fraction $\frac{f_i}{f}$ we adjust the filter size to m'_i bits in order to maintain the same contribution to the overall false positive.

It is possible that f'_i becomes so small that $p' > 1$. In this case, we set $p' = 1$ and consequently $m = 0$. Note that if $m = 0$, then the Bloom filter always returns true.

Chapter 4

Generic Typed Value Indices

We describe a collection of indices for XML text, element, and attribute node values that *(i)* consume little storage, *(ii)* have low maintenance overhead, *(iii)* permit fast equi-lookup on string values, and *(iv)* support range-lookup on any XML typed value (e.g., double, dateTime). The equi-lookup string value index depends on an *elaborate hash function* and on an *associative combination function* to facilitate updates on both mixed-content and element nodes. We also present techniques for creating range-lookup indices supporting any *ordered* XML typed value. These indices rely on a *finite state machine* that accepts the type specific language, and on a *state combination table* for combining states to speed-up updates. We evaluate the stability of the hash function, the storage overhead, and the indices creation and maintenance time in the context of the open-source XML database system MonetDB/XQuery.

4.1 Motivation

The semantics of XQuery are designed to facilitate querying both typed and untyped XML data, whose contents in turn may vary from strongly structured data to very loosely structured mixed-content data. For instance,

```
doc("persons.xml")//person[.//age = 42]
```

will return `<person>` nodes, that have at least one `<age>` node with integer value `42`. In absence of an XML Schema that would give `<age>` a specific type, the equality predicate will match all `<age>` nodes with a *string value* that is castable to an `xs:double` with value `42.0`, e.g.,

```
<age>42</age>
```

However, other matching instances exist, such as

```
<age>42.0</age> and <age>      +4.2E1</age>
```

That is, any text node containing a decimal number, or a double in various syntactical forms (including leading white space characters) all cast to `42.0`.

To further complicate things, the string value of an element node or a mixed-content node, is the concatenation of the string values of all descendant text nodes [86], such that the following node also matches the predicate.

```
<age><decades>4</decades>2<years/></age>
```

While this flexibility is one of the crucial advantages that XML offers over relational data management technology, it complicates the generic use of value indices for general comparisons. As a result, XQuery systems typically require a system administrator to specify the document sub-paths and the value casts to be indexed. In case of DB2 PureXML [59], for example, one could do

```
create index myindex on items(person) generate key using
        xmlpattern "//person//age" as sql double
```

However, this approach has the following disadvantages: *(i)* only queries that use the specific listed paths can be accelerated, *(ii)* the index is type specific; i.e., an index on double values cannot be used in case of string lookups, and *(iii)* in contrast to current trends in data management systems, it requires explicit (DBA-induced) index configuration.

In our view, the above mixed XML content example of `age` being decomposed in `decades` and `years`, yet accidentally mapping onto `42`, is a rather unintended consequence of the XPath and XQuery standards, that may even be called undesired. However, it *is* part of these well-entrenched standards and should thus be supported by XML database systems. This work addresses the above by investigating indexing techniques that enable *self-tuning* index management, by creating generic XML value indices that cover an entire document, not just a single path with one particular value type, and allow equi-comparisons on string values, as well range-lookup on any XML typed value. The index structures presented in this chapter are optimized for the cases where values represented by a single node are looked up, but also are able to correctly deal with mixed XML content.

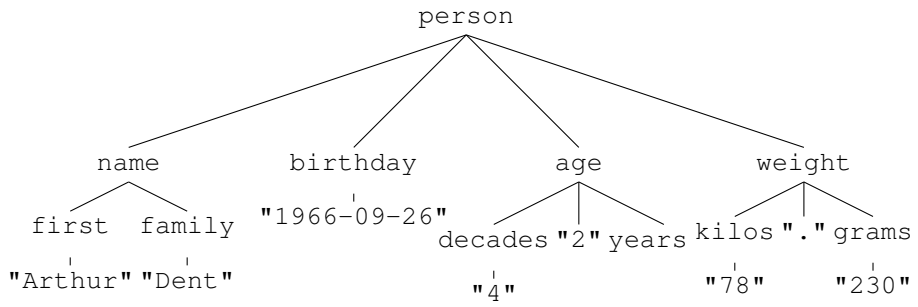


Figure 4.1: XML Document about persons

4.1.1 String Equi-Index

The string value index depends on a specialized hash function $H(\texttt{str}) : \textit{int}$, carefully designed to map arbitrary length string values into integer hash values in such a way that hash collisions are kept low. The hash function can be used to index XML text, element, and attribute node values. A (B-tree) index, constructed on the hash values, can be used for accelerating string value lookups, e.g., for evaluating the XPath expression

```
doc("persons.xml")//person[.//first="Arthur"]
```

that returns all <person> nodes with first name "Arthur".

In addition, we define an associative combination function $C(\textit{int}, \textit{int}) : \textit{int}$ that can be used to derive the hash value of the concatenation of a list of string values. This is achieved by using the *hash values* computed over the individual string values of the list, thus avoiding accessing the actual string data which can be costly. Figure 4.1 depicts the tree representation of an XML document about a person. In this example, the string value of node <name> is "ArthurDent". Thus, the XQuery query

```
doc("persons.xml")//*[fn:data(name)="ArthurDent"]
```

returns node <person>. In effect, the hash value of an element can be computed by combining, through function C, all hash values of its direct children. When updates are also considered, reconstructing the hash value index requires visiting all ancestors of the updated nodes and their immediate children, in order to recompute the hash values with function C. This means that on well-shaped trees, with $\log n$ depth and a low fanout, index maintenance costs remain limited.

4.1.2 Typed Range-Index

Moreover, we construct indices for range-lookup on other XML typed values, and still respect the mixed-content and element node semantics. Of particular interest are the types `xs:dateTime` and `xs:double` [87]. Note that an index on `xs:double` can be used to accelerate predicates on all numerical XQuery types.

This group of range-lookup indices relies on *finite state machines (FSM)* that recognize the language that accepts the specific type. For example, if double values are considered, an *FSM* can be defined such that it recognizes (*potential*) valid lexical representations. This *FSM* will return a *reject* state if an illegal sequence of characters is encountered, or the *state* in which the *FSM* terminated. In case of doubles and for typical XML data, the majority of all text nodes that do not represent a numeric value will be rejected immediately, avoiding waste of indexing resources. A minority of nodes will contain *potential* valid lexical representations, i.e., those that although they cannot be casted to a double value, do not contain an illegal sequence of characters (they are just incomplete). For example, the text node `"78"` rooted at node `<kilos>` in Figure 4.1 is a valid lexical representation that can be casted to a double. On the other hand, the text node `"."` rooted at node `<weight>`, although not castable to a double value, is a potential valid representation – it misses one or more leading and trailing digits.

In addition to an *FSM*, a *state combination table (SCT)* is defined for each supported XML typed value. The role of the *SCT* is similar to that of function C for the string value index: to efficiently decide if the combination of two nodes is a (potential) valid lexical representation of the XML typed value and to return their combined state. For example, concatenating all descendant text nodes of node `<weight>`, i.e.,

```
<kilos>78</kilos>.<grams>230</grams>
```

produces a valid lexical representation of a double value, namely `78.230`. As in the case of function C, *SCT* is used during the creation and updates of the range-lookup index. Finally, based on the selection of only those nodes that contain a valid lexical representation of the indexed type provided by the *FSM*, a (B-tree) index is built on the values of those nodes to facilitate fast range lookups.

4.1.3 Contributions

The main characteristics and novelty of the indices presented in this chapter can be summarized in the following:

- Cover the entire XML document: all element, attribute, and text nodes are indexed, independently from their path.

- Respect the XQuery semantics: the indices works correctly even in the presence of intermediate and mixed-content nodes.
- Self-tuned: in the sense that no explicit index configuration, which defines a type and/or a path, is required.
- Consume little storage: no data replication is needed.
- Updatable and low maintenance overhead: the design of the indices are driven by the need of efficient updatable generic indices.

4.1.4 Outline

The rest of the paper is organized as follows. Section 4.2 presents related work. Section 4.3 and 4.4 details the indexing techniques. Section 4.5 describes the implementation details for creating and updating the indices. The experimental justifications of our claims are presented in Section 4.6. Section 4.7 summarizes this chapter.

4.2 Related Work

We focus our study of related work to fully functional XML engines, and their implementation of value indices. To the best of our knowledge there is no other work that address the problem of generic XML value indices that work well, are updatable and are able to deal with the abnormality of the mixed-content node data values. All systems we examined require a user-originated definition of specific paths to be indexed, in contrary to our solution which covers the entire document.

In the context of DB2 native XML engine [10, 59] value indices are supported only for path-specific values and for predefined types. A query is been accelerated by those indices only if the path and the type match. The key of such XML indices is `[pathid, value, nodeid, rid]`. The order in which these tuples are indexed (e.g., on pathid or value) facilitate different kind of query acceleration.

In [65] two different types of XML indices are constructed, namely *primary* and *secondary*. The primary index is a B^+-tree on tuples of the form `[ordpath, tag, nodetype, value, pathid]`, ordered according to the document order. The secondary indices are defined on top of the B^+-tree keys, and on the columns of interest. For example, the XML value index is defined on `[value, pathid, ordpath]`. Such indices allow to speed up evaluation for specific queries, however, as also pointed out in [79], they do not respect the XQuery Data Model.

4.3 String Value Index

We present a fully updatable index covering equality lookups on XML string values. The index is based on a specialized hash function H that maps string values to hash values. Hash values are then indexed for fast lookup during query time. Each hash value is associated with a set of *candidate* XML node ids, which can then be further processed to select those nodes whose value and path structure are relevant to the query.

The hash function H accepts a *sequence of characters* (i.e., an XML string value) and outputs a *32-bit hash value*. The 27 most significant bits of the hash value, called in the sequel *c-array*, are used for hashing characters. Since an XML string value may be of arbitrary length we base the hash function on a *circular XOR operation*. The *circular XOR operator* works by applying the XOR operator between the 7 least significant bits of the value[1] of each character and the c-array, starting from the right most position (i.e., position 0), and gradually incrementing the offset by 5 bits to the left. When the end of the c-array is reached, that is when the offset is set to 25, 2 bits of the next character are XOR-ed to positions 25 and 26, while the remaining 5 bits are XOR-ed back to positions 0-4 of the c-array. The next character is then XOR-ed with the offset set to $(25 + 5) \mod 27 = 3$, thus circling around the c-array. The process is repeated until all characters of the sequence are processed. To produce hash values with such circular XOR operation, the 5 least significant bits of the 32-bit hash value are reserved for storing the offset information of the circle, i.e., the offset where the next character should be XOR-ed. More specific, the format of the hash value produced by the hash function H is

$$\underbrace{\text{C27}\cdots 1}_{\text{27-bits}} \,|\, \underbrace{\text{OFFC}}_{\text{5-bits}}$$

where **C**1...27 are the 27 most significant bits, forming the c-array, and reserved for hashing all characters of the XML string value. The c-array is built by applying the *circular XOR operator*. And **OFFC** are the least 5-bits form the OFFC field that encode the 27 different offset positions of the c-array (i.e., the elements of set $\mathbb{Z}_{27}$).

Algorithm 4.1 details how to compute the hash value of an XML string. The input is an XML string value `str`. The algorithm populates the c-array by iterating over all characters of `str` (line 2). The circular XOR operator is implemented by first shifting `offset` times to the left the 7 least bits of the current character, and then XOR-ing with h_{val} (line 3). If the offset is larger than 20, then the remaining bits, computed as $(\texttt{offset} + 7) \mod 27$, are XOR-ed back at the beginning of the c-array. Next, the

[1] ASCII or UTF value depending on the implementation

Algorithm 4.1 Hash Function H

Input: XML string value `str` as `sequence of chars`
Output: `32-bit` hash value $\mathtt{h}_{val}$

```
1: h_val = 0; /* initialize h_val */
2: /* populate the c-array */
   for (offset = 0; *str!='\0'; str++)
       c = *(str) & 127;
3:     /* circular_XOR operation*/
       h_val ^= c << offset;
       if (offset > 20)
          h_val ^= c >> (27 - offset);
4:     offset += 5; /* update offset */
       if (offset > 26) offset -= 27;
5: /* set the OFFC bits */
   h_val <<= 5; h_val |= offset;
6: return h_val;
```

offset is incremented by 5 for the next iteration (line 4). When all characters of `str` are consumed, the OFFC bits are set (line 5) and the h_{val} is returned (line 6).

Figure 4.2 depicts the iteration steps of function H for computing the hash value of `"Arthur"`, the text node of element `<first>` in the XML document of Figure 4.1. The procedure starts with `offset=0`. When the `offset` is 25, character `"r"` has to be processed. The 7 least significant bits of `"r"` are 1110010. From those, the two least significant bits (10) are XOR-ed with h_{val} at positions 25 and 26, while the 5 remaining bits (11100) are XOR-ed to positions 0 to 4. The algorithm ends and the h_{val} is returned. The OFFC bits are set to 3 (00011), indicating the next value of `offset`.

We define an *associative* function $C(\mathit{int}, \mathit{int}) : \mathit{int}$ to combine hash values during index creation and updates. Function C is designed such that given two string values $\mathtt{str}_{left}$ and $\mathtt{str}_{right}$ it holds that

$$H(\mathit{concat}(\mathtt{str}_{left}, \mathtt{str}_{right})) = C(H(\mathtt{str}_{left}), H(\mathtt{str}_{right}))$$

where *concat*() is the *string concatenation* function.

The combination function C is used during the creation of the index, as well in the event of updates. Suppose that the string value of node `<family>` in Figure 4.1 is updated from `"Dent"` to `"Prefect"`. After computing the new hash value of node `<family>` with $H($`"Prefect"`$)$, the hash value of node `<name>` must be updated

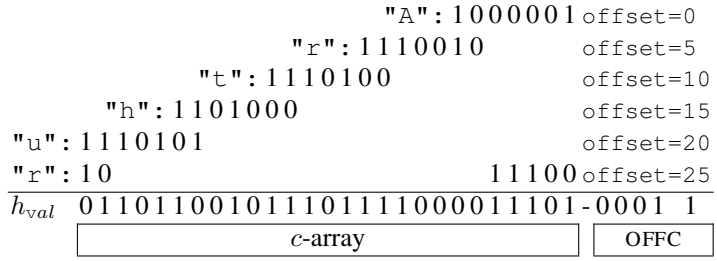


Figure 4.2: Example of computing $H(\texttt{"Arthur"})$

too, and consequently node <person>. Without the combination function C that would call for evaluating once more

$$h_{\texttt{<name>}} = H(\texttt{"ArthurPrefect"}), \text{ and}$$
$$h_{\texttt{<person>}} = H(\texttt{"ArthurPrefect1966-09-264278.230"}).$$

Obviously, for large documents this is very inefficient since the string values of all nodes in the document have to be visited in order to reconstruct the hash values. However, with function C we only need to invoke function H once for the updated text node. The hash values of all ancestors of the updated node are reconstructed by visiting only the siblings of the ancestors, and reading their hash values, as opposed to reconstructing their string values. For example the new hash value of node <name> will be computed by

$$h_{\texttt{<name>}} = C(h_{\texttt{<first>}}, h_{\texttt{<family>}})$$

where $h_{\texttt{<first>}}$ and $h_{\texttt{<family>}}$ are the already computed hash values of nodes <first> and <family>. Similarly, the new hash value of node <person> will be

$$h_{\texttt{<person>}} = C(h_{\texttt{<name>}}, C(h_{\texttt{<birthday>}}, C(h_{\texttt{<age>}}, h_{\texttt{<weight>}}))).$$

Algorithm 4.2 details how to combine two hash values, namely h_{left} (the left operand) and h_{right} (the right operand). The algorithm outputs the combined hash value h_{comb} of the input hash values. First, the c-array of the left operand is copied to h_{comb} (line 2). In order to combine the c-array of the right operand with h_{comb}, we apply the standard *circular left shift* operation to the c-array of h_{right}. The c-array of h_{right} is shifted to the left by as many positions as indicated by the OFFC bits of h_{left}. The result is then XOR-ed back to h_{comb} (line 3). Then, the OFFC bits of h_{left} and h_{right} are added, to update the offset information of the result hash value h_{comb} (line

Algorithm 4.2 Combination Function C

Input: hash values $\texttt{h}_{\texttt{left}}$ and $\texttt{h}_{\texttt{right}}$
Output: combined hash value $\texttt{h}_{\texttt{comb}}$

```
1: h_comb = 0; /* initialize h_comb */
2: /* copy the c-array of the left operand to h_comb */
   h_comb |= mask27(h_left)
3: /* circular left shift */
   h_comb ^= (mask27(h_right) << mask5(h_left)) |
        mask27(mask27(h_right) >> (27-mask5(h_left)));
4: /* add the OFFC bits of the left and right operands */
   h_comb |= (mask5(h_left) + mask5(h_right)) % 27;
5: return h_comb;
mask5(h) := h & 31
mask27(h) := h & (~31)
```

4). Finally, the combined hash value h_{comb} is returned (line 5). The algorithm uses functions *mark5*() and *mark27*(), which apply bit operations to separate the c-array and the OFFC bits from the input 32-bit hash values.

The correctness of the output of function C is based on the observation that the XOR operator has the associative property, i.e., if i, j, k are integers, then $(i \hat{} j) \hat{} k = i \hat{} (j \hat{} k)$. By shifting the right operand to the left, we permute the position 0 of the c-array to the position indicated by the offset of the left operand. Recall that the OFFC bits of the hash value indicate the offset where the next character should be XOR-ed. Effectively, function C continues the circular XOR operation of function H but in a different order of applying the XOR operation. However, because the XOR operation has the associative property the result is guaranteed to be correct.

Next, we prove by induction the associative property of function C, that is:

$$\begin{aligned} H(a_1 \cdots a_n) = & \qquad \text{(eq. 1)} \\ & = C(C(\ldots C(H(a_1), H(a_2)) \ldots, H(a_{n-1})), H(a_n)) \\ & = C(H(a_1), C(H(a_2), \ldots C(H(a_{n-1}), H(a_n)) \ldots)) \end{aligned}$$

proof. The proof is by induction on n, the number of string values $a_1, \ldots, a_n$ that form the concatenated hash value
$H(a_1 \cdots a_n)$. The base case is $n = 2$, that is $H(a_1 a_2) = C(H(a_1), H(a_2))$, which holds from the definition of function C. For n greater than 2, assume that (eq. 1) holds for all $k \geq 2$ such that $k < n$.

$$H(a_1 \cdots a_{n-1}) = C(\ldots C(H(a_1), H(a_2)) \ldots, H(a_{n-1}))$$
(Ind. Hyp. with $k = n-1$)

$$H(a_1 \cdots a_{n-1}a_n) = C(H(a_1 \cdots a_{n-1}), H(a_n))$$ (by base case)

$$H(a_1 \cdots a_{n-1}a_n) = \\ = C(C(\ldots C(H(a_1), H(a_2)) \ldots, H(a_{n-1})), H(a_n))$$
(by base case and Ind. Hyp.)

Similarly, we can prove the second equality of equation (eq. 1). Thus, changing the order of *operations* (i.e., the order of applying function C) does not produce a different hash value H.□

In Section 4.5 we present an efficient and simple algorithm for visiting the relevant nodes and (re)compute the hash values, during index creation and updates.

4.4 Typed Range-Lookup Index

We describe an updatable index covering range lookups which can be defined over any XML typed value. We detail the index over *double values*, however any other XML built-in type can be supported by applying the same ideas. The index is exact: it does not return false positives, neither it misses any nodes with value that matches the query.

This family of indices is based on *finite state machines* (FSM) that recognize the language that accepts the syntax (i.e., lexical representation) of the indexed XML type. Figure 4.3 illustrates the corresponding FSM for double values. Each distinct state of the FSM is depicted with a circle and each transition from one state to an other by an arrow. The arrows are labeled with the symbol of the language that permits the specific transition (E stands for the exponent character of doubles, and ws for whitespace characters). Double lined circles signify final states and states with incoming edges without a source signify initial states. Each state is marked with a unique label. In this content, all finite state machines that recognize an XML type are deterministic.

Notice that although the FSM of Figure 4.3 appears to have 5 initial states and 3 independent state transition graphs, each initial state is marked with a different symbol. Thus, depending the first character of the node string value, a different initial state is considered, implying the existence of a virtual empty initial state that redirects to the correct initial state.

Moreover, if there is more than one path leading to the same state, we expand the FSM in such a way that these paths lead to different copies of the same state, which

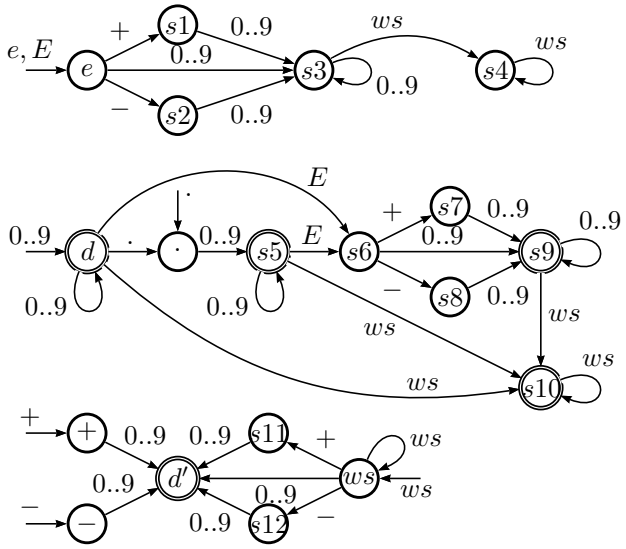


Figure 4.3: Finite State Machine for double values

in turn, the copied states are marked with a new unique label. This *normalization* of the FSM allows to uniquely identify the initial state and the path that leads to each state. For the proposed index, only through this normalization of the FSM, we can properly determine the consequences of concatenating arbitrary node string values by examining the states and not their string values. For presentation reasons, we have omitted the detailed expansion of the FSM in Figure 4.3: state d' redirects the FSM to state d, and for each distinct incoming edge of state d' (e.g., +, -, $s12$, etc.) different new labels are applied to all states following d. As a result, there are 60 different states including the *reject* state. The reject state (not visible in Figure 4.3) is reached always when the next character is not part of the language or it does not infer a valid state transition.

