Progressive Mergesort: Merging Batches of Appends into Progressive Indexes

Pedro Holanda
CWI, Amsterdam
holanda@cwi.nl

Stefan Manegold
CWI, Amsterdam
manegold@cwi.nl

ABSTRACT

Interactive exploratory data analysis consists of workloads that are composed of filter-aggregate queries with highly selective filters [1]. Hence, their performance is dependent on how much data they can skip during their scans, with indexes being the most efficient technique for aggressive data-skipping. Progressive Indexes are the state-of-the-art on automatic index creation for interactive exploratory data analysis. These indexes are partially constructed during query execution, eventually refining to a full index. However, progressive indexes have been designed for static databases, while in exploratory data analysis updates — usually batch-appends of newly acquired data — are frequent.

In this paper, we propose *Progressive Mergesort*, a novel merging technique to make Progressive Indexes cope with updates. Progressive Mergesort differs from other merging techniques for partial indexes as it incorporates the index budget strategy design from Progressive Indexing. It follows the same three principles as Progressive Indexes: (1) fast query execution, (2) high robustness,(3) guaranteed convergence.

Our experimental evaluation demonstrates that Progressive Mergesort is capable of achieving a 2x speedup when merging updates and up to 3 orders of magnitude lower variance than the state of the art.

1 INTRODUCTION

Data scientists perform interactive exploratory data analysis to discover unexpected patterns in large collections of data. This process is done using hypothesis-driven trial-and-error queries [10]. Given the result of a query, the data scientists refine their original hypothesis and either zoom in on the same data segment or move to a different one depending on the insights gained.

In the typical interactive exploratory data analysis workload, the data scientist inspects a massive amount of data by issuing selective analytical queries (usually via a visualization tool) to test their hypothesis. Battle et al. [1] depict that the most demanding type of interactive queries are cross filter applications (i.e., grouping data after applying selective filters). In these workloads, users expect almost immediate responses from the system, and each movement on the visualization tool will immediately submit another query to the database system.

Figure 1 depicts an example of a cross filter application. Here the data scientist uses a dataset that contains multiple attributes of flight information. The user visualizes each attribute as one histogram figure (e.g., departure time or airtime in minutes). The range slider on the top of the figure allows the users to change the filter used to construct these histograms, and the graphs are automatically updated depending on the new filter input.

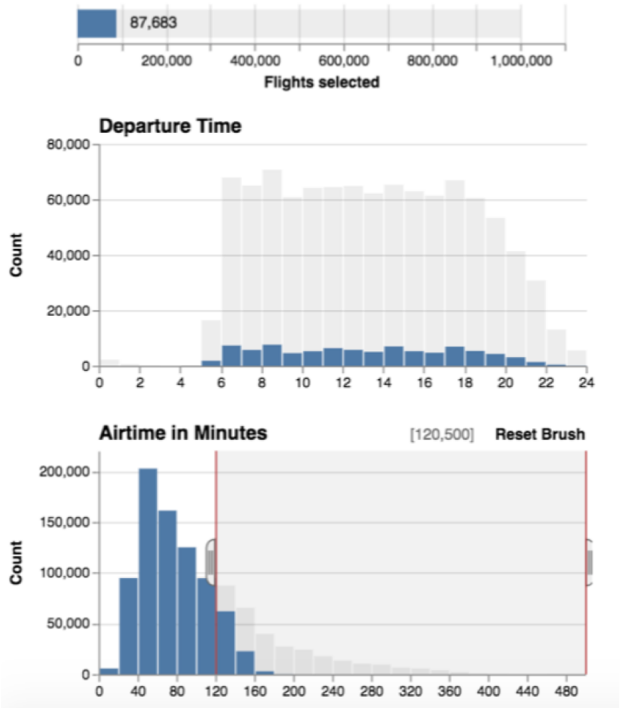


Figure 1: Interactive Data Analysis Example [1]

Since these workloads are dependent on a filter, when these filters are selective (e.g., wanting to know the information of a small number of flights), aggressive data skipping techniques can significantly improve the query performance.

Index structures are frequently used to boost workloads that depend on data skipping. There are two main strategies that automatically create indexes for interactive data analysis.

