Short Paper

MetisIDX - From Adaptive to Predictive Data Indexing

Elvis Teixeira
Federal University of Ceará
Fortaleza, Brazil
elvis.teixeira@lsbd.ufc.br

Paulo Amora
Federal University of Ceará
Fortaleza, Brazil
paulo.amora@lsbd.ufc.br

Javam C. Machado
Federal University of Ceará
Fortaleza, Brazil
javam.machado@lsbd.ufc.br

ABSTRACT

Exploratory data analysis characterized by analytic query workloads over large databases is now commonplace on both academia and industry. In these scenarios, data production velocity and unknown and drifting access patterns make the choice of access methods a challenging task. In this context, adaptive indexing techniques propose the use of partial indexes that are incrementally built in response to the actual query sequence and as a byproduct of query processing to optimize the access only to the key ranges of interest. This work presents a further development to this principle by leveraging the recent query history to predict the next key ranges and index them in advance, so the queries arriving in the near future find data in its final representation and higher placed in the storage hierarchy, since data must be loaded into main memory in order to be indexed. Adaptive merging is used as base architecture for the data structures and merge operations are executed in parallel with query execution instead of being the same operation. An extreme learning machine is used to perform key range forecasting and undergo continuous training by the indexing thread. The experiments show up to 38% lower query response times over a 1000 queries than adaptive merging, therefore lower overall response times and the decoupling of indexing operations during scan executions.

1 INTRODUCTION

Important modern database applications do not have a known workload in terms of access patterns. Data subsets which are the focus of query attention change over time, and ad hoc queries should be expected. Examples of this kind of application are found in scientific work or in exploratory analysis, which is an increasingly common daily task in many business areas today. No assumptions can be made about the workload, and database systems must still be able to answer the queries from these dynamic workloads efficiently, searching through and updating suitable data structures to speed up query processing.

To accomplish this task, adaptive indexing [7] was introduced. By adapting the DBMS internal structures to quickly answer queries that follow the current trends, it enables the system to perform better according to the dynamic workload. The fundamental idea is that each time data is scanned to answer a user query, an incremental step is performed to provide an index structure or refine it, which will permit subsequent scans to prune the search space and perform better. Such operations must be simple in order to avoid adding a prohibitive overhead to query execution [2] while still being useful to speed up queries and save access to slow storage devices.

Changing database physical layout in response to the workload can be powerful if used properly. The advantages of optimizing access only for the records of interest is based on the fact that, in most applications, a subset of the records is accessed more often than the rest. Partial indexes recognize this fact by focusing layout tuning in a subset of the indexed relation, but they lack the adaptive behavior and the possibility of incremental change. Instead of relying in periodic statistics observation, our approach continuously tracks the workload, since the access patterns and the set of records requested more frequently changes over time.

On the other hand, consider the situation in which, while trying to follow the query trends, the system organizes data in response to single queries which deviate from the underlying workload pattern. This wastes time and compromises performance. Additionally, the current query may not provide sufficient information to figure out the best key range to index in order to enable the next queries to take full advantage from the structure. More contextual information is needed. A workload model, instead of the advice from the current request, provides better guidance. These issues are similar to the problems of overfitting and generalization in pattern recognition tasks.

Improvements to adaptive indexing can be achieved by indexing key ranges not strictly equal to those of the query responses, possibly adding a stochastic component to the indexing process [4]. Another possibility is using periods of time when computing resources are not being exhaustively used to index key ranges not yet touched [10]. The main advantage of these approaches is the possibility of indexing a region of data that will be queried in the near future. When this happens, the response time will be optimal and the effort of indexing can be done independently of query processing, thus not incurring any extra overhead to it.

This work develops this analogy further by using an actual machine learning technique to create and continuously update a model of the query sequence. In other words, it leverages an adaptive structure and adds a predictive behavior to the index building operations based on a dynamic model of the workload, using the most recent requests as training data. By indexing data expected to be requested next, our index builder is less sensitive to anomalies, and avoids the effort of indexing uninteresting records. Such intelligent access structures fit naturally in the context the emerging self-driving systems [1], which promise to be able to handle highly dynamical and hybrid workloads while requiring even less manual operation than traditional systems.

