# Progressive Indexes

Timbó Holanda, P.T.

CHAPTER 5

# Progressive Merges

## 1 Introduction

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. If we take the flight dataset example presented in Chapter 2 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 intermediate incremental indexes. 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 chapter, we introduce *Progressive Mergesort*. Progressive Mergesort is designed to efficiently merge batches of appends while following Progressive Indexing's core design decisions. It presents a low-query impact even for large batches, high robustness, and guaranteed convergence (i.e., all elements are merged into one array).

### 1.1 Contributions

The main contributions of this chapter are:

- We introduce a novel Progressive Indexing technique that focuses on merging batches of appends into our main Progressive Indexing run.
- We experimentally verify that the Progressive Mergesort provides a more robust, predictable, and faster performance through various batch sizes and update frequencies.
- We provide Open-Source implementations of Progressive Mergesort.[1]

### 1.2 Outline

This chapter is organized as follows. In Section 2, we investigate related research on updating Adaptive Indexes, called Adaptive Merges. In Section 3, we describe our novel Progressive Mergesort technique and discuss its benefits and drawbacks. In Section 4, we perform an experimental evaluation of each of the novel methods we introduce, and we compare it against adaptive merging techniques. Finally, in Section 5 we draw our conclusions.

## 2 Related Work

There are three main algorithms designed to efficiently merge appends into adaptive indexes [34], 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. [29] presents a strategy to prune cold data from the cracker index to boost updates. However, we do not explore this strategy in this work since it directly goes against our full convergence philosophy.

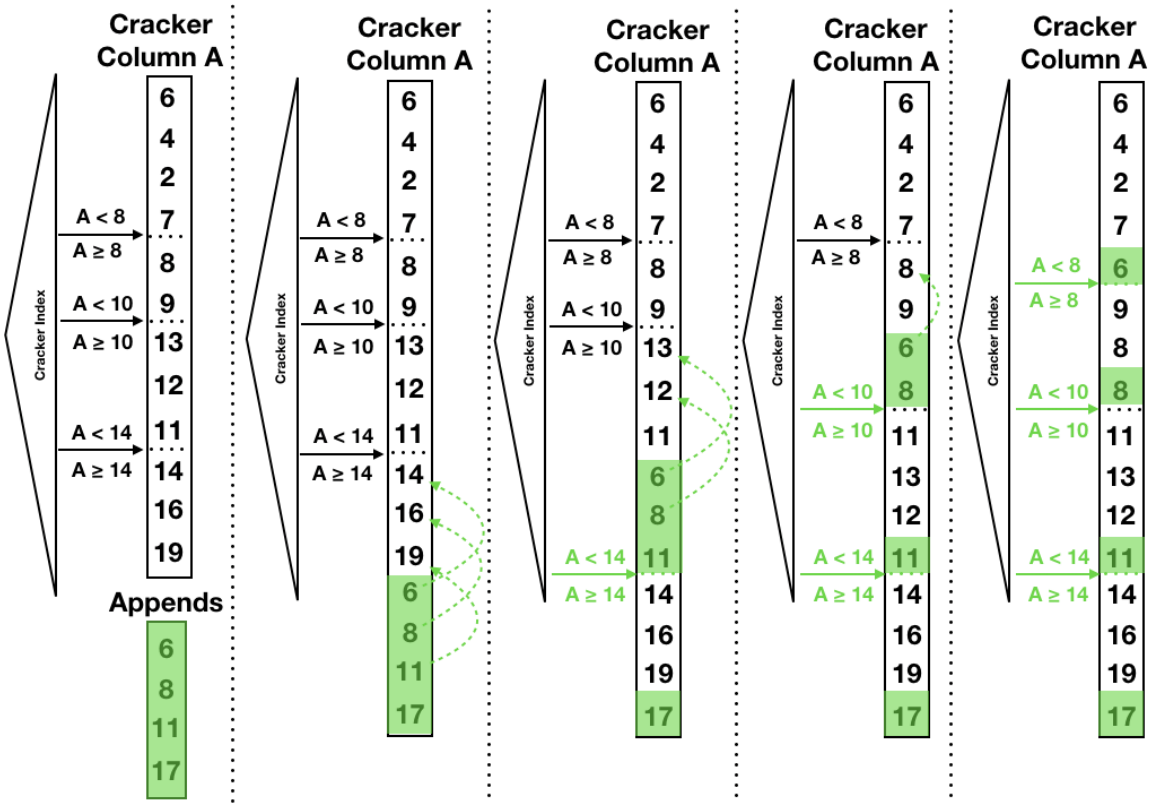



Figure 5-1: Merge Complete on query A ¡ 8

## 2.1 Merge Complete (MC)

This algorithm completely merges the full *Appends* vector into the *Cracker Column* as soon as a query requests data that is also present in the *Appends* vector.