(1) Adaptive Indexing [6, 9] automatically creates indexes based on query predicates of range queries. They perform quicksort iterations with query predicates as pivots, indexing the accessed pieces during query execution, efficiently smearing out the index creation cost over a workload.

Adaptive Indexing follows a philosophy of only indexing the minimum amount of data necessary to the currently executing query. Although this strategy allows for fast convergence on skewed workloads (i.e., workloads where the same piece is frequently accessed), it has no control over the amount of indexing that one query can perform. When accessing pieces with different levels of refinement, query execution time spikes, resulting in a highly unpredictable query cost, which is undesirable for interactive data analysis since the user expects the query to be executed within a time limit.

(2) Progressive Indexes [3, 5, 8] are designed to be highly robust, have a predictable convergence, and present a low total cost during the entire workload execution. Their main difference

from Adaptive Indexing is the introduction of an indexing budget δ. With this indexing budget, the data scientist sets a value for δ, and the index invests a fixed amount of time into per-query index creation, being the state-of-the-art algorithm developed for interactive exploratory data analysis.

The major drawback of Progressive Indexes is that they are only designed for static-databases. However, in the interactive data analysis scenario, the data is not static but rather frequently updated with batches of data that must be appended. As an example, in our flight dataset we can consider the scenario where batches of data are regularly appended since new flights happen all the time (e.g., either data is appended every few minutes, hours, days, depending on how critical is to analyze recent data).

One way of adapting the current Progressive Indexing strategy to support updates is to use the techniques developed for merging updates on Adaptive Indexes since they produce similar partial-indexes up-to full index convergence. However, these merging techniques follow Adaptive Indexing's philosophy of lazy query execution, drastically decreasing robustness (i.e., it creates performance spikes that vary the per-query response time in orders of magnitudes up and down), with no guaranteed convergence and high penalties for larger batches of appends.

In this paper, we introduce *Progressive Mergesort*. Progressive Mergesort is designed to efficiently merge batches of appends while following the core design decisions of progressive indexing. It presents a low-query impact even for large batches, high robustness, and guaranteed convergence (i.e., all elements are merged into one array).

2 RELATED WORK

In this section, we will cover how Progressive Indexing, in particular progressive quicksort, works and will present the Adaptive Merges algorithms that merge updates into Adaptive Indexing.

2.1 Progressive Indexing

Progressive Indexes are inspired by Adaptive Indexes. Both techniques perform index creation during query execution, aiming to smear out the index creation cost over the workload. Consequently, the indexes produced by both techniques have a similar format (i.e., both are partial indexes), except that once fully converged, progressive indexing turns into a standard B-tree. One major difference between Progressive Indexes and Adaptive Indexes is that Progressive Indexes use an index budget constraint δ that indicates the amount of data that can be indexed, in one sorting iteration, per query, while Adaptive Indexes only performs full sorting iterations (e.g., adaptive indexing will fully partition one column around a pivot in one query, while progressive indexing will partition a δ fraction of the column.).

Progressive Indexes come in two flavors, the fixed-δ where the user picks a fraction of the data, and the same fraction is indexed per query, and a greedy version, where the user sets a desired query execution time. The greedy algorithm uses a cost-model to select a suitable δ tailored for each query automatically. In this paper, we will focus on the fixed-δ version of Progressive Indexing and will leave the greedy version as future work.

Figure 2 depicts an example of Progressive Quicksort with $\delta = 0.5$ (i.e., half the data is pivoted in each query), a progressive indexing technique inspired by the quicksort algorithm. Progressive quicksort is triggered when a filter is executed on a column. In its first phase (*Initialize*), an uninitialized array is allocated with the same space as the original column. A pivot is then selected, and the data is copied to either the bottom or the top of the new array considering the pivot. The subsequent step (*Initialize 2*) can already take advantage of this information and only scan the necessary parts (either top, bottom, neither, or both) relevant to the query. It also continues the copy process until our progressive indexing array is completely populated. At the end of the *initialize phase*, the column is partitioned into two pieces. In the example ≤ 10 and > 10. We now start the *refinement phase* where pivots are selected for each piece, and they are ordered in-place. When pieces are sufficiently small, we fully sort them. This phase results in a completely sorted array. When reaching a completely sorted array, the *consolidation phase* starts building a bottom-up B-Tree on top of the array.