According to the semantics of the XQuery Data Model, each value of an XML node, if it does not evaluate to the reject state, is a potential accepted value. That is, because of the mixed content semantics: a non-final state may be evaluated to a final state if its siblings are considered. Therefore, the first step of creating the typed range-lookup index is to associate each XML node with a state. More specific, during

	rej	**e**	**.**	**d**	**s**1	**s**2	**...**
rej	rej	rej	rej	rej	rej	rej	...
e	rej	rej	rej	s3	rej	rej	...
.	rej	rej	rej	s5	rej	rej	...
d	rej	s6	s5	d	s7	s8	...
s1	rej	rej	rej	s3	rej	rej	...
s2	rej	rej	rej	s3	rej	rej	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Figure 4.4: State Combination Table (SCT)

the creation of the index, the lexical value of each text node is fed to the FSM. The FSM returns the state at which the recognition process has stopped. For example, if the lexical value of the given text node is `"E+93 "`, state $s4$ is returned. Similarly, if the lexical value is `" +32.3"` then the state labeled $s11 + s5$ is returned (state is not shown in Figure 4.3, but implied by the jump from d' to d). Finally, if the lexical value is not a (potential) valid representation of the typed value, for example `"42 text"`, then the reject state is returned. The result of this first step of the index creation is that each text node is assigned a state accordingly to the returned state of the FSM. Notice, that since the total number of states is small – in the case of doubles only 60 – the state can be saved with only one byte for each node. Moreover, the nodes that are evaluated to the reject state, which will be in most cases the majority, do not need to store any state – the absence of a state signifies the reject state.

The next step for creating (or updating) the index is to determine the state of the intermediate nodes. The desire is to be able to efficiently compute the state of an intermediate node without reconstructing the lexical representation of that node. Moreover, it should be possible to early reject intermediate nodes that do not have a valid lexical representation and thus avoid unnecessary traversal of the XML tree. This desire is fulfilled by defining a *state combination table* (SCT). The SCT is a succinct representation of all possible valid combinations of the states. Valid combinations are the ones that do not result in a reject state. The SCT for the double type is depicted in Figure 4.4. The size of the complete table is 60×60. However, most of the combinations result to a reject state. The succinct representation of the SCT omits all pairs that result in a reject state. Out of the 3600 different combinations, only 389 are non-reject. Moreover, since each state can be represented by a byte, the SCT consumes even less memory space. Finally, the normalization of the FSM described in the beginning of this section ensures that such a SCT always exists.

To facilitate fast range lookups a clustered (B-tree) index is built on top of the

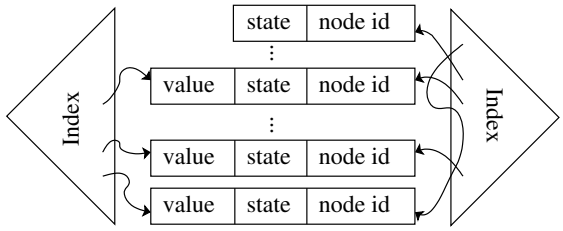


Figure 4.5: Indices for Range-Lookup on Typed Values

typed values which are associated with the id of the node they belongs to. In addition, a second index is built on the node ids. This index is used during the creation and updates of the typed value index for retrieving the state of a node id. The keys of the (B-tree) index are tuples of the form `[value, state, node id]`[2]. Moreover, some nodes may have a (potential) valid state but for storage efficiency do not need to store a value. For example if the state is $s2$, then there is no valid double value, and the state label itself is sufficient to deduce the lexical representation, namely `"E-"` or `"e-"` (which of the two equivalent representations is irrelevant for constructing the combined double value). In addition, the (B-tree) indices are used during creation or update of the typed XML indices to reconstruct the lexical representation of a specific node, without accessing the document data. For example if the value of a node is `"26"` while the state is $s7$, then the lexical representation of that node is `"26E+"`.

4.5 Implementation Details

In this section we present the implementation details for creating and updating the indices by employing the functionality and data structures introduced in the previous sections. We have implemented both indices in the context of the open-source XML database system MonetDB/XQuery [58]. Although some details are system specific and are tightly coupled with the internal data structures of the XQuery engine at hand, we present the skeleton of the algorithms as generic as possible. In general, only small adjustments would be necessary to successfully apply these ideas to other systems.

[2] depending the design of the XQuery engine – focusing on space or computational efficiency – the second (B-tree) can be clustered also, thus having tuples of the form `[value, node id]` and `[node id, state]`

4.5.1 Index Creation and Updates

Internally, MonetDB/XQuery stores an XML document in such a way that permits efficient depth-first traversal. This is achieved by using a range encoding on the documents nodes, similar to the pre/post encoding. For more details we refer the reader to [13]. It is realistically assumed that every XQuery engine provides a similar interface for efficient traversal over the XML document tree.

All indices are created and updated with the same skeleton algorithm, only the functions called in each case are changed. The algorithm is based on a depth-first traversal of the XML document and since all indices are independent of each other, creating and updating multiple defined indices can be done simultaneously with only one pass.

Algorithm 4.3 details the code for creating both the string equality index and the XML typed range index. Intuitively, the algorithm works as follows. The depth-first traversal starts at the root of the document until all text nodes are visited (lines 3-5 of Algorithm 4.3). The traversal is guided downwards until the first text node is found, while each visited intermediate node is pushed into a stack (lines 9-11). When a text node is located, function H or the FSM is called (lines 6-8). Next, and to locate the next text node, either the node on the head of the stack is popped (line 16), or the sibling nodes of the current node are considered (line 12). These cases traverse the XML document tree either upwards or rightwards to locate the next text node. However, in both cases the parent node of the current node has to be updated with the combined hash value or state. This is achieved by invoking function C or probing the SCT (lines 13-15 and 17-19). Finally, and after all text nodes are visited, the stack is emptied and all nodes popped from the stack are updated accordingly (lines 21-24). Notice that the functions calls on the DFS module appearing in Figure 4.3 (e.g., `DFS.nextSiblingNode()`) are always evaluated against the current node.

The update algorithm is outlined in Algorithm 4.4. It works similar to the create algorithm of Algorithm 4.3. The first difference between the two algorithms is that when a new node is added in the stack, its field is reset since it is not valid any more (line 9). This is, because if the depth-first traversal reached that node, it means that some of its descendants have been updated. The second difference is that when a node is popped from the stack, its new hash value or state is the combination of all of its children. In other words, its combined hash value or state must be recomputed across all its immediate children (lines 14-16 and 19-21). A side effect of this recomputation is that during the traversal of siblings nodes (line 11) – contrary to the create algorithm – there is no need to update the father node, this will happen eventually when that node is popped from the stack in a later step.

The algorithm presented in Algorithm 4.4 expects only updates on the value of a

Algorithm 4.3 Index Creation

Input: a sequence of all XML text nodes of a document as `ctx`
Output: hash value or state for all XML nodes of the document

```
01: /* initialize variables */
02: init all nodes.field to 0 or undef
03: cur_node = DFS.getRoot();
04: /* loop while all text nodes are consumed */
05: while (ctx.hasNext())
06:       if (ctx.current == cur_node)
07:         cur_node.field = H(cur_node)|FSM(cur_node);
08:         ctx.next();
09:       else if (ctx.current descendantOf cur_node)
10:          stack.push(cur_node);
11:          cur_node = DFS.nextChildNode();
12:       else if (ctx.current siblingOf cur_node)
13:          father_node = DFS.getFatherNode();
14:          father_node.field =
              = C(father_node.field, cur_node.field)/
                SCT[father_node.field][cur_node.field];
15:          cur_node = DFS.nextSiblingNode();
16:       else
17:          pop_node = stack.pop();
18:          pop_node.field =
              = C(pop_node.field, cur_node.field)|
                SCT[pop_node.field][cur_node.field];
19:          cur_node = pop_node;
20: /* empty stack with visited nodes */
21: while (stack.notEmpty())
22:      pop_node = stack.pop();
23:      pop_node.field =
          = C(pop_node.field, cur_node.field)|
            SCT[pop_node.field][cur_node.field];
24:      cur_node = pop_node;
```

Algorithm 4.4 Index Update

Input: a sequence of updated XML text nodes as `ctx`
Output: updated hash value or state for all affected XML nodes

```
01: /* initialize variables */
02: cur_node = DFS.getRoot();
03: /* loop while all text nodes are consumed */
04: while (ctx.hasNext())
05:       if (ctx.current == cur_node)
06:         cur_node.field = H(cur_node)|FSM(cur_node);
07:         ctx.next();
08:       else if (ctx.current descendantOf cur_node)
09:           cur_node.field = 0|undef;
              stack.push(cur_node);
10:           cur_node = DFS.nextChildNode();
11:       else if (ctx.current siblingOf cur_node)
12:           cur_node = DFS.nextSiblingNode();
13:       else
14:           pop_node = stack.pop();
15:           while (DFS.hasSiblingNode())
                  cur_node = DFS.nextSiblingNode();
                  pop_node.field =
                  = C(pop_node.field, cur_node.field)|
                    SCT[pop_node.field][cur_node.field];
16:           cur_node = pop_node;
17: /* empty stack with visited nodes */
18: while (stack.notEmpty())
19:     pop_node = stack.pop();
20:         cur_node = DFS.leftMostSibling();
            while (DFS.hasSiblingNode())
                cur_node = DFS.nextSiblingNode();
                pop_node.field =
                = C(pop_node.field, cur_node.field)|
                  SCT[pop_node.field][cur_node.field];
21:     cur_node = pop_node;
```

Data	**S**ize (MB)	#Nodes	#Text Nodes		#Double Values		#non-leaf
XMark1	112	4,690,640	3,024,328	(64%)	377,123	(8%)	0
XMark2	224	9,394,467	6,056,817	(64%)	754,936	(8%)	0
XMark4	448	18,827,157	12,138,505	(64%)	1,514,227	(8%)	0
XMark8	896	37,642,301	24,269,192	(64%)	3,026,029	(8%)	0
EPAGeo	170	6,558,707	4,372,404	(66%)	517,862	(7%)	0
DBLP	474	34,799,707	23,198,402	(66%)	3,748,565	(10%)	21
PSD	685	58,445,809	37,139,989	(63%)	2,441,791	(4%)	902
Wiki	2024	94,672,619	53,564,889	(56%)	104,059	(0.1%)	0

Table 4.1: Statistical information for the data sets

text node. However, in the case of a node or subtree deletion, a slight change in the algorithm solves the problem. More precisely, the algorithm gets as input the node that served as the root of the subtree that was deleted. In that case, the text value of that node –after the deletion of the subtree– is either the empty string or a new value. In either case, the update algorithm is invoked with the new value of the node, oblivious of the deleted subtree.

4.5.2 Transaction Management

A final consideration for the update implementation is transaction management overhead, in particular its locking requirements. A general challenge in XML value indexing is that the value of a node is (potentially) influenced by all its descendants. This implies that each update may impact the root node, and locking the root for each transaction can easily become a bottleneck.

A first remark on the proposed typed XML range indices, is that in typical XML documents, only leaf nodes and not intermediate nodes (let alone the root) have a type such as *xs:double* (see also Table 4.1). The proposed indexing scheme only stores nodes with a potential valid typed lexical value, which in general means that it contains only leaf nodes. Therefore, an adaptive locking scheme that supports fine-grained locks and is able to gradually enlarge the lock granularity [32] should work rather efficiently with this index.

For our other proposal concerning the equality index on strings, we do have the situation that all XML nodes have a string value and must be indexed, the root node inclusive. Each single update changes the hash value of all its ancestors, thus always affects the root node! However, in the context of structural updates for the MonetDB/XQuery system [15] it was shown that locking ancestors can be avoided if updates are *com-*

mutative, and the combination function C has indeed been developed specifically with this property in mind. It is in fact possible to avoid locking any ancestors of updated nodes during transaction processing until the commit point. A committing transaction should re-read the latest value of all ancestor nodes of an update (and their direct children, per the update algorithm) to recompute their new hash values. Even if siblings of the updated node where changed in the meantime and thus affected these ancestors, the commutativity of the C function ensures that we will compute the correct hash values in the end.

4.6 Experimental Evaluation

In this section we present our experiments for evaluating the index creation time, the disk storage overhead, and the maintenance overhead (i.e., update time), for both the equality string and the range typed index. We also studied the collisions introduced by the hash function H to asses the *stability* of the string index and identify which cases of abnormal input text values affect this stability.

The experiments were conducted on an Intel^{R} Core^{TM}2 Quad CPU Q6600 machine, running at 2.40GHz with 4MB cache memory. The machine had 8GB of RAM and 2 hard disks with active raid level 0, capable of reading(writing) from(to) disk with approximately 100MB/sec.

We used 8 different documents of varying size and structure. The first 4 documents were synthetically created with XMark[3] with scale factors 1, 2, 4, and 8, respectively. The remaining 4 documents reflect *"real life"* data, since they were download from online databases. The data set EPAGeo[4] contains geospatial data, the PSD[5] dataset contains protein sequence data, while DBLP[6] and Wiki[7] contain text data about publications and abstracts of articles, respectively. Table 4.1 details the size (in MBs) of each data set before shredding in the database, the number of nodes in the document, the number of text nodes, and the number of text nodes that have a (potential) valid double lexical representation. Next to the number of text and double nodes, the percentage compared to the total nodes of the data set is given to ease the comparison. The last column depicts the number of non-leaf nodes that have a (potential) valid double value. The datasets generated with XMark do not have any such nodes, while only the DBLP and PSD datasets have a few number of them. This observation strengths our

[3] http://www.xml-benchmark.org/
[4] http://www.epa.gov/enviro/geo_data.html
[5] http://pir.georgetown.edu/
[6] http://dblp.uni-trier.de/xml/
[7] http://download.wikimedia.org/

claim that intermediate nodes that cast to a specific XML type is a rare phenomenon, nevertheless an XML value index should respect the semantics of the XQuery Data Model. Our indices are semantically correct, and, as we will illustrate in this section, this is achieved without introducing any significant overhead.

The first set of experiments study the time and space overhead for creating the indices during shredding, that is when the document is processed and stored in the database. All runs were done in cold memory – none of the XML data resided in memory – thus the total shredding time includes the time needed to read the data from disk. We repeated the same experiments three times and report the average times. The deviation of each run from the average time was minor (less than 10ms). The two upper graphs of Figure 4.6 list the time in milliseconds needed by MonetDB/XQuery to shred the document, and the time needed by our create algorithm to construct the string and double indices. The bars in Figure 4.6 give a visual of the overhead percentage introduced in the shredding process from the index creation. For the string index, the overhead never exceeds 10% in the worst case, while for half of the cases is less than 5%. For the double index, the creation time overhead is less than 2% in all cases. This is expected, since the combination step is cheaper than that of the string index – probing an array vs. invoking a function.

The lower part of Figure 4.6 depicts the storage consumed by the string and double index compared to the storage demands of the database for the specific documents. The storage needs of the string index is at most 20% (e.g., EPAGeo, PSD) over the total document and 10% in the best cases (e.g., XMark, Wiki). The difference is explained by the distribution in each document of the string data and the number of nodes. Small number of text nodes with large size of string data resort to less storage consumption than large number of text nodes with few data in each node. On the other hand, the storage demands of the double index is limited. It never exceeds 2-3% of the total size of the database. This is mainly because a) each state is only 1 byte and b) there are few text nodes that have a valid double lexical representation compared to the total number of nodes.

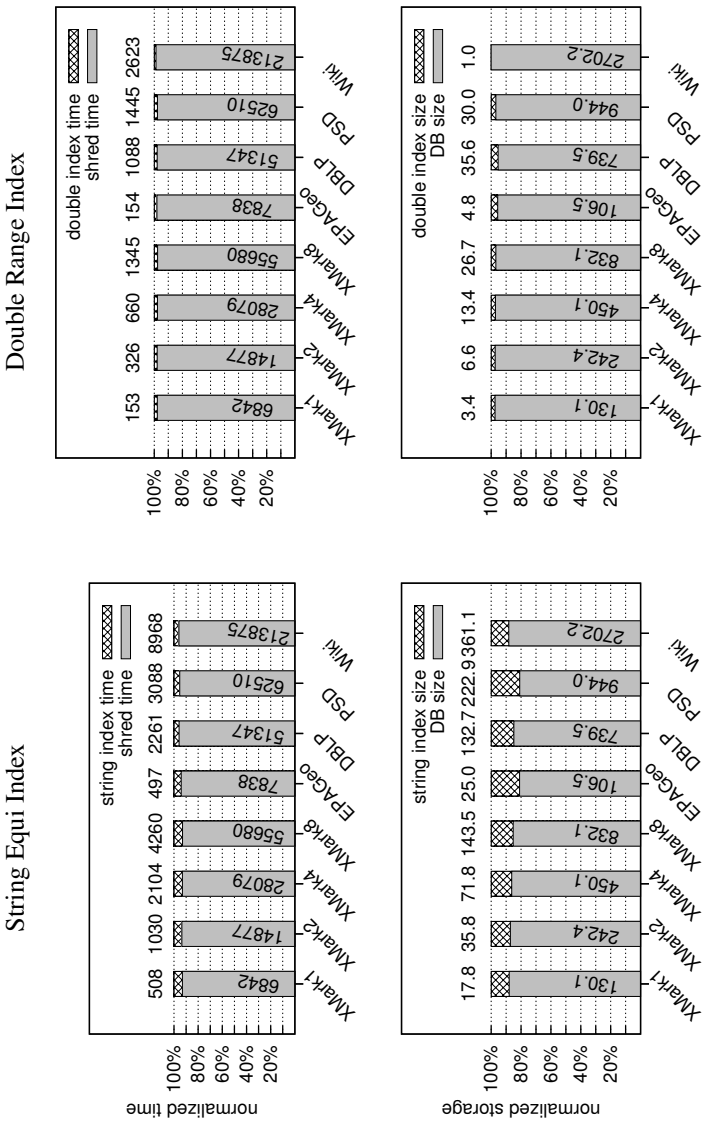


Figure 4.6: String and Double Index Creation Time and Storage Overhead

We next evaluate the update performance of the indices. The update queries were created by first defining the number of text nodes whose values should be updated, and then randomly picking the specified number of the text nodes for each document in the database. The number of updated nodes varied from 1 to 1 million. Each update query was run 20 times for each document and the average time is reported. Figure 4.7 depicts the update times observed for both the string and the double indices. As expected, because of the faster combination step, the double index performs slightly better than the string index. More importantly, for both indices, even in the case of 1 million updated nodes and for a document of size bigger than 2 Gigabytes (i.e., the Wiki dataset), the update time is less than 400 ms. On the other end, the time for updating a small number of nodes is kept less than 50 ms for the smaller documents. The experimental results show that the indices presented in this work are particularly suited for both cases of a) large number of updated nodes in large datasets, and b) small (transactional) updates with few updated nodes.

Finally, we study the stability of the hash function H, used in the equi-lookup index of strings. Many hash functions produce a fixed size output from an arbitrarily long input. In such a design, there will always be collisions, because any given hash has to correspond to a very large number of possible inputs [44]. Figure 4.8 depicts for all 8 documents the distribution of the number of distinct strings that are associated with the same hash value. Almost all of the strings produce a different hash value. Less than 1% of the total string values collide with another one for most of the documents, except the last larger ones, namely PSD and Wiki. But even for those the collisions are kept to less than 10%. Especially for the Wiki document there are cases where 9 distinct strings all hash to the same value. This is observed in the case of data representing URLs, were the different characters between two distinct URLs are repeated every 27 positions, while the rest data remain the same to all strings, such as `http://www..` In this case the hash function fails to produce distinct values because each different character is been eliminated by appearing twice exactly after 27 positions.

In conclusion, the experiments presented in this section show that the time and space overhead introduced by our indices is acceptable, while the update time is kept to the minimum and the XQuery Data Model is respected. Our indices are able to support large documents and update millions of nodes while keeping the false positives – due to hash collisions – during query time to a minimum.

4.7 Summary

We presented a family of generic updatable XML value indices capable of answering equality lookups on string values and range lookups on any XML typed value. These

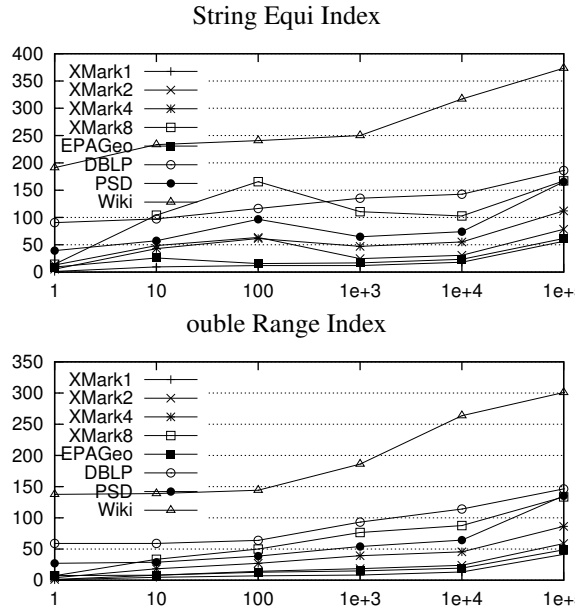


Figure 4.7: Update Time for String and Double Index

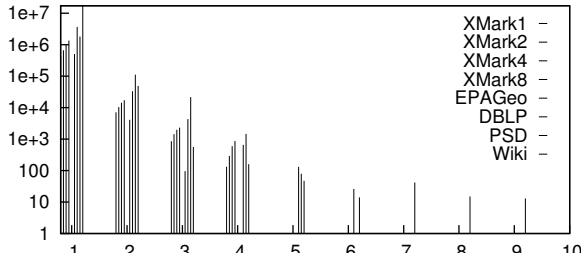


Figure 4.8: Hash Stability

indices are novel compared to prior solutions because they cover the entire document, permit fast updates, and conform with the semantics of the XQuery Data Model. The case were a mixed-content/intermediate node cast to a specific XML type is rare, nevertheless, an XML value index should support it.