Figure 5-1 depicts an example of merge complete executing the query $A < 8$. In our example, the column is already partitioned around three pivot points 8, 10, and 14. Since the appends vector contains element 6 (i.e., an element that qualifies the query), the whole appends vector is merged. The first step of the merge is to resize our cracker column to *cracker_column.size() + appends.size()*, followed by a copy of the appends elements to the end of the column and the deletion of the appends vector. Then we must swap the newly added elements that are in the wrong piece to their correct piece. In this case, elements 6, 8, and 11 are swapped with elements in the current piece's border with the last piece. After performing the swaps, we update the cracker index pointer for 14 to point at the correct place, considering the newly inserted elements. This process is repeated until all inserted elements are placed in the correct pieces. In our example, we perform 6 swaps, and we update all 3 nodes of

[1]Our implementations and benchmarks are available at `https://github.com/pdet/ProgressiveMergesort`

our cracker index. At the end of the execution, the appends list is empty.

## 2.2 Merge Gradual (MG)

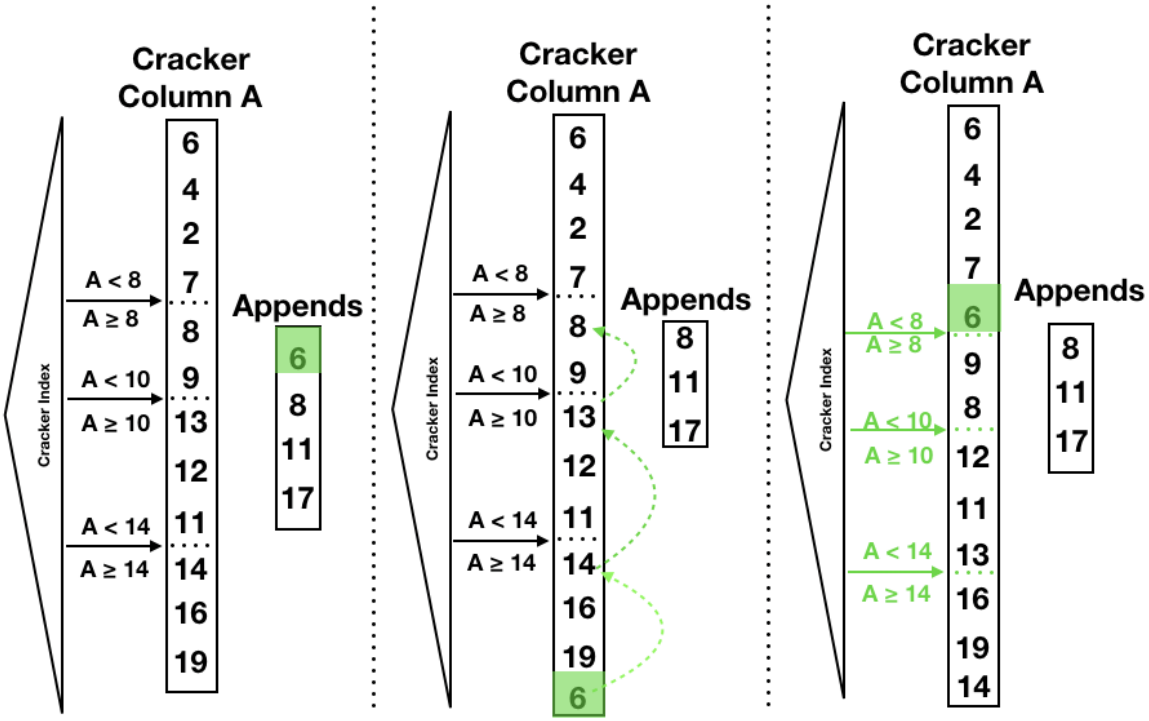



Figure 5-2: Merge Gradual on query A ¡ 8

Merge Gradual differs from Merge Complete concerning the amount of data merged per query. It only merges elements that qualify for the currently executing query.

