Progressive Indices: Indexing without prejudice

Pedro Holanda
supervised by Stefan Manegold
Centrum Wiskunde & Informatica
Amsterdam, Netherlands
holanda@cwi.nl

ABSTRACT

Database cracking is a method to create partial indices as a side-effect of processing queries. Cracking efficiently smears out the cost of creating a full index over a stream of queries, creating an index that is overfitted to queried parts of the data. This core characteristic of cracking leads to unpredictable performance and unreliable convergence towards a full index. These problems are aggravated when considering updates and multidimensional queries.

We envision a new indexing technique, *Progressive Indexing* that improves database cracking by strictly limiting per-query indexing cost to a budget (e.g., a user-defined fraction of scan costs), allowing the first and subsequent queries to complete without heavy penalties. At the same time, all indexing effort is spent towards predictable convergence towards a full index. We discuss different algorithms to deal with multidimensional queries and updates while maintaining a robust and convergent index, we then explore the research space and new challenges that arise from this new technique.

1. INTRODUCTION

Index creation is one of the major difficult decisions in database schema design [4]. Based on the workload, the database administrator needs to decide whether creating a specific index is worth the overhead of creating and maintaining it. This considerable up-front cost creates a trade-off that requires careful consideration and experimentation.

Automatic physical design tuning [2] aims to release the user of having to manually choose which indexes to create. They attempt to find the optimal set of indices given a query workload, by balancing the benefits of having an index versus the added costs of creating the index and maintaining it during modifications to the database. However, these tools are not able to work on dynamic systems due to the unpredictability and lack of idle time for a priori index creation.

Adaptive indexing techniques, such as database cracking [11], attempt to solve this problem by presenting an adaptive partial indexing approach for relational databases. It works by building a partial index as a co-product of query processing. The index is built the first time a column is queried and is continuously refined as subsequent queries are executed. This way the cost of creating an index is distributed over a stream of queries.

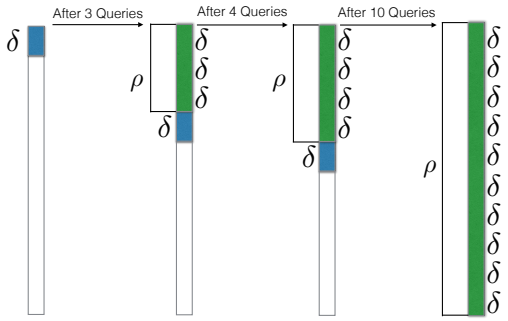

Figure 1: Progressive indexing.

However, database cracking and its variations are not robust against varying workload patterns. Since the index is only refined in the areas targeted by the workload. Queries that deviate from them will target unrefined sections of the index, leading to large and unpredictable spikes in performance. This problem is exacerbated when dealing with updates. In case the most refined area is also the most updated, and multidimensional queries (i.e., queries with selections in multiple columns) since one must maintain multiple indices, one for each column.

To address these needs we propose a novel approach for incremental indexes, *Progressive Indexing*. Where every query that is issued to the database results in a fixed refinement of the index, leading to a robust query execution and full convergence towards a full index. While maintaining a low extra cost per query.

Figure 1 depicts how progressive indexing works. Considering an unindexed column, a pivot ϵ and a budget $\delta = 0.1$, we start by indexing a δ fraction of the data while scanning the remaining $1 - \delta$ piece of data. After 3 queries, 30% of our column is already indexed around ϵ. The fourth query starts by performing an index lookup on the ρ fraction of the data that has already been indexed while scanning the unindexed $1 - \rho$ piece and expanding the index by another δ fraction of the total column. Finally, after 10 queries, we fully indexed ϵ and we continue this process by selecting another ϵ.

Proceedings of the VLDB 2018 PhD Workshop, August 27, 2018. Rio de Janeiro, Brazil.

Paper Structure. The rest of this paper is structured as follows. Section 2 provides an overview of related work. Then, section 3, describes progressive indexing. Section 4 presents a brief proof of concept and experimental analysis. Finally, in section 5, we discuss our research plan.

2. RELATED WORK

Automatic physical database design has been an active research field for the past twenty years. These work resulted in two different areas; self-tuning tools and adaptive indexing.

Self-Tuning Tools [1, 3, 16, 6] attempt to solve this problem by automatically recommending a set of indexes to optimize a known workload of the system. However, these systems depend on previous workload knowledge and are only able to create full indexes. Unsuitable for unpredictable workloads or when there is no idle time to be invested in a priori index creation.