The benefits of performing incremental indexing detached from query processing occurs when the forecasting succeeds and a key range is placed in the final index form before it is queried. In this case the access is logarithmic in the size of the indexed data, not on the total amount of data. If the forecasting predicts the wrong key range then the cost of the next query will be that of a scan in the partial index structure built in the first query, a partitioned B+ tree, as discussed in the next section. Even in this case, the scan will not be as costly as it would be in a strictly adaptive indexing scheme, because the scan algorithm will not have to move data around, but only to find the qualifying records. This scheme keeps the hypothesis that the workload is unknown, as in previous adaptive indexes, but it recognizes that application queries are not random, and an underlying pattern should exist.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

2 METISIDX

MetisIDX is an indexing mechanism for relational data that targets secondary storage (HDDs, SSDs, etc) and uses the accumulated knowledge from previous queries in the workload to guide the index construction. The partitioned B+ tree, and some of the index creation and maintenance procedures are similar to those developed for adaptive merging [3]. A B-tree based structure was chosen to exploit its paged data transfer, which is naturally performed with block-addressed devices and permits meaningful data blocks to be exchanged between the levels of a memory hierarchy. However, differently from the adaptive merging approach, MetisIDX indexing routine is decoupled from data scanning and is less sensitive to workload anomalies since it leverages the information provided by multiple requests rather than the current query alone.

The learning algorithm must be lightweight enough to be executed in time frames that do not exceed the order of magnitude required to answer a single query. This constraint makes it impractical to use many state-of-art machine learning techniques such as massive deep neural networks, since using these methods would result in a training time long enough to make the resulting model already outdated, in the sense that it reflects a workload pattern that is already gone. Additionally, it must be able to update the model using new available data, new queries in this context, without discarding the previous model version altogether. A suitable technique that fulfills these requirements is the Extreme Learning Machine (ELM) [9], a class of neural networks which always has a single hidden layer, and only the weights of the connections between the hidden neurons and the output neurons must be trained.

The training data used in model updates is the sequence of key range boundaries from the last N queries processed by the system. N is a hyper parameter that has to be chosen. The trained network is the current model of the workload, used to forecast index range candidates and trigger a new merging operation. In the next training step the model is updated to keep track of possible workload trends shift. After that, the model is used to forecast the key range to be queried next, and a new merging operation is triggered. The process of training and triggering merges is carried out cyclically in a dedicated execution thread.

The first query triggers a full scan over the non-indexed data. During this first scan execution, data is read from secondary storage in chunks, called runs hereafter. Each run is then sorted using an algorithm suitable for main memory resident data. The records that belong to the query response set are then collected and returned to the user. The sorted runs are written back to disk as the leaves of a partitioned B+ tree and global ordering is achieved by introducing an artificial leading attribute whose value is the runs creation order.

The size of the runs is limited by the amount of main memory available and should be as big as possible, since longer runs provide fewer index partitions and thus faster access, because the tree must be traversed from root to leaf level for each partition. This partitioned index is already able to speed up the processing of subsequent queries, each time a search operation is executed the partitioned tree is traversed from the root to leaf level once per partition. In order to achieve optimal read access, the partitions must be merged into a single one so that the index becomes a regular B+ tree (not partitioned). In the adaptive merging [3] approach, partition merging and query processing are a single operation.

Algorithm 1: Forecast And Index Thread

```
   Data: Predicates of last N queries
   Result: Predicted query key ranges indexed continuously
 1 while True do
 2     if at least N new queries observed then
 3         train neural network;
 4         discard training data;
 5     else
 6         predict next key range;
 7         if range is not locked by query thread then
 8             acquire latch on key range;
 9             merge key range to final index;
10             release latch;
11     end
12 end
```

In MetisIDX, two separate structures are used, one is the partitioned tree resulting from the first query, and the second, another B+ tree structure for the final index, which is composed of the results of the merging operations performed by the indexing thread. This design was chosen to minimize the efforts of structure maintenance, as a result, no merges are performed on the nodes of the partitioned tree, i. e. they are permitted to underflow and the height of the tree is fixed. In the final index, however, overflow checks happen and nodes are split as more data is moved from the partitioned tree to their final position. Adaptive merging, on the other hand, has to deal with structure maintenance for merges and splits, because the transition from the partitioned structure to the final index is accomplished by collecting the records that belong to the response set of the query and merging them into a final partition, which becomes the full index after a number of queries. All the partitions compose a single tree structure, including the final partition.