We evaluated our design and algorithms in a widely used open-source XML database system, MonetDB/XQuery. The indices presented in this work will be part of a future stable release of MonetDB/XQuery, thus putting our ideas to an every-day test. We intend to expand our work by designing indices capable of answering queries that involve substring matching and regular expressions.

Chapter 5

Space-Economical Partial Gram Indices for Exact Substring Matching

Exact substring matching queries on large data collections can be answered using q-*gram* indices, that store for each occurring q-byte pattern an (ordered) *posting list* with the positions of all occurrences. Such gram indices are known to provide fast query response time and to allow the index to be created quickly even on huge disk-based datasets. Their main drawback is relatively large storage space, that is a constant multiple (typically > 2) of the original data size, even when compression is used. In this chapter, we study methods to conserve the scalable creation time and efficient exact substring query properties of gram indices, while reducing storage space. We first propose a *partial gram* index based on a reduction from the problem of omitting indexed q-grams to the *set cover problem*. While this method is successful in reducing the size of the index, it generates false positives at query time, reducing efficiency. We then increase the accuracy of partial grams by splitting posting lists of frequent grams in a frequency-tuned set of *signatures* that take the bytes surrounding the grams into account. The resulting *qs-gram* scheme is tested on big data collections (up to 426GB) and is shown to achieve an almost 1:1 data to index size, and query performance even faster than normal gram methods, thanks to the reduced size and access cost.

5.1 Motivation

Finding all instances of an ordered sequence of bytes (i.e., a string) in a large file is a fundamental pattern matching problem. Efficient solutions with real-time performance are attractive to many applications, e.g., online string search, word processing software, computational biology and digital forensic pattern search. Both the database and IR communities have investigated many string search topics, for instance approximate (e.g., edit distance based) string selection and top-K joins [29, 88, 50, 42], full text search [4, 51], and digital forensic search [67]. In this work we stick to the basic problem of exact substring matching [52, 55], but note that techniques for solving this are typically the building blocks to address elaborate approximate string matching problems.

We focus on the following efficiency aspects of the exact substring search indexing problem:

- index creation time,
- index space, and
- query response time.

These aspects are influenced by the following dimensions:

- data size (scalability),
- data distribution, in particular the frequency distribution of co-located letters,
- the length of the queries.[1]

Ideally, one would like to achieve an index that takes similar space (or less) as the input data, can be created in time close to a sequential I/O pass over that data, and provides query response time close to a sequential I/O pass over the query result volume; no matter the data and query distribution. These objectives are very hard to achieve all together, and get even harder when considering datasets with a uniform rather than a skewed distribution.

5.1.1 Suffix Techniques

In the algorithms community, the suffix tree [55, 22] and suffix array [52, 25] are being studied as indices to allow fast substring searching. Due to the large amount of information carried in each node and edge of a suffix tree, the storage overhead is 10-20x

[1] We assume a distribution of co-located query letters similar to the data distribution.

		suffix array	full qgram	partial gram	**qs gram**
creation:	time	– –	++	++	**+**
	space	– –	–	**++**	**+**
querying:	short (≤ 5)	+	+	–	**–**
	medium	+	+	–	**+**
	long (> 10)	++	+	+	**++**
result-use:	merge	– –	+	+	**+**

Table 5.1: Exact Substring Index Comparison

the input data in good implementations, which renders suffix trees impractical in most applications. The suffix array is a more space efficient variant which simply constructs an array of positions, where the positions point to the suffixes in the string in lexicographical order. For both suffix trees as well as suffix arrays, algorithms are known whith space and time complexity of $O(|\sigma|)$, where $|\sigma|$ is the length of input data string σ. The excellent property of suffix arrays is that regardless the query (long or short, frequent or rare), lookup is simply a binary search of fixed cost. Sometimes this binary search, which leads to $log|\sigma|$ (on string σ) I/Os, is held against suffix arrays in favour of potentially O(1) hash-based solutions. However, this disadvantage can be mitigated by creating a sparse RAM resident B-tree index on top of the array. Regrettably, however, all known suffix array construction algorithms do *not* truly scale to large datasets. Once the data size exceeds main memory, index creation performance strongly deteriorates due to the need for close to $|\sigma|$ random disk writes. As a result, the best scaling reported "linear" suffix array construction algorithm (using special Linux kernel patches to improve random I/O) has been demonstrated on only 5GB of data [25]. Even on this effectively small – potentially RAM resident – data size, suffix array construction takes many hours. Our work focuses on truly scalable techniques that can be used to manage huge disk-based datasets, where the input data set can – in the case of forensic data search – be a set of full hard drive images (terabyte scale and beyond). As the scalability aspect of suffix methods is several orders of magnitude off target for our objectives, we focus on alternative methods.

5.1.2 Gram Techniques

The basic idea of q-grams is to construct an index on all occurring patterns of q co-located letters (we stick to bytes, in this work – depending on the alphabet a byte can either contain more or less than one character). Figure 5.1 shows a string for a

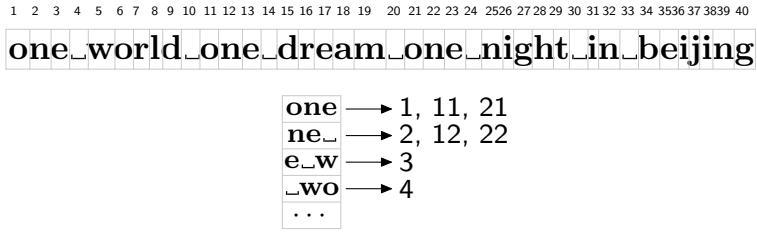


Figure 5.1: A String and Its Posting Lists

slogan of the Beijing 2008 Olympic games. The entire string is indexed using q-grams, with each q-gram associated with a list of postings e.g. 3-gram postings as shown in Figure 5.1. Postings can be stored using IR engines specialising in inverted list processing, but it has been shown that a DBMS can also be used to store and query such data efficiently [29]. A given query ϱ (for exact match) is also decomposed to several q-grams $\mathcal{G}$. By performing merge join on the posting lists associated with each q-gram in $\mathcal{G}$, we can identify all occurrences of ϱ. For example, to find a phrase with the first word 'one' and the second word starting with 'w', we can submit a 5-byte query 'one␣w'. By doing merge join of the three posting lists of 'one' $(1, 11, 21)$, 'ne␣' $(2, 12, 22)$ and 'e␣w' (3), we could identify one match at posting 1. IR systems routinely optimize posting list processing by choosing a merge join that puts the shortest lists first. Also, it is not strictly required to join the posting lists of *all* occurring grams; posting lists may be pruned (omitted from the plan) as long as all letters in the query remain covered by at least one non-pruned gram. A typical approach in gram query processing is thus to order the shortest posting lists first in the plan and prune the longest lists as long as the query remains covered.

5.1.3 Comparison

The creation of a gram-based (inverted) index boils down to using a fast scalable sort method, meeting our scalability objectives. As for the querying aspect, the query processing complexity is linear in the volume of all accessed (non-pruned) posting lists. While this can be substantially more than the final result volume, the positive point is that the access pattern is sequential, thus efficient. In text datasets, there is skew in the q-gram distribution, which results in long posting lists being pruned from query plans, strongly improving query time. The main disadvantage of q-gram based indices is storage space: (i) generally speaking, each byte is stored q times in index entries, excluding the first/last $(q - 2)$ bytes in the head/tail of the string, e.g., the byte 'e'

	3-gram	4-gram	5-gram	6-gram
wikipedia 2.1GB	2.6×10^5	2.0×10^6	8.2×10^6	2.2×10^7
INEX 4.5GB	4.4×10^5	3.7×10^6	1.5×10^7	4.1×10^7
aquaint 3.0GB	1.8×10^5	1.4×10^6	7.0×10^6	2.2×10^7
XMark 7.4GB	4.9×10^4	4.0×10^5	1.8×10^6	5.9×10^6
movie.avi 699MB	4.2×10^6	7.9×10^7	8.9×10^7	9.0×10^7
movie.rmvb 1.2GB	4.2×10^6	1.3×10^8	1.5×10^8	1.5×10^8

Table 5.2: q-gram statistics for various datasets

is stored three times at 'one', 'ne␣' and 'e␣w'; (ii) each byte is stored as a posting, and in case of terabyte datasets one needs 6 bytes for each. Posting lists consist of monotonic increasing numbers, hence these are routinely stored as differences (gaps) and compressed [36, 5]. Skewed data distributions compress well, giving a >2x space reduction. Even with compression, gram indices have size of 2-3x the input in case of skewed data. In more uniform data such as hard disk dumps with many binary video files, there are orders of magnitude more different q-grams (see Table 5.2), and most posting lists are short and thus cannot be compressed, such that the storage space deteriorates to 6x (additionally, the gain of pruning lists during query processing is much less). Note that suffix arrays, which consist of a fully permuted array of postings, are by definition not compressible, hence also take 6x space. Another drawback of suffix arrays is that when the result of a query is used in complex query processing, e.g., in case of regexp string matching, or when in XML databases keyword constraints are combined with structural constraints, multiple results need to be merged. Given the order in gram indices, merging is cheap, whereas suffix arrays need to re-sort the result.

Table 5.1 summarises the strengths and benefits of the discussed approaches, and also introduces the qs-gram method proposed in this chapter. The primary purpose of the qs-grams is to reduce the size of the index further, close to a 1:1 relationship with the input size, while conserving the other good properties of gram indices.

5.1.4 Contributions and Outline

The problem of gram indices for substring matching is formalised in Section 5.2. In Section 5.3 we investigate the possibility of omitting certain grams from the index ("partial grams"), mapping the gram selection problem to a *weighted set covering problem*. For example, in Figure 5.1, in the partial gram approach, we might index 'one', 'e␣w', $\cdots$ (omitting 'ne␣'). A consequence of the partial gram storage is that we now get false positives of string occurrences that may not fully match the beginning

and ending letters of the query, affecting performance. To reduce false positives, we then introduce the qs-gram structure, that adaptively augments the partial grams with signatures covering adjacent bytes. Section 5.4 describes implementation details. The performance is studied in Section 5.5, covering both text and binary data, with sizes up to 426GB (the largest experiment reported in literature so far) showing that the qs-gram approach combines good query performance with strongly reduced index space. After discussing related work in Section 5.6, we sumarize in Section 5.7.

5.2 Preliminaries

A *string* σ is defined to be an ordered sequence of bytes, and its *length* to be $|\sigma| = n$. We denote $\sigma[i, j], 1 \leq i \leq j \leq n$ the substring of σ of length $j - i + 1$ starting at position i. If $i = j$ then we write $\sigma[i]$, referring to the i-th byte of string σ.

A *q-gram* is a string of length q. A q-gram g *appears* at position i of the string σ, if $g = \sigma[i, i + q - 1]$. The set of all q-grams that appear in a string σ is denoted by $\mathcal{G} = \{g_1, \ldots, g_k\}$. Each q-gram $g_i \in \mathcal{G}$ is associated with a list of *postings* $\mathcal{P}(g_i) = \{p_1, \ldots, p_l\}$, where each *posting* $p_i \in \{1, \ldots, n - q + 1\}, i \in \{1, \ldots, l\}$ refers to a position in the string σ where g_i appears. Evidently, a q-gram g_i can appear more than once in a string σ.

A q-gram g *covers* the bytes $\sigma[i], \ldots, \sigma[i + q - 1]$ if there is a posting $p \in \mathcal{P}(g)$ such that $p = i$. A set $\mathcal{G}$ of q-grams *covers* the entire string σ, if every byte $\sigma[i]$ of the string σ is covered by at least one q-gram. In other words, we require each byte of the string σ to be referred by at least one posting of a q-gram.

Finally, a *string matching query* is defined to be the query which given the string ϱ of length $|\varrho|$, requests all positions i of the string σ such that $\varrho = \sigma[i, i + |\varrho| - 1]$. For simplicity, we will refer to the string matching query as *query* ϱ.

5.2.1 Q-Gram based indices

Q-Gram based indices utilise the set of q-grams $\mathcal{G}$ to efficiently answer queries on approximate and exact substring match, text auto-completion, and for error-correction. The general approach, which will be referred to in the sequel as the *full q-gram index (FG)*, covers each byte of the string σ exactly q times. This is achieved by including in the set $\mathcal{G}$ all possible q-grams of the string σ, thus each byte $\sigma[i]$ will be covered by those q-grams that have postings equal to $i - q + 1, \ldots, i$. The most common choice for q is 3. The total number of postings in a full q-gram index is $|\sigma| - q + 1$.

In this work, we argue that for *space-economical* q-gram based indices it is better to use *partial q-gram indices (PG)* instead of full. However, for providing complete

answers to substring matching queries, we have to ensure that each byte in the string σ is covered at least once. The problem at hand is formally defined as follows: *Given a string σ and its set $\mathcal{G}$ of q-grams, find a minimal subset $\mathcal{G}' \subseteq \mathcal{G}$, such that $\mathcal{G}'$ covers the entire string σ*. The *coverage* requirement is necessary to ensure that no matching substrings are missed, thus resulting in false negatives. However, a partial q-gram index will return exact matches plus a small list of candidate substrings because only a subset of $\mathcal{G}$ is indexed. In Section 5.5 we experimentally compare the number of candidates and the actual matches for the partial q-gram indices.

Another important difference between a full q-gram index and a partial one is that the former can match any substring that is no shorter than q, while the latter loses some expressive power: it can find matching candidates for a substring that is no shorter than $2q - 1$.

Lemma 1 *If $\mathcal{G}'$ is a partial set of q-grams for string σ and ϱ is a* query *on σ that can be evaluated correctly, then the length $|\varrho|$ of ϱ is no shorter than $2q - 1$.*

Proof. There are 3 cases. First, if $|\varrho| < q$ then it is impossible to match any q-gram with the query string ϱ since the length of ϱ is smaller than the length of the indexed q-grams[2]. The second case is when $q \leq |\varrho| \leq 2q - 2$. Let query ϱ match on string σ at positions $\sigma[i, i + 2q - 3]$. Also, let assume that these bytes of σ are covered by the two q-grams g_1, g_2 with postings $\mathcal{P}(g_1) = \sigma[i-1, i+q-1]$ and $\mathcal{P}(g_2) = \sigma[i+q, i+2q-2]$. In this case, no q-gram that can be extracted from query ϱ will match any q-gram used to cover the bytes of string σ from position i to $i + 2q - 3$, and thus false negatives will appear. Finally, the third case is when $|\varrho| \geq 2q - 1$, then there should always be a q-gram that appears on both the query ϱ and the string σ. If no such q-gram exists, then the query result is empty.□

5.2.2 Positional Merge Join

Consider two q-grams g_1, g_2 and their respective postings $\mathcal{P}(g_1) = \{p_1, \ldots, p_l\}$ and $\mathcal{P}(g_2) = \{p'_1, \ldots, p'_n\}$ over string σ. Also, consider a query ϱ where the q-gram g_1 appears in position i of ϱ, and q-gram g_2 in position $i+k$ of ϱ, where $i+k < |\varrho|-q+1$. In order to find which postings of g_1 and g_2 are valid candidates for matching query ϱ to string σ, we have to join $\mathcal{P}(g_1)$ with $\mathcal{P}(g_2)$. This join must take into consideration also the distance k between g_1 and g_2 in ϱ because the same distance must be preserved in σ, too. The *positional merge join* is the join operation of two posting lists and an offset k.

[2]We discount the non-practical approach of merging all superset grams $\{P_x | x \in \mathcal{G}' : x \subset q\}$ into a list of candidates.

For the two posting lists $\mathcal{P}(g_1)$, $\mathcal{P}(g_2)$ and an offset k, the operation *PosMerge-Join* $(\mathcal{P}(g_1), \mathcal{P}(g_2), k)$ is defined as:

$$\begin{array}{ll} \text{SELECT} & \mathcal{P}(g_1).posting \\ \text{FROM} & \mathcal{P}(g_1).posting, \mathcal{P}(g_2).posting \\ \text{WHERE} & \mathcal{P}(g_1).posting = \mathcal{P}(g_2).posting - k \end{array}$$

Let assume that there is a partial 3-gram index for the string in Figure 5.1. This partial index includes the 3-gram 'one' with postings list $\mathcal{P}(\text{'one'}) = \{1, 11, 21\}$ and the 3-gram 'e␣w' with postings list $\mathcal{P}(\text{'e␣w'}) = \{3\}$. Let the query ϱ ='one␣w'. Query ϱ is decomposed to three 3-grams 'one', 'ne␣', and 'e␣w'. Since only 3-gram 'one' and 'e␣w' exist in the partial index, we apply the positional merge join operator on the postings lists of these two 3-grams with an offset 2. The result is posting $\{1\}$, which is the position that ϱ matches the string in Figure 5.1.

The positional merge join will return a candidate list of positions in the string σ for the query ϱ to be verified. This happens when there is no q-gram in the index that covers the first and last $q-1$ bytes of the query ϱ. However, all bytes in between should be matched with some q-grams in the partial index. If this is not the case, then it is safe to conclude that there is no occurrences of query ϱ in σ. This provides opportunity during query evaluation, to reject mismatches early, and to only verify candidates in the raw data (i.e., the string σ) at the very end, only for those queries whose borders were not fully covered.

5.3 Partial Q-Gram Indices

This section presents techniques for choosing the appropriate set of q-grams to be included in a partial q-gram index. We use existing techniques from the set-theory field and extend them to meet practical requirements in terms of indexing and querying time. In addition, we introduce *signature* and *hash based* q-grams and then propose the novel qs-gram approach. These variations refer to the techniques used to store the posting lists $\mathcal{P}$ of q-grams. The goal is to minimize the number of the candidate list returned during query matching, and thus achieve faster execution time.

5.3.1 Partial Q-Gram Selection

The selection of the q-grams to be included in the partial index should satisfy the following two objectives: *i)* each q-gram must have a sorted list of postings in order to minimise I/O during posting fetching, and *ii)* the final candidate list must be small enough to reduce the cost of verification against the raw data, i.e., the input

string σ. These two requirements converge to index only highly selective q-grams, i.e., the ones with a small number of postings. More formally, we minimise the function $\sum_{g_i \in \mathcal{G}'} \mathcal{F}(g_i)$, where $\mathcal{F}(g_i)$ denotes the frequency of q-gram g_i, i.e., the number of postings in the list $\mathcal{P}(g_i)$. This problem can be reduced to the *weighted set covering problem*. In terms of the set covering problem, the universe to be covered is all positions of the string σ, i.e., all $p_i \in S = \{1, \cdots, n\}$, where n is the length of σ. The collection of all posting lists $\mathcal{P}(g_i)$ of the q-grams g_i appearing in σ are used to cover the universe $S = \{1, \cdots, n\}$. The weight of each $\mathcal{P}(g_i)$ is the number of elements in the list, i.e., $\mathcal{F}(g_i)$.

Algorithm 5.1 Selection of Partial Grams CHOOSEPARTIALGRAMS(σ, q)

Input: σ = the input string, q = the length of the grams
Output: a set of q-grams $\mathcal{G}'$ whose postings cover σ

```
G' ← ∅;
scan σ and derive all posting lists P(g_i), g_i ∈ G;
calculate F(g_i) = |P(g_i)| for all g_i ∈ G;
sort all F(g_i) in descending order;
cnt[i] ← 0 for i ∈ {1, ..., |σ|};
for each g_i in descending order of F(g_i)
        selected ← false;
        if g_i is the first or the last q-gram in σ
          G' ← G' ∪ {g_i};
          selected ← true;
        else
          for each posting p in P(g_i) and selected is false
                  for k from 0 to q − 1
                    if cnt[p + k] is equal to q − 1
                      selected ← true;
                      G' ← G' ∪ {g_i};
                      break;
        for each posting p in P(g_i)
                for k from 0 to q − 1
                  if selected is true
                    cnt[p + k] ← q;
                  else if cnt[p + k] < q − 1
                          cnt[p + k] ← cnt[p + k] + 1;
return G';
```

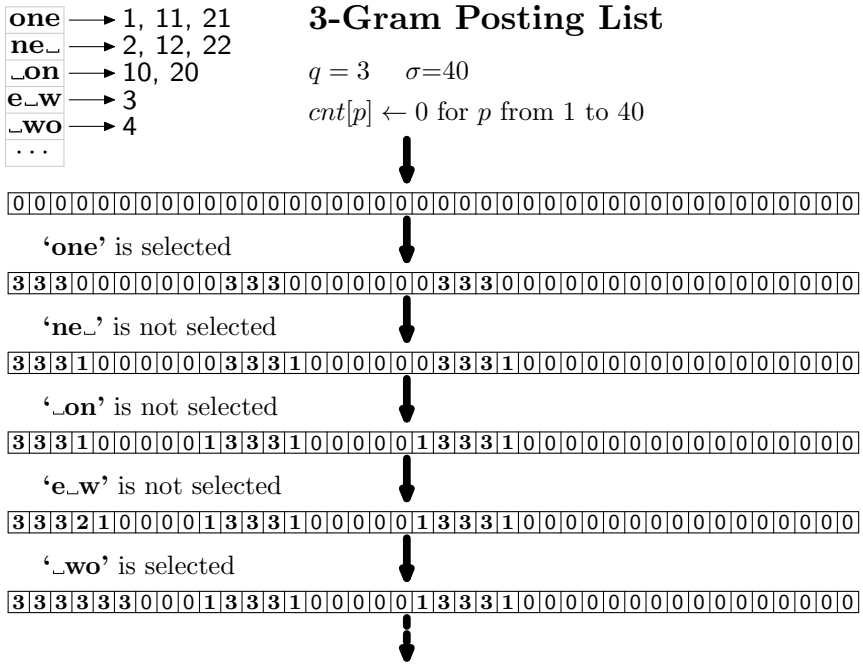


Figure 5.2: Algorithm Sample

The weighted set covering problem is a well studied problem and the optimal solution can be approximated by the following greedy algorithm, within an approximation factor of $H_n(\leq \ln n + 1)$. Let $\gamma(g_i) = \mathcal{F}(g_i)/|\mathcal{P}(g_i) \cap S|$ be a weight function, which considers both the q-gram frequency $\mathcal{F}(g_i)$ and the number of postings $|\mathcal{P}(g_i) \cap S|$ to be covered next. At each step, the q-gram g_i with the minimum value of the weighted function $\gamma(g_i)$ is picked and added to the partial index. Next, all postings that the posting list $\mathcal{P}(g_i)$ could cover are removed from the set S. Set S contains all the uncovered positions up until this point. Finally, the function γ is re-computed for all remaining unused q-grams. The process is repeated by choosing the next smallest weighted q-gram, until all positions are covered, i.e., until $S = \emptyset$.

