UNIVERSITY OF CALIFORNIA,
IRVINE

# Optimizing Repartitioning Parallel Sort in AsterixDB

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

by

Mikhail Lychagin

2020

## DEDICATION

To my parents who have supported me and my siblings for being cute.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

## ACKNOWLEDGMENTS

The most honorable mention goes to Prof. Michael Carey for mentoring for his mentorship and expertise. He taught me a lot about database principles, proper experiment setup, and writing. Many thanks.

I would like to thank Sandy Irani, Wail Y. Alkowaileet, and Ali Alsuliman for their collaboration and invaluable input on this project. I could not have done it without them.

I would like to thank Prof. Chen Li for teaching me about database principles in CS222 and being part of my thesis committee.

Finally, a special thanks to my dad for being supportive and proofreading my papers.

# ABSTRACT OF THE THESIS

Optimizing Repartitioning Parallel Sort in AsterixDB

By

Mikhail Lychagin

Master of Science in Computer Science

University of California, Irvine, 2019

Professor Michael J. Carey, Chair

As big data evolves, more and more databases are incorporating a parallel architecture. Sorting a dataset on a key value is a required operation of any database. The hope is that a linear increase in computing power given to a database results in a linear increase in performance. One of these databases is AsterixDB, which has Repartitioning Parallel Sort, or RPS, as its sort operator.

The goal of this thesis is to optimize RPS to fully utilize the parallel nature of AsterixDB in all cases. Currently, the sort operator performs poorly when faced with an input dataset whose sort attribute is skewed on one or more identical values. In this thesis, we first discuss the current state of sorting in AsterixDB and the problems associated with it. Second, we go over a proposed optimization to the sort operator. Third, we compare the old approach with the new one with performance testing. Finally, we discuss some future work that can be done to further improve sorting in AsterixDB.

# Chapter 1

# Introduction

The desire for parallelism has skyrocketed over the last several decades. At first, it started on a small scale, with computers having multiple processors running in parallel and sharing memory. It became more cost-effective to bundle multiple processors (now cores) together than to have one higher-speed processor. The challenges of coding shared memory systems then led architects to build message-based multicomputers (now clusters) to create larger systems. Engineers began writing programs to fully utilize all the processors available to them within a single machine. Then, with the advent of high-speed networks, the internet, and long-range communication, data/computing centers were able to be linked to solve very large computing problems. Database management systems have long incorporated the use of multiple partitions and partitioned parallelism to speed up computation [2]. Partitions are logical units of a database that can be accessed in parallel with each other. These partitions store data and execute operators but each only do a fraction of the storage/computing work needed for the entire database.

Nowadays, a database system typically runs on a cluster of machines to handle large amounts of data. Each machine can house multiple partitions. One database system that incorporates such a structure is AsterixDB [3]. Data in AsterixDB is hash-partitioned across all machines in an attempt to even out the computing and storage load. However, with data being spread across multiple machines, extra steps must be taken to assemble the correct final result for a query. AsterixDB queries are broken down into a series of operators [5] that partitions execute to compute a result. A set of operators can run in either a parallel or serial mode. In parallel mode, each partition concurrently executes an instance of the operator locally, and in serial mode, only a single partition is executing while the others are waiting. The operator of interest in this thesis is the sort operator. AsterixDB currently uses a Repartitioning Parallel Sort [6] as a basis for its sort operator. The specifics of RPS will be discussed later on in the thesis, but for now, know that RPS assigns each partition a range of data values to sort. Each partition sorts the data given to it, and the next operator then picks up the result set based on the ranges each partition was given. A problem arises if a majority of the data has an identical sort key value (or a small set of such values). In such a case, a large amount of data gets sent to one partition instead of being equally distributed amongst all partitions. In this thesis, we attempt to equally load balance all partitions to get the fastest execution time possible when confronted with data skewed on one or more values. We analyze two algorithms that deliver promising results when compared to the original AsterixDB sort implementation.

The rest of the thesis is laid out in the following manner. Chapter 2 goes over related work in the area of parallel sorting with skewed values. Then, in Chapter 3, we go over the current architecture of AsterixDB as well as an outline of RPS, followed by our work to enhance the sort operator. In Chapter 4, we present the results of performance testing with the previous approach and our new ones. Finally, in Chapter 5, we conclude the thesis and briefly discuss possible future work.

# Chapter 2

# Related Work

The idea of range partitioning during a sort has been around for a long time, dating back to the 1990s. The problem addressed in this thesis involves data distribution with a splitting vector. The work below might have different algorithms or mediums, but the idea is the same. For example, [7] outlines an identical algorithm to the one AsterixDB currently uses, except its medium is a multiprocessor. A single node first collects local samples from all other participating nodes and produces a splitting vector, i.e., a plan for how to spread the sort across all nodes. The splitting vector is then sent back to each node and is used to redistribute all the data. However, there is no protection against heavy hitter values or any mention of a protocol to employ when multiple split values are the same. The authors have two different splitting modes, “probabilistic splitting” and “exact splitting.” The “probabilistic” approach relies on sampling a small subset of the data, while the “exact” approach examines all the data before redistribution. As we will see here later, there is an inherent flaw in using a splitting vector, as it does not address the distribution of duplicate values at each of the split points. In [9], the authors follow a similar sorting approach, but they provide a custom splitting vector to the sort operator. (There is a similar feature in AsterixDB today, which allows users to give the sort operator a custom splitting vector.) In this case, there is no need for a sample aggregation step, and each partition receives the custom splitting vector. The data is then repartitioned and sorted. However, there is no new information on how to deal with skewed values in that paper.

Joining is known to be negatively affected by skewed datasets. Although sorting and joining are different operations, they both require data redistribution. In [8], the authors propose doing joins with range partitioning instead of the typical hash partitioning. The authors found an elegant solution for handling repeat data values in a splitting vector: they follow a "weighted range partitioning" approach in which percentages at split points are also taken into account when redistribution occurs. If two split points share the same value, then that value will be sent in appropriate cardinalities to the partitions associated with those two split points. This approach is the closest that we have seen to an actual cardinality distribution.

# Chapter 3

# Design and Implementation

## 3.1 Overview of AsterixDB

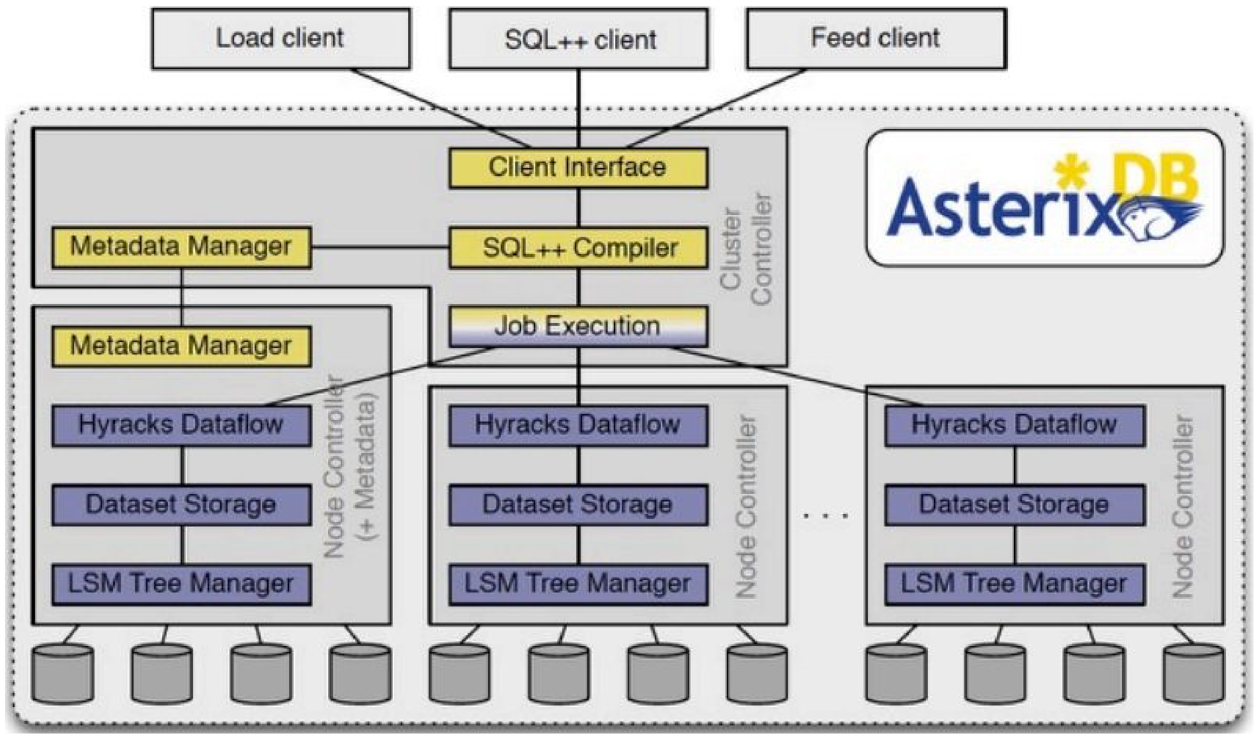


Figure 3.1: AsterixDB System Architecture

This section gives a brief introduction to the architecture "under the hood" of AsterixDB. Figure 3.1 shows an overview of the system architecture for AsterixDB [3]. As stated before, AsterixDB runs on a cluster of machines. There is a single node, known as the cluster controller, which interacts with the user and sends query plans composed of operators to the other nodes. AsterixDB follows a shared-nothing model in which each node has separate memory and storage. A hash function, based on the primary key of a dataset, is employed to distribute data amongst the partitions. The hope is that data will be distributed evenly and in an unbiased form across the partitions.

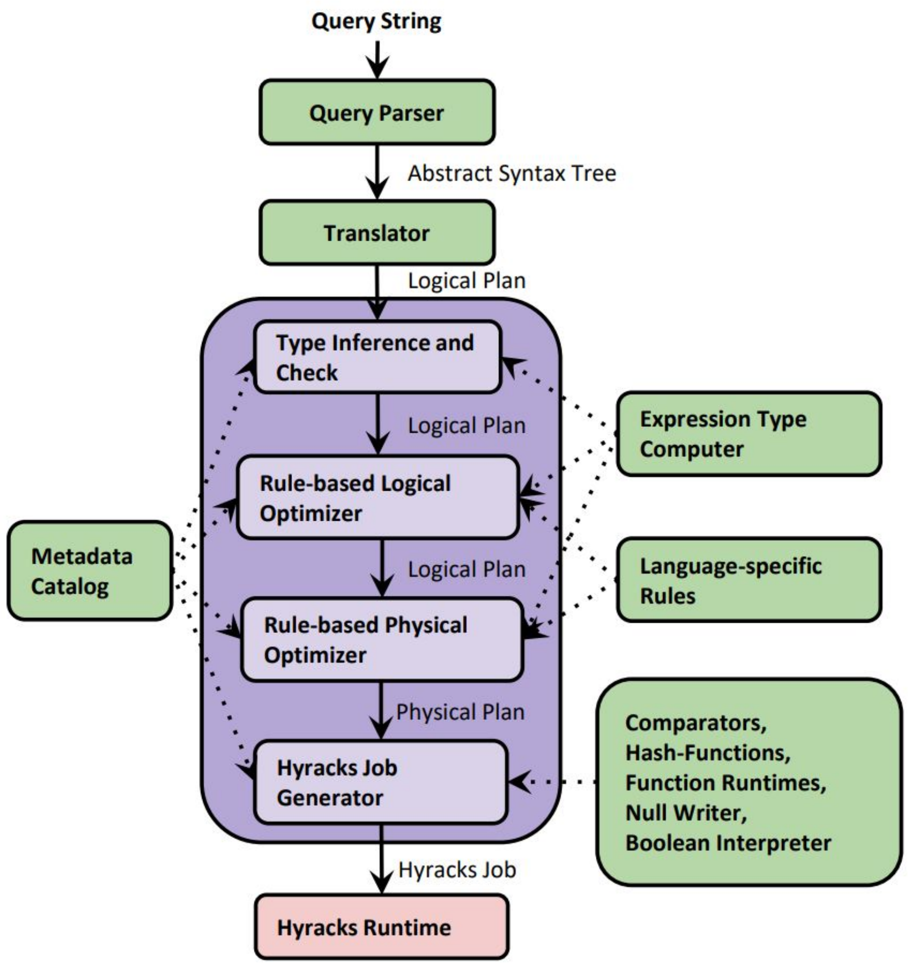


Figure 3.2: Query Processing Steps

As shown in Figure 3.1, data enters AsterixDB through loading, insertion queries, or a feed of continuous data. Figure 3.2 outlines the process by which a query is broken down into operators. The cluster controller is the main point of contact which handles all user requests. Once a query has been received, an abstract syntax tree is created. Then, Algebricks [4] breaks down the syntax tree into logical operators which are manipulated in the query optimization step. Finally, the resultant logical operators are compiled into Hyracks [5] jobs, which are expressed as Directed Acyclic Graphs (DAGs) consisting of operators and connectors. Each node controller then executes the job in a pipelined fashion. Finally, the result is stored on all nodes and returned to the user when requested [12].