In order to answer a range query, such as the ones used in the experiment, the query processing thread traverses the partitioned tree once for each partition in the tree and collects the records of interest, then it proceeds to scan the final index to account for the case in which the required records have already been merged to that location. The predictive behavior of MetisIDX minimizes the need for the operation of scanning the partitioned tree since the merging operations are made in anticipation and eventually make entire partitions empty.

The decisions on which key ranges to merge and the actual merging operations are done independently and in parallel to query processing. Such decision process comes from the extreme learning machine that is continuously trained in background by a dedicated thread. That same thread is used to perform merges at the end of each training mini-batch. After each merging operation the indexing thread attempts to perform a new model update if enough new queries have been observed. An important difference between a predictive system and a strictly adaptive one is the fact that, by indexing a key range before it is queried, the records in the indexed range will be brought up to cache. This means that predictive indexing is also a predictive cache prefetching.

Algorithm (1) depicts this process. The whole procedure is executed in an infinite loop in the indexing thread, that continuously tries to perform merge operations if the required resources are not protected by a latch, and performs a training step if a number of queries have been observed. This number must be chosen by the user, for our setup we used 10.

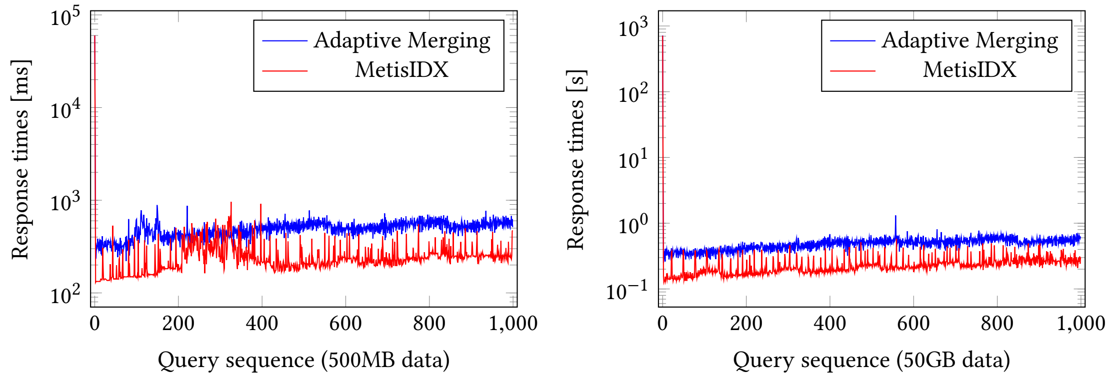

Figure 1: Response times

3 EXPERIMENTAL EVALUATION AND SETUP

MetisIDX and Adaptive Merging were both implemented for comparison in a custom storage engine named MetisDB, which implements the usual B+ tree access operations including persistence. The inclusion of adaptive merging and MetisIDX implementations in well known and complete database management systems would be a valuable way to fully evaluate and compare the performances and the implications for transaction processing by making the use of standard benchmarks possible. However, as pointed out by the authors of adaptive merging, conventional database architectures make a clear distinction between scans and index updates, treating the first as read-only, and the modifications needed to integrate the query-and-index strategies use in adaptive approaches call for whole new system architectures built with adaptive components in mind. This is the primary motivation for the MetisDB.

Since MetisIDX is not a strictly adaptive technique, and it adds indexing steps which are not part of query processing, it would make sense to compare it with Holistic Indexing, which has similar attributes. However, that technique is designed to work in the context of main memory column stores, while our approach targets tuple-based systems in secondary storage. For this reason Adaptive Merging is used as a baseline, as its application domain and data structures used are the same. The machine in which the experiments were carried out consists of a 3.1GHz Intel i5 processor, a 500GB, 7200RPM Seagate hard disk drive, and a 2x4GB DDR3 memory. The software stack is composed of a Debian GNU/Linux version 9 operating system and MetisDB was compiled using the GNU C++ compiler version 6.3.