Figure 5-2 presents the algorithm executing the $A < 8$ query in the same cracker column as before. A binary search using the query predicates is performed in the Appends vector. The elements that qualify for the query, in this case only the value 6, are merged to the cracker index. As before, value 6 is initially placed at the end of the cracker column and erased from the appends vector. Value 6 is then swapped until it reaches its correct piece, with the nodes in the cracker index being updated accordingly. Note that 3 swaps are done in this case, all 3 nodes from the cracker index are updated, and 25% of the values in the appends vector are merged.

## 2.3 Merge Ripple (MR)

**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

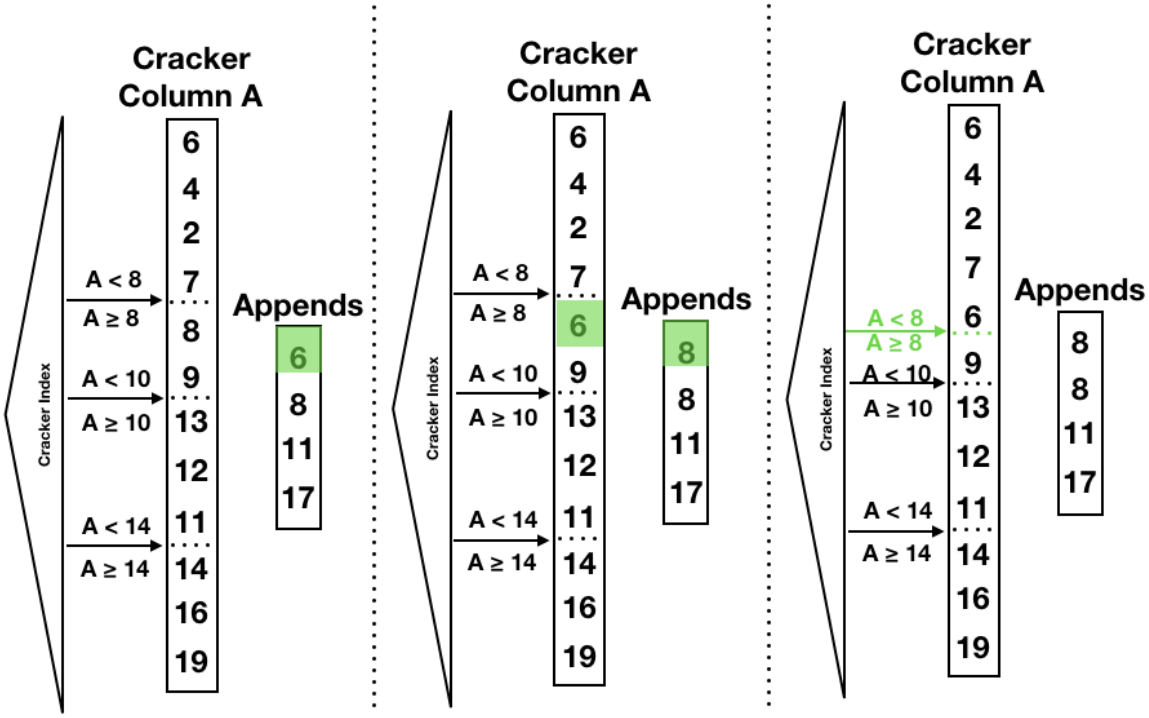



Figure 5-3: Merge Ripple on query A ¡ 8

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 5-3 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. However, our append list remains with the same size it had at the start of the algorithm. Merge Ripple performs fewer swaps and updates than the previous algorithms while merging the necessary amount of data to our index.

**Discussion.** The Merge Complete algorithm presents the highest convergence since it fully merges the appends list whenever the appends vector has elements that qualify for the query. However, it will potentially present high-performance spikes when performing such merges. The Merge Ripple is expected to present lower performance spikes since it only merges what is necessary, avoiding column resizes,

swaps, and index node updates. However, it also presents a slow convergence and can present large performance spikes when the workload shifts to a piece where many elements must be merged. The Merge Gradual seems to be the best balance between robustness and convergence, but robustness issues similar to the Merge Ripple are still expected. Another major problem of these algorithms is the necessity of having a fully sorted appends list to merge the data efficiently. In the original paper, only small batches were used in the experiments. However, when facing large appends, the necessary a-priori sort of the append list will present a major performance bottleneck.