The operators in a query DAG often have dependencies between them, as is the case with RPS. In such instances, nodes will have to wait for previous operators to finish before continuing with their operators. AsterixDB incorporates a push-based architecture in which data is pushed between operators in the form of frames. A frame is a collection of tuples. In AsterixDB, the fields in a tuple may have heterogeneous data types. Furthermore, some attribute values may be completely missing from a tuple.

## 3.2 Parallel Sort Global Merge

Before [6] introduced Repartitioning Parallel Sort, the default sort operator in AsterixDB was Parallel Sort Global Merge (PGSM). Figure 3.3 depicts the logical plan for PGSM. Each partition does a local scan and generates sorted runs for data relevant to the search predicate.

Each partition then merges all of its generated runs into one run. As each partition generates its final run, they push the result to a single partition. This partition merges the resulting runs from all of the partitions and generates a final sorted run with all of the data from all partitions present.

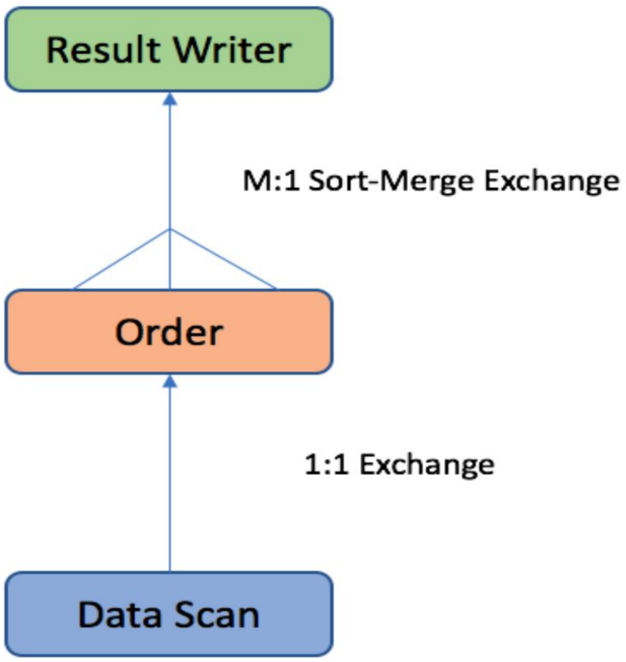


Figure 3.3: PGSM Plan

## 3.3 Repartitioning Parallel Sort

### 3.3.1 Overview

The current AsterixDB sorting operator has seven phases, as seen in Figure 3.4 (taken from [6]). Initially, a scan is done to collect all relevant data to the sort query. Then, the data is sent to the local sampling operator and the forward operator. To achieve sending data to two different operators, AsterixDB requires the use of a replicate operator. The forward operator's job is to block sort execution on each node until all incoming operations are completed. Note that there is a blocking edge between the forward operator and the range computation step. As a result, the

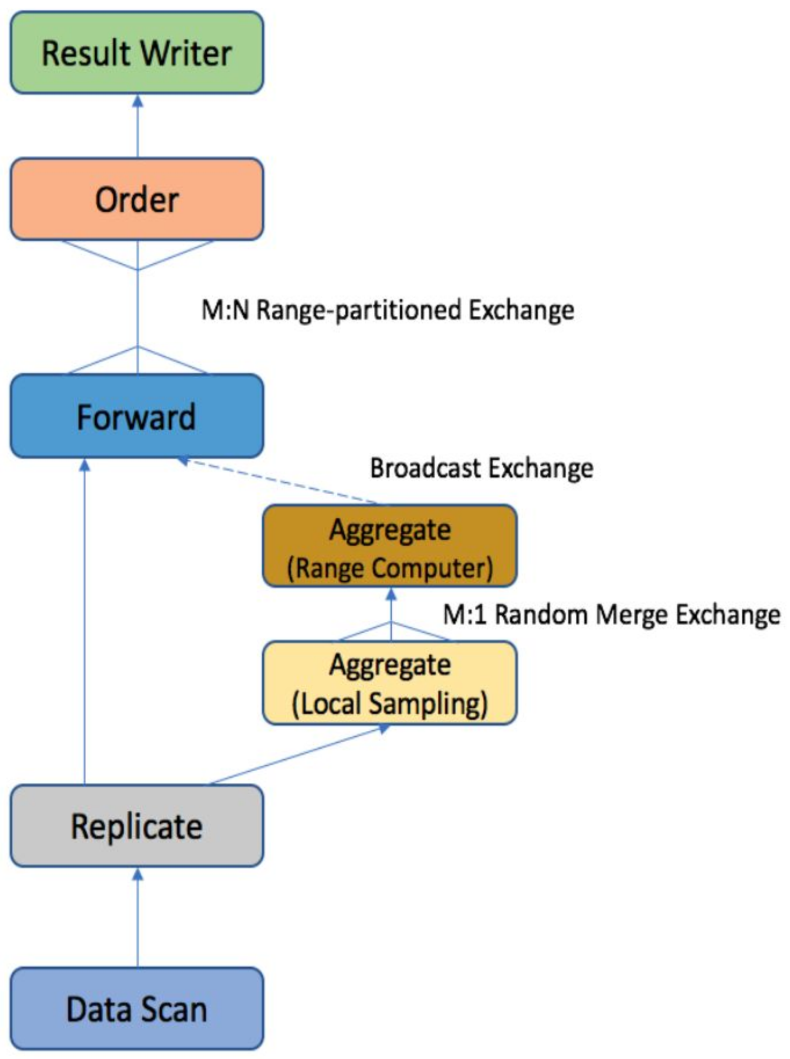

Result Writer
Order
M:N Range-partitioned Exchange
Forward
Broadcast Exchange
Aggregate (Range Computer)
M:1 Random Merge Exchange
Aggregate (Local Sampling)
Replicate
Data Scan

Figure 3.4: RPS Plan

local sampling and range computation will always finish before the forward operator executes. In the local sampling step, each partition takes a sample of the dataset. Currently, the default number of samples per partition is 100, but this number can be configured. Then, the samples from all partitions are sent to a single partition using the M:1 random merge exchange. The receiving partition constructs a RangeMap based on the incoming samples in the range computation step. (The rest of the partitions wait while the RangeMap is being computed.) As of right now the RangeMap only holds the splitting vector. Once created, the RangeMap is broadcast back to each partition. Now, armed with the RangeMap, each partition's forward operator participates in an M:N exchange operator in which the data to be sorted is distributed based on the RangeMap. Each partition then sorts the data they were given by the previous step. Finally, the next operator can pick up the sorted data by accessing the resulting partitions in an ordered fashion. We will discuss the steps relevant to this thesis in greater depth below, along with the proposed changes.

### 3.3.2 Sampling

Collecting samples from a partitioned dataset is an area that has had a lot of attention from the research community [7,10]. Prior work has mainly focused on finding the optimum number of samples to take from a dataset when constructing a splitting vector. While there can be optimizations done to the number of samples collected in the current sort operator, we'll leave such improvements to future work. The problem we are addressing here is the way (order) in which samples are collected. Currently, a linear scan is done, and the first 100 samples are taken per partition. This shouldn't cause an issue if the data is uniformly distributed. However, if the input is already sorted, the RangeMap will assign ranges to partitions based upon an unrepresentative sample. In this case, the last partition will receive all values not present in the initial sample, which typically is the bulk of the dataset. Therefore, it's essential to ensure that the RangeMap is constructed with data points that are indeed randomly selected. Fortunately, the current AsterixDB sampling stage iterates through all the tuples, so the only necessary improvement is to enforce a better selection policy. For such a task, we have chosen Algorithm R (for "reservoir sampling") from [11] (pp. 144-145).

The improved sampling algorithm is quite simple, and Figure 3.5 provides its basic pseudocode. In our case, k is always 100. After the k-item sample array is full, each new incoming value is given a chance to replace a value in the sample array. (Note that this algorithm could also be modified to increase k as the number of values seen increases.)

```
(* S has items to sample, R will contain the result *)
ReservoirSample(S[1..n], R[1..k])
  // fill the reservoir array
  for i := 1 to k
      R[i] := S[i]

  // replace elements with gradually decreasing probability
  for i := k+1 to n
    (* randomInteger(a, b) generates a uniform integer from the inclusive range {a, ..., b} *)
    j := randomInteger(1, i)
    if j <= k
        R[j] := S[i]
```

Figure 3.5 Pseudocode for Reservoir Algorithm

### 3.3.3 Splitting Vector

Now that we possess a much more random sample set, it's time to create a splitting vector. First, we define the exact semantics of the splitting vector. The definition below is adapted from [7]. A splitting vector, v[i], is defined as having $1 \leq i < k$, such that all tuples assigned to partition 1 have a sort key less than or equal to v[1]. All tuples assigned to partition 2 have a sort key value greater than v[1] but less than or equal to v[2], and so on, until all tuples assigned to partition k have a sort key greater than v[k-1]. In RPS, the splitting vector is found by sorting the samples and dividing them into k equally sized buckets. Then, the values at the borders of the buckets are selected as entries in the splitting vector.

In RPS today, assigning a tuple to a partition of the sort is reasonably straightforward. A tuple, with its sort key t, is assigned using a linear scan on the splitting vector until v[i] > t to determine the tuple's partition assignment. If multiple split values are equivalent, the partition on the right of the last such split value will be chosen. As a result all the tuples with identical sort keys will be routed to the same partition. In an extreme case where 100% of the tuples have identical sort key values, they will all end up in the last partition. The last partition will then be forced to sort the entire dataset locally. As we can see, RPS doesn't have a protocol in place for handling a massive influx of identical values.