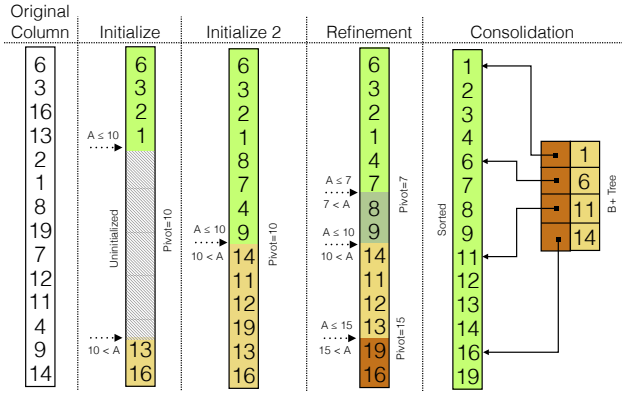


Figure 2: Progressive Quicksort [5]

2.2 Adaptive Merges

There are three main algorithms designed to efficiently merge appends into adaptive indexes [7], the *Merge Complete*, *Merge Gradual*, and *Merge Ripple*, and we will refer to these algorithms as *Adaptive Merges* from now on. They follow the same philosophy of Adaptive Indexing by only merging appends when necessary. They differ from each other in terms of what data they will merge and how they merge it. In the following subsections, we overview each algorithm and present an example of their execution. Besides the strategies to efficiently merge appends into the index's column, Holanda et al. [4] present a strategy to prune cold data from the cracker index to boost updates. However, we do not explore this strategy in this paper since it directly goes against our full convergence philosophy.

Merge Complete (MC) This algorithm completely merges the full *Appends* vector into the *Cracker Column* (i.e., the cracker column is a full copy of the original column owned by the adaptive indexing structure) as soon as a query requests data that is also present in the *Appends* vector.

Merge Gradual (MG) Merge Gradual differs from Merge Complete with respect to the amount of data merged per query. It only merges items that qualify for the currently executing query.

Merge Ripple (MR) Like Merge Complete, the Merge Ripple algorithm only merges the elements that qualify for the query predicates. They differ on how they merge them. In the Merge Ripple, instead of resizing the Cracker Column and appending the element to its end as its first step, it starts by swapping the to-be inserted element with the first element in the next greater-neighboring piece from its correct piece.

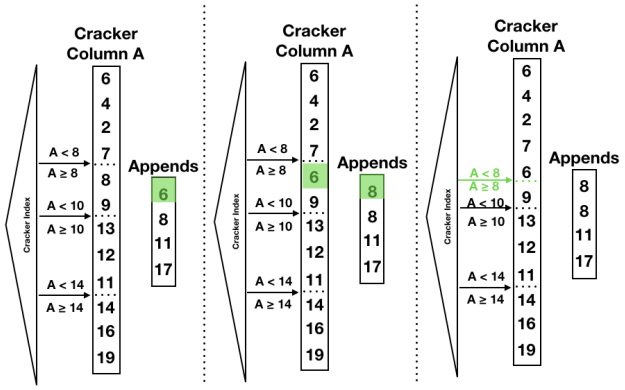


Figure 3: Merge Ripple on query A < 8

Figure 3 depicts an example of Merge Ripple executing the query $A < 8$. In our example, the column is already partitioned around three pivot points (8, 10, 14) and the appends array contains four values (6, 8, 11, 17). Since we only need to insert element 6 from the appends array, we perform a cracker index lookup and identify the element's piece (i.e., the first piece holding 6, 4, 2, and 7). We then go to the successor piece (i.e., piece 2 with elements 8 and 9) and swap the first element of that piece (8) with the element in our appends (6). After that, we only need to update the cracker index node that points to the value 8. In this case, we only had to perform 1 swap and update 1 node in the cracker index to merge 25% of our appends. Merge Ripple performs fewer swaps and updates than the previous algorithms while merging the necessary amount of data to our index.