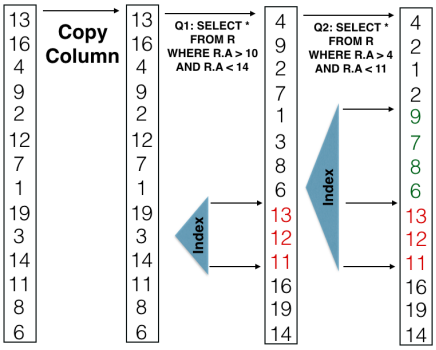

Figure 2: Database cracking.

Adaptive Indexing [15] is an alternative to the self-tuning tools. It is especially useful in scenarios where the workload is unpredictable and there is no idle time to invest in index creation. It tackles these problems by creating indexes that are workload dependent in an incremental fashion. Figure 2 depicts an example of database cracking [11]. Query Q_1 starts by triggering the creation of the cracker column(i.e., initially a copy of column A) where the tuples are clustered in three pieces reflecting the range predicate of Q_1. The result of Q_1 is then retrieved as a view on the Piece colored in red (i.e., $10 < A < 14$). Later, query Q_2 requires a refinement of Pieces 1 and 3 (i.e., respectively indexing $A > 7$ and $A \leq 16$), splitting each in two new pieces.

Database cracking has multiple issues: (1) poor convergence towards a full index, (2) inefficient tuple reconstruction, (3) unpredictable performance and (4) inefficient updates. Below we briefly discuss the research that addresses these issues.

Convergence. Hybrid cracking [5, 12, 15] mitigate the issue of poor convergence towards a full index by executing many initial cracking runs with random pivots. Although this provides better convergence and higher robustness it greatly impacts the cost of the first query.

Tuple Reconstruction. Sideways cracking [10, 15] address the inefficient tuple reconstruction problem. It minimizes the tuple reconstruction cost by using a "cracker maps" data structure. They provide a mapping between attributes that are combined in queries. However, the strategy is only applied to tuple reconstruction and not to multidimensional queries.

Robustness. Stochastic cracking [7, 15] address the unpredictable performance problem by creating partitions using a random pivot element instead of pivoting around the query predicates. However, the actual cracking still occurs in the pieces where the query predicates fall into.

Generalization. Adaptive adaptive indexing [14] attempts to be a general-purpose algorithm for adaptive indexing. It has multiple parameters that can be tuned to mimic the data access of multiple adaptive indexing techniques (e.g., database cracking, sideways cracking, hybrid cracking).

Updates. SPST-Index [8] extends the original cracking work on updates [9] by rotating nodes when they are accessed in order to cluster cold-data on the leaves. When updates are executed the leaves are pruned, consequently the index constraints are relaxed resulting in faster updates. However this work still alleviates the cost of updates by increasing the cost of cracking for the subsequent queries, bringing more unpredictability to the query costs.

3. PROGRESSIVE INDICES

The previous section gave us the necessary motivation for progressive indexing. Motivated by: (1) The workload dependent pivot selection causes performance spikes and does not guarantee convergence towards a full index. (2) Cracking is not able to efficiently handle multidimensional queries since it must maintain multiple indices and intersect their points, and (3) the robustness is penalized when dealing with updates due to partial index removal and the possibility of updates being focused in refined cracked pieces.

We propose progressive indexing. Progressive indexing is designed to be efficient when dealing with multiple types of queries and updates without sacrificing robustness or convergence. We believe that the following modifications from adaptive indexing must be taken: (1) the pivot-selection does not need to be workload dependent, every query must have a budget to spend on indexing creation or maintenance and when an indexed piece is smaller than L1 cache it is fully ordered. (2) Progressive indexing generates a unique index for multiple columns, and (3) other sorting algorithms, besides quick-sort adaptations, are exploited in order to efficiently merge updates while maintaining robustness and convergence.

As a result of the small initial cost, progressive indexing occurs without significant impact on query performance and has a near-immediate return on investment over performing naive scans. Even if the column is only queried a few times, progressive indexing will still provide a performance benefit. On the other hand, if the column is queried thousands of times, the index will reliably converge towards a full index and queries will be answered at the same performance as if a full index had been built.

Single Column. Figure 3 depicts an example of progressive quick-sort. In this example, we define a budget $\delta = 0.5$. Query Q_1 starts by triggering the *initialize* phase from progressive quick-sort. First, it allocates an uninitialized column of the same size of the original column and then selects 9 as a pivot ϵ. The original column is scanned and $n * \delta$, where n is the column size, elements are copied either to the top or the bottom of the copied column, depending on the pivot, while doing so we also select the elements that fulfill Q_1 predicates. A binary search tree (BST) is also formed to

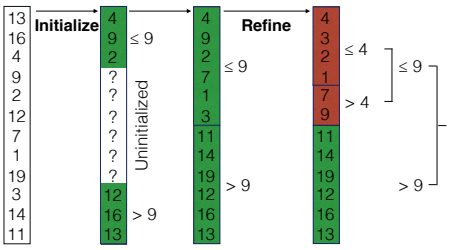

Figure 3: Progressive quick-sort.

keep track of the pivot points. The subsequent queries can already leverage from the sorted data by performing lookups in the BST. Later, query Q_2 triggers the *refine* phase fully indexing the column around ϵ. Q_3 then select another ϵ, and the refinement process continues until we reach a full index. The main disadvantage of progressive quick-sort is that fast convergence relies entirely on good pivot point selection.