### 3.3.4 Distribution

As we have just seen in the previous section, there is a major flaw in the way distribution is handled in RPS. To improve the tuple distribution, our goal would be to correctly spread the identical values to all partitions that can accept them. One way to accomplish such a goal is to incorporate the weighted policy mentioned in [8]. To do so, an auxiliary structure to the splitting vector will need to be created to hold the percent distribution at each split point. We will refer to it as the “percentage list” or PL for short. The PL list, p[i], is defined as having 1 ≤ i < k, such that the i’th partition has p[i] percent of keys with value v[i]. Note that the PL and the splitting vector have the same number of entries. Constructing the PL is fairly simple and requires a linear scan of the samples list once the splitting vector has been determined. Figure 3.6 shows the algorithm to calculate the percentage list.

```
(*  S is the sorted array of samples
    V is the splitting vector *)
calculatePercentageList(S[1..n], V[1..k])
    for i := 1 to k
        int smallestIndexEqualToSample = V[i];
        while(smallestIndexEqualToSample >= 0 && S[V[i]] == S[smallestIndexEqualToSample])
            smallestIndexEqualToSample--;

        smallestIndexEqualToSample++;

        int largestSplitIncludingSample = i;
        while(largestSplitIncludingSample <= k && S[V[i]] == S[V[largestSplitIncludingSample]])
            largestSplitIncludingSample++;

        largestSplitIncludingSample--;

        int largestIndexEqualToSample = splitPoints[largestSplitIncludingSample];
        while (largestIndexEqualToSample <= n && S[V[i]] == S[largestIndexEqualToSample]
            largestIndexEqualToSample++;

        largestIndexEqualToSample--;

        double count = largestIndexEqualToSample - smallestIndexEqualToSample;
        double waterMark = smallestIndexEqualToSample;
        for j := i to largestSplitIncludingSample
            percentages[j] = ((V[j] - waterMark) * 100) / (count);
            waterMark = V[j];

        i = largestSplitIncludingSample;
```

Figure 3.6 Percentage List Algorithm

A change must be made to the current distribution method in order to utilize the PL list. Let’s return to the previous example with a tuple that has a sort key of t. A linear scan on the splitting vector is still performed. However, there is a special case when t = v[i]. Figure 3.7 displays pseudocode for the algorithm that is executed after the first equivalent split value is found. The algorithm returns the index of the partition that the tuple is to be assigned to.

```
(* V is the splitting vector, P is the percentage list, and k is the startIndex  *)
AssignTuple(V[1 ... n], P[1 .. n], int i)

    (* randomDouble(a,b) generates a uniform double from the inclusive range {a, ..., b} *)
    double a := randomDouble(0,100)
    int j := i

    while V[i] == V[j] && j <= n
        a -= P[j]
        if a <= 0
            return j
        j++

    return j
```

Figure 3.7 Percent Distribution Algorithm

Figure 3.8 shows a small example of a splitting vector of [2,2,3] with a percentage list of [20,80,50]. Twenty percent of the values of 2 will go to Partition 1 while the remaining eighty percent will go to Partition 2. The values of 3 will be evenly split between Partitions 3 and 4.

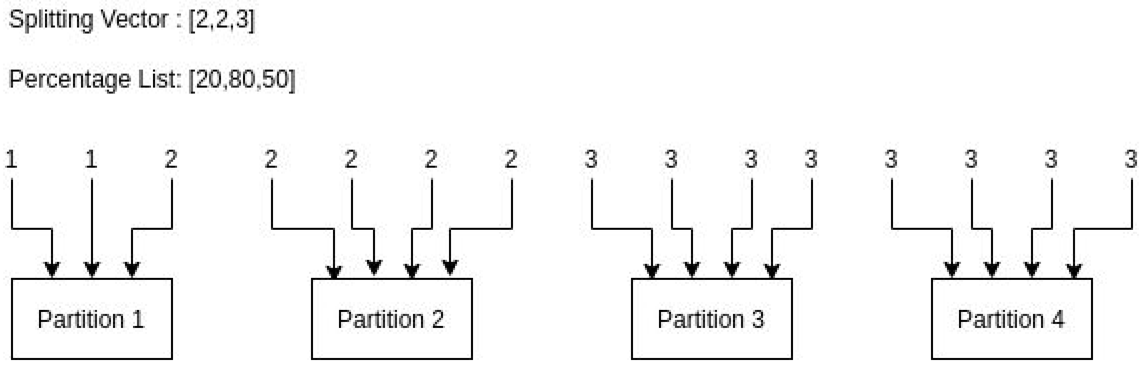


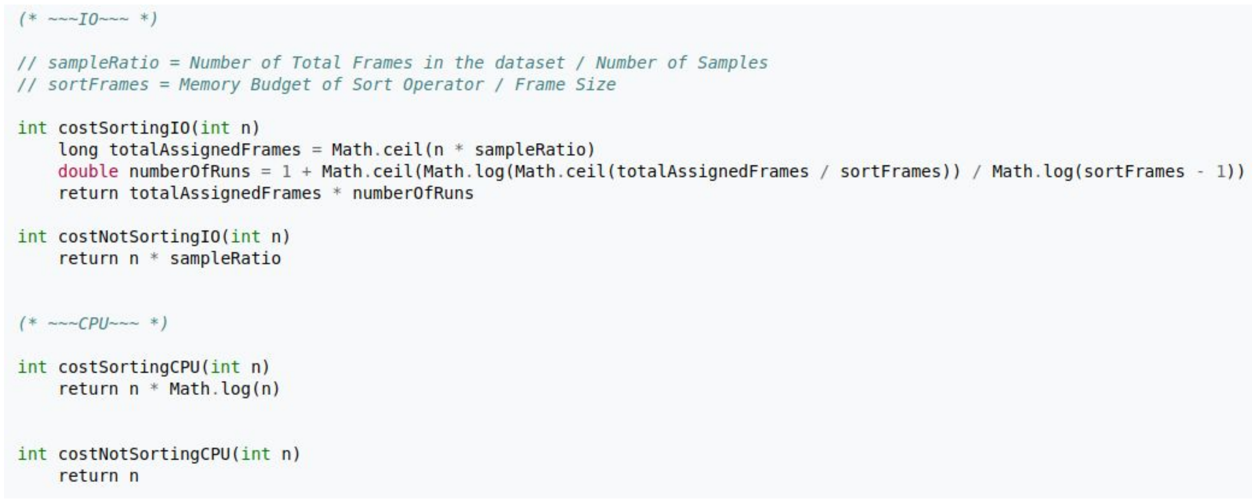
Figure 3.8 Splitting Vector & Percentage List Example

## 3.4 Skewed Range Map Algorithm

The goal of this thesis is to optimize RPS to achieve the fastest runtime possible, given any data distribution. The runtime of RPS will be dependent upon the highest sorting time of any single partition. Therefore, we are going to be constructing an algorithm that minimizes the maximum sorting time of any partition. We will call this algorithm the Skewed Range Map Algorithm, or SRM for short. Apart from RPS, SRM can designate partitions as “sorting” or “non-sorting.” A skewed value in a dataset can be assigned to separate partition(s) that will only receive that particular value. Since they receive only one value, they won’t have to sort the incoming data, leading to faster linear execution time. The result is a RangeMap based upon a work analysis that accounts for skewed datavalues. It is also noteworthy that the resulting sorted data will be eventually distributed over the nodes (partitions) in the cluster, avoiding the potential for overloading one node’s storage capacity.

### 3.4.1 Cost Functions

Similarly to RPS, SRM assigns samples to partitions to construct a RangeMap. Since SRM does a work analysis, it needs to know the “cost” of a partition when it’s assigned a particular set of samples. Partitions come in two flavors, “sorting” and “non-sorting,” which require separate cost functions. The goal is that these cost functions should accurately estimate the execution time of the sort. Figure 3.9 shows the required cost functions. The number of samples defaults to 100 but is adjustable by setting ‘compiler.sort.samples’.



```
(* ~~~IO~~~ *)

// sampleRatio = Number of Total Frames in the dataset / Number of Samples
// sortFrames = Memory Budget of Sort Operator / Frame Size

int costSortingIO(int n)
    long totalAssignedFrames = Math.ceil(n * sampleRatio)
    double numberOfRuns = 1 + Math.ceil(Math.log(Math.ceil(totalAssignedFrames / sortFrames)) / Math.log(sortFrames - 1))
    return totalAssignedFrames * numberOfRuns

int costNotSortingIO(int n)
    return n * sampleRatio


(* ~~~CPU~~~ *)

int costSortingCPU(int n)
    return n * Math.log(n)


int costNotSortingCPU(int n)
    return n
```

Figure 3.9 Cost Functions

For this thesis, we have chosen to experiment with two cost functions. First, there is a CPU cost function. Given an input size of n samples, the CPU cost function returns a value of n for a non-sorting partition and a value of n * log(n) for a sorting partition. Second, there is an IO cost function which is more complex and takes into account the total number of frames encompassing the dataset. Given n samples, the IO cost function determines how many frames those samples represent. Then, if the partition is non-sorting, the number of frames is returned. If the partition is sorting, the number of input frames and memory buffer size are used to calculate the total number of sort passes needed. Finally, the result is the number of frames multiplied by this number of passes.