# 3 Progressive Mergesort

Progressive Mergesort is a Progressive Indexing technique inspired by the mergesort algorithm [17] 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 $\delta$ that represents the percentage of the indexed per-query data, guaranteeing that the same amount of effort will be distributed for the entire workload.

In practice, during query execution, the $\delta$ 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 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 chapter 3 depending on the algorithm executing the refinement. In this work, 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 these chunks.

Figure 5-4 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

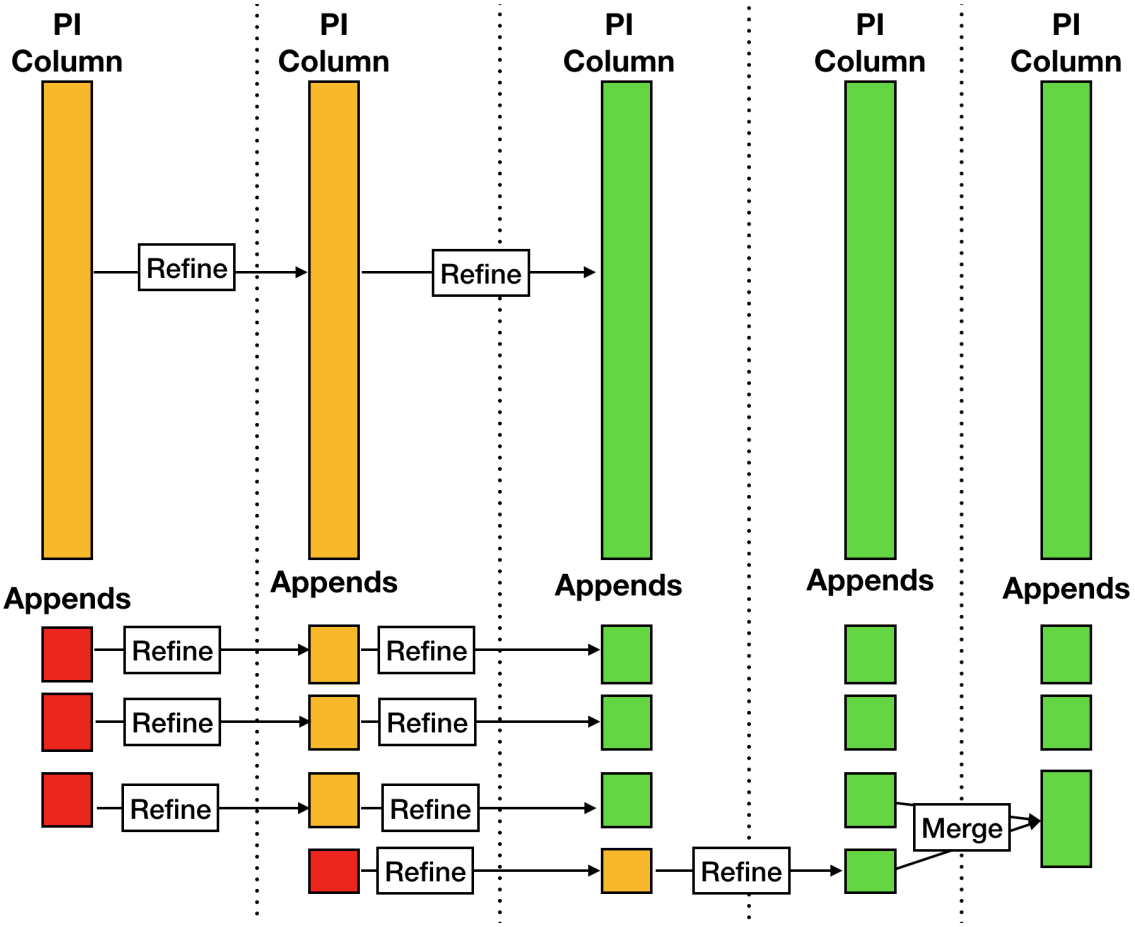



Figure 5-4: Progressive Mergesort

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 $\delta$ 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 5-5 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