The greedy algorithm produces a good approximation of the optimal solution in polynomial – to the size of σ and number of posting lists – time. However, the complexity remains quadratic, thus when both n and $\mathcal{G}$ are large, the classical greedy algorithm fails: it already takes hours to select 3-grams from a 64MB document. In the

following we describe a more greedy algorithm with worse approximation factor but with linear execution time in the size of the input σ.

5.3.2 A Scalable Set-Cover Algorithm

The complexity of the previous greedy algorithm originates from the requirement that in each step the *new global minimum* value of $\gamma(g_i)$ is selected. However, for the applications we study, it is preferable to sacrifice the approximation factor bounds of the set cover problem, and instead gain on the indexing time. The basic idea of Algorithm 5.1 is to scan all q-grams in descending frequency order and greedily choose which grams to keep without recomputing the global weights in each step. More specifically, a q-gram g_i is *not* included in the partial index if it has a large frequency $\mathcal{F}(g_i)$, and only if all positions can be still covered by the remaining q-grams. The technical details of Algorithm 5.1 are best described with the following example.

Going back to the example in Figure 5.1, the string σ = 'one␣world␣one ␣dream␣one ␣night␣in␣beijing' has 40 positions to be covered and 33 distinct 3-grams. A single pass over string σ will extract all q-grams and their posting lists. The q-grams are then sorted in descending order of their frequencies . Next, a counter cnt$[i]$ for each position in the string σ is initialized to 0.

Figure 5.2 depicts the first few steps of the CHOOSEPARTIALGRAMS algorithm for the string σ of Figure 5.1. The most frequent q-gram is 'one', but it has to be selected because it covers the first position of string σ. Thus, $\mathcal{G}' = \{$'one'$\}$. Next, the count *cnt* of each position in string σ covered by 'one' is set to 3. The second row of Figure 5.2 shows the values of the count array *cnt* for each position in the string σ after 3-gram 'one' has been selected.

The second most frequent 3-gram is 'ne␣'. The positions that 3-gram 'ne␣' covers are either already covered (e.g., position 2) or the count value *cnt* is less than q-1. Therefore, 3-gram 'ne␣' can safely be disregarded since it is too frequent and other 3-grams do exist for covering the uncovered positions. However, before disregarding 3-gram 'ne␣', each of the positions $\{4, 14, 24\}$ is incremented by 1 to signify that there are still q-1 q-grams left that can potentially cover these positions. The third row of Figure 5.2 depicts the new state of the count array.

The next two 3-grams are 'ne␣' and 'e␣w'. Both of them can be disregarded, because they are high in the frequency list and none of the positions that they cover on string σ have a count *cnt* equal to q-1. However, '␣wo' has to be selected to ensure that position 4 of string σ is covered.

Algorithm 5.1 terminates with thirteen 3-grams selected for the partial 3-gram index: $\mathcal{G}' = \{$one, ␣wo, rld, d␣o, ␣dr, eam, m␣o, ␣ni, ght, ␣in, ␣be, iji, ing$\}$.

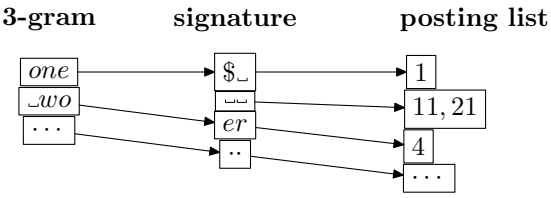


Figure 5.3: Signature-based Posting List

The complexity of algorithm CHOOSEPARTIALGRAMS is $O(|\sigma| + |\mathcal{G}|ln|\mathcal{G}|)$. First, a linear scan is performed on σ to derive all q-grams, their postings lists, and the frequencies $\mathcal{F}(g_i)$. The cost of sorting all q-grams is $O(|\mathcal{G}|ln|\mathcal{G}|)$. The big loop is a linear in the size of σ, since each position of σ will be visited at most q times.

5.3.3 Signature-based Gram Indices

There are two main limitations regarding partial q-gram indices: *i)* some very frequent q-grams are unavoidably selected to ensure that each byte of the string σ is covered; *ii)* during query evaluation, the indexed partial q-grams might not be sufficient to fully answer a query ϱ, thus resulting in a candidate list. In order to prune the false positives from the candidate list, the string σ has to be examined, causing time consuming I/O access.

To alleviate the aforementioned limitations, we propose the use of *signatures*. A signature s of a q-gram g is defined to be the concatenation of the two bytes *guarding* g in string σ. More specifically, let a q-gram be g_i and its posting list be $\mathcal{P}(g_i) = \{p\}$. Then, the signature s of g_i in position p is $s = \sigma[p-1]\sigma[p+q]$. A q-gram may have as many signatures as the number of postings in its posting list $\mathcal{P}$. The collection of all the signatures of a q-gram g, together with its posting list will be referred to as an s-gram. The q-gram g is also called the *infix* of its s-gram. Finally, we will refer to the first byte of a signature s as $s[1]$ and the second as $s[2]$. If $s[1]$ or $s[2]$ do not correspond to a byte (i.e., they point to the positions before the start or after the end of the string σ) then the symbol \$ is used.

Figure 5.3 shows the signatures s of the first two q-grams of the string σ of Figure 5.1. The 3-gram 'one' has the posting list $\{1, 11, 21\}$ and the list of signatures $\{\$␣,␣␣\}$. Consequently, the postings list of 3-gram 'one' can be split to two smaller ones, namely $\{1\}$ and $\{11, 21\}$, each one referring to a different signature of 'one'.

Signatures are used to split long posting lists, in order to reduce both the number of postings fetched and the number of candidates produced during query processing. This

is true because s-grams are more discriminative than q-grams. However, signatures may have a considerable storage overhead. In theory, the number of signatures for each q-gram is at most 2^{16} (for 2 bytes), but in practice the number of signatures for most q-grams is far below this upper bound.

The frequency distribution of q-grams in the text data sets we used loosely follows Zipf's law. This indicates that many q-grams will have short posting lists. For these q-grams, the overhead of maintaining their signatures is higher than retrieving the whole posting list during query evaluation. Moreover, some long posting lists may be split to very fine grained pieces, as a result of a large number of different signatures. For these cases, it is preferable to combine the fine grained lists into groups and thus save storage overhead. The next section presents techniques on how to dynamically make the decision on which posting lists to keep unmodified, which ones to split with the use of signatures and which ones to be combined into groups.

5.3.4 QS-Grams

We propose a novel approach to marry the merits of q-gram and s-gram, named qs-gram, with the following objectives: maintain less grams on the index entries, split long posting lists for improved query performance, and merge short postings to further save index space.

Figure 5.4 gives an overview of qs-gram. Two histograms are required, H_s and H_q, for s-grams and q-grams, respectively. Assume that $q = 3$, the number of distinct q-grams is up to 2^{24} and for s-grams 2^{40}. The histogram H_s might be too large to fit in memory. A *lossy* technique will be discussed in Section 5.4 on how these histograms can be efficiently built in the secondary memory.

As depicted in Figure 5.4, there are less q-grams than s-grams, and in average, the frequency of a q-gram is higher than that of an s-gram. Suppose that a threshold t is used to determine whether a gram is frequent or not, the grams are classified as follows:

1. Frequent s-grams ($frequency \geq t$) will end up with their own posting list. Referring to the left part of H_s in Figure 5.4, they reside in an s-gram dictionary.

2. Infrequent q-grams ($frequency < t$) will not be split at all. These q-grams are maintained in a q-gram dictionary, as shown in Figure 5.4, the right part of H_q.

3. For the grams in between, i.e., q-grams with frequencies greater than t and s-grams having frequencies less than t, a hash-based approach is utilised.

If a q-gram g has a long posting list, but has many s-grams with short posting lists, the hash function $h()$ is used to combine s-gram lists in a single one. To construct

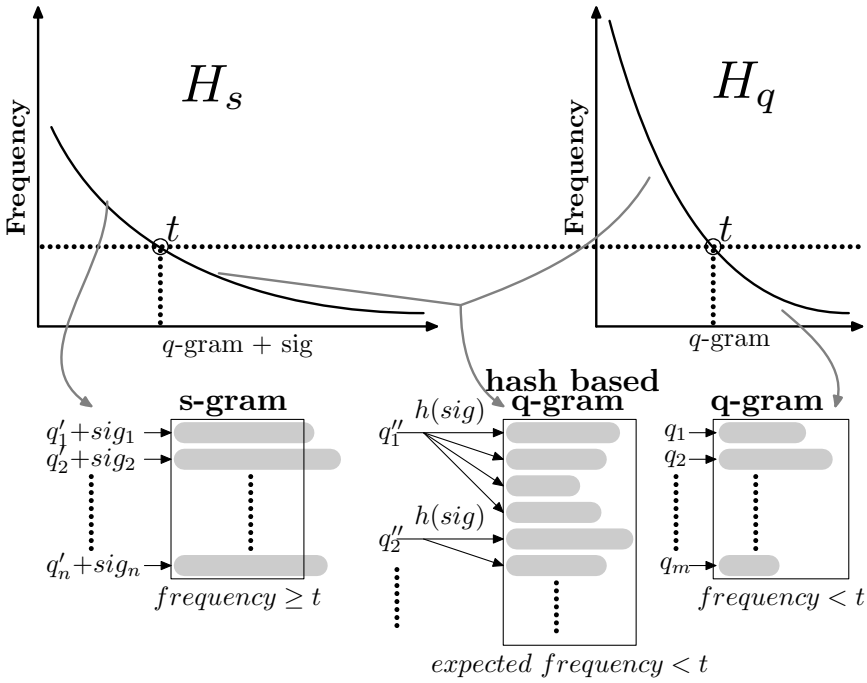


Figure 5.4: An Overview of QS-Gram

the hash-based q-gram dictionary, a number of buckets is dynamically allocated for each q-gram according to the frequencies of q-grams. For instance, if a q-gram g_e with frequency 35 is to be split and the threshold is $t = 10$, four buckets will be allocated for g_e. As the hash-function randomly combines infrequent signatures, the amount of postings falling in each hash bucket approximates t. The keys in the hash-based q-gram index are just q-grams stored in order, each leading to an array of buckets. To look up an infrequent s-gram, its infix q-gram g is first found, after which the hash function $h()$ is applied on s to compute the specific bucket, as shown in Figure 5.4. The main reason for using a hash function to combine buckets is that it requires no additional storage space.

The qs-gram index is designed to merge short posting lists of s-grams associated with one q-gram, so as to get almost equally length t posting lists. The certainty that posting lists have a certain minimum length ensures that even in skewed gram distributions storage volume is dominated by postings, and not by the dictionary, and also ensures that compression is functional on all postings. As twice the list length t is an upper bound on the average number of false positives, it should be chosen relatively small.

5.4 Implementation Details

This section describes the implementation details for building the q-gram indices. Since the input data is larger than the available memory, we present *chunk based* algorithms for full and partial q-gram indices that efficiently divide the process to fit in memory. We then present how a query is evaluated against the proposed q-gram indices.

5.4.1 Gram Generation

We describe not only effective, but also efficient and scalable generation algorithms for full q-gram, partial q-gram and qs-gram indices. Since, the input data may not fit entirely in main memory, and in order to achieve good scalability, a *chunk based* generation strategy is used. Specifically, the input data is partitioned into chunks, i.e., a sequence of bytes with equal length, and each chunk is processed independently. The algorithm can be executed either sequentially or in parallel over each chunk. After the termination of the chunk based generation, the *local* posting lists of each chunk are merged using a multi-way merge-union operation. The size of each chunk can be tuned, in order to render the algorithm cache or memory resident.

To decrease storage overhead, as well as to improve I/O system performance, all posting lists are compressed before being written to disk. Since each posting list is

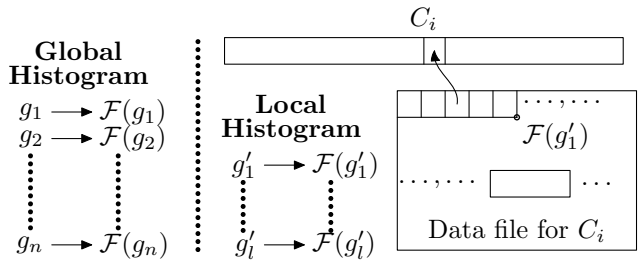


Figure 5.5: Global and local data structures

sorted, delta coding [36, 5] is adopted. Delta coding computes the difference between consecutive postings, and records only the value of the first posting and the following differences. Other compression techniques are orthogonally applicable.

For the chunk-wise processing, the input string σ is partitioned to $\lceil |\sigma|/m \rceil$ chunks $C_1, C_2, \ldots, C_{\lceil |\sigma|/m \rceil}$, each of size m. Each chunk C_i also includes $q-1$ bytes from the previous chunk C_{i-1} to ensure that no q-gram is missed during index creation. To illustrate this, assume that $q = 3$ and $m = 2$, thus string σ is partitioned into 2 chunks, C_1 and C_2:

$$\sigma = \sigma[1] \cdots \sigma[m-2] \underbrace{\sigma[m-1]\sigma[m]}_{q-1=2} \Big| \sigma[m+1]\sigma[m+2] \cdots \sigma[2m]$$

The first chunk C_1 is from position 1 until m, while the second chunk C_2, from position $m+1$ up to $2m$. However, this way each chunk is processed separately, and the 3-gram $\sigma[m-1]\sigma[m]\sigma[m+1]$ would never be found. To avoid this *miss*, C_2 is redefined to also include $q - 1 = 2$ bytes from the previous chunk, thus C_2 starts at $m-1$ and ends at $2m$.

A *global* and a *local* histogram are used to record the frequencies of the q-grams found in string σ. The global histogram records the total frequency of each q-gram found on the *entire* input data. This histogram remains always in memory, since it is updated by every chunk. The left part of Figure 5.5 depicts the structure of the global histogram. It only consists of the q-grams and their frequencies.

On the other hand, a different local histogram exists for each chunk, which stores only the frequencies of the q-grams found in a specific chunk. To free up memory, each local histogram is flushed to disk after we finished processing the chunk. In addition to the local histogram, each chunk has its own *data file*, which contains all posting lists $\mathcal{P}$, i.e., the positions in the input string where the q-grams are located. The local

histogram is used to navigate in the data file by computing the offset and the length for the posting list of each q-gram. The right part of Figure 5.5 depicts the structure of the local histogram and the data file for chunk C_i.

Algorithm 5.2 Chunk-Wise Processing CHUNKPROCESSING(C_i, q)
Input: current chunk C_i, gram length q
Output: a local histogram H_i and a data file $\mathcal{D}_i$

$k \leftarrow 0$;
define struct *temp* $= \{$*q-gram*, *position*$\}$;
while $C_i[k]$ not the end of chunk
    temp$[k] \leftarrow C_i[k, k+q-1], k$;
    $k \leftarrow k+1$;
radix_sort(*temp*) on *q-gram, position*;
$H_i \leftarrow$ merge the same grams and compute the frequencies;
populate $\mathcal{D}_i$ from *temp.position*;
update global histogram with H_i;
return $H_i, \mathcal{D}_i$;

Algorithm 5.2 details how the local histogram and the data file is populated for each chunk. A temporary structure is used that records each q-gram and its position in the chunk C_i. First, chunk C_i is sequentially scanned and every q-gram is extracted and stored to *temp*. Afterwards, the *temp* structure is sorted on the value of the q-grams by using the radix cluster algorithm [53] (a radix sort on the $8q$ lowest significant bits when viewing the concatenated bytes of a gram as a number) in $O(qm)$ time. Next, the *temp* structure is merged to contain all the postings of the same q-gram and the frequencies are computed. This is possible, because radix cluster will group all same q-grams together. Next, the data file $\mathcal{D}_i$ is populated by dumping sequentially the *merged* and *sorted temp* structure. Finally, the global histogram is updated by merging the current local histogram.

The above process is repeated for all chunks. Afterwards, the local data files $\mathcal{D}_i$ are merged to one single global data file $\mathcal{D}$. Each posting list of the local data files is written to the correct offset of the single global data file by consulting the global histogram. The resulting global data file contains the entire posting list of each q-gram in the input string σ.

An alternative approach is to scan the entire input string and collect the global histogram without writing data to disk. In a second scan, data could be written to

the final global data file without merging local posting lists. However, this alternative approach writes data using random I/O, which renders the algorithm I/O bound. For the proposed approach, all disk reads/writes entail sequential I/Os, which is over an order of magnitude faster than random.

Index generation thus produces two global structures: a histogram H and a data file $\mathcal{D}$. The global histogram contains entries of the form $[q\text{-}gram|data\_offset]$, where *data_offset* refers to the position in the data file that the q-grams' posting list can be found. The q-grams are sorted to facilitate binary search during query evaluation. The data file is the concatenation of the posting lists of all q-grams.

The generation procedure for partial q-grams is similar to full q-grams. The only difference is that in each chunk, instead of indexing all q-grams, partial q-grams are selected using the algorithm proposed in Section 5.3.2. The decision is made locally by consulting only the local histograms.

For generating the qs-gram index, two global histograms are required, as depicted in Figure 5.4. One global histogram is needed for q-grams and one for s-grams. These are used to assist the gram classification by examining their frequencies. The two global histograms are produced in the same way as described until now. However, the s-gram histogram is an array of $[(q+2)\text{-}gram, posting\_list]$ entries sorted first on the s-gram level then on the postings. Recall that the number of s-grams is up to 2^{40} when $q = 3$, which could render the s-gram histogram to be out of memory. A lossy *probabilistic counting* technique is adopted for this case, similar to [56], where there is a limited histogram buffer. In principle all s-grams are collected, but when the buffer gets full, s-grams with a small count are removed to make space. The adaptive approach works as follows. A lower bound, initially set to 1, is used to guard whether an s-gram should be removed or not. If after removing entries with only 1 occurrence, there is still not enough space, the lower bound is doubled and the above removal operation is repeated until there is enough space in the buffer. This leads to an incomplete histogram on the s-gram level, but generally only s-grams that occur very infrequently will be missing.

After the first scan of the input string, we determine using threshold t whether a q-gram should be split with signatures: $i)$ very frequent s-grams will end up with a private posting list, recorded in an s-gram dictionary; $ii)$ very infrequent q-grams will not be split at all, stored in a q-gram dictionary; $iii)$ the hash based approach is used in between.

Next, the input string is scanned for the second time. In this second pass, the local histograms and data files are generated for each chunk. We point out two differences with full q-gram generation. First, for hash based q-grams, instead of maintaining for each q-gram a global offset, each bucket records an offset. Secondly, posting lists are sorted in s-gram level. Therefore, during the construction of the q-grams and hash

based q-grams, the posting list is re-sorted.

The final multi-way merge-union on sorted posting lists is similar to that for q-grams. The only difference is that the three types of dictionaries must be handled separately.

5.4.2 Query Processing

Consider a query ϱ of size $|\varrho| = k$ where $k \geq 2q - 1$. The query ϱ is decomposed into a set of q-grams $\mathcal{G}_\varrho = \{g_1, \ldots, g_{k-q+1}\}$, where each $g_i = \varrho[i] \cdots \varrho[i + q - 1]$ for $1 \leq i \leq k - q + 1$. If signatures are used, the signature s_i of each q-gram g_i is $s_i = \varrho[i-1]\varrho[i+q]$. The first byte s_1 of the first signature and the second byte s_{k-q+1} of the last signature are unknown. They are set to a wildcard $*$ that matches any single byte, i.e. $s_1[1] = s_{k-q+1}[2] = *$. With the above information about q-grams and signatures for the query ϱ, and the stored histograms, a set of posting lists is fetched from disk and the positional merge join is applied.

In an early optimization step, *negative queries*, i.e., queries that have empty results, can be quickly identified before even the postings are fetched. In the case of full q-grams, if any posting list of $g_i \in \mathcal{G}_\varrho$ is empty ($\exists \mathcal{P}(g_i) = \emptyset$), ϱ is a negative query, since a query will have an empty result if at least one of its substring does not appear. For example, let a string $\sigma_e =$'*one␣world*' and two queries ϱ_1 :'*one␣w*', 'ϱ_2 :*one␣v*'. For query ϱ_1, all its q-grams are indexed by the q-grams of σ_e. With regards to ϱ_2, one of its q-gram. i.e., $g_3 =$'*e␣v*', is not indexed, and thus ϱ_2 is a negative query over σ_e.

In the case of partial q-gram and qs-gram, the criteria are different since some q-grams are pruned: *i)* if all posting lists are empty ($\forall \mathcal{P}(g_i) = \emptyset$) where $g_i \in \mathcal{G}_\varrho$, ϱ is a negative query, and *ii)* if the i-th byte of the query ϱ is not covered by any q-gram, and $q - 1 < i < |\varrho| - q + 3$, then ϱ is a negative query. Notice that we can not determine if a query is negative if the bytes that are not covered are the $q - 1$ bytes located at the borders of the given query.

The next step, and since it has been established that the query ϱ might have a non-empty result set, the relevant posting lists must be fetched. For the full q-gram index, the posting lists of all q-grams are indexed. However, in order to further optimize the query evaluation, q-grams with very long posting lists can be omitted during query evaluation. The basic requirement is that each byte of the query is covered at least by one q-gram. For instance, to cover a seven-byte query '*one␣wor*', the grams 'one' and 'wor' can be used. To cover the remaining byte '␣', any of the q possible grams (i.e., '*ne␣*', '*e␣w*' and '*␣wo*') are checked, and the one with the smallest frequency is selected.