## 3.4.2 Non-Sorting Restrictions

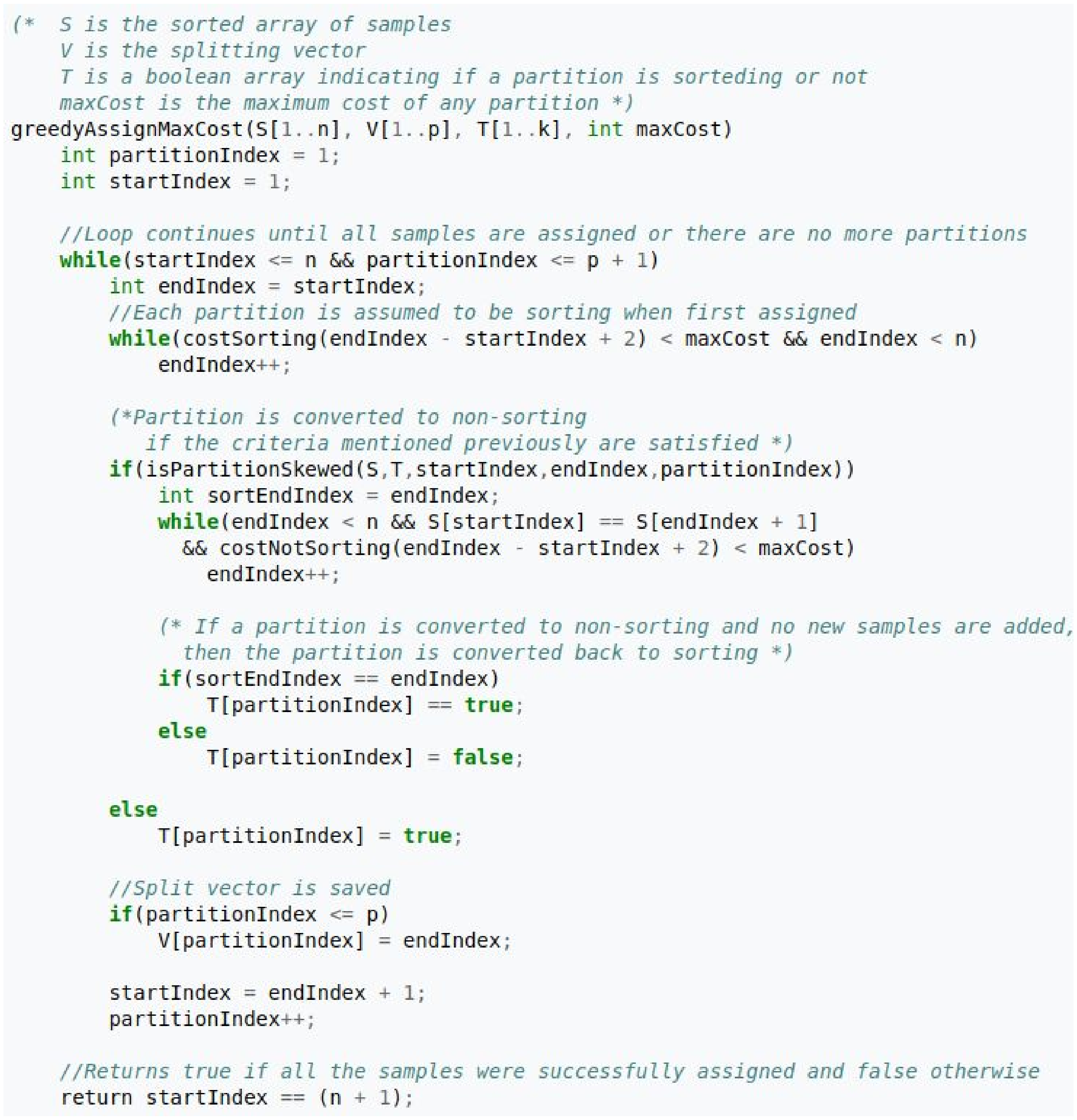
If a partition receives only one value, e.g.,, all NULLs, a sort does not have to be done. All of the tuples share an identical sort key value and are therefore already in an acceptable order. Now, a distinction can be made between partitions that must sort and partitions that don't have to sort. Not all SRM partitions can be non-sorting; several prerequisites must be met for a partition to avoid having to sort. First, the partition with the lowest index always has a lower bound value of negative infinity, and the partition with the greatest index has an upper bound of positive infinity. (This approach also works for heterogeneous data values as long as they can be compared.) The reason is that since the splitting vector is created using a relatively small sample of values, the entire spectrum of possible domain values must still be accounted for. Second, two adjacent partitions can only both be non-sorting if they share an identical sort key value. Otherwise, there could exist unsampled in-between values that would compromise the non-sorting integrity of one of the partitions. Figure 3.10 puts the above restrictions into pseudocode.  The function returns true if a partition is able to be non-sorting and false otherwise.

```
(*  S is the sorted array of samples
    T is a boolean array indicating if a partition has sorted or not
    startIndex is the index of the first sample assigned to the partition
    endIndex is the index of the last sample assigned to the partition
    partitionIndex is the index of the partition *)
isPartitionSorting(S[1..n], T[1..k], int startIndex, int endIndex, int partitionIndex)
    //Partition can't be skew if it's the first or last partition
    if(partitionIndex == 1 || partitionIndex == k)
        return true;

    //First and last index must be the same value
    if(S[startIndex] != S[endIndex])
        return true;

    //Last partition must have sorted or the last partition is skewed over an identical value
    if(T[partitionIndex - 1] || S[startIndex - 1] != S[startIndex])
        return false;

    return true;
```

Figure 3.10 Partition Sorting Restrictions

### 3.4.3 Greedy Assignment

Given a maximum per-partition cost, the origin of which we will explain in the next section, calculating the splitting vector to balance the sorting process can be achieved using a greedy approach guided by the maximum cost. Starting from the first partition, each partition is assigned as many samples as it can hold before it exceeds the maximum cost. Each partition is assumed to be sorting when first assigned and is converted to a non-sorting partition if the criteria mentioned previously are satisfied. If a partition is converted to non-sorting, more samples will then be assigned if they share identical values. In addition, if a partition is converted to non-sorting and no new samples are added, then the partition is converted back to sorting.(Otherwise, even if the next partition met the "non-sorting" criteria, it would be forced to sort as it would be "non-sorting" over a different data value than the current partition.) Since there are n samples total, the complexity of assigning all samples to partitions is O(n). The greedy assignment returns true if all the samples were successfully assigned and returns false otherwise. Figure 3.11 has the pseudocode for the greedy step.

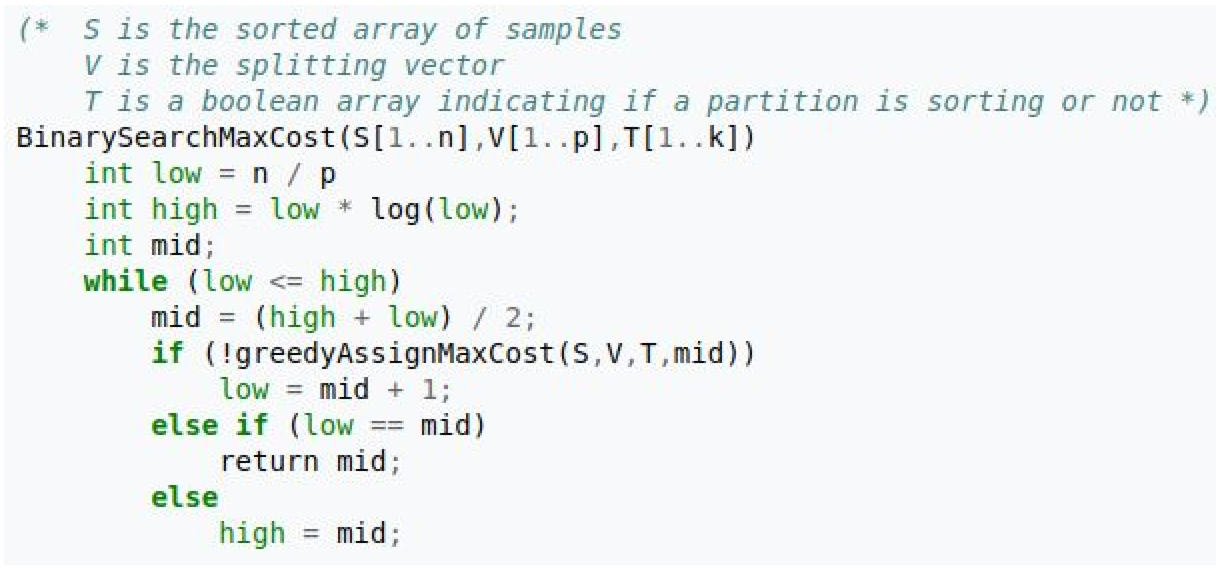


```
(*  S is the sorted array of samples
    V is the splitting vector
    T is a boolean array indicating if a partition is sorteding or not
    maxCost is the maximum cost of any partition *)
greedyAssignMaxCost(S[1..n], V[1..p], T[1..k], int maxCost)
    int partitionIndex = 1;
    int startIndex = 1;

    //Loop continues until all samples are assigned or there are no more partitions
    while(startIndex <= n && partitionIndex <= p + 1)
        int endIndex = startIndex;
        //Each partition is assumed to be sorting when first assigned
        while(costSorting(endIndex - startIndex + 2) < maxCost && endIndex < n)
            endIndex++;

        (*Partition is converted to non-sorting
           if the criteria mentioned previously are satisfied *)
        if(isPartitionSkewed(S,T,startIndex,endIndex,partitionIndex))
            int sortEndIndex = endIndex;
            while(endIndex < n && S[startIndex] == S[endIndex + 1]
              && costNotSorting(endIndex - startIndex + 2) < maxCost)
                endIndex++;

            (* If a partition is converted to non-sorting and no new samples are added,
              then the partition is converted back to sorting *)
            if(sortEndIndex == endIndex)
                T[partitionIndex] == true;
            else
                T[partitionIndex] = false;

        else
            T[partitionIndex] = true;

        //Split vector is saved
        if(partitionIndex <= p)
            V[partitionIndex] = endIndex;

        startIndex = endIndex + 1;
        partitionIndex++;

    //Returns true if all the samples were successfully assigned and false otherwise
    return startIndex == (n + 1);
```

Figure 3.11 Greedy Assignment Algorithm

## 3.4.4 Binary Search

The SRM algorithm does a binary search using the greedy algorithm to compute the split vector. Specifically, the SRM algorithm is looking for the lowest maximum cost for which the greedy algorithm returns true. The cost-finding outer loop of the SRM algorithm is a simple binary search on a boolean function and is depicted in Figure 3.12. The return value for this algorithm is the minimum cost for which an assignment can be made. This cost can then be plugged into the greedy step to get the final splitting vector.



```
(*  S is the sorted array of samples
    V is the splitting vector
    T is a boolean array indicating if a partition is sorting or not *)
BinarySearchMaxCost(S[1..n],V[1..p],T[1..k])
    int low = n / p
    int high = low * log(low);
    int mid;
    while (low <= high)
        mid = (high + low) / 2;
        if (!greedyAssignMaxCost(S,V,T,mid))
            low = mid + 1;
        else if (low == mid)
            return mid;
        else
            high = mid;
```

Figure 3.12 Binary Search on Maximum Cost

## 3.4.5 Running Time

Let's analyze the running time of the algorithm. In the explanation below, n is the number of samples, and p is the number of partitions. If every partition is non-sorting, the lower bound for the maximum cost is n / p. In the case that every partition has to sort, the upper bound maximum cost is (n / p) * log (n / p). As stated above, the cost of computing one greedy assignment is O(n). There are a total of log( (n / p) * log (n / p) ) steps to the binary search. Therefore, the running time of the SRM algorithm is O(n * log((n/p) * log(n/p))).

### 3.4.6 Examples

Figure 3.13 displays an example assignment by the greedy algorithm. In this case, the chosen maximum cost is 5. A sorting partition can be assigned a maximum of 3 samples here, while a non-sorting partition can hold 5 samples. In the example, only the second partition can be labeled as non-sorting. Although the third partition has all identical sample values (5's), it is adjacent to a partition that is non-sorting on a different value (4's). If the full data set contains a (non-sampled) tuple with a key value of 4.5, that tuple would be assigned to the third partition during distribution. The resulting splitting vector, in this example, would be [3,4,5]. The percentage list is [100,100,75]. Also, since all of the samples were successfully assigned, the greedy algorithm would return a value of true.

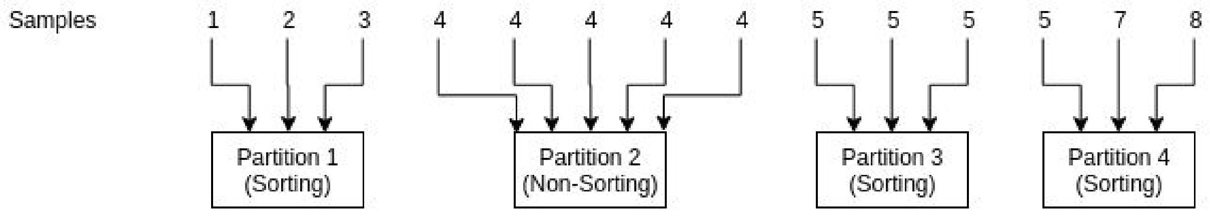


Figure 3.13 Adjacent Example

Another edge case of the greedy algorithm is demonstrated in Figure 3.14. The maximum cost in this example is again 5. When initially assigned, partition two was assumed to be sorting at first. It was assigned 3 samples (as is the maximum for this example). However, all three of those samples had an identical value of 3. The partition was converted to non-sorting, and more samples equivalent to 3 were attempted to be assigned. However, the next sample value is 5, and adding it to partition two would violate the non-sorting state of partition two. According to the greedy assignment algorithm, if a partition is converted to non-sorting, but no new samples were added, then that partition is reverted back to sorting. Keeping the partition as non-sorting wouldn't incur any benefit to execution time and would only prevent the next partition from being non-sorting. Note that if partition two were non-sorting, then partition three would have to be sorting since partition two and three contain different values. The last sample (9) isn’t assigned to any partition, and therefore the return value of the greedy algorithm is false - it wasn’t able to assign all samples to compute a splitting vector that keeps all partitions under the maximum specified cost.