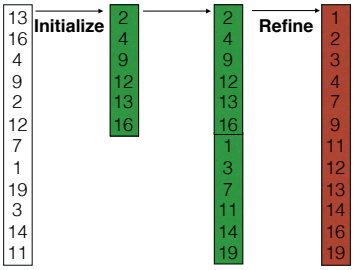

Figure 4: Progressive merge-sort.

Updates. Updates are stored in an extra vector. When a query is executed this extra column is also fully scanned. When our original column is fully sorted, we drop the BST, since we can now perform a binary search on the ordered column, and start the merging process with the column that holds the updates. To efficiently merge the updates into our original column we use progressive merge-sort. Progressive merge-sort consists of two build phases. In the first build phase, one unsorted chunk of size $n * \delta$ is sorted, where n is the column size and δ is our budget. In order to answer queries, we perform a binary search in the sorted chunks while scanning the unsorted data. In the second build phase, we merge the sorted chunks together using a cascading two-way merge. Note that we do not necessarily complete a full merge in one query, as this would result in large drops in performance when we merge two large chunks together. Instead, we merge at most $n * 2\delta$ elements and keep track of how far along the merge we are. After a merge is completed, we replace the two original chunks with the merged chunk. Figure 4 depicts the progressive merge-sort algorithm with $\delta = 0.5$, in the first phase half the column is ordered in a chunk. The second phase orders the remaining of the column in another chunk and finally both chunks are merged resulting in a fully ordered column. The main disadvantage of merge-sort is that while we have many sorted chunks, we are performing many random accesses, as we are doing a full binary search in each of the $1/\delta$ chunks.

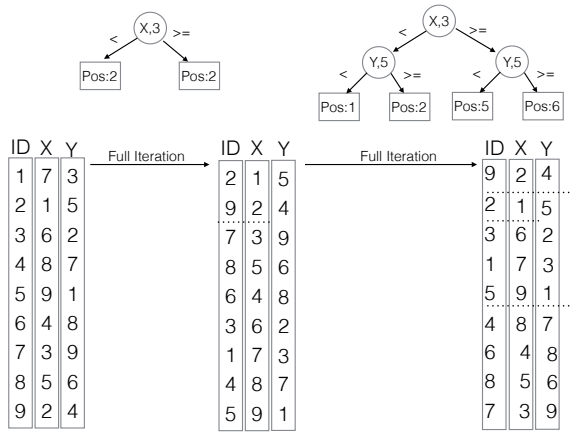

Figure 5: Multidimensional progressive quick-sort.

Multi-column. We adapt progressive quick-sort to work with multidimensional queries by generating a KD-Tree on top of the columns. In a KD-Tree every level of the tree consists of only one column. To maintain this property we interleave our pivots through the columns. Multidimensional progressive quick-sort can be visualized in figure 5. For simplicity in our example we set $\delta = 1$ (i.e., every iteration fully indexes one pivot). In the first iteration we select a $\epsilon = 3$ to use as a pivot for column X. Since our $\delta = 1$ at the end of the first query, we will have fully copied the columns and fully indexed the column X around 3, while keeping the alignment with column Y. A KD-Tree is then created to keep track of the indexed pivot. Later, when Q_2 is executed it triggers another iteration of progressive indexing. However, this time we are going to reorder the column Y over a new $\epsilon = 5$. Since we have already indexed X on 3, we need to reorder Y in the two pieces of X to maintain the alignment.T he main disadvantage of multidimensional progressive quick-sort is that it does not provide progressive usage of space, after fully indexing the first pivot, we have already made full copies of all the columns.

4. PRELIMINARY RESULTS

In this section, we present a brief experimental analysis to demonstrate the strong potential benefits of progressive indexing.

Setup. We implemented progressive quick-sort in a stand-alone program written in C++ and optimized using level 3. All experiments were conducted on a machine running Fedora 26, with an Intel Core i7-2600K CPU @ 3.40 GHz with 8 cores, 16 GB of main memory and 8192 KB L3 cache size.