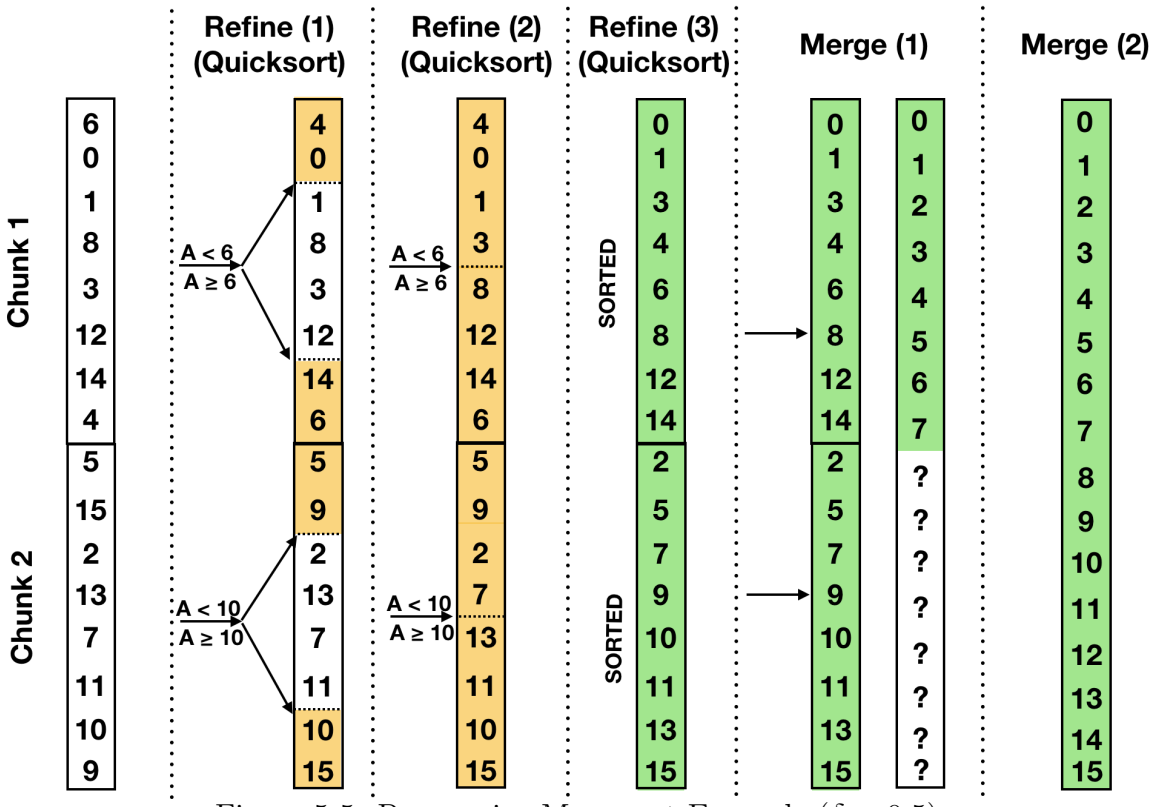



Figure 5-5: Progressive Mergesort Example ($\delta = 0.5$)

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 delta. 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 work, 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 chapter's scope, and we leave it as future work.

Listing 5 depicts a C++ like implementation of Progressive Mergesort. The Progressive Mergesort has as its input a vector of columns representing the chunks that are being refined, a Column representing the current set of updates, a double with the delta, the query predicates, the result structure, a pointer to the merge column, and a parameter indicating the minimum size the update column must have before entering the refinement phase. In the first *for* loop (lines 5-11), we iterate through all chunks and execute the query on each chunk. On line 6, we normalize our delta to the size of the chunk. Line 7 executes a Progressive Quicksort call that refines and returns the filtered elements of that chunk. These elements are then merged into our result structure. While checking each chunk, we also check if they are all sorted since we only start the merge phase after all chunks are already sorted.

In the second *for* loop (lines 12-15), we check if any of the elements in our current update column qualifies for the range query. If so, we add it to the result structure.