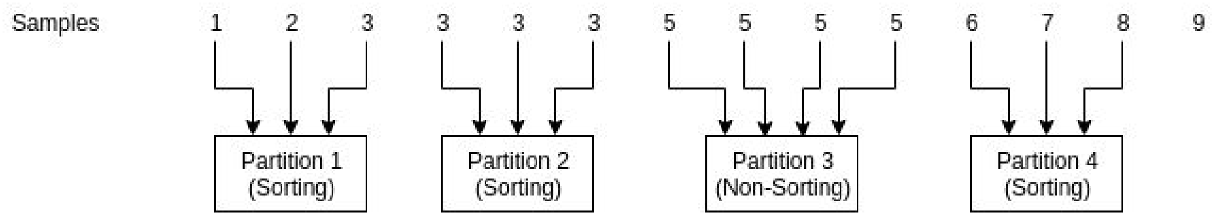


Figure 3.14 Sorting All Identical Samples Example

The final example in Figure 3.15 exemplifies what would happen in the case where all the samples share an identical value. Again, the maximum cost is 5. Notice that the first partition has to remain as a sorting partition in case the full dataset contains some entry with a key value less than 3 (and similarly the last partition remains sorting). The greedy function would return true with a splitting vector of [3,3,3]. The percentage list in this example is [21.5, 35.7, 35.7].

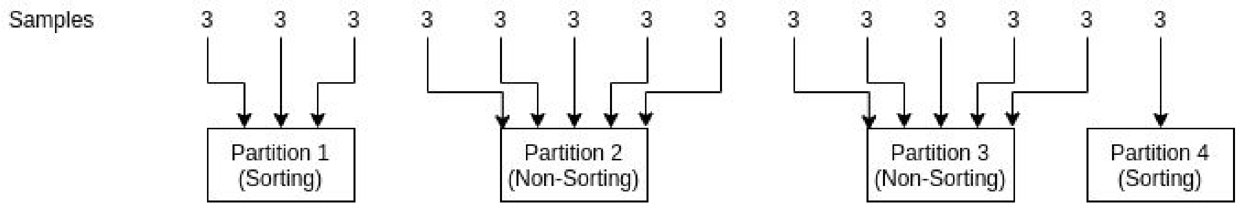


Figure 3.15 All Identical Sample Example

### 3.4.7 Optimistic Optimization

The SRM algorithm works reasonably well in most cases, as we will see later, but there are a few instances where some of the restrictions could be relaxed for better performance. One in particular is the requirement for the first and last partition to be sorting, and another is two non-sorting partitions only being adjacent when they share the same value. Figure 3.16 illustrates a problematic edge case. The maximum cost is 5 for this example as well. Notice how there are three partitions and 3 different values, all of which have cardinalities of 4.

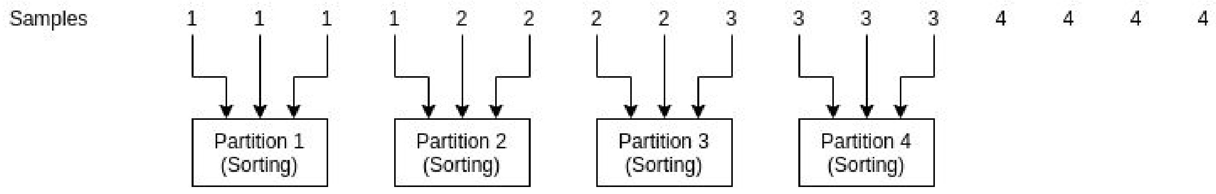
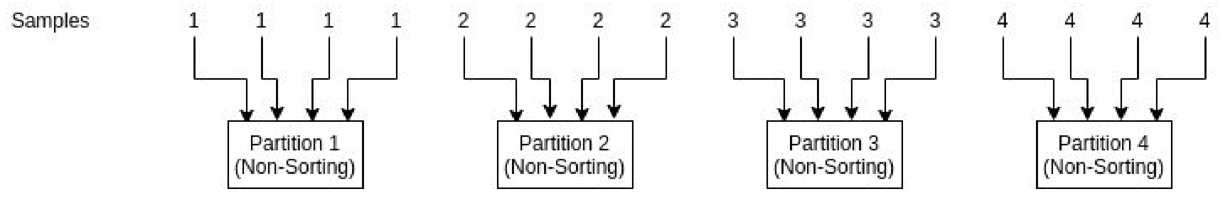


Figure 3.16 SRM Optimistic Pre

The first partition is forced to sort because it's the first partition. The second and third partitions sort because they didn't get assigned all of the same value, and the last one sorts because it's the last partition. The SRM algorithm is optimized for the underlying dataset to potentially have in-between values that could compromise a partition's ability to not have to sort the data given to it. However, as we can see in this example, such precautions can severely limit the number of samples that can be assigned. If we disable these extra precautions, we could risk in-between values getting into partitions labeled as "non-sorting." However, the local sort operator can be assigned to only behave as "non-sorting" if it can verify that all of the data values are the same. Therefore, the worst that can happen is that a partition labeled as "non-sorting" will end up having to sort. More importantly, the computed (sorted) result will still be correct. Let's look at an optimistic approach to the problem then, as featured in Figure 3.17. Now, the first and last partition can be marked as non-sorting, and two adjacent partitions can be non-sorting even if they contain different values.


Samples
1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4
Partition 1 (Non-Sorting)
Partition 2 (Non-Sorting)
Partition 3 (Non-Sorting)
Partition 4 (Non-Sorting)
Figure 3.17 SRM Optimistic Post

In this case are able to assign all of the samples given the same maximum cost constraint. However, now we aren't guaranteeing that a partition marked as non-sorting will ultimately always receive identical values. In such a case, the partition will have to sort (leading to a longer execution time).

# Chapter 4

# Performance Evaluation

In this chapter, we present the results of a set of  experiments that compare the old RPS implementation against the SRM algorithm. The SRM algorithm will be tested using a CPU estimated cost function as well as an IO estimated cost function. These two cost functions each come in two flavors, optimistic and pessimistic. Our experiments also include a “cardinality” and lead balancing approach, that consists of the PL (Percentage List) changes but not the cost-based SRM algorithm. Furthermore, for the “cardinality approach”, we adjusted the percentage algorithm to ignore thresholds below 5% (to avoid one sample causing a partition to sort). AsterixDB’s current handling of identical values in the RPS RangeMap is less than ideal and will serve as a strawman in our experiments. The cardinality approach and the SRM algorithm both address the skew value issue. However, it is not clear whether the extra overhead added by the cost based SRM algorithm merits its usage. In addition, all of the tests except for PGSM and RPS include the added sampling improvements discussed in 3.2.2. All of the tested algorithms are depicted in Table 4.1.

| Name | Splitting Vector | Percentage List |
|---|---|---|
| PGSM | None | No |
| RPS | Even Splitting Vector (RPS) | No |
| Cardinality | Even Splitting Vector (RPS) | Yes |
| Pess CPU | Pessimistic SRM with CPU cost functions | Yes |
| Pess IO | Pessimistic SRM with IO cost functions | Yes |
| Opt CPU | Optimistic SRM with CPU cost functions | Yes |
| Opt IO | Optimistic SRM with IO cost functions | Yes |

Table 4.1: Tested Algorithms

The chapter is outlined as follows. First, we will go over the experimental setup. Second, we will determine appropriate cost functions for AsterixDB in the testing environment. Third, we will go over various performance tests that pit the three approaches against each other. Finally, we will do a general analysis and discuss which method produces the best performance.

## 4.1 Experimental Environment and Setup

The experiments listed in this chapter were conducted on a cluster of 6 nodes. For some of the experiments, the number of nodes was varied to study speedup. Each node was a Dell PowerEdge 1435SC with 2 Opteron 2212HE processors and 8GB of DDR2 memory. Each node had two 1TB hard disk drives with a 7200 RPM speed. Since there were two disks per node, AsterixDB was configured to run with two partitions per node. The AsterixDB sort memory budget was set to 2048KB, and the JVM was given 4GB of RAM to operate with. The use of a low sort memory budget was critical in forcing the sort operator to do multiple passes. Otherwise, we wouldn't have tested the full functionality of the IO cost function. The data we sorted on had the schema presented in the Wisconsin Benchmark [14]. The only modification was to generate tuples of approximately 600 bytes [1]. Listing 4.1 depicts the AsterixDB DDL statements for the generated data.

```
CREATE TYPE WisconsinSchemaType AS {
unique1 : int ,
unique2 : int ,
two : int ,
four : int ,
ten : int ,
twenty : int ,
onePercent : int ,
tenPercent : int ,
twentyPercent : int ,
fiftyPercent : int ,
unique3 : int ,
evenOnePercent : int ,
oddOnePercent : int ,
stringu1 : string ,
stringu2 : string ,
string4 : string
} ;

CREATE DATASET Wisconsin (WisconsinSchemaType) PRIMARY KEY unique1 ;
```

Listing 4.1 Experiment dataset definition

The sizes and cardinalities of the datasets we used for testing are indicated in Table 4.2.

| Number of Tuples(Million) | Total Dataset Size(JSON) |
|---|---|
| 30 | 16 GB |
| 60 | 32 GB |
| 90 | 47 GB |
| 120 | 63 GB |
| 150 | 78 GB |
| 180 | 94 GB |

Table 4.2: Experiment Datasizes

The primary key of each dataset was unique1 and the sort key varied between experiments. Listing 4.2 shows the two query types executed in the experiments. The NULL query tests a sort key with unique values intermingled with NULL values. Normally, the Wisconsin dataset doesn't have any NULL values in the unique2 field. However, the data generator [1] allows the field to have a certain percentage of NULL values. The domain query tests a sort key which has a small subset of values.

```
//NULL Query
SELECT VALUE w
FROM Wisconsin
ORDER BY w.unique2;

//Domain Query
SELECT VALUE w
FROM Wisconsin
ORDER BY w.four;
```

Listing 4.2: SQL++ queries

## 4.2 Results and Analysis

### 4.2.1 Varying Degrees of Nulls

In this experiment, we measure the execution time for sorting datasets with varying degrees of data skew. For each query, we sent a request to the database via a curl command. We set the data mode to deferred, which causes the query not to return any values to the requester (to avoid counting return network traffic from the server). We stress-test the algorithms by increasing the number of NULL values being sorted on in increments of 20%. The sort key for this experiment is unique2 with the percentage of NULL values ranging from 0% to 100%. The test ran on 12 partitions with 180M tuples. The goal of this experiment is to determine which algorithms perform the best under stressed conditions. Figure 4.1 displays the results.