3 PROGRESSIVE MERGESORT

Progressive Mergesort is a progressive indexing technique inspired by the mergesort algorithm [2] and used for merging appends into the main progressive indexing structure. It follows the three pillars of progressive indexes: (1) low impact on query execution, (2) robust performance, and (3) guaranteed convergence. It relies on an index-budget δ that represents the percentage of the data that is indexed per-query, guaranteeing that the same amount of effort will be distributed for the entire workload.

In practice, during query execution, the δ defined for our Progressive Indexing algorithm is used for both the main index structure and progressive mergesort.

Progressive mergesort follows two distinct canonical phases, the refinement phase, and the merge phase, which are described in this section.

Refinement. In the refinement phase, we can use any of the other proposed progressive indexing algorithms, getting the most performance depending on data distribution and workload. Our budget is used as described in the original Progressive Indexing paper [5] depending on the algorithm executing the refinement. In this paper, we decided to experiment with Progressive Quicksort as our algorithm of choice. Utilizing the other algorithms is left as an engineering exercise for future work.

Merge. At the end of the refinement phase of any progressive indexing algorithm, the result is a sorted list. When all merge chunks are fully sorted, we progressively merge them into one sorted chunk. We perform a progressive two-way-merge in order to merge said chunks.

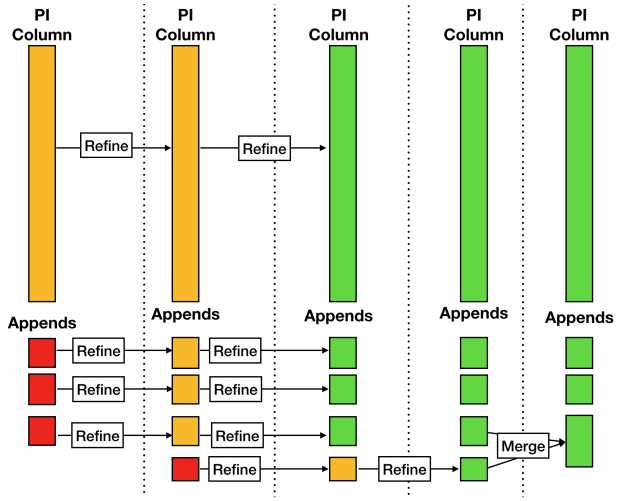


Figure 4: Progressive Mergesort

Figure 4 depicts a high-level concept of Progressive Mergesort. In this figure, red vectors are completely unsorted vectors, yellow are partially sorted vectors, and green are completely sorted. We start with our main index structure only partially sorted and with a new batch of appends.

It starts with the refinement phase. At this step, any Progressive Indexing technique can be used and will continue their execution until reaching completely sorted lists. When all chunks are entirely sorted, the second phase of Progressive Mergesort starts. Here, the *Appends* arrays are progressively merged into one array. One might note that new batches can be introduced while other batches are already being refined. In this case, a Progressive Mergesort run will be initiated to newly appended chunks. All these chunks use the same δ as our main progressive index but normalized to the chunk size. Only when the original Progressive Indexing column and the appends are fully sorted (i.e., we have one sorted column for the Progressive Indexing and one sorted column for all the appends) and the appends have the same or bigger size as the Progressive Indexing column we merge them.

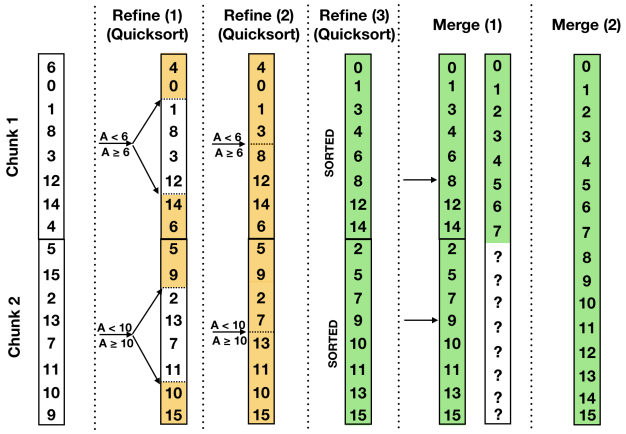


Figure 5: Progressive Mergesort Example ($\delta = 0.5$)