In the case of the partial q-gram index, unfortunately the same optimization techniques can not be applied. The q-grams are selected chunk-wise and based on local

decisions (i.e., the local histogram), thus there is no guarantee that a specific q-gram will be found across the entire input string. Therefore, no matter whether a query ϱ is fully covered or not, all posting lists must be fetched and examined.

The qs-gram index has three histograms. The infrequent q-gram histogram is processed similarly to the partial q-gram index. However, the histograms for the s-grams and hash based q-grams are treated differently, since the stored signatures must be taken into account. First, the signature s_i of each gram g_i is extracted. There are three cases:

1. $s_i = \varrho[i-1]\varrho[i+q]$, where $i \neq 1$ and $i \neq k-q+1$. The signatures are ordered, hence a binary search is used to identify the specific entry for s_i.

2. $s_i = \varrho[k-q+1]*$, which refers to the last q-gram of query ϱ. With the use of binary search all signatures whose first byte is $\varrho[k-q+1]$ will be identified.

3. $s_i = *\varrho[q+1]$ i.e., the first q-gram. The second byte of the signatures is not sorted, therefore a binary search can not be used and all signatures of g_i will be fetched.

If the q-gram is not found in the s-gram histogram, the hash based q-grams are searched. The hash value of s_i is calculated and used to identify the bucket that the q-gram should reside in. If it is found, the corresponding posting list is fetched. Notice that, if the signature contains a wildcard (for g_1 and g_{k-q+1}), the hash value cannot be decided and thus all posting lists of the current q-gram will be fetched. Finally, if the q-gram is not located either in the s-gram nor the hash based histogram, the q-gram histogram is queried.

All fetched relevant posting lists are then fed into a positional merge join operator in order to identify the valid occurrences of the query ϱ in string σ. For the full q-gram index, a standard multi-way positional merge join is sufficient. However, for the partial q-gram and qs-gram indices, the positional merge join must adaptively determine in a chunk-wise fashion which q-grams are present and which were (potentially) pruned by the set-cover algorithm:

1. When processing a chunk C_i, if all posting lists have no postings indexed in chunk C_i, or some byte $\varrho[k]$ of a query ϱ is not covered where $q-1 < k < |\varrho| - q + 3$, chunk C_i can be safely skipped,

2. If some q-grams have postings (are indexed) in the chunk C_i, and the corresponding q-grams cover all bytes of the query[3], the standard multi-way positional merge join is used.

[3]Note that if a certain q-gram is stored as an s-gram, we have information on $q+2$ query letters.

3. Otherwise, that is, if the present q-grams cover the inner part of a query but not the outer boundaries, a multi-way positional merge join is still performed, but these results are marked as candidates, requiring verification afterwards.

Note that for the qs-gram index, the candidate list of the third case will be shorter than that of the partial q-gram, thus less postings would have to be verified against the input string.

5.4.3 Extensions

Possible future optimisations in qs-gram query processing address the problem that when examining a chunk and seeing that a certain q-gram is *not* present, we pessimistically assumed in the above that the q-gram must have been pruned, while it could also be that there was simply no occurrence (in that case, there is no query result in this chunk, and whatever we emit above is a false positive). To address this, we could augment our indexing structure with a bit-string for each q-gram, that records whether a q-gram was preserved by the set-cover algorithm in a specific chunk or not. For example, the bit-string 11001 means that a q-gram g is present in chunks $1, 2$ and 5. This information can then be used to emit considerable less false positives, but raises the future research question as to the space/time trade-offs of keeping this extra information, especially on data sets with many q-grams. Even if no bit-strings are kept, a similar optimization can already be made in the qs-gram approach when merging a signature or hashed list (i.e. in case of a globally frequent q-gram). If this list turns out to have no postings in a chunk, we can still analyze the other hash and signature tables for the same infix q-gram in this chunk. Because the set-covering pruning decisions are made on the q-gram level, presence of any other infix matching information in the indices is proof that the q-gram was not pruned in C_i, and can be used to reduce false positives. Note, however that searching for this extra proof may cause additional index I/O at query time. One potential strategy would be to use only a limited set of (or a single) long signature list(s) with the same infix to steer this optimization. Alternatively, when processing hash lists, with all hash-bucketed postings list adjacent on disk, we might simply re-use whatever information is present in the large disk block I/O unit we read for opportunistic pruning.

Partial q-grams are selected separately in different chunks. In order not to miss any query result, the postings near the chunk boundaries should be particularly considered, since a query may cross between two chunks. We propose a simple solution. When the iterators of all posting lists are moved to the next chunk, for each posting list of the i-th q-gram g_i of the query ϱ, an iterator is moved backward to the last previous posting p_i. The candidate query match is ϱ' such that $\varrho[i] = p_i$. To check whether ϱ' crosses two

chunks, only the first byte $\varrho'[1] = p_i - i + 1$ and the last byte $\varrho'[|\varrho'|] = p_i + |\varrho'| - i$ need to be checked.

5.5 Performance Study

We conducted extensive experiments with all discussed q-gram indices, investigating index creation time, index space and query performance. The following datasets were used:

- wiki-article: the Wikipedia archive that contains current versions of article content (22GB)[4].
- wiki-meta: the complete Wikipedia archive including discussion and user pages (44GB).
- XMark[5], using scale factors from 10 to 500 to generate XML documents from 1.1GB to 55GB.
- GOV2: the dataset used in the 2006 TREC Terabyte Track[**?**] consisting of a crawl of the .gov domain (426GB).
- movie: we also experimented with a number of binary data files containing movies, and ranging in size from 1GB to 8GB, mainly to show the index creation and space behavior when there are many more different grams and the frequency is more uniform (we still lack application scenarios and queries that could be useful on such binary data, this is future work).

In the following, we use *FG*, *PG* and *QS* to denote full q-gram, partial q-gram and qs-gram indices. Most experiments were conducted on a PC with a 2.40GHz Intel Core2 Quad Q6600 CPU, 8GB of RAM and 2 hard disks in RAID-0. The experiments with the larger 426GB GOV2 dataset did not fit on the storage system of the PC, and were done instead on a server machine with two quad-core 2.8GHz Xeon (Nehalem) CPUs, 48GB of memory and a RAID-0 file system consisting of 16 SSDs (Intel X25-M). Note that in the index creation experiment, all I/O is sequential, and performance of SSDs is quite similar to normal hard drives. On this machine, we also performed a parallel experiment, where the 426GB dataset was split into eight partitions of 53GB, on which we ran our indexing program in parallel.

[4] http://download.wikimedia.org/enwiki/20090512
[5] http://monetdb.cwi.nl/xml/

elapsed time in sec. (write time in parentheses)			
	FG	**PG**	**QS**
wiki-article 22GB	4165 (1495)	4274 (778)	9673 (882)
wiki-meta 44GB	8665 (3133)	8901 (1646)	20042 (2193)
XMark 55GB	11122 (4151)	10964 (2018)	24057 (2346)
single PC, two disks			
GOV2-1core 426GB	91364 (32365)	87045 (13475)	174872 (5507.9)
GOV2-8core 426GB	12539 (4966)	12068 (2555.4)	25129 (1018.37)
8-core server, 16 SSDs			

Table 5.3: Index Creation Performance (sec)

5.5.1 Index Creation

Table 5.3 shows the overall time needed to create the various indices, where we give below each result between parenthesis the sub-component time needed for the multi-way merge union. The chunk size was 128MB for *FG*, *PG* and *QS* (we used frequency threshold $t = 2000$), causing a first phase of index generation for a 128MB chunk of the input data at-a-time. For the subsequent of merging local posting lists within chunk into a global index, the buffer size was again set to 128MB, causing our external merge sort to read each time 128MB of postings from all chunks, writing this out as a global postings lists. All algorithms run CPU-bound. For *FG* and *PG*, we achieve an indexing speed of 18GB per hour on the PC, and on the server machine in the 8-way parallel experiment, the 426GB of GOV2 data was indexed in less than 3.5 hours, achieving good speedup.

For *QS*, the performance is around 8GB per hour on the PC. *SG* and *PG* require additional time for running the set covering algorithm, but in case of *PG* this extra cost is offset by the fact that it writes less postings (this can be seen by comparing the write times between parentheses). *QS* takes much more time, since (1) the radix-sort runs on $q + 2$ bytes instead of q bytes, (2) to save space without writing intermediate data, the raw string is scanned twice. The first scan is to collect two global dictionaries for s-gram and q-gram. The second scan processes chunk-wise data.

	wiki 22GB	wiki 44GB	XMark 55GB	GOV2 426GB	binary data (movie) 1GB	2GB	4GB	8GB
FG	43	89	111	606.7	3.9	7.6	16	31
PG	23	47	58	425.3	2.4	4.7	9.5	19
QS	23.7	50.7	69.7	490.3	2.6	4.9	9.7	19.8
dict:	0	0	0	0	.2	.2	.2	.3

Table 5.4: Index Storage Space (GB)

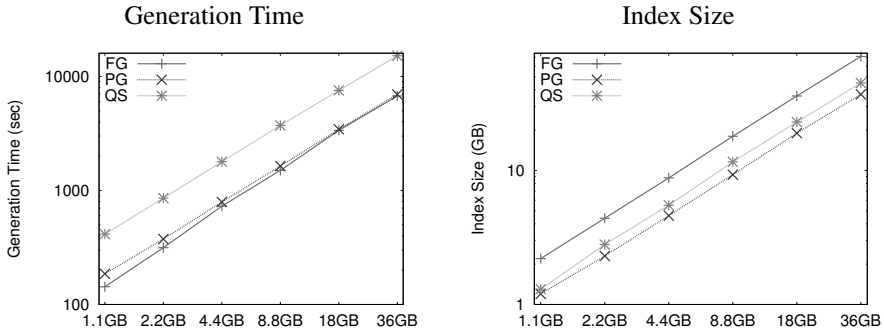


Figure 5.6: XMark data with factors 10, 20 to 320

5.5.2 Index Space

Table 5.4 shows the sizes of generated indices, including the space for the dictionaries. The dictionary sizes depend on the amount of different grams (see Table 5.2). As the amount of different grams in the index is small in textual data, the dictionary sizes do not play any role in the results on the textual data sets in Table 5.4. Table 5.4 shows in the additional "dict:" row that for *QS* (the scheme with the largest dictionaries), this overhead is still limited. The overall conclusion for textual data is that *FG* with compression typically achieves a 2x storage space, and *PG* and *SG* improve that to roughly 1x. In case of binary data, index space deteriorates to 4x in *FG*. Both partial approaches perform similar, and reduce storage by almost a factor 2.

Figure 5.6 shows the scalability on XMark using sizes 1.1GB, up to 36GB. Figure 5.6 (a) confirms linear scalability in creation time and Figure 5.6 (b) confirms linear scalability in index sizes.

5.5.3 Query Evaluation

For each data set, queries with variable lengths were tested, from 5 to 15. For queries with the same length, 100 positive queries were randomly drawn from the dataset, but we excluded queries with excessively long results (> 1M). In the figures, the results are ordered on *FG* score to help comparison. Due to space limitations, we concentrate here on the 44GB wiki-meta data. Figures 5.7 (a)-(d) shows that for short queries (e.g., 5), *FG* is generally faster than *PG* and *QS*. However, when the length of queries increases to 9, the difference between *FG* and *PG*, *QS* get smaller, and *QS* processes many queries faster than *FG*. For queries with lengths 11 and 15, *QS* outperforms *FG* for most queries (*PG* being slowest always).

To understand these results better, we first explain the other two groups of experiments. The first group is shown in Figures 5.7 (e)-(h), showing the number of postings loaded for different approaches. The number reported here was the posting under compression i.e. the data read from disk, but not the number of postings resulting from the positional merge join. The other group of experiments is the number of postings verified, which applies only to *PG* and *QS*, since they may generate false positives. These numbers are shown in Figures 5.7 (i)-(l).

Figures 5.7 (e)-(h) tell that *FG* and *PG* load many more postings than *QS*. The reason that *FG* sometimes loaded less data than *PG* is that *FG* was optimized to omit long posting lists from query plans. *PG*, however, cannot prune long posting lists, since that will potentially introduce many false positives. *QS*, however, always fetches less postings, since the posting lists of qs-grams are typically short.

The query evaluation for *FG* is CPU-bound, dominated by the number of postings in the positional merge join. The execution time of *PG* and *QS* is composed of two parts: (1) positional merge join, which is dominated by the number of postings loaded; (2) false positive checks, which will dominate the overall time if the number of candidates is large. Observe from Figures 5.7 (i)-(l) that *QS* always checks less candidates than *PG*. The value below 1 means that there is no candidates, i.e., all chunks are fully covered. When only few (or none) candidates are required to be checked, *QS* outperforms *FG* and *PG*. *PG* is normally slower than *FG*, since *PG* could load more postings and candidates are to be verified. Note that *PG* has the smallest storage space, roughly half of used by *FG*. *QS* needs slightly more space, but outperforms *PG* in query processing.

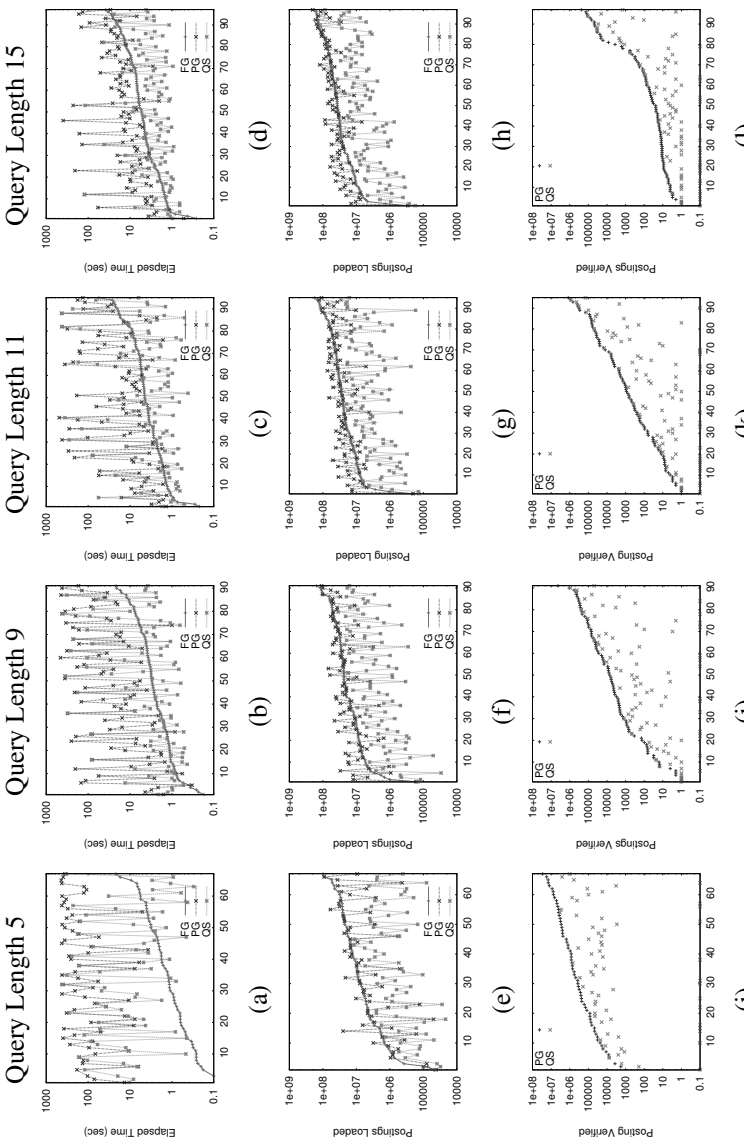


Figure 5.7: wiki-meta (44GB) (a-d) Elapsed Time (e-h) Postings Loaded (i-l) Postings Checked

5.6 Related Work

For answering exact substring matching, suffix tree [55, 22] and suffix array [52, 25] have been extensively studied, but this has not yet produced algorithms that could work on huge sizes such as the 426GB GOV2 dataset we indexed.

Q-gram based indices were first used to model sequences, using the statistical properties of q-grams. q-grams have been widely studied for efficient approximate string matching, also using DBMS as the storage and execution component [29]. In order to improve the performance of approximate queries, [50, 88] proposed variable-length grams, and [9, 42] worked on reducing the sizes of indices – note that our work concerns not approximate but with exact query processing. Many other work study the problem of approximate string joins using various similarity functions [6, 20, 70]. The above q-gram based approaches focus mainly on improving approximate string matching, and not robust enough to efficiently handle the problem of exact substring matching.

Suffix tree [55, 22] and suffix array [52, 25] are studied as indices to allow fast substring searching. A suffix tree could be constructed in $O(|\sigma|)$ time, versus $O(|\sigma|log^{|\sigma|})$ time for a suffix array. Finding all occurrences of a query ϱ over σ requires $O(|\varrho|+|\sigma|)$ time. Q-gram based indices are first used to model sequences, using the statistical properties of q-grams. Naturally, q-grams are widely studied for efficient approximate matching, by converting a sequence of strings of a set of q-grams, such that different sequences can be compared efficiently [29]. The storage overhead of a suffix tree is typically $10-20$ times of the raw string. Suffix array reduces the storage overhead to a factor of 4. They are unfavourable to real applications due to their large sizes.

5.7 Summary

We presented different gram-based indices for exact substring matching on huge data sets. Motivated by the fact that the full q-gram index has considerable storage overhead, impacting particularly the ease of manipulating such huge data sets, we aimed at space economical q-gram indices. The proposed partial q-gram indices have a very compact size (around 1x), but produce a relative large number of false positives that must be checked against the raw data. To alleviate this problem in partial q-grams, we proposed a novel approach called qs-gram index. The qs-gram index is designed to exploit the gram distribution, by splitting long posting lists and merge short posting lists with a frequency-adaptive signature approach, to ensure good data compression and query performance. The trade-off compared to partial q-grams is that qs-grams consume slightly larger space. We demonstrated excellent scalability on large data sets (up to

426GB), and showed query performance of qs-grams rivals or even improves that of traditional full q-grams.

As to future work, to reduce the number of postings to be verified in query processing, we plan to build to a cost model to judge when is benefit to load and combine multiple posting lists intead of verifying the raw data. Also, the dictionary size (and auxiliary bit-strings) for large binary data is problematic. We will investigate dictionary compression techniques.

Chapter 6

Concluding Remarks

Indexes have been an integral part of database management systems. They allow queries to be evaluated faster since they prevent scanning the entire base data each time. In this thesis we have presented four different indexes, each one designed to tackle a specific challenging application. Despite the different specifications, all indexes in this thesis share one design principle, they are *space efficient*.

To conclude this thesis, we iterate over the major contributions and we provide a roadmap for future research on the specific subject.

6.1 Contributions

6.1.1 Imprints

We first described column imprints, a light-weight secondary index with a small memory footprint suited for a main-memory setting. It is a bitvector indexes, Our extensive experimental evaluation showed significant query evaluation speed-up against pure scans and the established indexing approaches of zonemaps and bitmaps with bit- binning and WAH compression. The storage overhead of column imprints is just a few percent, with a max of 12 over the base column.

6.1.2 Split Bloom Filters

In this chapter we presented split Bloom filters, a collection of variable-size Bloom filters maintained in memory. Each filter covers a subset of the key values (records).

This approach makes it possible to adjust the size of the filters and use more bits for subsets that are larger and/or more frequently accessed. Rebuilding one or a few small filters in response to inserts, deletes or changes in access frequencies is also faster than rebuilding a single large filter. We also presented a mathematical model and methods to optimally size the filters in a collection, taking into account the number of distinct values covered by a filter and how frequently the filter is accessed. We also showed how and when to rebuild filters in response to changes in data or access frequencies. We performed an extensive experimental evaluation and found that our approach has several advantages compared with using a single large filter.

6.1.3 Generic Typed Value Indexes

In this chapter we described a collection of indices for XML text, element, and attribute node values that *(i)* consume little storage, *(ii)* have low maintenance overhead, *(iii)* permit fast equi-lookup on string values, and *(iv)* support range-lookup on any XML typed value (e.g., double, dateTime). We evaluated our design and algorithms in MonetDB/XQuery and they are now part of a *stable release of MonetDB/XQuery*, thus putting our ideas to an every-day test.

6.1.4 Partial Grams

We presented different gram-based indices for exact substring matching on huge data sets. Motivated by the fact that the full q-gram index has considerable storage overhead, impacting particularly the ease of manipulating such huge data sets, we aimed at space economical q-gram indices. The main contribution, qs-grams are designed to exploit the gram distribution, by splitting long posting lists and merge short posting lists with a frequency-adaptive signature approach, to ensure good data compression and query performance.

6.2 Future Work

All of the indexes presented in this thesis can be used in a distributed environment. However, the adaptation depends on both the application and the underlying hardware setting. We believe it is a fairly straight forward process, however there is plenty of room to explore new scientific questions and define the details for such an adaptation.

6.2.1 Imprints

The current implementation of imprints in MonetDB can cope with multithreaded scans. Each thread checks a different portion of imprints and produces a list of qualifying positions that are later merged to one list. Likewise, imprints can be used in a distributed environment.

Another extension of imprints currently into the workings is to perform parallel scans in multiple columns. Such scans are needed for GIS applications and other n-dimensional queries. Also, a single imprint may contain information for more than one column, instead of merging in parallel multiple imprints.

6.2.2 Split Bloom Filters

Split Bloom filters can be used in a distributed environment or cluster of computers to transmit information between nodes, in order to mark which ones contain data relevant to the query. The size of each Bloom filter can be determined not by the frequency of a specific key is accessed, but by the amount of the workload a node has. More specific, the smaller a bloom filter is, the more the false positives that have to be resolved during the step of checking the data. Therefore, if a node has little to no workload, it can transmit a smaller Bloom filter to the nodes that are more busy, thus *a)* reducing the amount of probing the busy nodes have to perform, and *b)* increasing the amount of false positive checking the less busy nodes have to perform.