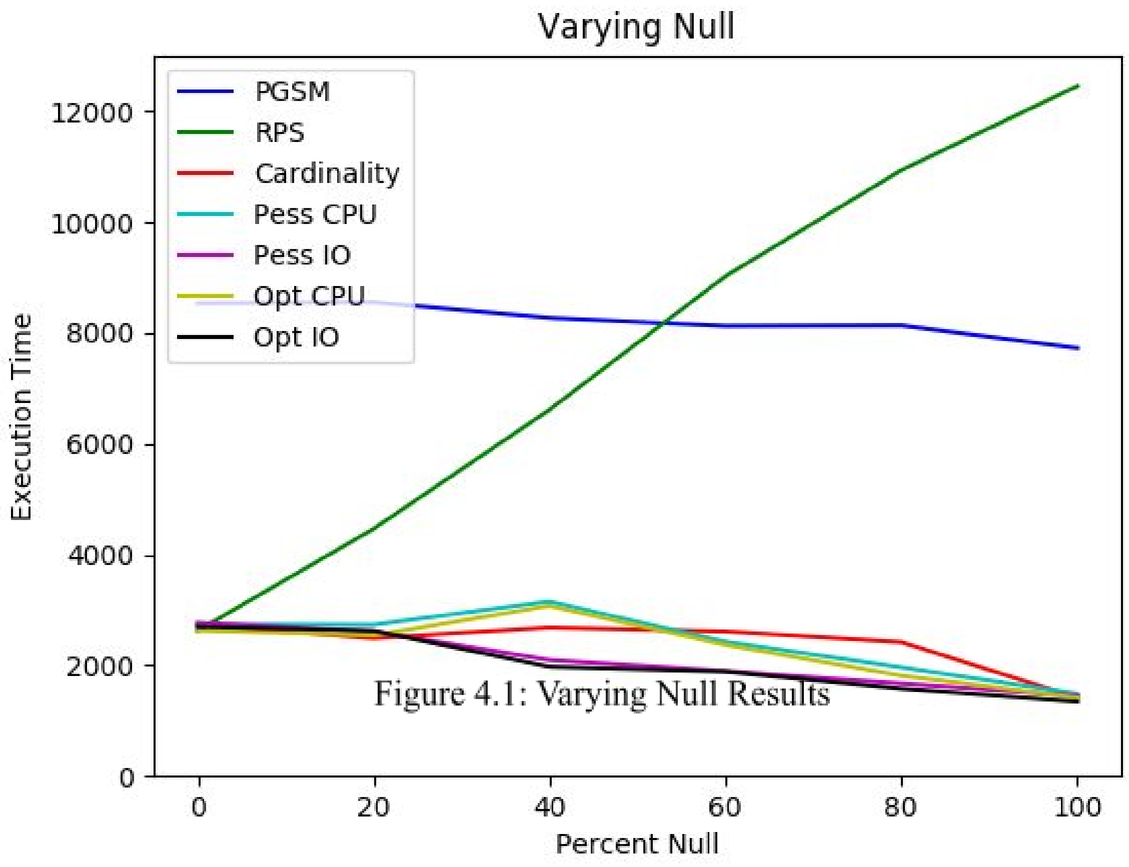


Figure 4.1: Varying Null

As explored in [6] RPS is much faster than PGSM for sorting. However, RPS introduced a new problem, as its tuple-to-partition distribution method is susceptible to skewed values. PGSM didn't have this problem, which is why PGSM remains consistent as the data becomes more and more skewed. RPS on the other hand does not remain consistent with the increasing amount of

NULL values, which is to be expected. As the number of nulls increases, RPS stops utilizing all of the available partitions, and the number of partitions that are actively sorting decreases. Eventually, RPS became significantly worse than PGSM, the sorting algorithm which it was supposed to be far superior to. Since the disparity between RPS/PGSM and the SRM algorithm was so large, Figure 4.2 zooms in to show how just the variants of the SRM algorithm did with respect to one another.

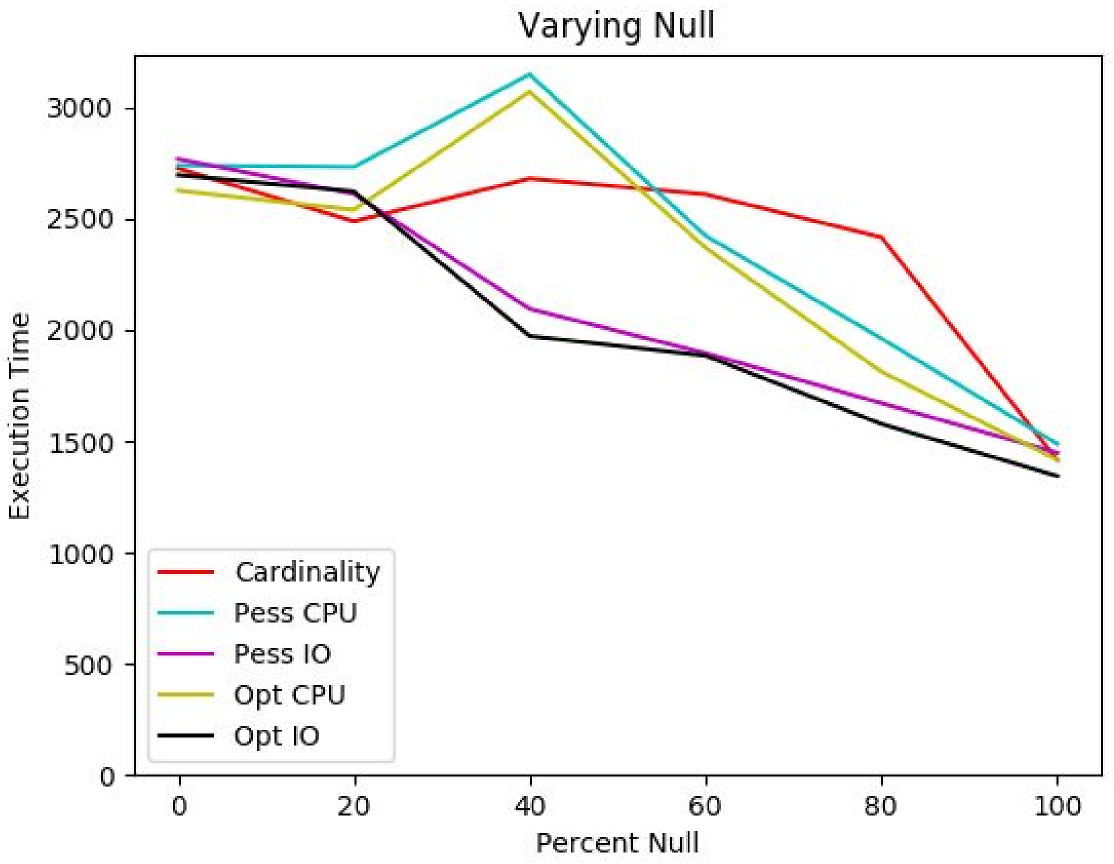


Figure 4.2: Varying Null Close Up

As we can see, within the SRM family, the IO cost functions did a lot better than CPU and plain cardinality. At 20% NULL values, there aren't enough null values to warrant an entire partition being assigned as non-sorting. At around 40% NULL values, there are enough null values in the sample set to assign an entire partition to non-sorting on NULL values. The IO cost function had a slight in execution time, while the CPU cost function had a noticeable spike. The CPU cost function overestimated the number of tuples that could “fit” into a partition that doesn’t sort, which caused the non-sorting partition to execute for longer than the sorting ones. The cardinality method performs worse at higher NULL percentages because it doesn't take into account the work difference between sorting and not sorting. Finally, at 100% NULL values none of the partitions are sorting anymore.

### 4.2.2 Speedup

The above experiment isn’t necessarily the most realistic scenario when it comes to a typical query. A benchmark by the name of PigMix [13] contains fields with 20% NULL values when dealing with NoSQL data. Accordingly, in this experiment, we used a Wisconsin-style dataset of 105M tuples with 20% of the unique2 field values being NULL. The number of partitions was varied between 4 and 12 (i.e, we used cluster sites of 2-6 nodes). Figure 4.3 shows the results of this experiment, which divided a total fixed dataset size of 56GB over an increasing number of nodes. Speedup here is defined by the ratio of the execution time at 4 partitions (the starting point on the left) over the execution time of any subsequent datapoint.

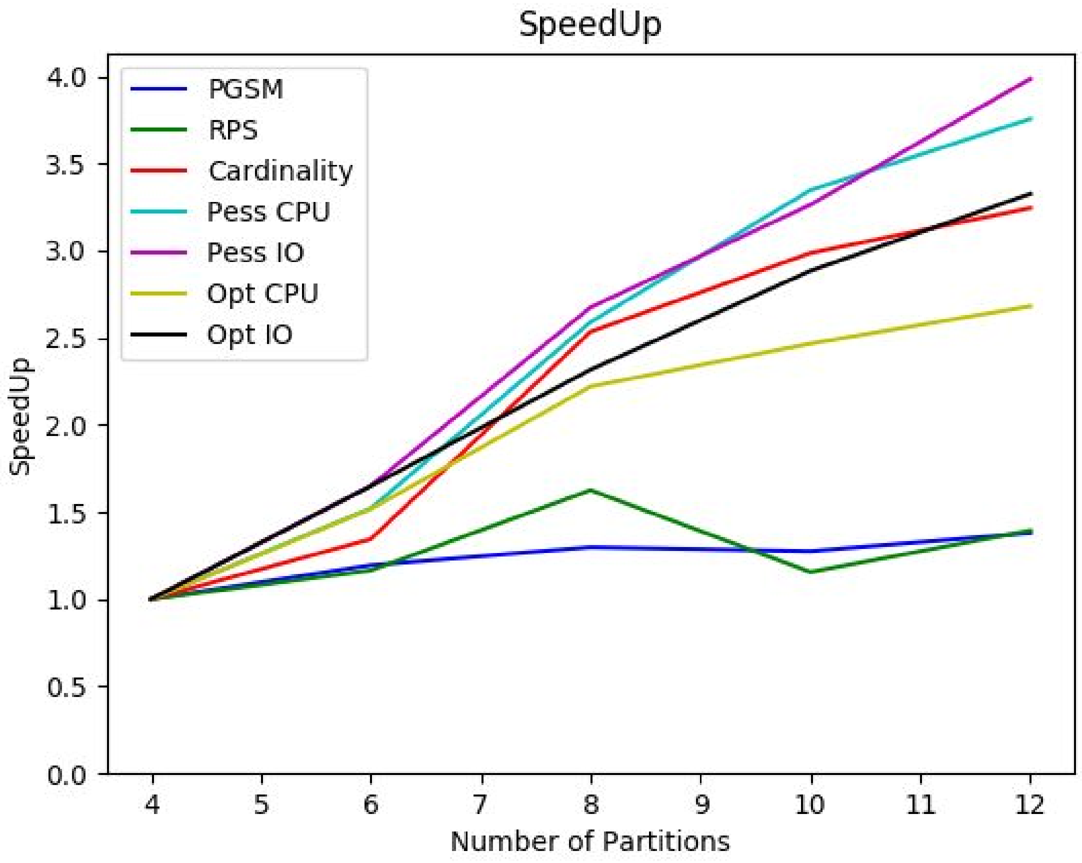


Figure 4.3: Speed Up Results

Since data distribution is irrelevant to PGSM, there was no significant change in the speedup. For RPS, there is a decrease in speedup at the 10 partition mark because there are then enough NULL values in the RangeMap to cause RPS not to send any tuples to the first of the ten partitions. In this experiment, the pessimistic cost functions outperformed the optimistic cost functions. When NULL values get sorted, they are placed at the start of the number line, before 0. At the 10 partition mark there are enough NULL values for the optimistic cost functions to dedicate an entire partition to NULL values. The pessimistic cost functions dedicate the second partition to NULL values because they cannot mark the first partition as non-sorting. In the pessimistic case, the NULL values are split between the first and second partition, while in the optimistic case, only the first partition receives the NULL values. However, the optimistic cost functions can load balance and spill the NULL values to adjacent partitions. If the cost functions accurately predicted the difference between sorting and non-sorting, then the optimistic variants would always outperform the pessimistic variants in this example. Therefore, the cost functions are not perfect, although we can say that the IO cost function outperforms the CPU cost function.