Figure 5 depicts an example of Progressive Mergesort with *delta* = 0.5. We start with two batches of updates. In the initial iterations, we execute Progressive Quicksort as the refinement phase. In *Refine (1)*, a Progressive Quicksort iteration is initiated

for each chunk, since $\delta = 0.5$ both iterations index half of each chunk around one pivot. In *Refine (3)* both Progressive Quicksort iterations ended, and both chunks are fully sorted. Hence we will start the merge phase of Progressive Mergesort in the following query. In *Merge (1)* we start to merge both lists using a two-way merge algorithm, and we stop when the resulting list is half complete due to our δ. For the chunks that are being merged, we must store the offsets where we stopped merging. Finally, in *Merge (2)* we end the merge phase with one completely sorted append list and delete the previous chunks.

Query Processing. When executing a query on a column with progressive indexing, we might encounter several arrays (i.e., the original Progressive Indexing column and batches of appends that started to be refined but are not yet merged) with different levels of refinement.

During the query execution, each array must be checked to return the elements that fit the query predicates. If the array is already fully sorted, a binary search will be executed to return the result. Otherwise, the array will be at some step of the refinement phase. Hence a lookup on the binary tree is necessary to return the offsets that match the query predicates.

When to Merge. In this paper, we decided to first completely merge all appends into one, fully sorted, append array. If this array has a size equal to or bigger than the current Progressive Indexing column, we merge both. This decision was made to avoid frequent resizes of large arrays (e.g., if we merged the Progressive Indexing column with every append first, this would result in a resize for the progressive column at every batch, which would be prohibitively expensive).

However, this decision is not necessarily optimal for all workloads. Having multiple arrays increase the random access to respond to the workload while diminishing the merge costs creating a trade-off depending on when and how these merges are performed. Creating an algorithm that decides when is the appropriate moment to merge these different arrays and which arrays should be merged is out of this paper's scope, and we leave it as future work.

4 EXPERIMENTAL ANALYSIS

This section provides an experimental evaluation of Progressive Mergesort and compares it with the Adaptive Merges techniques.

4.1 Setup

We implemented the Progressive Mergesort algorithm and the Adaptive Merges in a stand-alone program written in C++. The Progressive Mergesort uses Progressive Quicksort in its refinement phase.

Compilation. This application was compiled with GNU g++ version 7.2.1 using optimization level -O3.

Machine. All experiments were conducted on a machine equipped with 256 GB main memory and an 8-core Intel Xeon E5-2650 v2 CPU @ 2.6 GHz with 20480 KB L3 cache.

Appends. All experiments have 3 parameters regarding the appends, (1) the *batch_size* that represents the size of a batch of appends, (2) the *frequency* which represents an interval of queries where a new batch of appends is executed, and (3) *start_after* that describes how many queries need to be executed before the first append happens. With these 3 parameters we calculate the number of appends that will be executed $total\_appends = \frac{total\_queries - start\_after}{frequency} * batch\_size$, and divide our data set into the *original_column* set that represents our initially loaded column and the *appends* set that represent the appends that will be inserted.

Data set. We generate a synthetic data set composed of $N + total\_appends$ unique 8-byte integers, with $N \in \{10^7, 10^8, 10^9\}$ and representing the original column size. After generating the data set, we shuffle it following a uniform-random distribution and divide it into our original column and a list of appends.

Workload. Unless stated otherwise, all experiments consist of a synthetic workload with 10^4 queries in the form `SELECT SUM(R.A) FROM R WHERE R.A BETWEEN` V_1 `AND` V_2. A random value is selected for V_1 and $V_2 = V_1 + (N + total\_appends) * 1\%$.

Configuration. We experiment with 3 main configurations.

