Space efficient indexes for the big data era
Sidirourgos, L.

Citation for published version (APA):
Sidirourgos, E. (2014). Space efficient indexes for the big data era

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
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. **FUTURE WORK**

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.