The MetisDB storage engine contains a buffer manager that uses an LRU cache replacement policy on $8KB$ pages, LRU was chosen to favor recently loaded pages to remain in memory, as this is the case for recently indexed key ranges. Response times are used as a performance metric for the purpose of comparison. We use this metric instead of the usual I/O operations count used in disk-based access method evaluation because this quantity is not as directly linked to response times in MetisIDX as it is in other access methods. The reason is that, as we index data before query processing occurs and the indexing process must bring data up to the main memory cache, the select operator is expected to find its response set in the buffer pool. In other words, it does not matter how many disk accesses were performed if the data is in cache by the time one needs it.

The synthetic data used in the experiments consists of tuples with a 64bit integer used as the index key, and a 48B random string added to increase volume. Two tables were created, one containing 500MB and 7,106,208 tuples with the format described above, hereafter called $T1$, and another containing 50GB and 727,483,873 tuples, called $T2$. Two different data sizes were used to assess the effects of the domain size for the neural network predictions and cache invalidation, since the 500MB data can be entirely accommodated on cache while the larger one can not.

The query workload consists of 1000 range queries of the form `SELECT COUNT(*) ... WHERE A =>` Q_{LOW} `AND A <=` Q_{HI}`;` where the key range limits, Q_{LOW} and Q_{HI}, are generated from a function Q that maps each query to a point in the search key space. Let the domain of the key be $[0, M)$ and j be the order of a query, then the Q used is

$$Q(j) = M \cdot (j/C)^2 \tag{1}$$

where M is the maximum value of the search key and C is the order of the last query, 1000 in the given setup. In this form Q will distribute the queries over the entire key range. From this function (1) Q_{LOW} and Q_{HI} are derived by

$$\begin{aligned} Q_{LOW}(j) &= Q(j) - R_1 \\ Q_{HI}(j) &= Q(j) + R_2 \end{aligned} \tag{2}$$

where R_1 and R_2 in (2) are random positive values generated by a normal distribution with standard deviation equal to 1% of the key range. This parameter determines the selectivity of the queries and additional executions with different values were performed with similar results. These are the functions the neural networks learn. A quadratic function was chosen to define the access pattern as it is a simple non-linear function, and the goal is not to test the ELM forecasting capabilities for complex functions as it is done elsewhere [5]. This quadratic function, even though non-linear, is a sequential access, the worst case for adaptive indexes since, as a key range is never queried more than once, each one faces the non-indexed part of the data.

Two ELMs were used, one learns Q_{LOW} and the other learns Q_{HI}, both as functions of j. Each network has one neuron in the input and output layers, since the target function is one-dimensional, and four neurons in the hidden layer. A test with 4 neurons was carried out and yielded the same results, since 4 is enough for such simple function. In a real application where the access pattern may be more complicated, the use of more neurons in the hidden layer is advisable.