- High Frequency Low Volume (HFLV): A batch of appends with $batch\_size = 0.001\% * N$ executed every 10 queries.
- Medium Frequency Medium Volume (MFMV): A batch of appends with $batch\_size = 0.01\% * N$ executed every 100 queries.
- Low Frequency High Volume (LFHV): A batch of appends with $batch\_size = 0.1\% * N$ executed every 1000 queries.

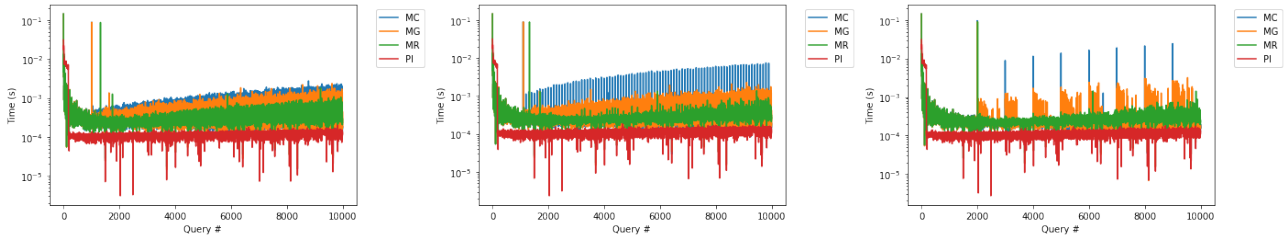
4.2 Performance Comparison

In this paper, we decided to use the Adaptive Merges algorithms only with Adaptive Indexing due to the increased complexity of implementing them to work with Progressive Indexing and leave this task as an engineering exercise for future work. Since the base indexing algorithm is different for the Adaptive Merges and Progressive Mergesort, we decided to start appending data after 1000 queries to have refined indexes and better isolate the actual append cost from early index creation. Hence we avoid the noise of partitioning the *original_column* and focus on the actual merges from the appends. Our Progressive Mergesort uses a fixed δ of 0.1 in all experiments. It is also important to notice that our Progressive Indexing implementation stops its convergence when it becomes a fully sorted list. We leave merging appends into the concise B+-Tree format as future work.

Figure 6 depicts a per-query performance comparison of Progressive Mergesort and Adaptive Merges. In this experiment, we use a data set with $N = 10^7$ and run all 3 configurations described in the previous section. We continue this section by describing two observations present in all experiments, (1) regarding the column resizes and (2) an overall analysis of query robustness.

Resizes. In all three configurations, HFLV, MFMV, and LFHV, we can notice that all three Adaptive Merges present a performance spike right after the start of the updates around query 1000. The main reason for this spike is the need to resize the *Cracker Column* when appending new data. Since this resize reserves 2 times the space of the original *Cracker Column*, it only happens once. It is also possible to notice that with Merge Ripple, the spike occurs 100 queries later than with Merge Complete and Merge Gradual. This is because Merge Ripple avoids resizing the *Cracker Column* by swapping the data from the *Appends* and the Column with the actual resize only happening when we are in the last piece. This problem does not exist with Progressive Mergesort since we perform a *vector.reserve()* to allocate memory to the merge vector, and filling out the merge vector is completed over multiple queries.

Robustness. The Merge Complete presents the lowest robustness from all algorithms. Whenever a merge happens, it has a big spike upwards since it completely merges it. Merge Gradual is the second-worst. Since it completely merges all elements that qualify the predicate, it does not have one big performance spike,



(a) HFLV (b) MFMV (c) LFHV

Figure 6: Progressive Mergesort and Adaptive Merges ($N = 10^7$ and $start\_after = 1000$)

spreading those merges through many queries. This is particularly visible in Figure 6c that depicts the low-frequency high volume experiment (i.e., at every 1000 queries, a batch of size 10^4 is inserted. One can see that at every 1000 queries, there is an upwards spike that slowly decreases for 500 queries, and then has a slop down since most of the *Appends* array was merged by that point. From the Adaptive Merges, the Merge Ripple presents the least variance. All queries slightly increase their cost with increasing updates. Finally, the Progressive Mergesort presents the lowest variance, with no performance spikes up.