We use an 8-byte integer array with 10^8 uniformly distributed values as our dataset. All queries are of the form: `SELECT SUM(R.A) FROM R WHERE R.A BETWEEN` V_1 `AND` V_2. All the queries have selectivity equal to 0.1 and we define our budget $\delta = 0.1$. All the progressive quick-sort pivots are randomly selected.

In Figure 6 we depict the per-query performance evaluation of progressive quick-sort, database cracking, stochastic cracking, coarse-granular index (i.e., a stochastic cracking variant), a B+tree as a full index and column scans. We can observe that adaptive indexing techniques show a very

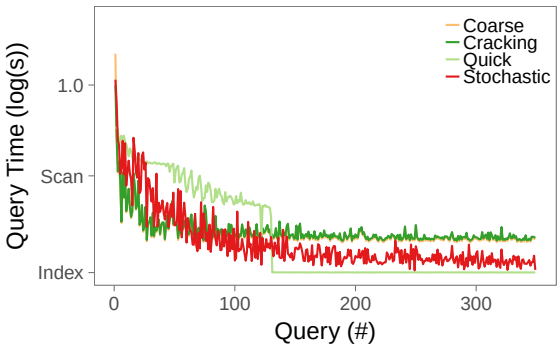

Figure 6: Progressive Quick-sort.

volatile performance with much more spikes throughout our query stream. This is due to the index refined by the workload. One can observe that progressive quick-sort has a more robust performance but still presents performance spikes due to the partial index creation depending entirely on the pivot selection. Additionally one can observe that progressive quick-sort converges to a full index around query 120, whereas the adaptive index techniques never converge.

5. FUTURE RESEARCH

Progressive indexing introduces new aspects that were unexplored by adaptive indexing and that require further investigation. We envision, as our ultimate goal, the creation of a formal cost-model with a decision tree that is able to choose and interleave from different progressive indexes. In order to optimize the query response time without penalizing robustness or convergence. We describe the following as the aspects that shall be explored:

- **Sorting-algorithms.** We discussed two different algorithms adapted to work as progressive indexes. However, many other well-known algorithms (e.g., radix-sort, bucket-sort, heap-sort) can be adapted to work in a progressive fashion as well.

- **Pivot-selection.** Some sorting algorithms (e.g., quick-sort) require a good pivot point selection. Different strategies can be applied to select a pivot. It can be workload dependent, data dependent, completely random or even a mix from all of the above;

- **Data Structure.** Different data structures can be used to exploit modern processes and boost access to the ordered data, like the ART-tree [13], or even just having a fully ordered vector and dropping any extra structure might be beneficial for faster updates;

- **Index Budget.** We used as a budget a fixed user-defined constant from a scan as our budget. However, this budget can adapt itself to keep a more robust cost, above the scan, until the index is fully generated and ready to be completely exploited.

We point out the following as the research steps that we will follow in the next coming years:

1. Adapt other sorting algorithms to work progressively and analyze their advantages and disadvantages defining a cost-model with a decision tree to unify all solutions;

2. Explore how progressive indexing can be leveraged in more complex database operations, such as joins and aggregations.

Acknowledgments. This work was funded by the Netherlands Organisation for Scientific Research (NWO), project "Data Mining on High-Volume Simulation Output" (Holanda).

6. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*, 2000.

[2] N. Bruno. *Automated Physical Database Design and Tunning*. CRC-Press, 2011.

[3] S. Chaudhuri and V. R. Narasayya. An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*, volume 97, 1997.

[4] D. Comer. The Difficulty of Optimum Index Selection. *TODS*, 3(4):440–445, 1978.

[5] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, pages 371–381. ACM, 2010.

[6] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index Selection for OLAP. In *Data Engineering*, 1997.

[7] F. Halim, S. Idreos, P. Karras, and R. H. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *VLDB*, 5(6):502–513, 2012.

[8] P. Holanda and E. C. de Almeida. SPST-Index: A Self-Pruning Splay Tree Index for Caching Database Cracking. In *EDBT*, pages 458–461, 2017.

[9] S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. In *SIGMOD*, 2007.

[10] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing Tuple Reconstruction in Column-stores. *SIGMOD*, pages 297–308, 2009.

[11] S. Idreos, M. L. Kersten, S. Manegold, et al. Database Cracking. In *CIDR*, volume 3, pages 1–8, 2007.

[12] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *VLDB*, 4(9):586–597, 2011.

[13] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *ICDE*, 2013.

[14] F. M. Schuhknecht, J. Dittrich, and L. Linden. Adaptive adaptive indexing. *ICDE*, 2018.

[15] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The Uncracked Pieces in Database Cracking. *Proc. VLDB Endow.*, 7(2):97–108, Oct. 2013.

[16] G. Valentin, M. Zuliani, D. C. Zilio, G. Lohman, and A. Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *Data Engineering*, 2000.