6.2.3 Generic Typed Value Indexes

We presented the generic typed value indexes in the context of XML. However, they can equally be used for RDF data. RDF has become more popular and it is the model of choice for most modern application to share semi-structured data. RDF also uses the XML standard to donate the type of the values, and also share the same semantics. Thus, the generic typed value indexes can be used in a RDF repository with some adjustments.

6.2.4 Partial Grams

Partial grams are split in many chunks, thus providing a natural way to distribute them over multiple nodes. Pattern matching is a very important function for scientific applications such as genome sequences and internet security.

6.3 Another approach to Big Data

In this thesis we presented techniques to search big datasets for that piece of information that is relevant to the query. We done so by using indexes that are small enough to fit in today's main memories in order to speed up the search.

However, exploring big data can become a futile process. The amount of information is so large, and the speed that it is produced is so fast, that one can easily get lost in this flood of data, even with the use of small and fast indexes. Having that in mind, we argue that a different approach is needed towards the big data problem. Instead of gathering vast amounts of data that we cannot consume, *we should introduce a rotting factor to disregard old and unused data, while also introduce methods that transform data to fresh and new, but more concise, information.*

6.3.1 Big Data Space Rotting

For the sake of brevity consider a single table $R(t, f, A_1, \ldots, A_n)$ where A_i denotes the attributes, t the real-world time it was inserted, and a freshness property $f \in [0, 1]$ initially set to 1. So far, the relation R can be used like a normal relational table. However, rotting of the data may take place, i.e., *the extent of table R decays with a periodic clock cycle of T seconds using a data fungus f until it is completely rotten away*.

Each tuple in R rots over time, reflected in an ever decreasing freshness value. When the freshness reaches zero, the tuple is discarded from R. An simple decay fungus F would be to consider retention periods, where after the data will be discarded. However, many more data fungi can be considered, based on their rate of decay, what to decay, how to decay, etc.

6.3.2 Big Data Space Freshness

The evident approach to avoid rotten data is to eat it or cook it into useful information as soon as possible. It is a task normally undertaken by the data ingestion pipeline. But, we can go one step further by focusing on the queries over R. Consider the select-from-where queries $A = Q(T, R, P)$, the query Q over table R with predicate P, target expression T and answer set A. Then, *the extent of table R is replaced after each query Q into the combination of the answer set A and a reduced extent of R, where all tuples in R satisfying P have been discarded immediately.*

This rule stresses the point that once you take something out of R, you should distill it into useful knowledge, a summary, or consume it, or store it in a new container subject to different data fungi. The database is kept in optimal health condition if you

regularly can turn rotting portions into summaries for later consumption, or inspect them once before removal.

Bibliography

[1] Abadi, D. J., Myers, D. S., DeWitt, D. J., and Madden, S. R. Materialization Strategies in a Column-Oriented DBMS. In *Proceedings of the 23rd International Conference on Data Engineering* (2007).

[2] Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D. J., Silberschatz, A., and Rasin, A. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In *Proceedings of the 35th International Conference on Very Large Data Bases* (2009).

[3] Almeidaa, P. S., Baqueroa, C., Preguicab, N., and Hutchisonc, D. Scalable Bloom Filters. *Information Processing Letters 101*, 6 (2007).

[4] Amer-Yahia, S., Curtmola, E., and Deutsch, A. Flexible and efficient XML search with complex full-text predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2006).

[5] Anh, V. N., and Moffat, A. Inverted Index Compression Using Word-Aligned Binary Codes. *Information Retrieval 8*, 1 (2005).

[6] Arasu, A., Ganti, V., and Kaushik, R. Efficient Exact Set-Similarity Joins. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (2006).

[7] Beckham, J. L. The CNET E-Commerce Data Set, 2005.

[8] Beckmann, N., Kriegel, H. P., Schneider, R., and Seeger, B. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1990).

[9] Behm, A., Ji, S., Li, C., and Lu, J. Space-Constrained Gram-Based Indexing for Efficient Approximate String Search. In *Proceedings of the 25th International Conference on Data Engineering* (2009).

[10] Beyer, K., Cochrane, R. J., Josifovski, V., Kleewein, J., Lapis, G., Lohman, G., Lyle, B., Özcan, F., Pirahesh, H., Seemann, N., Truong, T., der Linden, B. V., Vickery, B., and Zhang, C. System RX: One Part Relational, One Part XML. In *Proceedings of the 24th ACM SIGMOD International Conference on Management of Data* (2005).

[11] Blanas, S., Li, Y., and Patel, J. M. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2011).

[12] Bloom, B. H. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM 13*, 7 (1970).

[13] Boncz, P., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., and Teubner, J. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proceedings of the 25th ACM SIGMOD International Conference on Management of Data* (2006).

[14] Boncz, P., Zukowski, M., and Nes, N. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of the 2nd Conference on Innovative Data Systems Research* (2005).

[15] Boncz, P. A., Manegold, S., and Rittinger, J. Updating the Pre/Post Plane in MonetDB/XQuery. In *Proceedings of the International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)* (2005).

[16] Bruck, J., Gao, J., and Jiang, A. Weighted Bloom Filter. In *IEEE International Symposium on Information Theory* (2006).

[17] Campbell, D. Breakthrough performance with in-memory technologies, at http://blogs.technet.com/b/dataplatform insider/archive/2012/11/08/breakthrough-performance-with-in-memory-technologies.aspx.

[18] Canahuate, G., Gibas, M., and Ferhatosmanoglu, H. Update Conscious Bitmap Indices. In *Proceedings of the 19th International Conference on Scientific and Statistical Database Management* (2007).

[19] Chan, C., and Ioannidis, Y. An efficient bitmap encoding scheme for selection queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1999).

[20] Chaudhuri, S., Ganjam, K., Ganti, V., and Motwani, R. Robust and Efficient Fuzzy Match for Online Data Cleaning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2003).

[21] Chazelle, B., Kilian, J., Rubinfeld, R., and Tal, A. The Bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms* (2004).

[22] Cheung, C.-F., Yu, J. X., and Lu, H. Constructing Suffix Tree for Gigabyte Sequences with Megabyte Memory. *IEEE Transactions on Knowledge and Data Engineering 17*, 1 (2005).

[23] Cohen, S., and Matias, Y. Spectral Bloom Filters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2003).

[24] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation* (2004).

[25] Dementiev, R., Karkkainen, J., Mehnert, J., and Sanders, P. Better external memory suffix array construction. *ACM Journal on Experimental Algorithmics 12* (2008).

[26] Diaconu, C., Freedman, C., Ismert, E., Larson, P., Mittal, P., Stonecipher, R., Verma, N., and Zwilling, M. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2013).

[27] Gosink, L., Wu, K., Bethel, E., Owens, J. D., and Joy, K. I. Data Parallel Bin-Based Indexing for Answering Queries on Multi-core Architectures. In *Proceedings of the International Conference on Scientific and Statistical Database Management* (2009).

[28] Goyal, N., and Sharma, Y. New binning strategy for bitmap indices on high cardinality attributes. In *Proc. of the 2nd Bangalore Annual Compute Conference* (2009).

[29] Gravano, L., Ipeirotis, P. G., Jagadish, H. V., Koudas, N., Muthukrishnan, S., Pietarinen, L., and Srivastava, D. Using q-grams in a DBMS for Approximate String Processing. *IEEE Data Eng. Bull. 24*, 4 (2001).

[30] Grund, M., Krüger, J., Plattner, H., Zeier, A., Cudre-Mauroux, P., and Madden, S. HYRISE - A Main Memory Hybrid Storage Engine. In *Proceedings of the 38th International Conference on Very Large Data Bases* (2012).

[31] Hao, F., Kodialam, M., and Lakshman, T. V. Building high accuracy bloom filters using partitioned hashing. In *Proceedings of the SIGMETRICS* (2007).

[32] Haustein, M. P., Harder, T., and Luttenberger, K. Contest of XML Lock Protocols. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (2006).

[33] Hellerstein, J., Naughton, J., and Pfeffer, A. Generalized Search Trees for Database Systems. In *Proceedings of the 21th International Conference on Very Large Data Bases* (1995).

[34] Heman, S., Zukowski, M., Nes, N., Sidirourgos, L., and Boncz, P. Positional Update Handling in Column Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2010).

[35] Hey, T., Tansley, S., and Tolle, K. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[36] Holloway, A. L., Raman, V., Swart, G., and DeWitt, D. J. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2007).

[37] IBM solidDB, information available at http://www.ibm.com/.

[38] Idreos, S., Kersten, M., and Manegold, S. Database Cracking. In *Proceedings of the 3rd Conference on Innovative Data Systems Research* (2007).

[39] Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P. C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., and Abadi, D. J. H-store: a high-performance, distributed main memory transaction processing system. In *Proceedings of the 34th International Conference on Very Large Data Bases* (2008).

[40] Kemper, A., and Neumann, T. HyPer – A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the 27th International Conference on Data Engineering* (2011).

[41] Kersten, M. L., Idreos, S., Manegold, S., and Liarou, E. The Researcher's Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds. *PVLDB 4*, 12 (2011).

[42] Kim, M.-S., Whang, K.-Y., Lee, J.-G., and Lee, M.-J. n-Gram/2L: A Space and Time Efficient Two-Level n-Gram Inverted Index Structure. In *Proceedings of the 31st International Conference on Very Large Data Bases* (2005).

[43] Kirsch, A., and Mitzenmacher, M. Less hashing, same performance: Building a better Bloom filter. *Journal of Random Structures and Algorithms 33*, 2 (2008).

[44] Knuth, D. E. *Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)*. Addison-Wesley Publishing Company, 1997.

[45] Koudas, N. Space efficient bitmap indexing. In *Proceedings of the 9th Conference on Information and Knowledge Management* (2000).

[46] Laney, D. 3D Data Management: Controlling Data Volume, Velocity and Variety. Gartner, 2001.

[47] Larson, P.-Å., Blanas, S., Diaconu, C., Freedman, C., Patel, J. M., and Zwilling, M. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. In *Proceedings of the 38th International Conference on Very Large Data Bases* (2012).

[48] Levandoski, J., Larson, P.-Å., and Stoica, R. Identifying Hot and Cold Data in Main-Memory Databases [PDF]. In *Proceedings of the 29th International Conference on Data Engineering* (2013).

[49] Levandoski, J., Lomet, D., and Sengupta, S. The Bw-Tree: A B-tree for New Hardware. In *Proceedings of the 29th International Conference on Data Engineering* (2013).

[50] Li, C., Wang, B., and Yang, X. VGRAM: Improving Performance of Approximate Queries on String Collections Using Variable-Length Grams. In *Proceedings of the 33rd International Conference on Very Large Data Bases* (2007).

[51] Lu, J., and Callan, J. P. User modeling for full-text federated search in peer-to-peer networks. In *Proceedings of the ACM SIGIR International Conference on on Information Retrieval* (2006).

[52] Manber, U., and Myers, G. Suffix Arrays: A New Method for On-Line String Searches. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms* (1990).

[53] Manegold, S., Boncz, P. A., and Nes, N. Cache-Conscious Radix-Decluster Projections. In *Proceedings of the 30th International Conference on Very Large Data Bases* (2004).

[54] Manegold, S., Kersten, M., and Boncz, P. Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct. In *Proceedings of the 35th International Conference on Very Large Data Bases* (2009).

[55] McCreight, E. M. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM (JACM) 23*, 2 (1976).

[56] Metwally, A., Agrawal, D., and Abbadi, A. E. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Transactions on Database Systems 31*, 3 (2006).

[57] Mitzenmacher, M. Compressed Bloom Filters. In *Proceedings of the 20th ACM symposium on Principles Of Distributed Computing* (2001).

[58] MonetDB. http://www.monetdb.org.

[59] Nicola, M., and van der Linden, B. Native XML support in DB2 universal database. In *Proceedings of the 31st International Conference on Very Large Data Bases* (2005).

[60] O'Neil, P. Model 204 Architecture and Performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems* (1987).

[61] O'Neil, P., and Graefe, G. Multi-table joins through bitmapped join indices. *SIGMOD Rec. 24*, 3 (1995).

[62] O'Neil, P., and Quass, D. Improved Query Performance with Variant Indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1997).

[63] Oracle TimesTen In-Memory Database, information availa-ble at http://www.oracle.com.

[64] Pagh, A., Pagh, R., and Rao, S. S. An Optimal Bloom Filter Replacement. In *Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms* (2005).

[65] Pal, S., Cseri, I., Seeliger, O., Schaller, G., Giakoumakis, L., and Zolotov, V. Indexing XML data stored in a relational database. In *Proceedings of the 30th International Conference on Very Large Data Bases* (2004).

[66] Pandis, I., Tozun, P., Johnson, R., and Ailamaki, A. PLP: Page Latch-free Shared-everything OLTP. In *Proceedings of the 37th International Conference on Very Large Data Bases* (2011).

[67] Petrovic, S., and Bakke, S. Application of q-Gram Distance in Digital Forensic Search. In *Proceedings of the 2nd International Workshop Computational Forensics* (2008).

[68] Pettey, C., and Goasduff, L. Gartner Says Solving 'Big Data' Challenge Involves More Than Just Managing Volumes of Data. http://www.gartner.com/newsroom/id/1731916, 2011.

[69] Putze, F., Sanders, P., and Singler, J. Cache-, hash-, and space-efficient bloom filters. *ACM Journal of Experimental Algorithmics 14* (2009).

[70] Sarawagi, S., and Kirpal, A. Efficient set joins on similarity predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2004).

[71] Scilens Platform. http://www.scilens.org/content/platform.

[72] Sewall, J., Chhugani, J., Kim, C., Satish, N., and Dubey, P. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. In *Proceedings of the 37th International Conference on Very Large Data Bases* (2011).

[73] Sidirourgos, L., Kersten, M., and Boncz, P. SciBORQ: Scientific data management with Bounds On Runtime and Quality. In *Proceedings of the 5th Conference on Innovative Data Systems Research* (2011).

[74] Sidirourgos, L., Kersten, M., and Boncz, P. Scientific Discovery through Weighted Sampling. In *Proceedings of the 1st IEEE International Conference on Big Data* (2013).

[75] Sinha, R. R., and Winslett, M. Multi-resolution bitmap indexes for scientific data. *ACM Trans. Database Syst. 32*, 3 (2007).

[76] Stonebraker, M., Abadi, D., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., and Zdonik, S. C-Store: A Column Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases* (2005).

[77] Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases* (2007).

[78] VoltDB, information available at http://www.voltdb.com/.

[79] Vyas, A., Fernandez, M. F., and Simeon, J. The Simplest XML Storage Manager Ever. In *Proceedings of the First International Workshop XIME-P, in cooperation with ACM SIGMOD* (2004).

[80] Wong, H., Liu, H., Olken, F., Rotem, D., and Wong, L. Bit Transposed Files. In *Proceedings of the 11th International Conference on Very Large Data Bases* (1985).

[81] Wu, K., Madduri, K., and Canon, S. Multi-level bitmap indexes for flash memory storage. In *Proceedings of the 14th IDEAS* (2010).

[82] Wu, K., Otoo, E. J., and Shoshani, A. Compressing Bitmap Indexes for Faster Search Operations. In *Proceedings of the 14th International Conference on Scientific and Statistical Database Management* (2002).

[83] Wu, K., Otoo, E. J., and Shoshani, A. On the performance of bitmap indices for high cardinality attributes. In *Proceedings of the 30th International Conference on Very Large Data Bases* (2004).

[84] Wu, K., Otoo, E. J., and Shoshani, A. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst. 31*, 1 (2006).

[85] Wu, K., Shoshani, A., and Stockinger, K. Analyses of multi-level and multi-component compressed bitmap indexes. *ACM Trans. Database Syst. 35*, 1 (2008).

[86] XQuery 1.0 and XPath 2.0 Data Model (XDM). http://www.w3.org/TR/xpath-datamodel/.

[87] XML Schema Part 2: Datatypes Second Edition. http://www.w3.org/TR/xmlschema-2/.

[88] Yang, X., Wang, B., and Li, C. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2008).

[89] Zhong, M., Lu, P., Shen, K., and Seiferas, J. Optimizing Data Popularity Conscious Bloom Filters. In *Proceedings of the 27th ACM symposium on Principles Of Distributed Computing* (2008).

List of Figures

List of Tables

List of Algorithms

Summary

The *big data* era is characterized by the challenges put forward by both data-intensive scientific discovery and the e-commerce surge. Scientific discovery has shifted from being an exercise of theory and computation, to become the exploration of an ocean of observational data. State-of-the-art astronomical observatories and modern scientific instruments produce every day petabytes of information. Moreover, e-commerce sales have grown in the last decade and expected to account for more than 10% of retail sales in the near future. Analyzing user generated data and web logs is crucial for increasing competitiveness, creating targeted advertisement and advanced recommendation systems, and providing quality of service. Enterprises gather huge amounts of data, to be continuously queried and updated for fast user experience, but also batch analyzed over long periods to design business plans and economical strategies. The big data challenges in the size of the data collections, the speed of updates, and the diversity of the data models are summarized in the so called "3Vs" *volume, velocity, variety*.

The predominant answer to the big data challenge, given by the data management community, is the raw power of big data center installations, complemented by new technologies focusing on scalable distribution of data and operations, such as MapReduce and Hadoop. In addition, system designers have build database warehouses to work on top of these distributed environments, such as Hive, Pig, Impala, and more. These systems are spread across multiple machines and therefor should be easy to deploy and initialize. Moreover, accessing local partitioned data remains a bottleneck, thus there is still a clear need for *space efficient indexes* that are lightweight to build and maintain. Such space efficient indexes consume only sub-linear to the indexed data space, are fast to create (usually a single scan), and also are easy to update (linear to the size of the appended data). In addition, usually they are secondary structures, meaning that they do not dictate the placement of the indexed data, thus not requiring expensive read and write steps during creating or updates.

In this thesis we present four space efficient indexes that satisfy the above requirements. Each one of them is designed to address the requirements of a specific applica-

tion. First, we present *column imprints*, a cache conscious secondary index for column stores capable of answering range queries fast. Imprints are particularly handy for numerical domains in scientific databases with a large number of attributes per table. We next introduce a new variant of Bloom filters called *split Bloom filters*, designed to restrict access to cold stores in the presence of skew access. The applicability of such index is in large e-commerce sites that keep in memory hot sets of users and products, while old and outdated data are stored in slower memories. We then introduce a type independent index that produces an order preserving hash function. It is used to index large XML repositories. The main challenge to overcome is that XML elements are typeless and predicates type agnostic, e.g., a predicate matches both strings and numerical values. Finally, we present a space efficient string index based on grams, called *qs-grams*. This index is capable of answering sub-string queries on huge collections of BLOB's or documents, faster than the state-of-the-art *n-grams* and by using a 1-1 ratio of storage. *qs-grams* are designed for pattern matching applications, such as genome sequence alignment or detecting malicious snippets of binary code on disks.

Samenvatting

Het *big data* tijdperk wordt gekenmerkt door de uitdagingen die voortkomen uit zowel data-intensief wetenschappelijke onderzoek als door de opkomst van de elektronische handel, de *e*-commerce. De wetenschappelijke ontdekking is verschoven van zijnde een uitoefening van theorie en berekening, naar de bestudering van overvloeden aan observationele data. Geavanceerde sterrenkundige observatoria en andere moderne wetenschappelijke instrumenten produceren elke dag petabytes aan informatie. Daarnaast zijn de verkopen uit de elektronische handel het laatste decennium gegroeid en men verwacht dat deze 10% gaan uitmaken van de detailhandel in de nabije toekomst. De analyse van gegenereerde gebruikersgegevens en logbestanden van het web is cruciaal bij een toenemende concurrentie, het maken van doelgerichte reclame en moderne aanbevelingssystemen, en voor de kwaliteit van te leveren diensten. Ondernemingen verzamelen enorme hoeveelheden gegevens, die niet alleen continu opgevraagd en aangepast worden voor een snelle gebruiksbelevenis, maar die ook over een langere termijn groepsgewijs ontleed worden om bedrijfsplannen en economische strategieën te ontwerpen. De big data uitdagingen kunnen naar omvang van de gegevens, de snelheid van aanpassingen, en de verscheidenheid aan datamodellen samengevat worden in de zogenaamde "3V's", naar het Engelse *volume, velocity, variety*.

Het overheersende antwoord op de big data uitdaging, voortgebracht door de gemeenschap van gegevensbeheerders, is de ruwe kracht van de installatie van een big data center, aangevuld met nieuwe technologieën gericht op schaalbare verspreiding van de data en berekeningen, zoals MapReduce en Hadoop. Daarbij hebben systeemontwerpers datawarehouses gebouwd die bovenop deze gedistribueerde omgevingen werken, zoals Hive, Pig, Impale, en andere. Deze systemen zijn verspreid over meer dan één machine en zouden daarom eenvoudig uit te rollen en te initialiseren zijn. Bovendien blijft de toegankelijkheid van locale gepartitioneerde data een knelpunt, wat nog steeds de noodzaak van *ruimte-efficiënte idices* duidelijk maakt, die makkelijk te bouwen en te onderhouden zijn. Het verbruik van zulke ruimte-efficiënte idices (Eng: space efficient indexes) is slechts sublineair ten opzichte van de geïndexeerde data

space, ze zijn snel te maken (gewoonlijk in een enkele scan) en ook makkelijk te updaten (lineair ten opzichte van de toegevoegde data). Daarnaast zijn het gewoonlijk secundaire structuren, wat betekent dat ze niet de plaatsing van de geïndexeerde data vastleggen en dus geen dure lees- en schrijfstappen vereisen gedurende het aanmaken en updaten.