### 4.2.3 Scaleup

Another standard test we did was the scaleup test, which scales the cluster size and the data size proportionally. This test ran with 30M tuples per node with the data-set having 20% null values. Scaleup is defined by the ratio of the first data point's execution time over the execution time of any subsequent datapoint, and the ideal ratio is 1.0. The results are shown in Figure 4.4 and Figure 4.5.

As demonstrated in [6] PGSM has poor scaleup because the final step involves a single partition merging data from all other partitions. Again, at around the 10 partition mark, RPS starts to decline rapidly. For SRM, the optimistic functions start assigning a single partition to NULL values a lot earlier than the pessimistic functions. At the 8 partition mark, the optimistic cost functions have already dedicated a single partition to the NULL values. In comparison, it takes the pessimistic functions until the 10 partition mark because pessimistic functions must label the first partition as sorting. However, once the pessimistic functions do detect the NULL values, they are still comparable in scaleup to the optimistic functions. In this specific test, as the number of partitions is increased, the difference between the optimistic and pessimistic functions decreases. There is only one skewed value in the dataset, and the pessimistic functions are only missing out on the first partition being able to be non-sorting. As we will see in the next experiment, pessimistic performance degrades as the number of skewed values increases.

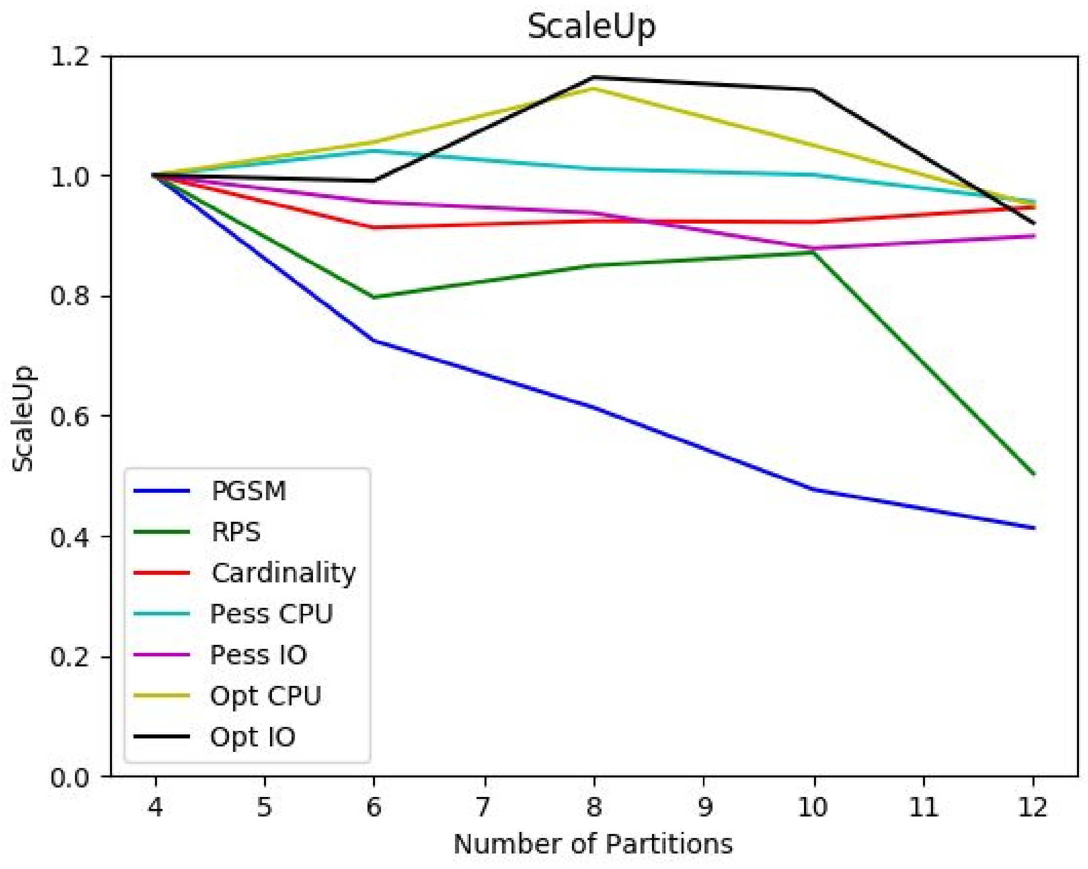


Figure 4.4: Scale Up Results

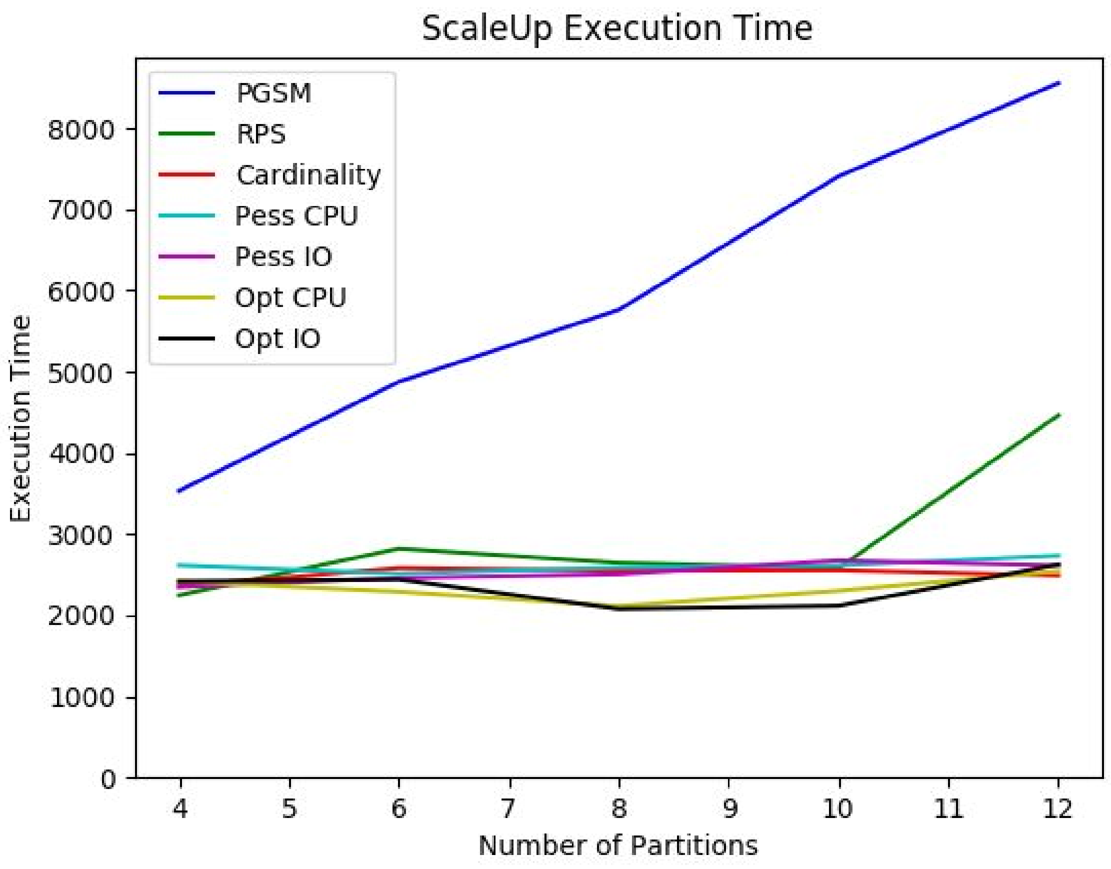


Figure 4.5 Scale Up Execution Time

### 4.2.4 Varying Domain Size

The goal of this experiment is to see how the cost functions perform with a very limited sort field domain size. For example, a domain size of 1 has all of the same values, while a domain size of 2 has two different values present in the sort key. The experiment ran with 180M tuples sorted over 12 partitions. The results are in Figure 4.6.

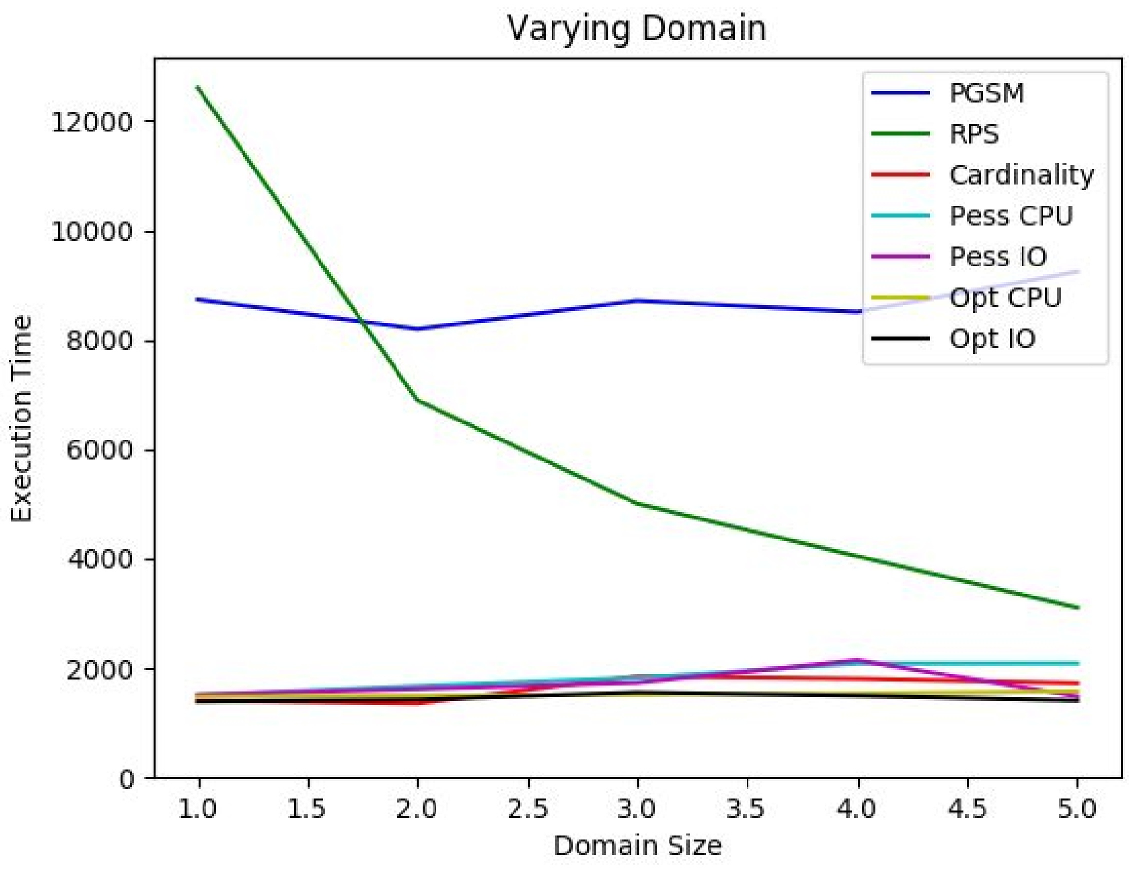


Figure 4.6: Varying Domain Results

Since PGSM isn't reliant on the domain or distribution of the data, PGSM stays at around the same execution time. However, RPS drops dramatically. When the domain size is one, RPS sends all of the values to the same partition. The number of partitions RPS fully utilizes is equal to the domain size. It's perhaps surprising to see that RPS with two partitions is better than PGSM with 12 partitions. In the end, RPS is able to use 6 out of the 12 partitions available for the range of domains considered here.