In the first *if* (lines 16-20), we initiate a merge of the two last chunks in our vector if no merge is currently happening and all chunks are sorted. The second *if* (lines 21-29) performs the actual merge, we calculate a normalized budget for the size of the *merge_column* and progressively build it. Lines 33-37 check if the merge is already finished. If it is already done, we delete the merged chunks from our chunk vector and add the newly merged chunk to the vector. We also set the pointer to the *merge_column* to null to indicate that we can initiate other merges.

The final *if* (lines 30-34) check if the *updates* column has reached a size bigger than the minimum necessary for it to become a chunk. If so, we initiate a Progressive Quicksort refinement that will be refined in the following queries. We add it to our chunk vector and create a new update column to hold the next appends.

**Listing 5** Progressive Mergesort Body

```
void progressive_mergesort(vector<Column>& chunks, Column& updates,
    double delta,Query query,Result& result,MergeColumn* merge_column,
    size_t min_update_size){
    bool all_sorted = true;
    for (auto& c: chunks){
        auto budget = c.size()*delta;
        result.merge(chunk->execute(query,budget));
        if (!c.sorted){
            all_sorted = false;
        }
    }
    for (auto&u:updates){
        int match = query.match(u);
        result.maybe_push_back(u, match);
    }
    if (!merge_column && all_sorted && chunks.size() > 1){
        auto l = chunks.size() - 2;
        auto r = chunks.size() - 1;
        merge_column = new MergeColumn(chunks[l],chunks[r]);
    }
    if (merge_column){
        auto budget = merge_column.size()*delta;
        merge_column.merge(budget);
            if (merge_column.finished()){
                chunks.erase(chunks.begin()+l,chunks.begin()+r+1);
                chunks.insert(chunks.begin,merge_column);
                merge_column = nullptr;
            }
    }
    if (updates->size > min_update_size){
        auto pq = new ProgressiveQuicksort(updates);
        chunks.push_back(pq);
        updates = new Column();
    }
}
```

# 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 three 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 three 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.

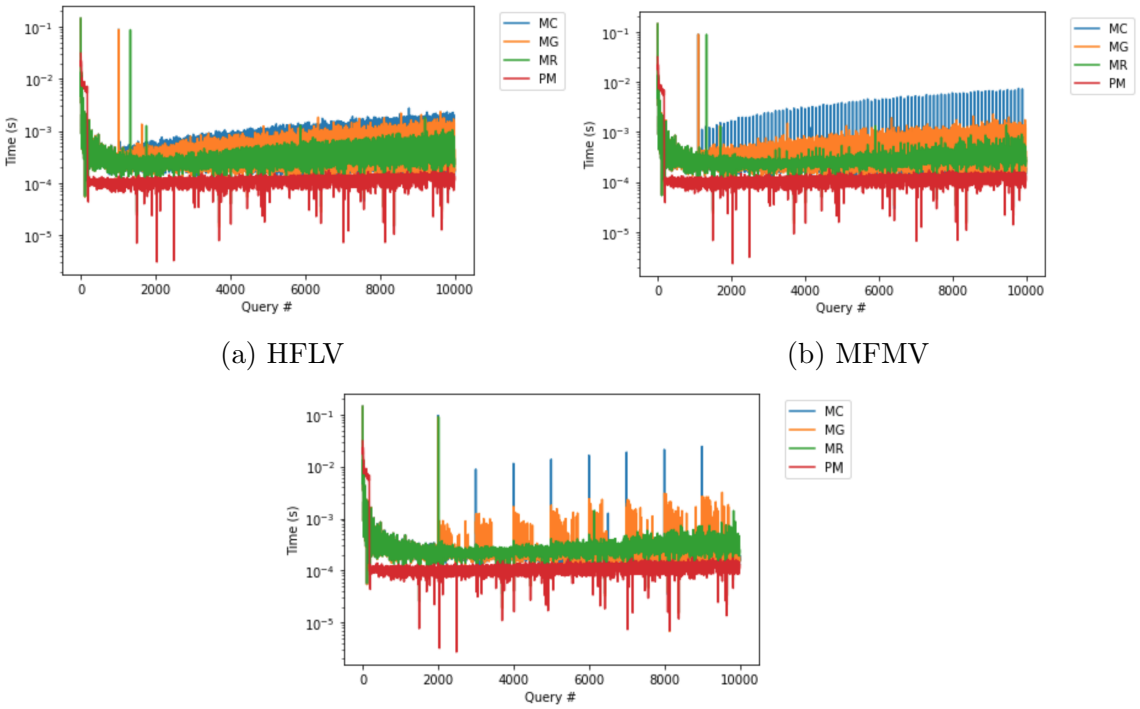
**Workload.** Unless stated otherwise, all experiments consist of a synthetic workload with $10^4$ queries in the form `SELECT SUM(R.A) FROM R WHERE R.A BETWEEN` $V_1$ `AND` $V_2$. A random value is selected for $V_1$ and $V_2 = V_1 + (N + total\_appends) * 1\%$.