In dit proefschrift presenteren we vier ruimte-efficiënte indices die aan bovengenoemde voorwaarden voldoen. Elke afzonderlijk is ontworpen om zich op een specifieke toepassing toe te leggen. Als eerste presenteren we *kolomafdrukken* (Eng: *column imprints*), een cache-bewuste secundaire index voor kolomgeoriënteerde databases die in staat zijn *r*ange queries snel te beantwoorden. *I*mprints zijn bijzonder handig in het numerieke domein van wetenschappelijke databases waar het aantal attributen per tabel groot is. Vervolgens stellen we een nieuwe variant van Bloom filters voor, namelijk *split Bloom filters*, ontworpen om de toegang tot *c*old stores te beperken in de aanwezigheid van een *s*kew toegang. De toepasbaarheid van zulke indices is voor de grotere e-commerce sites, die *h*ot sets van gebruikers en producten in het geheugen houden, terwijl oude en achterhaalde gegevens opgeslagen worden in langzamere geheugens. Daarna introduceren we een type-onafhankelijke index, die een hash functie produceert die de volgorde behoudt. Deze wordt gebruikt om grote XML repositories te indexeren. De grootste uitdaging is om te bereiken dat de XML elementen typeloos zijn en de predicaten type-agnostisch zijn, bijv. een predicaat dat voor zowel alfanumerieke als numerieke waarden werkt. Tenslotte presenteren we een ruimte-efficiënte string index, gebaseerd op *g*rams en genaamd *qs-grams*. Deze index is in staat om *s*ubstring queries op grote verzamelingen van BLOB's of documenten sneller te beantwoorden dan de moderne *n-grams* en door een 1-op-1 verhouding van opslag te gebruiken. *qs-grams* zijn ontworpen voor toepassingen op het gebied van patroonherkenning, zoals de sequencebepaling van het genoom of bij het opsporen van kwaadwillende stukjes binaire code op schijven.

Acknowledgments

I could not be happier for finishing my thesis, it was a long journey with ups and downs, with a couple of ”return to the beginning” cards but also with equal amount of ”get out of jail” ones. Nothing would have been possible without prof. Dr. Peter Boncz believing in me and offering a position at CWI when I had no proof of my abilities. My enormous gratitude also goes to prof. Dr. Martin Kersten for the endless discussions and brainstorming, for having new and amazing ideas for research every single week, and for being always there for me in all aspects of my life, both professional and personal. I have been blessed with two amazing advisors and without them nothing would have been possible.

The Database Architecture group at CWI, also known as the MonetDB team, created the environment that was needed to flourish as a researcher and achieve knowledge. Dr. Stephan Manegold was always there to provide solutions for my research, but also to protect me from managerial bureaucracy so I can continue my research uninterrupted. Dr. Niels Nes with whom I shared a wall, and Sjoerd Mullender down the hall, were my ”go to” guys for all of my algorithmic and coding problems. But also the rest of the group, Stratos Idreos, Romulo Goncalves, Marcin Zukowski, Erietta Liarou, Ying Zhang, Fabian Groffen, Milena Ivanova, Eleni Petraki, Hannes Muhleisen, Holger Pirk, they all helped me when I asked them. Bart Scheers, our in-house astronomer, who taught us the wonders of the sky and provided the dutch translation of the summary of this thesis.

I also want to express my gratitude to the two committee members of my thesis prof. Dr. Anastasia Ailamaki and Dr. Paul Larson who gladly accepted my invitation and traveled a long distance to support me. Anastasia Ailamaki has been there for me many times and I consider her both a mentor and a cherished friend. Paul Larson offered me a warm welcome at Microsoft Research and showed me the ways of industry research.

My friends in Amsterdam, who are too many to name but they know who they are, have made my life easier and happier. I consider them my family away from home. We

have partied a lot, got drunk, laughed, and loved. Six years in Amsterdam would have not been possible without them. Spiros has been a great companion all of these years, a windsurf, snowboarding, traveling, and most importantly drinking buddy. Elina has filled an enormous gap in my life. Being with her is more important than any life achievement, and for that I love her with all my heart.

Finally, I want to thank my family, my mother Sophia and my father Vangelis. Without them no child would ever have managed to achieve what I did. Their support, their believe in me, their love and their understanding can not be valued or expressed with words. I owe them my life.

SIKS Dissertatiereeks

1998-1 Johan van den Akker (CWI) DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2 Floris Wiesman (UM) Information Retrieval by Graphically Browsing Meta-Information

1998-3 Ans Steuten (TUD) A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective

1998-4 Dennis Breuker (UM) Memory versus Search in Games

1998-5 E.W.Oskamp (RUL) Computerondersteuning bij Straftoemeting

1999-1 Mark Sloof (VU) Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products

1999-2 Rob Potharst (EUR) Classification using decision trees and neural nets

1999-3 Don Beal (UM) The Nature of Minimax Search

1999-4 Jacques Penders (UM) The practical Art of Moving Physical Objects

1999-5 Aldo de Moor (KUB) Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems

1999-6 Niek J.E. Wijngaards (VU) Re-design of compositional systems

1999-7 David Spelt (UT) Verification support for object database design

1999-8 Jacques H.J. Lenting (UM) Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.

2000-1 Frank Niessink (VU) Perspectives on Improving Software Maintenance

2000-2 Koen Holtman (TUE) Prototyping of CMS Storage Management

2000-3 Carolien M.T. Metselaar (UVA) Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.

2000-4 Geert de Haan (VU) ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5 Ruud van der Pol (UM) Knowledge-based Query Formulation in Information Retrieval.

2000-6 Rogier van Eijk (UU) Programming Languages for Agent Communication

2000-7 Niels Peek (UU) Decision-theoretic Planning of Clinical Patient Management

2000-8 Veerle Coup (EUR) Sensitivity Analyis of Decision-Theoretic Networks

2000-9 Florian Waas (CWI) Principles of Probabilistic Query Optimization

2000-10 Niels Nes (CWI) Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 Jonas Karlsson (CWI) Scalable Distributed Data Structures for Database Management

2001-1 Silja Renooij (UU) Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 Koen Hindriks (UU) Agent Programming Languages: Programming with Mental Models

2001-3 Maarten van Someren (UvA) Learning as problem solving

2001-4 Evgueni Smirnov (UM) Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 Jacco van Ossenbruggen (VU) Processing Structured Hypermedia: A Matter of Style

2001-6 Martijn van Welie (VU) Task-based User Interface Design

2001-7 Bastiaan Schonhage (VU) Diva: Architectural Perspectives on Information Visualization

2001-8 Pascal van Eck (VU) A Compositional Semantic Structure for Multi-Agent Systems Dynamics.

2001-9 Pieter Jan 't Hoen (RUL) Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes

2001-10 Maarten Sierhuis (UvA) Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design

2001-11 Tom M. van Engers (VUA) Knowledge Management: The Role of Mental Models in Business Systems Design

2002-01 Nico Lassing (VU) Architecture-Level Modifiability Analysis

2002-02 Roelof van Zwol (UT) Modelling and searching web-based document collections

2002-03 Henk Ernst Blok (UT) Database Optimization Aspects for Information Retrieval

2002-04 Juan Roberto Castelo Valdueza (UU) The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05 Radu Serban (VU) The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents

2002-06 Laurens Mommers (UL) Applied legal epistemology; Building a knowledge-based ontology of the legal domain

2002-07 Peter Boncz (CWI) Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications

2002-08 Jaap Gordijn (VU) Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas

2002-09 Willem-Jan van den Heuvel(KUB) Integrating Modern Business Applications with Objectified Legacy Systems

2002-10 Brian Sheppard (UM) Towards Perfect Play of Scrabble

2002-11 Wouter C.A. Wijngaards (VU) Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12 Albrecht Schmidt (Uva) Processing XML in Database Systems

2002-13 Hongjing Wu (TUE) A Reference Architecture for Adaptive Hypermedia Applications

2002-14 Wieke de Vries (UU) Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems

2002-15 Rik Eshuis (UT) Semantics and Verification of UML Activity Diagrams for Workflow Modelling

2002-16 Pieter van Langen (VU) The Anatomy of Design: Foundations, Models and Applications

2002-17 Stefan Manegold (UVA) Understanding, Modeling, and Improving Main-Memory Database Performance

2003-01 Heiner Stuckenschmidt (VU) Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU) Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD) Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petkovic (UT) Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UVA) Causation in Artificial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT) Development and specification of virtual environments

2003-07 Machiel Jansen (UvA) Formal Explorations of Knowledge Intensive Tasks

2003-08 Yongping Ran (UM) Repair Based Scheduling

2003-09 Rens Kortmann (UM) The resolution of visually guided behaviour

2003-10 Andreas Lincke (UvT) Electronic Business Ne-

gotiation: Some experimental studies on the interaction between medium, innovation context and culture

2003-11 Simon Keizer (UT) Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

2003-12 Roeland Ordelman (UT) Dutch speech recognition in multimedia information retrieval

2003-13 Jeroen Donkers (UM) Nosce Hostem - Searching with Opponent Models

2003-14 Stijn Hoppenbrouwers (KUN) Freezing Language: Conceptualisation Processes across ICT-Supported Organisations

2003-15 Mathijs de Weerdt (TUD) Plan Merging in Multi-Agent Systems

2003-16 Menzo Windhouwer (CWI) Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses

2003-17 David Jansen (UT) Extensions of Statecharts with Probability, Time, and Stochastic Timing

2003-18 Levente Kocsis (UM) Learning Search Decisions

2004-01 Virginia Dignum (UU) A Model for Organizational Interaction: Based on Agents, Founded in Logic

2004-02 Lai Xu (UvT) Monitoring Multi-party Contracts for E-business

2004-03 Perry Groot (VU) A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving

2004-04 Chris van Aart (UVA) Organizational Principles for Multi-Agent Architectures

2004-05 Viara Popova (EUR) Knowledge discovery and monotonicity

2004-06 Bart-Jan Hommes (TUD) The Evaluation of Business Process Modeling Techniques

2004-07 Elise Boltjes (UM) Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes

2004-08 Joop Verbeek(UM) Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politile gegevensuitwisseling en digitale expertise

2004-09 Martin Caminada (VU) For the Sake of the Argument; explorations into argument-based reasoning

2004-10 Suzanne Kabel (UVA) Knowledge-rich indexing of learning-objects

2004-11 Michel Klein (VU) Change Management for Distributed Ontologies

2004-12 The Duy Bui (UT) Creating emotions and facial expressions for embodied agents

2004-13 Wojciech Jamroga (UT) Using Multiple Models of Reality: On Agents who Know how to Play

2004-14 Paul Harrenstein (UU) Logic in Conflict. Logical Explorations in Strategic Equilibrium

2004-15 Arno Knobbe (UU) Multi-Relational Data Mining

2004-16 Federico Divina (VU) Hybrid Genetic Relational Search for Inductive Learning

2004-17 Mark Winands (UM) Informed Search in Complex Games

2004-18 Vania Bessa Machado (UvA) Supporting the Construction of Qualitative Knowledge Models

2004-19 Thijs Westerveld (UT) Using generative probabilistic models for multimedia retrieval

2004-20 Madelon Evers (Nyenrode) Learning from Design: facilitating multidisciplinary design teams

2005-01 Floor Verdenius (UVA) Methodological Aspects of Designing Induction-Based Applications

2005-02 Erik van der Werf (UM)) AI techniques for the game of Go

2005-03 Franc Grootjen (RUN) A Pragmatic Approach to the Conceptualisation of Language

2005-04 Nirvana Meratnia (UT) Towards Database Support for Moving Object data

2005-05 Gabriel Infante-Lopez (UVA) Two-Level Probabilistic Grammars for Natural Language Parsing

2005-06 Pieter Spronck (UM) Adaptive Game AI

2005-07 Flavius Frasincar (TUE) Hypermedia Presentation Generation for Semantic Web Information Systems

2005-08 Richard Vdovjak (TUE) A Model-driven Approach for Building Distributed Ontology-based Web Applications

2005-09 Jeen Broekstra (VU) Storage, Querying and Inferencing for Semantic Web Languages

2005-10 Anders Bouwer (UVA) Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments

2005-11 Elth Ogston (VU) Agent Based Matchmaking and Clustering - A Decentralized Approach to Search

2005-12 Csaba Boer (EUR) Distributed Simulation in Industry

2005-13 Fred Hamburg (UL) Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen

2005-14 Borys Omelayenko (VU) Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics

2005-15 Tibor Bosse (VU) Analysis of the Dynamics of Cognitive Processes

2005-16 Joris Graaumans (UU) Usability of XML Query Languages

2005-17 Boris Shishkov (TUD) Software Specification Based on Re-usable Business Components

2005-18 Danielle Sent (UU) Test-selection strategies for probabilistic networks

2005-19 Michel van Dartel (UM) Situated Representation

2005-20 Cristina Coteanu (UL) Cyber Consumer Law, State of the Art and Perspectives

2005-21 Wijnand Derks (UT) Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006-01 Samuil Angelov (TUE) Foundations of B2B Electronic Contracting

2006-02 Cristina Chisalita (VU) Contextual issues in the design and use of information technology in organizations

2006-03 Noor Christoph (UVA) The role of metacognitive skills in learning to solve problems

2006-04 Marta Sabou (VU) Building Web Service Ontologies

2006-05 Cees Pierik (UU) Validation Techniques for Object-Oriented Proof Outlines

2006-06 Ziv Baida (VU) Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling

2006-07 Marko Smiljanic (UT) XML schema matching – balancing efficiency and effectiveness by means of clustering

2006-08 Eelco Herder (UT) Forward, Back and Home Again - Analyzing User Behavior on the Web

2006-09 Mohamed Wahdan (UM) Automatic Formulation of the Auditor's Opinion

2006-10 Ronny Siebes (VU) Semantic Routing in Peer-to-Peer Systems

2006-11 Joeri van Ruth (UT) Flattening Queries over Nested Data Types

2006-12 Bert Bongers (VU) Interactivation - Towards an e-cology of people, our technological environment, and the arts

2006-13 Henk-Jan Lebbink (UU) Dialogue and Decision Games for Information Exchanging Agents

2006-14 Johan Hoorn (VU) Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change

2006-15 Rainer Malik (UU) CONAN: Text Mining in the Biomedical Domain

2006-16 Carsten Riggelsen (UU) Approximation Methods for Efficient Learning of Bayesian Networks

2006-17 Stacey Nagata (UU) User Assistance for Multitasking with Interruptions on a Mobile Device

2006-18 Valentin Zhizhkun (UVA) Graph transformation for Natural Language Processing

2006-19 Birna van Riemsdijk (UU) Cognitive Agent Programming: A Semantic Approach

2006-20 Marina Velikova (UvT) Monotone models for prediction in data mining

2006-21 Bas van Gils (RUN) Aptness on the Web

2006-22 Paul de Vrieze (RUN) Fundaments of Adaptive Personalisation

2006-23 Ion Juvina (UU) Development of Cognitive Model for Navigating on the Web

2006-24 Laura Hollink (VU) Semantic Annotation for Retrieval of Visual Resources

2006-25 Madalina Drugan (UU) Conditional log-likelihood MDL and Evolutionary MCMC

2006-26 Vojkan Mihajlovic (UT) Score Region Algebra:

A Flexible Framework for Structured Information Retrieval

2006-27 Stefano Bocconi (CWI) Vox Populi: generating video documentaries from semantically annotated media repositories

2006-28 Borkur Sigurbjornsson (UVA) Focused Information Access using XML Element Retrieval

2007-01 Kees Leune (UvT) Access Control and Service-Oriented Architectures

2007-02 Wouter Teepe (RUG) Reconciling Information Exchange and Confidentiality: A Formal Approach

2007-03 Peter Mika (VU) Social Networks and the Semantic Web

2007-04 Jurriaan van Diggelen (UU) Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach

2007-05 Bart Schermer (UL) Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance

2007-06 Gilad Mishne (UVA) Applied Text Analytics for Blogs

2007-07 Natasa Jovanovic' (UT) To Whom It May Concern - Addressee Identification in Face-to-Face Meetings

2007-08 Mark Hoogendoorn (VU) Modeling of Change in Multi-Agent Organizations

2007-09 David Mobach (VU) Agent-Based Mediated Service Negotiation

2007-10 Huib Aldewereld (UU) Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols

2007-11 Natalia Stash (TUE) Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System

2007-12 Marcel van Gerven (RUN) Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty

2007-13 Rutger Rienks (UT) Meetings in Smart Environments; Implications of Progressing Technology

2007-14 Niek Bergboer (UM) Context-Based Image Analysis

2007-15 Joyca Lacroix (UM) NIM: a Situated Computational Memory Model

2007-16 Davide Grossi (UU) Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems

2007-17 Theodore Charitos (UU) Reasoning with Dynamic Networks in Practice

2007-18 Bart Orriens (UvT) On the development an management of adaptive business collaborations

2007-19 David Levy (UM) Intimate relationships with artificial partners

2007-20 Slinger Jansen (UU) Customer Configuration Updating in a Software Supply Network

2007-21 Karianne Vermaas (UU) Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005

2007-22 Zlatko Zlatev (UT) Goal-oriented design of value and process models from patterns

2007-23 Peter Barna (TUE) Specification of Application Logic in Web Information Systems

2007-24 Georgina Ramrez Camps (CWI) Structural Features in XML Retrieval

2007-25 Joost Schalken (VU) Empirical Investigations in Software Process Improvement

2008-01 Katalin Boer-Sorbn (EUR) Agent-Based Simulation of Financial Markets: A modular, continuous-time approach

2008-02 Alexei Sharpanskykh (VU) On Computer-Aided Methods for Modeling and Analysis of Organizations

2008-03 Vera Hollink (UVA) Optimizing hierarchical menus: a usage-based approach

2008-04 Ander de Keijzer (UT) Management of Uncertain Data - towards unattended integration

2008-05 Bela Mutschler (UT) Modeling and simulating causal dependencies on process-aware information systems from a cost perspective

2008-06 Arjen Hommersom (RUN) On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective

2008-07 Peter van Rosmalen (OU) Supporting the tutor in the design and support of adaptive e-learning

2008-08 Janneke Bolt (UU) Bayesian Networks: Aspects of

Approximate Inference

2008-09 Christof van Nimwegen (UU) The paradox of the guided user: assistance can be counter-effective

2008-10 Wauter Bosma (UT) Discourse oriented summarization

2008-11 Vera Kartseva (VU) Designing Controls for Network Organizations: A Value-Based Approach

2008-12 Jozsef Farkas (RUN) A Semiotically Oriented Cognitive Model of Knowledge Representation

2008-13 Caterina Carraciolo (UVA) Topic Driven Access to Scientific Handbooks

2008-14 Arthur van Bunningen (UT) Context-Aware Querying; Better Answers with Less Effort

2008-15 Martijn van Otterlo (UT) The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains.

2008-16 Henriette van Vugt (VU) Embodied agents from a user's perspective

2008-17 Martin Op 't Land (TUD) Applying Architecture and Ontology to the Splitting and Allying of Enterprises

2008-18 Guido de Croon (UM) Adaptive Active Vision

2008-19 Henning Rode (UT) From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search

2008-20 Rex Arendsen (UVA) Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven.

2008-21 Krisztian Balog (UVA) People Search in the Enterprise

2008-22 Henk Koning (UU) Communication of IT-Architecture

2008-23 Stefan Visscher (UU) Bayesian network models for the management of ventilator-associated pneumonia

2008-24 Zharko Aleksovski (VU) Using background knowledge in ontology matching

2008-25 Geert Jonker (UU) Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency

2008-26 Marijn Huijbregts (UT) Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled

2008-27 Hubert Vogten (OU) Design and Implementation Strategies for IMS Learning Design

2008-28 Ildiko Flesch (RUN) On the Use of Independence Relations in Bayesian Networks

2008-29 Dennis Reidsma (UT) Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans

2008-30 Wouter van Atteveldt (VU) Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content

2008-31 Loes Braun (UM) Pro-Active Medical Information Retrieval

2008-32 Trung H. Bui (UT) Toward Affective Dialogue Management using Partially Observable Markov Decision Processes

2008-33 Frank Terpstra (UVA) Scientific Workflow Design; theoretical and practical issues

2008-34 Jeroen de Knijf (UU) Studies in Frequent Tree Mining

2008-35 Ben Torben Nielsen (UvT) Dendritic morphologies: function shapes structure

2009-01 Rasa Jurgelenaite (RUN) Symmetric Causal Independence Models

2009-02 Willem Robert van Hage (VU) Evaluating Ontology-Alignment Techniques

2009-03 Hans Stol (UvT) A Framework for Evidence-based Policy Making Using IT

2009-04 Josephine Nabukenya (RUN) Improving the Quality of Organisational Policy Making using Collaboration Engineering

2009-05 Sietse Overbeek (RUN) Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality

2009-06 Muhammad Subianto (UU) Understanding Classification

2009-07 Ronald Poppe (UT) Discriminative Vision-Based Recovery and Recognition of Human Motion

2009-08 Volker Nannen (VU) Evolutionary Agent-Based Policy Analysis in Dynamic Environments

2009-09 Benjamin Kanagwa (RUN) Design, Discovery and Construction of Service-oriented Systems

2009-10 Jan Wielemaker (UVA) Logic programming for knowledge-intensive interactive applications

2009-11 Alexander Boer (UVA) Legal Theory, Sources of Law & the Semantic Web

2009-12 Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin) Operating Guidelines for Services

2009-13 Steven de Jong (UM) Fairness in Multi-Agent Systems

2009-14 Maksym Korotkiy (VU) From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)

2009-15 Rinke Hoekstra (UVA) Ontology Representation - Design Patterns and Ontologies that Make Sense

2009-16 Fritz Reul (UvT) New Architectures in Computer Chess

2009-17 Laurens van der Maaten (UvT) Feature Extraction from Visual Data

2009-18 Fabian Groffen (CWI) Armada, An Evolving Database System

2009-19 Valentin Robu (CWI) Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets

2009-20 Bob van der Vecht (UU) Adjustable Autonomy: Controling Influences on Decision Making

2009-21 Stijn Vanderlooy (UM) Ranking and Reliable Classification

2009-22 Pavel Serdyukov (UT) Search For Expertise: Going beyond direct evidence

2009-23 Peter Hofgesang (VU) Modelling Web Usage in a Changing Environment

2009-24 Annerieke Heuvelink (VUA) Cognitive Models for Training Simulations

2009-25 Alex van Ballegooij (CWI) "RAM: Array Database Management through Relational Mapping"

2009-26 Fernando Koch (UU) An Agent-Based Model for the Development of Intelligent Mobile Services

2009-27 Christian Glahn (OU) Contextual Support of social Engagement and Reflection on the Web

2009-28 Sander Evers (UT) Sensor Data Management with Probabilistic Models

2009-29 Stanislav Pokraev (UT) Model-Driven Semantic Integration of Service-Oriented Applications

2009-30 Marcin Zukowski (CWI) Balancing vectorized query execution with bandwidth-optimized storage

2009-31 Sofiya Katrenko (UVA) A Closer Look at Learning Relations from Text

2009-32 Rik Farenhorst (VU) and Remco de Boer (VU) Architectural Knowledge Management: Supporting Architects and Auditors

2009-33 Khiet Truong (UT) How Does Real Affect Affect Affect Recognition In Speech?