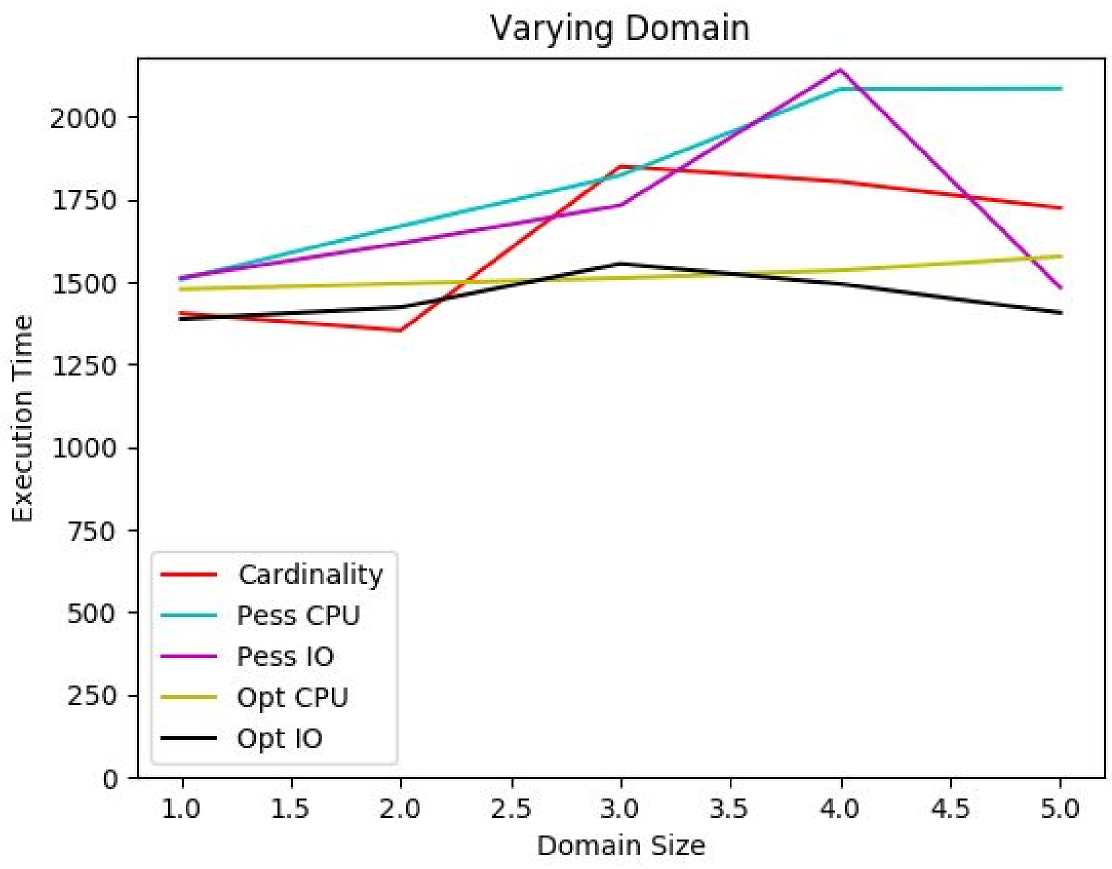


Figure 4.7: Varying Domain Close Up

Taking a zoomed-in look at SRM, in Figure 4.7, it’s evident that the optimistic functions do a lot better than the pessimistic ones. The cardinality function ignores percent split point percentages that are above a certain threshold to protect non-sorting partitions from receiving a small number of values that would force a sort. However, at the domain size of 3 mark, this threshold was broken, and we see an increase in the execution time. For the rest of this experiment, the cardinality method performed as expected. On the pessimistic side, the CPU cost function did a fair bit worse than the IO cost function. However, the optimistic cost functions stayed reasonably even.

### 4.2.5 Worst Case Optimistic

The optimistic cost functions have performed fairly well up to this point. This test is aimed to expose the flaws in the optimistic approach. As explained in section 3.4.7, the optimistic algorithm allows for two non-sorting partitions that are assigned different values to be adjacent to one another. However, in-between values could linger in the dataset and not be represented in the samples. One of these in-between values could get into a non-sorting partition and force it to sort. (Recall that the optimistic approach doesn't provide a guarantee that non-sorting partitions will never have to sort.)  In this test, "Worst Case CPU" and "Worst Case IO" have their sort operator always conduct a sort, even if all the input values are identical.. This experiment was done identically to the varying null experiment. There are 12 partitions which will sort 180M tuples. Figure 4.8 holds the results.

When forced to always sort, the CPU cost function shows as much as a 100% increase in execution while the IO cost function has a slightly smaller increase of 30%. The dramatic spikes in the CPU cost function results are due to an overestimation of how many tuples can be assigned to a non-sorting partition. For the optimistic CPU, the bottleneck (at around the 40% NULL mark) was the non-sorting partition being assigned all of the NULL values. Therefore, when that partition was actually forced to sort, it doubled the overall execution time. The IO cost function predicted the difference in load balancing between non-sorting partitions and sorting partitions more accurately than the CPU cost function. As a result, the optimistic IO cost function wasn't bottlenecked by the non-sorting partition and therefore suffered less penalty when forced to sort.

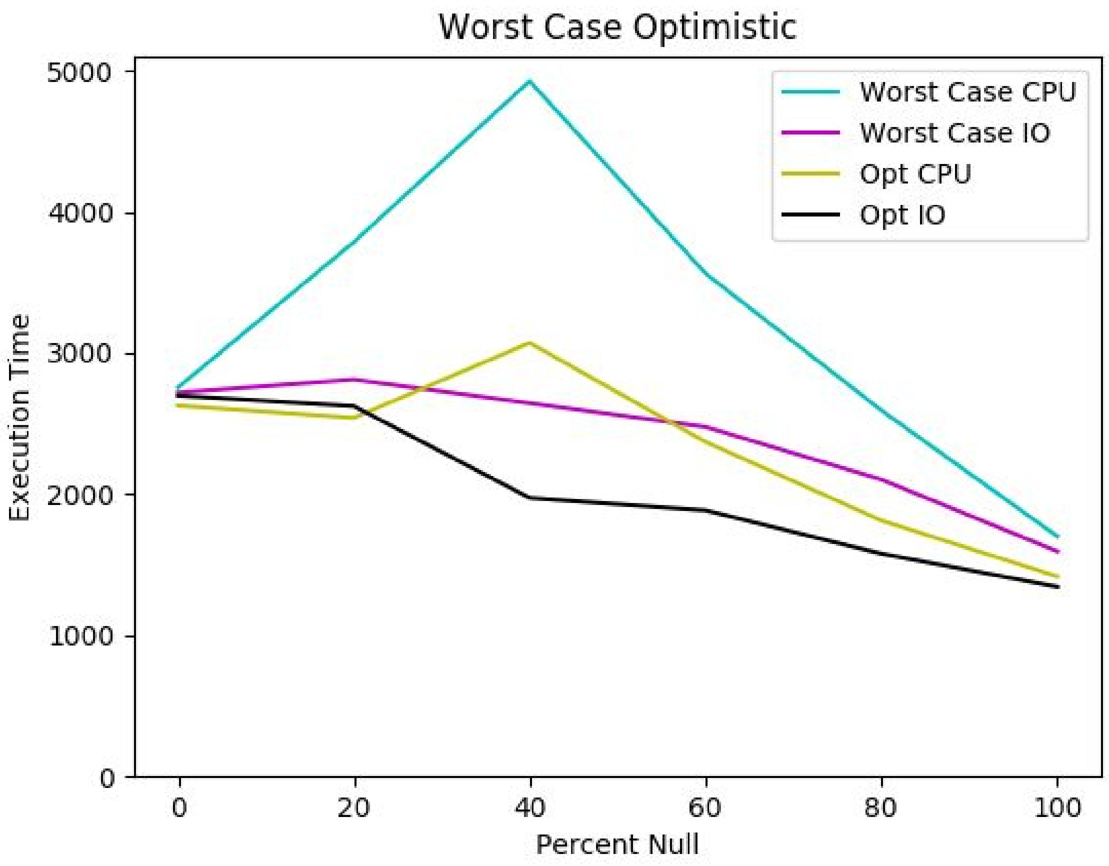


Figure 4.8: Optimistic Control Results

# Chapter 5

# Conclusion and Future work

## 5.1 Conclusion

This thesis has presented an optimized repartitioning parallel sort for skewed data. We started by highlighting “holes” in the current AsterixDB RPS implementation, namely that skewed data values are all sent to the same partition. Then, we discussed the SRM algorithm. SRM accounts for skewed data and assigns values to partitions based upon a work analysis in which partitions receiving one value have the option to not sort their data. We discussed various cost functions which attempt to predict the work difference between a sorting partition and a non-sorting partition. Then, we tested the SRM algorithm, with its various cost functions, against PGSM, RPS, and a basic cardinality distribution approach. It is apparent that our methods provide a better execution time in the use cases we have tested them on so far. The cardinality approach did fairly well in all realistic scenarios, beating out the pessimistic functions in most cases. We believe that more work needs to be done on the cost function for the SRM algorithm in order for it to be a truly viable solution.

## 5.2 Future Work

For SRM to be viable, there needs to be more research done to create an accurate cost function. The IO cost function performed well in all tests except for the speedup test. A cost function could be created that incorporates both the CPU cost and the IO cost of a query. Furthermore, it could be adjusted based upon the observed execution times of previous queries. In a different direction, each partition has to scan the entire dataset to produce the samples before the RangeMap is created. There could be an architecture in place to prevent having to do such an expensive full scan.

# Bibliography

[1] Datagen. https://github.com/shivajah/datagen.

[2] David DeWitt and Jim Gray. 1992. Parallel database systems: the future of high performance database systems. Commun. ACM 35, 6 (June 1992), 85–98..

[3] Alsubaiee, Sattam, et al. "AsterixDB: A scalable, open source BDMS." *Proceedings of the VLDB Endowment* 7.14 (2014): 1905-1916.

[4] Borkar, Vinayak, et al. "Algebricks: a data model-agnostic compiler backend for Big Data languages." *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015.

[5] Borkar, Vinayak, et al. "Hyracks: A flexible and extensible foundation for data-intensive computing." *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 2011.

[6] Alsuliman, Ali. *Optimizing External Parallel Sorting in AsterixDB*. Masters Thesis, Computer Science Department, UC Irvine, 2018.

[7] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. 1991. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In Proceedings of the first international conference on Parallel and distributed information systems (PDIS '91). IEEE Computer Society Press, Washington, DC, USA, 280–291.

[8] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. 1992. Practical Skew Handling in Parallel Joins. In Proceedings of the 18th International Conference on Very Large Data Bases (VLDB '92). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 27–40..

[9] Graefe, Goetz, "Parallel external sorting in Volcano" (1989). CSETech. 194.

[10] Vojnovic, Milan, Fei Xu, and Jingren Zhou. "Sampling based range partition methods for Big Data analytics." *Technical report, Microsoft Research* (2012).

[11] Knuth, Donald E.. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Third Boston: Addison-Wesley, 1997.

[12] Cheelangi, Madhusudan. *Result Distribution in Big Data Systems*. Computer Science Department, UC Irvine, 2013.

[13] Apache Pigmix website. http://cwiki.apache.org/confluence/ display/PIG/PigMix

[14] D. J. DeWitt. The Wisconsin benchmark: Past, present, and future. In J. Gray, editor, The Benchmark Handbook. Morgan Kaufmann, 1993.