One can notice that all algorithms present downwards spikes at the same queries overall 3 configurations. These are caused by noise due to the way we select our query predicates to fix our workload selectivity. Since we create our second query predicate as $V_2 = V_1 + (N + total\_appends) * 1\%$. Queries might not have exactly 1% selectivity if the data is not completely merged in the column. Since the figures are with the y-axis in log scale, small differences in the selectivity produce these downwards performance spikes.

4.3 Varying Data Sizes

	Workload	MC	MG	MR	PM
10^7	HFLV	2.72	3.52	2.57	1.07
	MFMV	2.18	3.39	2.45	1.07
	LFHV	2.00	2.55	2.34	1.06
10^8	HFLV	22.76	26.16	26.61	10.64
	MFMV	20.25	26.14	25.19	10.72
	LFHV	22.14	22.42	23.89	10.63
10^9	HFLV	209.25	221.67	295.39	104.77
	MFMV	206.39	219.39	267.94	104.96
	LFHV	197.89	200.62	250.62	103.95

Table 1: Cumulative Time (s)

Table 1 depicts the total execution cost for the workload, excluding the initial 1000 queries. On all experiments, Progressive Mergesort presented approximately 2x better performance than the best performing Adaptive Merge algorithm. The main reason for this performance difference is that all Merge Adaptive algorithms must keep the appends sorted to merge them efficiently. This problem impacts Merge Ripple the most since it tends to keep a larger appends array due to its lazier merging property. That means that a larger array must be re-sorted at every append insertion. One might notice that the results of Adaptive Merges seem to directly contradict Idreos et al. [7], where Merge Ripple was the best performing algorithm of the three. The HFLV with $N = 10^7$ is the only experiment with the same parameters as the original paper and showcases a similar result, with Merge Ripple being the fastest of the Adaptive Merges. However, as discussed before, with larger appends Merge Ripple starts to lose its benefit of fewer swaps to keep the append vector sorted.

One other interesting result is the variance in the total cost depending on the configuration of the workload. The Adaptive Merges algorithms present a much higher variance than Progressive Mergesort for the same data size. This is more prominent with larger data sizes. Taking $N = 10^9$ as an example, Merge Complete presents a variance of 11.36s, Merge Gradual of 21.05s, Merge Ripple of 44.72s, and Progressive Mergesort of 1.01s.

Compared to the Adaptive Merges algorithms, Progressive Mergesort has a very low variance from configurations at the same data size. This is due to the Progressive Mergesort algorithm not performing a complete sort in the append list but rather properly refining and merging it depending on their data size.

	Workload	MC	MG	MR	PM
10^7	HFLV	e-07	e-07	e-07	e-10
	MFMV	e-06	e-07	e-07	e-10
	LFHV	e-06	e-07	e-07	e-10
10^8	HFLV	e-05	e-05	e-05	e-07
	MFMV	e-05	e-05	e-05	e-07
	LFHV	e-04	e-05	e-05	e-07
10^9	HFLV	e-03	e-03	e-03	e-06
	MFMV	e-03	e-03	e-03	e-06
	LFHV	e-02	e-03	e-03	e-06

Table 2: Robustness (Orders of Magnitude)

Table 2 depicts the order of magnitude of the query variance of each workload on all 3 data sizes. We only calculate the query variance after executing the first 1000 queries. Note that the lower the variance the more robust the algorithm is. As expected, Merge Complete presents the lowest robustness since it completely merges the *Appends* array to the *Cracker Column* causing a huge performance spike. The Merge Gradual/Ripple are better than the Merge Complete, since it only merges that tuples that qualify the query predicates. Progressive Mergesort present the highest robustness due to its indexing budget, effectively offering a more fine-grained control over the stream of queries.

4.4 Appends during Index Creation

To perform a fair comparison of the Adaptive Merges and Progressive Mergesort, we only initiated the updates after 1000 queries to minimize the initial index creation cost of Adaptive Indexing and Progressive Indexing. However, after 1000 queries, the progressive indexing is already fully converged (i.e., the main index is a sorted list).