2009-34 Inge van de Weerd (UU) Advancing in Software Product Management: An Incremental Method Engineering Approach

2009-35 Wouter Koelewijn (UL) Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling

2009-36 Marco Kalz (OUN) Placement Support for Learners in Learning Networks

2009-37 Hendrik Drachsler (OUN) Navigation Support for Learners in Informal Learning Networks

2009-38 Riina Vuorikari (OU) Tags and self-organisation: a metadata ecology for learning resources in a multilingual context

2009-39 Christian Stahl (TUE, Humboldt-Universitaet zu Berlin) Service Substitution – A Behavioral Approach Based on Petri Nets

2009-40 Stephan Raaijmakers (UvT) Multinomial Language Learning: Investigations into the Geometry of Language

2009-41 Igor Berezhnyy (UvT) Digital Analysis of Paintings

2009-42 Toine Bogers (UvT) Recommender Systems for Social Bookmarking

2009-43 Virginia Nunes Leal Franqueira (UT) Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients

2009-44 Roberto Santana Tapia (UT) Assessing Business-

IT Alignment in Networked Organizations

2009-45 Jilles Vreeken (UU) Making Pattern Mining Useful

2009-46 Loredana Afanasiev (UvA) Querying XML: Benchmarks and Recursion

2010-01 Matthijs van Leeuwen (UU) Patterns that Matter

2010-02 Ingo Wassink (UT) Work flows in Life Science

2010-03 Joost Geurts (CWI) A Document Engineering Model and Processing Framework for Multimedia documents

2010-04 Olga Kulyk (UT) Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments

2010-05 Claudia Hauff (UT) Predicting the Effectiveness of Queries and Retrieval Systems

2010-06 Sander Bakkes (UvT) Rapid Adaptation of Video Game AI

2010-07 Wim Fikkert (UT) Gesture interaction at a Distance

2010-08 Krzysztof Siewicz (UL) Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments

2010-09 Hugo Kielman (UL) A Politiele gegevensverwerking en Privacy, Naar een effectieve waarborging

2010-10 Rebecca Ong (UL) Mobile Communication and Protection of Children

2010-11 Adriaan Ter Mors (TUD) The world according to MARP: Multi-Agent Route Planning

2010-12 Susan van den Braak (UU) Sensemaking software for crime analysis

2010-13 Gianluigi Folino (RUN) High Performance Data Mining using Bio-inspired techniques

2010-14 Sander van Splunter (VU) Automated Web Service Reconfiguration

2010-15 Lianne Bodenstaff (UT) Managing Dependency Relations in Inter-Organizational Models

2010-16 Sicco Verwer (TUD) Efficient Identification of Timed Automata, theory and practice

2010-17 Spyros Kotoulas (VU) Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications

2010-18 Charlotte Gerritsen (VU) Caught in the Act: Investigating Crime by Agent-Based Simulation

2010-19 Henriette Cramer (UvA) People's Responses to Autonomous and Adaptive Systems

2010-20 Ivo Swartjes (UT) Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative

2010-21 Harold van Heerde (UT) Privacy-aware data management by means of data degradation

2010-22 Michiel Hildebrand (CWI) End-user Support for Access to Heterogeneous Linked Data

2010-23 Bas Steunebrink (UU) The Logical Structure of Emotions

2010-24 Dmytro Tykhonov Designing Generic and Efficient Negotiation Strategies

2010-25 Zulfiqar Ali Memon (VU) Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective

2010-26 Ying Zhang (CWI) XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines

2010-27 Marten Voulon (UL) Automatisch contracteren

2010-28 Arne Koopman (UU) Characteristic Relational Patterns

2010-29 Stratos Idreos(CWI) Database Cracking: Towards Auto-tuning Database Kernels

2010-30 Marieke van Erp (UvT) Accessing Natural History - Discoveries in data cleaning, structuring, and retrieval

2010-31 Victor de Boer (UVA) Ontology Enrichment from Heterogeneous Sources on the Web

2010-32 Marcel Hiel (UvT) An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems

2010-33 Robin Aly (UT) Modeling Representation Uncertainty in Concept-Based Multimedia Retrieval

2010-34 Teduh Dirgahayu (UT) Interaction Design in Service Compositions

2010-35 Dolf Trieschnigg (UT) Proof of Concept: Concept-based Biomedical Information Retrieval

2010-36 Jose Janssen (OU) Paving the Way for Lifelong Learning; Facilitating competence development through a learning path specification

2010-37 Niels Lohmann (TUE) Correctness of services and their composition

2010-38 Dirk Fahland (TUE) From Scenarios to components

2010-39 Ghazanfar Farooq Siddiqui (VU) Integrative modeling of emotions in virtual agents

2010-40 Mark van Assem (VU) Converting and Integrating Vocabularies for the Semantic Web

2010-41 Guillaume Chaslot (UM) Monte-Carlo Tree Search

2010-42 Sybren de Kinderen (VU) Needs-driven service bundling in a multi-supplier setting - the computational e3-service approach

2010-43 Peter van Kranenburg (UU) A Computational Approach to Content-Based Retrieval of Folk Song Melodies

2010-44 Pieter Bellekens (TUE) An Approach towards Context-sensitive and User-adapted Access to Heterogeneous Data Sources, Illustrated in the Television Domain

2010-45 Vasilios Andrikopoulos (UvT) A theory and model for the evolution of software services

2010-46 Vincent Pijpers (VU) e3alignment: Exploring Inter-Organizational Business-ICT Alignment

2010-47 Chen Li (UT) Mining Process Model Variants: Challenges, Techniques, Examples

2010-48 Withdrawn

2010-49 Jahn-Takeshi Saito (UM) Solving difficult game positions

2010-50 Bouke Huurnink (UVA) Search in Audiovisual Broadcast Archives

2010-51 Alia Khairia Amin (CWI) Understanding and supporting information seeking tasks in multiple sources

2010-52 Peter-Paul van Maanen (VU) Adaptive Support for Human-Computer Teams: Exploring the Use of Cognitive Models of Trust and Attention

2010-53 Edgar Meij (UVA) Combining Concepts and Language Models for Information Access

2011-01 Botond Cseke (RUN) Variational Algorithms for Bayesian Inference in Latent Gaussian Models

2011-02 Nick Tinnemeier(UU) Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language

2011-03 Jan Martijn van der Werf (TUE) Compositional Design and Verification of Component-Based Information Systems

2011-04 Hado van Hasselt (UU) Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference learning algorithms

2011-05 Base van der Raadt (VU) Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.

2011-06 Yiwen Wang (TUE) Semantically-Enhanced Recommendations in Cultural Heritage

2011-07 Yujia Cao (UT) Multimodal Information Presentation for High Load Human Computer Interaction

2011-08 Nieske Vergunst (UU) BDI-based Generation of Robust Task-Oriented Dialogues

2011-09 Tim de Jong (OU) Contextualised Mobile Media for Learning

2011-10 Bart Bogaert (UvT) Cloud Content Contention

2011-11 Dhaval Vyas (UT) Designing for Awareness: An Experience-focused HCI Perspective

2011-12 Carmen Bratosin (TUE) Grid Architecture for Distributed Process Mining

2011-13 Xiaoyu Mao (UvT) Airport under Control. Multiagent Scheduling for Airport Ground Handling

2011-14 Milan Lovric (EUR) Behavioral Finance and Agent-Based Artificial Markets

2011-15 Marijn Koolen (UvA) The Meaning of Structure: the Value of Link Evidence for Information Retrieval

2011-16 Maarten Schadd (UM) Selective Search in Games of Different Complexity

2011-17 Jiyin He (UVA) Exploring Topic Structure: Coherence, Diversity and Relatedness

2011-18 Mark Ponsen (UM) Strategic Decision-Making in complex games

2011-19 Ellen Rusman (OU) The Mind ' s Eye on Personal Profiles

2011-20 Qing Gu (VU) Guiding service-oriented software engineering - A view-based approach

2011-21 Linda Terlouw (TUD) Modularization and Specification of Service-Oriented Systems

2011-22 Junte Zhang (UVA) System Evaluation of Archival Description and Access

2011-23 Wouter Weerkamp (UVA) Finding People and their Utterances in Social Media

2011-24 Herwin van Welbergen (UT) Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior

2011-25 Syed Waqar ul Qounain Jaffry (VU)) Analysis and Validation of Models for Trust Dynamics

2011-26 Matthijs Aart Pontier (VU) Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots

2011-27 Aniel Bhulai (VU) Dynamic website optimization through autonomous management of design patterns

2011-28 Rianne Kaptein(UVA) Effective Focused Retrieval by Exploiting Query Context and Document Structure

2011-29 Faisal Kamiran (TUE) Discrimination-aware Classification

2011-30 Egon van den Broek (UT) Affective Signal Processing (ASP): Unraveling the mystery of emotions

2011-31 Ludo Waltman (EUR) Computational and Game-Theoretic Approaches for Modeling Bounded Rationality

2011-32 Nees-Jan van Eck (EUR) Methodological Advances in Bibliometric Mapping of Science

2011-33 Tom van der Weide (UU) Arguing to Motivate Decisions

2011-34 Paolo Turrini (UU) Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations

2011-35 Maaike Harbers (UU) Explaining Agent Behavior in Virtual Training

2011-36 Erik van der Spek (UU) Experiments in serious game design: a cognitive approach

2011-37 Adriana Burlutiu (RUN) Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference

2011-38 Nyree Lemmens (UM) Bee-inspired Distributed Optimization

2011-39 Joost Westra (UU) Organizing Adaptation using Agents in Serious Games

2011-40 Viktor Clerc (VU) Architectural Knowledge Management in Global Software Development

2011-41 Luan Ibraimi (UT) Cryptographically Enforced Distributed Data Access Control

2011-42 Michal Sindlar (UU) Explaining Behavior through Mental State Attribution

2011-43 Henk van der Schuur (UU) Process Improvement through Software Operation Knowledge

2011-44 Boris Reuderink (UT) Robust Brain-Computer Interfaces

2011-45 Herman Stehouwer (UvT) Statistical Language Models for Alternative Sequence Selection

2011-46 Beibei Hu (TUD) Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work

2011-47 Azizi Bin Ab Aziz(VU) Exploring Computational Models for Intelligent Support of Persons with Depression

2011-48 Mark Ter Maat (UT) Response Selection and Turn-taking for a Sensitive Artificial Listening Agent

2011-49 Andreea Niculescu (UT) Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality

2012-01 Terry Kakeeto (UvT) Relationship Marketing for SMEs in Uganda

2012-02 Muhammad Umair(VU) Adaptivity, emotion, and Rationality in Human and Ambient Agent Models

2012-03 Adam Vanya (VU) Supporting Architecture Evolution by Mining Software Repositories

2012-04 Jurriaan Souer (UU) Development of Content Management System-based Web Applications

2012-05 Marijn Plomp (UU) Maturing Interorganisational Information Systems

2012-06 Wolfgang Reinhardt (OU) Awareness Support for Knowledge Workers in Research Networks

2012-07 Rianne van Lambalgen (VU) When the Going

Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions

2012-08 Gerben de Vries (UVA) Kernel Methods for Vessel Trajectories

2012-09 Ricardo Neisse (UT) Trust and Privacy Management Support for Context-Aware Service Platforms

2012-10 David Smits (TUE) Towards a Generic Distributed Adaptive Hypermedia Environment

2012-11 J.C.B. Rantham Prabhakara (TUE) Process Mining in the Large: Preprocessing, Discovery, and Diagnostics

2012-12 Kees van der Sluijs (TUE) Model Driven Design and Data Integration in Semantic Web Information Systems

2012-13 Suleman Shahid (UvT) Fun and Face: Exploring non-verbal expressions of emotion during playful interactions

2012-14 Evgeny Knutov(TUE) Generic Adaptation Framework for Unifying Adaptive Web-based Systems

2012-15 Natalie van der Wal (VU) Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes.

2012-16 Fiemke Both (VU) Helping people by understanding them - Ambient Agents supporting task execution and depression treatment

2012-17 Amal Elgammal (UvT) Towards a Comprehensive Framework for Business Process Compliance

2012-18 Eltjo Poort (VU) Improving Solution Architecting Practices

2012-19 Helen Schonenberg (TUE) What's Next? Operational Support for Business Process Execution

2012-20 Ali Bahramisharif (RUN) Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing

2012-21 Roberto Cornacchia (TUD) Querying Sparse Matrices for Information Retrieval

2012-22 Thijs Vis (UvT) Intelligence, politie en veiligheidsdienst: verenigbare grootheden?

2012-23 Christian Muehl (UT) Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction

2012-24 Laurens van der Werff (UT) Evaluation of Noisy Transcripts for Spoken Document Retrieval

2012-25 Silja Eckartz (UT) Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application

2012-26 Emile de Maat (UVA) Making Sense of Legal Text

2012-27 Hayrettin Gurkok (UT) Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games

2012-28 Nancy Pascall (UvT) Engendering Technology Empowering Women

2012-29 Almer Tigelaar (UT) Peer-to-Peer Information Retrieval

2012-30 Alina Pommeranz (TUD) Designing Human-Centered Systems for Reflective Decision Making

2012-31 Emily Bagarukayo (RUN) A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure

2012-32 Wietske Visser (TUD) Qualitative multi-criteria preference representation and reasoning

2012-33 Rory Sie (OUN) Coalitions in Cooperation Networks (COCOON)

2012-34 Pavol Jancura (RUN) Evolutionary analysis in PPI networks and applications

2012-35 Evert Haasdijk (VU) Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics

2012-36 Denis Ssebugwawo (RUN) Analysis and Evaluation of Collaborative Modeling Processes

2012-37 Agnes Nakakawa (RUN) A Collaboration Process for Enterprise Architecture Creation

2012-38 Selmar Smit (VU) Parameter Tuning and Scientific Testing in Evolutionary Algorithms

2012-39 Hassan Fatemi (UT) Risk-aware design of value and coordination networks

2012-40 Agus Gunawan (UvT) Information Access for SMEs in Indonesia

2012-41 Sebastian Kelle (OU) Game Design Patterns for Learning

2012-42 Dominique Verpoorten (OU) Reflection Amplifiers in self-regulated Learning

2012-43 Withdrawn

2012-44 Anna Tordai (VU) On Combining Alignment Techniques

2012-45 Benedikt Kratz (UvT) A Model and Language for Business-aware Transactions

2012-46 Simon Carter (UVA) Exploration and Exploitation of Multilingual Data for Statistical Machine Translation

2012-47 Manos Tsagkias (UVA) Mining Social Media: Tracking Content and Predicting Behavior

2012-48 Jorn Bakker (TUE) Handling Abrupt Changes in Evolving Time-series Data

2012-49 Michael Kaisers (UM) Learning against Learning - Evolutionary dynamics of reinforcement learning algorithms in strategic interactions

2012-50 Steven van Kervel (TUD) Ontologogy driven Enterprise Information Systems Engineering

2012-51 Jeroen de Jong (TUD) Heuristics in Dynamic Sceduling; a practical framework with a case study in elevator dispatching

2013-01 Viorel Milea (EUR) News Analytics for Financial Decision Support

2013-02 Erietta Liarou (CWI) MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing

2013-03 Szymon Klarman (VU) Reasoning with Contexts in Description Logics

2013-04 Chetan Yadati(TUD) Coordinating autonomous planning and scheduling

2013-05 Dulce Pumareja (UT) Groupware Requirements Evolutions Patterns

2013-06 Romulo Goncalves(CWI) The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience

2013-07 Giel van Lankveld (UvT) Quantifying Individual Player Differences

2013-08 Robbert-Jan Merk(VU) Making enemies: cognitive modeling for opponent agents in fighter pilot simulators

2013-09 Fabio Gori (RUN) Metagenomic Data Analysis: Computational Methods and Applications

2013-10 Jeewanie Jayasinghe Arachchige(UvT) A Unified Modeling Framework for Service Design.

2013-11 Evangelos Pournaras(TUD) Multi-level Reconfigurable Self-organization in Overlay Services

2013-12 Marian Razavian(VU) Knowledge-driven Migration to Services

2013-13 Mohammad Safiri(UT) Service Tailoring: User-centric creation of integrated IT-based homecare services to support independent living of elderly

2013-14 Jafar Tanha (UVA) Ensemble Approaches to Semi-Supervised Learning Learning

2013-15 Daniel Hennes (UM) Multiagent Learning - Dynamic Games and Applications

2013-16 Eric Kok (UU) Exploring the practical benefits of argumentation in multi-agent deliberation

2013-17 Koen Kok (VU) The PowerMatcher: Smart Coordination for the Smart Electricity Grid

2013-18 Jeroen Janssens (UvT) Outlier Selection and One-Class Classification

2013-19 Renze Steenhuizen (TUD) Coordinated Multi-Agent Planning and Scheduling

2013-20 Katja Hofmann (UvA) Fast and Reliable Online Learning to Rank for Information Retrieval

2013-21 Sander Wubben (UvT) Text-to-text generation by monolingual machine translation

2013-22 Tom Claassen (RUN) Causal Discovery and Logic

2013-23 Patricio de Alencar Silva(UvT) Value Activity Monitoring

2013-24 Haitham Bou Ammar (UM) Automated Transfer in Reinforcement Learning

2013-25 Agnieszka Anna Latoszek-Berendsen (UM) Intention-based Decision Support. A new way of representing and implementing clinical guidelines in a Decision Support System

2013-26 Alireza Zarghami (UT) Architectural Support for Dynamic Homecare Service Provisioning

2013-27 Mohammad Huq (UT) Inference-based Framework Managing Data Provenance

2013-28 Frans van der Sluis (UT) When Complexity becomes Interesting: An Inquiry into the Information eXperience

2013-29 Iwan de Kok (UT) Listening Heads

2013-30 Joyce Nakatumba (TUE) Resource-Aware Business Process Management: Analysis and Support

2013-31 Dinh Khoa Nguyen (UvT) Blueprint Model and Language for Engineering Cloud Applications

2013-32 Kamakshi Rajagopal (OUN) Networking For Learning; The role of Networking in a Lifelong Learner's Professional Development

2013-33 Qi Gao (TUD) User Modeling and Personalization in the Microblogging Sphere

2013-34 Kien Tjin-Kam-Jet (UT) Distributed Deep Web Search

2013-35 Abdallah El Ali (UvA) Minimal Mobile Human Computer Interaction Promotor: Prof. dr. L. Hardman (CWI/UVA)

2013-36 Than Lam Hoang (TUe) Pattern Mining in Data Streams

2013-37 Dirk Brner (OUN) Ambient Learning Displays

2013-38 Eelco den Heijer (VU) Autonomous Evolutionary Art

2013-39 Joop de Jong (TUD) A Method for Enterprise Ontology based Design of Enterprise Information Systems

2013-40 Pim Nijssen (UM) Monte-Carlo Tree Search for Multi-Player Games

2013-41 Jochem Liem (UVA) Supporting the Conceptual Modelling of Dynamic Systems: A Knowledge Engineering Perspective on Qualitative Reasoning

2013-42 Lon Planken (TUD) Algorithms for Simple Temporal Reasoning

2013-43 Marc Bron (UVA) Exploration and Contextualization through Interaction and Concepts

2014-01 Nicola Barile (UU) Studies in Learning Monotone Models from Data

2014-02 Fiona Tuliyano (RUN) Combining System Dynamics with a Domain Modeling Method

2014-03 Sergio Raul Duarte Torres (UT) Information Retrieval for Children: Search Behavior and Solutions

2014-04 Hanna Jochmann-Mannak (UT) Websites for children: search strategies and interface design - Three studies on children's search performance and evaluation

2014-05 Jurriaan van Reijsen (UU) Knowledge Perspectives on Advancing Dynamic Capability

2014-06 Damian Tamburri (VU) Supporting Networked Software Development

2014-07 Arya Adriansyah (TUE) Aligning Observed and Modeled Behavior

2014-08 Samur Araujo (TUD) Data Integration over Distributed and Heterogeneous Data Endpoints

2014-09 Philip Jackson (UvT) Toward Human-Level Artificial Intelligence: Representation and Computation of Meaning in Natural Language

2014-10 Ivan Salvador Razo Zapata (VU) Service Value Networks

2014-11 Janneke van der Zwaan (TUD) An Empathic Virtual Buddy for Social Support

2014-12 Willem van Willigen (VU) Look Ma, No Hands: Aspects of Autonomous Vehicle Control

2014-13 Arlette van Wissen (VU) Agent-Based Support for Behavior Change: Models and Applications in Health and Safety Domains

2014-14 Yangyang Shi (TUD) Language Models With Meta-information

2014-15 Natalya Mogles (VU) Agent-Based Analysis and Support of Human Functioning in Complex Socio-Technical Systems: Applications in Safety and Healthcare

2014-16 Krystyna Milian (VU) Supporting trial recruitment and design by automatically interpreting eligibility criteria

2014-17 Kathrin Dentler (VU) Computing healthcare quality indicators automatically: Secondary Use of Patient Data and Semantic Interoperability

2014-18 Mattijs Ghijsen (VU) Methods and Models for the Design and Study of Dynamic Agent Organizations

2014-19 Vincius Ramos (TUE) Adaptive Hypermedia Courses: Qualitative and Quantitative Evaluation and Tool Support

2014-20 Mena Habib (UT) Named Entity Extraction and Disambiguation for Informal Text: The Missing Link

2014-21 Kassidy Clark (TUD) Negotiation and Monitoring in Open Environments