CHAPTER 6

Big Picture

This chapter discusses the main challenges of implementing Progressive Indexing in a database system and points out future work for the general area of incremental indexes. We also dive in specifically on unidimensional Progressive Indexing, multidimensional Progressive Indexing, and Progressive Merges.

1 The Elephant In The Room

The fact of the matter is, no database system took into production Adaptive Indexing, even though the first paper of Adaptive Indexing, *Cracking the Database Store*[38] dates from 2005. Some of the reasons, like unpredictable query response times, high penalty over initial queries, and lack of full index convergence, have been mitigated with the Progressive Indexing approach. However, other issues permeate adaptive and progressive indexes that make it unlikely for them to be picked up by a production-ready database system.

Tuple Reconstruction In most of the Adaptive/Progressive Indexing experiments, the columns must be grouped in advance when constructing the index structure, which leads to a lack of usability of the index. At the same time, real-life queries tend to filter and project over different groups of columns. For the unidimensional adaptive/progressive index, this problem is obvious. Only one column is indexed when selecting any other column. We must perform tuple reconstruction, which hides any potential benefit from having the data skipping from the index, except for point queries and high selective queries that would not be classified as Analytical Processing. Multidimensional Adaptive/Progressive Indexing presents the same problem since it only groups the data if they have filters. Hence selections on columns that are not being filtered would have to perform tuple reconstruction. One way of mitigating the tuple reconstruction would be to create the index by not only copying the filtered columns but the whole table. Of course, this presents problems in itself since it will cause a storage blow-up (i.e., now every index must own a copy of the full table) and increase the updates' costs.

Overhead of Storage/Maintenance Every query with a filter will produce either a unidimensional or a multidimensional Adaptive/Progressive Indexing, depending on the number of filters the query has. In the worst case, at some point, every column will have a unidimensional index created, and one multidimensional index will be created for every unique combination of multidimensional filters. This, of course, will cause a storage blow-up and a maintenance overhead that will make it impossible to use these techniques on a real exploratory dataset.

Is there hope? We believe that the next step to the grand area of Adaptive/Progressive Indexes is to move from secondary index creation to Adaptive/Progressive Table Partitioning. The basic idea is to perform the partitioning used to create indexes and reorganize the table's data instead of creating a secondary index structure. This would increase the usability of the data reorganization since the multidimensional indexes will suffer from tuple reconstruction costs when accessing non-indexed tuples.

2 Future Work

In this section, we will present potential future research directions in the area of Progressive Indexes. We split up this section by Progressive Indexes and Progressive Merges.

2.1 Progressive Indexes

We point out the following as the main aspects to be explored in Progressive Indexes future work:

- **Approximate Query Processing.** One could also resort to using approximate query processing techniques [12] to allow for a faster convergence (i.e., by spending less time scanning data for the query, we can invest more time indexing data). We can then build a progressive index as a by-product of the approximate query processing, leading to better accuracy and faster responses as the data is queried more often.

- **Indexing Methods.** Other techniques can be adapted to work progressively with different benefits. For example, instead of constructing the complete hash table, we only insert $n * \delta$ elements and scan the column's remainder. The partial hash table can be used to answer point queries on the indexed part of the data. Another example is column imprints [54] where instead of immediately building imprints for the entire column, only build them for the first fraction δ of the data.

- **Interleaving Progressive Strategies.** As depicted in our decision tree, different progressive strategies can be more efficient in different scenarios. When the indexing budget is small, the indexes can take longer to converge fully. This longer period increases the chances of sudden changes in the workload patterns before the index is fully built. Detecting these changes and changing the progressive strategy on the fly can be beneficial for these cases.

- **Indexing Structures.** Different data structures can be used to exploit modern hardware and boost access to more selective queries. In chapter 3, we choose to progressively build a B+-Tree in our consolidation phase. However, other structures like the ART-tree [40] can also be built progressively, with more careful considerations on their creation costs and query performance.

- **Complex Database Operations.** Much like regular indexes, progressive indexes could also be used for other database operations such as joins and aggregations.

2.2 Progressive Merges

In chapter 5 we introduce a novel algorithm for merging appends into progressive indexes. The work has still several engineering and research steps that must be taken as future work:

- **Integrating Merge Ripple With Progressive Indexing.** In our experiments, we compare against adaptive indexing using the merge gradual/complete/ripple algorithms. However, this comparison would be even more significant if these algorithms were implemented directly into Progressive Indexing. For example, if the main index algorithm is Progressive Quicksort, by using an AVL-Tree, similar merge algorithms could be used.

- **Refinement Method.** In chapter 5, we only use Progressive Quicksort as our refinement strategy within Progressive Mergesort. However, in the Progressive Indexing work, it is demonstrated that different Progressive Indexing algorithms can present better performance depending on the data distribution and workload. With mergesort, we can select a different algorithm for each chunk in the refinement step. Deciding which algorithm to use could drastically improve performance.

- **Merge Strategy.** Deciding when to merge and which arrays to merge can be beneficial to the cumulative cost of the workload since there is a trade-off on the random access versus merging costs (i.e., keeping many smaller arrays or frequently merging them in order to maintain only a small number of bigger arrays). An algorithm that takes that this trade-off into consideration is left as future work.

- **Greedy Progressive Mergesort.** Our current implementation of Progressive Mergesort relies on a fixed δ for the entire workload. The development of a cost-model to the merge phase will integrate it with greedy Progressive Indexing algorithms. Hence, as future work, a greedy version of our Progressive Mergesort can bring even fewer performance spikes to our algorithm.

- **Handling Updates.** We describe how to efficiently merge appends since these are the most common types of updates in interactive data analysis. However, although deletes and updates are not frequent, they might still occur. Therefore Progressive Mergesort must be capable of properly handling them.

- **Multidimensional Updates.** Until now, we only focused on unidimensional Progressive Indexing. However, multidimensional Progressive Indexing [43] was recently proposed to efficiently index columns for queries with multiple selective filters. In this algorithm, a KD-Tree is used to store and navigate the partitions created by Progressive Indexing. To support updates on this structure, Progressive Mergesort must be extended to consider the KD-Tree nodes to merge multiple batches of updates correctly.

- **Real Benchmarks.** The Sloan Digital Sky Survey [1] is an open-source project

[1] `https://www.sdss.org/`

that maps the universe with an open data set and interactive-exploratory query logs. Capturing the updates on this database can represent real patterns of updates on interactive data.