**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 work, 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 $\delta$ of 0.1 in all experiments.




(a) HFLV

(b) MFMV

(c) LFHV

Figure 5-6: Progressive Mergesort and Adaptive Merges ($N = 10^7$ and $start\_after = 1000$)

Figure 5-6 depicts a per-query performance comparison of Progressive Mergesort and Adaptive Merges. This experiment uses a data set with $N = 10^7$ and runs all three 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 query robustness analysis.

**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 two 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 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, spreading those merges through many queries. This is particularly visible in Figure 5-6c 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 spikes downwards at the same queries overall three 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

Table 5.1 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

| | 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 5.1: Cumulative Time (s)

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. [34], 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.

Table 5.2 depicts the order of magnitude of each workload's query variance 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 and Merge Ripple are better than the Merge Complete since they only merge tuples that qualify the query predicates. Progressive Mergesort presents the highest robustness due to its indexing budget, effectively offering more fine-grained

| | 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 5.2: Robustness (Orders of Magnitude)

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 ten queries. They differ on the batches' size, with batches of size 100, 1000, and 10000.

Figure 5-7 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 delta (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 chapter 3 to automatically generate a value for $\delta$ to reduce query variance. We leave that algorithm as an exercise for future work.

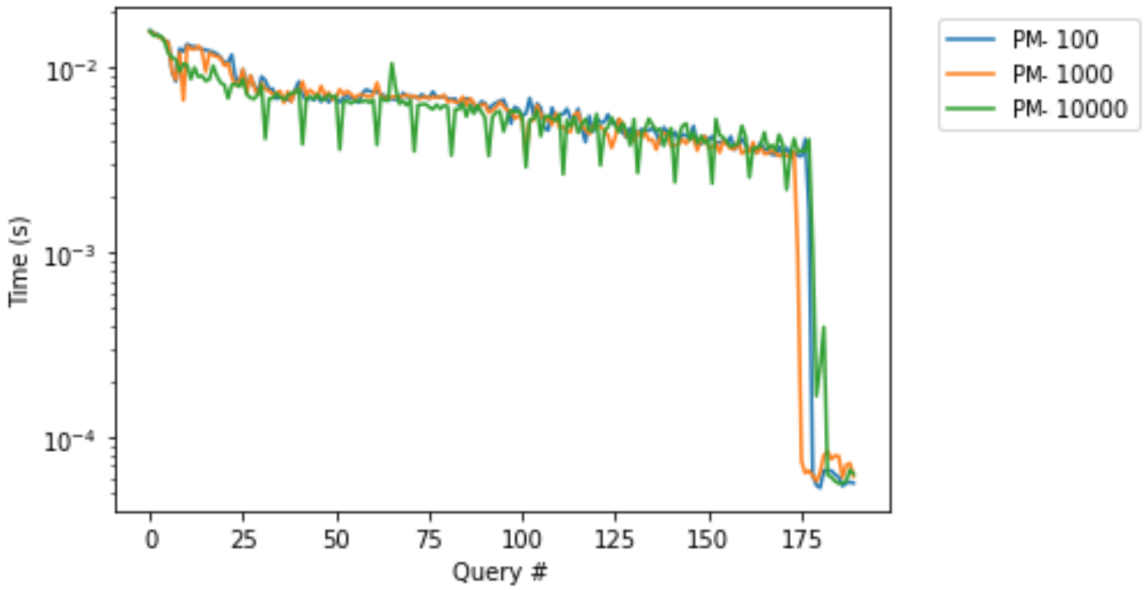



Figure 5-7: Progressive Mergesort before index convergence.

# 5 Summary

This chapter introduces 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.