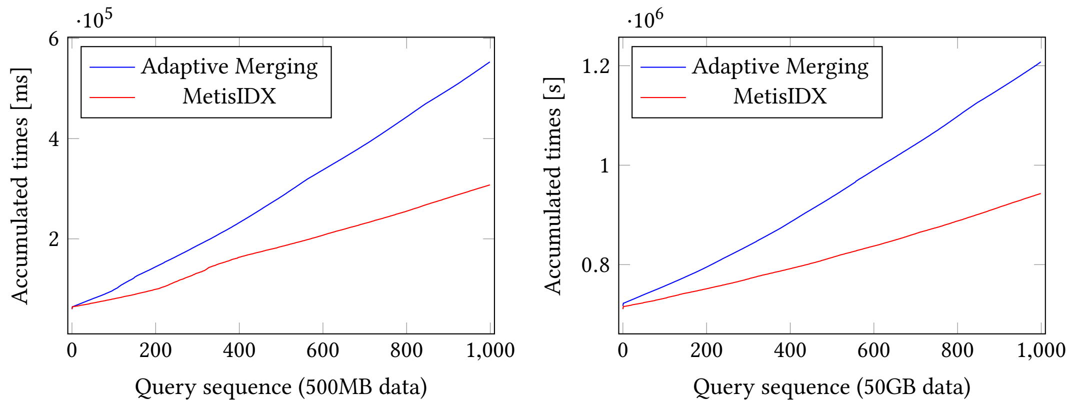

Figure 2: Accumulated response times

Figure (1) shows the response times of MetisIDX and adaptive merging for the two data sizes. For both experiments the fact that the first query has a cost much higher than the ones that follow is visible, this result is compatible with the related work in adaptive indexing in general. The queries over the 500MB data have a noisy behavior likely due to the CPU usage of the training thread and other processes running in parallel, since the data is small enough to be all in memory after the first query. On the other hand, the queries over 50GB data have a more consistent behaviour and the advantage of decoupling indexing from query processing becomes clear. The soft increase seen along the query sequence is due to the the growing number of cached pages, which increases the addressing costs.

Figure 2 Shows the accumulated query processing time, that is, the time spent to perform the first N queries as a function of N. The behavior of the curves show that as more queries are processed the gain in overall performance increases so that long running applications benefit the most from the technique. This also points a common trait of adaptive indexes, that is, as more queries are processed, the system gains knowledge of the data stored and of the workload.

In a few queries, the response times of MetisIDX jumps to the level of adaptive indexing. This happens when a key range starts being indexed and then is requested by a query before the indexing action finishes. It is the only situation in which the query processing thread waits. Since the observation window used is 10 queries and the indexing action for the next immediate query happens after each observation window, this is expected to be the periodicity of these peaks.

4 CONCLUSIONS AND FUTURE WORK

MetisIDX is able to speed up access by range queries by decoupling index building efforts from query processing while maintaining the workload oriented behavior of adaptive indexes. In this technique not only the current query is treated as a hint on how to physically organize the data but the entire query sequence is taken to be a sample of the underlying workload patterns. The workload is still assumed to be unknown, only the existence of an underlying pattern is required.

It shares the concurrency concerns of other adaptive techniques in terms of application access. Additionally, indexing occurs in parallel to query processing when the key ranges do not overlap. These considerations call for a latch free alternative to data structure access in order to alleviate latch contention, or even make the two actions race free, and increase throughput. An option to address these challenges is the adaptation of the approach presented here to the context a latch-free B-tree based structure, such as the Bw-tree [8] and a multiversion strategy to distinguish the data accessed by the current query and that being indexed may also be an option.

An in-depth analysis of the overall concurrency issues would be valuable not only for this particular technique but for any machine learning and pattern recognition based access methods. These are interesting issues for big data exploration as building full indexes upfront is an increasingly less attractive option [6] as data volumes grow. Learning from the data not only about the information it carries but also about the best ways to access it is a reasonable and, as far as we know, open research problem.

Acknowledgements

This research was partially supported by FUNCAP/CE-Brazil and LSBD/UFC.

REFERENCES

[1] J. Arulraj, A. Pavlo, and P. Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 583–598, 2016.

[2] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing access methods: The RUM conjecture. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016.*, pages 461–466, 2016.

[3] G. Graefe and H. A. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, pages 371–381, 2010.

[4] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *PVLDB*, 5(6):502–513, 2012.

[5] G. Huang, Q. Zhu, and C. K. Siew. Extreme learning machine: Theory and applications. *Neurocomputing*, 70(1-3):489–501, 2006.

[6] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *PVLDB*, 4(9):585–597, 2011.

[7] M. L. Kersten and S. Manegold. Cracking the database store. In *CIDR*, pages 213–224, 2005.

[8] J. J. Levandoski and S. Sengupta. The bw-tree: A latch-free b-tree for log-structured flash storage. *IEEE Data Eng. Bull.*, 36(2):56–62, 2013.

[9] N. Liang, G. Huang, P. Saratchandran, and N. Sundararajan. A fast and accurate online sequential learning algorithm for feedforward networks. *IEEE Trans. Neural Networks*, 17(6):1411–1423, 2006.

[10] E. Petraki, S. Idreos, and S. Manegold. Holistic indexing in main-memory column-stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1153–1166, 2015.