In this experiment, we want to evaluate Progressive Mergesort's impact during Progressive Indexing's creation phase (i.e.,

Initialization and Refinement phases). In our setup, we use a dataset with $N = 10^7$, a workload with 1% selectivity and 200 queries, and three different update setups. All update setups start at the first query and perform appends at every 10 queries, they differ on the size of the batches, with batches of size 100, 1000 and 10000.

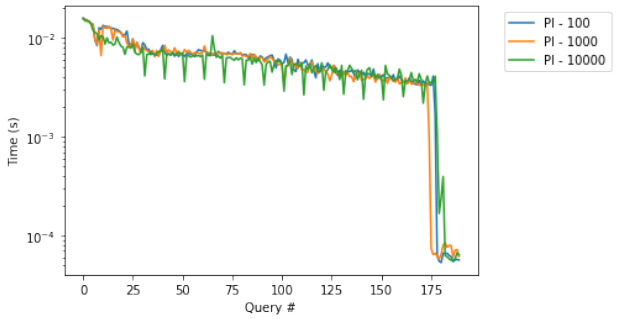


Figure 7: Progressive Mergesort before index convergence.

Figure 7 depicts the per-query cost for the 200 queries. The height of the performance spikes are strongly correlated to the batch sizes, with larger batches introducing a higher spike. This happens due to our strategy using a fixed δ (i.e., a % of the total size of the data that is indexed per-query) for the entire workload. Hence the more data we ingest, the actual per-query cost will increase since the data size increases. One way of minimizing this issue is to extend the cost models proposed in Holanda et al. [5] to automatically generate a value for δ to reduce query variance. We leave that algorithm as an exercise for future work.

5 CONCLUSION & FUTURE WORK

In this paper, we introduce the *Progressive Mergesort*, a novel progressive algorithm used to merge batches of appends. We compare it to the state-of-the-art merging algorithms from adaptive indexing techniques and show how they perform under multiple synthetic benchmarks. Our solution is more robust and faster than the state-of-the-art.

We point out the following as the main aspects to be explored in Progressive Mergesort's 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 this paper, 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 also have the opportunity of selecting 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 vs merging costs (i.e., keeping many smaller arrays or frequently merging them in order to maintain only a small number of bigger arrays). Designing 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 allow it to also be integrated with progressive indexing algorithms that use an interactivity threshold and automatically adapt the δ value to boost robustness. Hence, as future work, a greedy version of our progressive mergesort can bring even fewer performance spikes to our algorithm.
- **Handling Updates.** In this paper, 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 [8] 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 that maps the universe with an open data set and interactive-exploratory query logs. Capturing the updates on this database can depict a good representation of real patterns of updates on interactive data.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Leilani Battle, Philipp Eichmann, Marco Angelini, Tiziana Catarci, Giuseppe Santucci, Yukun Zheng, Carsten Binnig, Jean-Daniel Fekete, and Dominik Moritz. 2020. Database Benchmarking for Supporting Real-Time Interactive Querying of Large Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1571–1587.
[2] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.
[3] Pedro Holanda. 2018. Progressive Indices: Indexing Without Prejudice.. In *PhD@ VLDB*.
[4] Pedro Holanda and Eduardo Cunha de Almeida. 2017. SPST-Index: A Self-Pruning Splay Tree Index for Caching Database Cracking. In *EDBT*. 458–461.
[5] Pedro Holanda, Mark Raasveldt, Stefan Manegold, and Hannes Mühleisen. 2019. Progressive indexes: indexing for interactive data analysis. *PVLDB* 12, 13 (2019), 2366–2378.
[6] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR*. 68–78.
[7] Stratos Idreos, Martin L Kersten, and Stefan Manegold. 2007. Updating a cracked database. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 413–424.
[8] Matheus Nerone, Pedro Holanda, Eduardo Almeida, and Stefan Manegold. 2021. Multidimensional Adaptive & Progressive Indexes. In *ICDE*.
[9] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. 2013. The Uncracked Pieces in Database Cracking. *PVLDB* 7, 2 (2013), 97–108. https://doi.org/10.14778/2732228.2732229
[10] Thibault Sellam, Emmanuel Müller, and Martin Kersten. 2015. Semi-Automated Exploration of Data Warehouses. In *CIKM*. 1321–1330.

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