UNIVERSITY OF CALIFORNIA,
IRVINE

Optimizing External Parallel Sorting in AsterixDB

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

by

Ali Alsuliman

Thesis Committee:
Professor Michael J. Carey, Chair
Professor Chen Li
Doctor Yingyi Bu

2018

DEDICATION

To my mother whose soul is watching over me wherever I go

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Prof. Michael Carey. I am honoured to have worked with him. Thanks for all the care and support you offered to me. He is indeed one of a kind.

I would like to thank my friends Abdullah Alamoudi and Murtadha Al Hubail for all the support and time they gave me. I could not have done it without them. I would like to thank everyone in AsterixDB's team for their support.

I would like to thank my thesis committee members, Prof. Chen Li and Dr. Yingyi Bu, for reviewing my thesis.

The work reported in this thesis has been supported in part by NSF CNS award 1305430 and industrial gift funding from Facebook, Google, Infosys, and Microsoft.

ABSTRACT OF THE THESIS

Optimizing External Parallel Sorting in AsterixDB

By

Ali Alsuliman

Master of Science in Computer Science

University of California, Irvine, 2018

Professor Michael J. Carey, Chair

In parallel database systems, when a query is evaluated, each operation in the query plan is typically parallelized. This allows a large cluster of machines to process a vast amount of data efficiently since all the machines work simultaneously. AsterixDB, a big data management system, leverages this fact to parallelize operations when executing a query, and sort is one such operation.

This work is an attempt to optimize the sort operation in AsterixDB to make the entire sort process parallel from start to finish without the need for an additional final merge step at the end of the sort operation. We state our problem and highlight some of the issues with the current approach to sorting. We present the architecture and high-level design of the proposed optimization. Then, we describe the implementation details and how the different components interact with each other. We conclude by showing the results of a performance evaluation before and after this optimization.

Chapter 1

Introduction

The concept of parallelism has a long history in the computing world. It has enabled us to do computing in an efficient manner. We see it in hardware where nowadays a typical CPU, for example, would host multiple cores running instructions in parallel. We see it in software where one would write a multi-threaded program to gain performance among other things. We also see it in the database management world where we have parallel database management systems that try to evaluate queries in a parallel fashion [5].

Over the years, data has been growing to unprecedented levels. This led to the birth of Big Database Management Systems (BDMS). Generally, such systems run on a cluster of commodity machines in order to handle the large volume of data. AsterixDB [2] is one example of such a system. In AsterixDB, data lives at many sites across the cluster. The data is hash-partitioned to achieve balance. This fits the parallel execution paradigm very well since an operation in a query plan can be carried out simultaneously at all the sites. Indeed, in AsterixDB, most of the time the operators of an evaluation plan are executed in "partitioned" mode, meaning there are multiple instances of the operator that are running in parallel. However, some operators require an additional serial step in order to produce the correct result. Particularly, an order-by operator in AsterixDB runs in parallel in all the nodes of the cluster where each node sorts its part of the data and produces a partition. However, there is no global ordering across the nodes. Therefore, one single node merges all of those partitions in order to enforce global ordering (this single node's step runs in "unpartitioned" mode, i.e., only one instance is running). This final merge step has been seen to be a bottleneck and wastes some of the benefits gained by sorting the data in parallel [7]. We will call the current sort behaviour "parallel sort global merge" (PSGM). In this thesis, we attempt to optimize the whole sort process by first repartitioning the data across the nodes of the cluster such that there is a global ordering to begin with. After that, all the nodes can proceed to sort their parts in parallel. By doing this, we eliminate the global merge step in exchange for a step with lower overhead, the repartitioning step. We will call this optimized version "repartitioning parallel sort" (RPS). The technique we have implemented to partition the data is simple yet decent enough that we are already observing promising performance gains.

The remainder of this thesis is organized as follows: in Chapter 2, we discuss related work in regard to parallel sorting in the realm of database management systems. In Chapter 3, we present our work to enhance the sort operation and describe its design and implementation. We show the results of an empirical performance evaluation in Chapter 4. In Chapter 5, we conclude and talk about future work.

Chapter 2

Related Work

[7] gives a detailed discussion about past work on parallel external sorting in the literature, and [10] explores parallel algorithms in database management systems in detail including external sorting. Several of the initial works relied on a final merge step to enforce a global ordering after sorting subfiles locally in parallel. This is similar to AsterixDB's current sort, PSGM. Much of the later work that followed was focused on the idea of redistributing the original data so that there is a global ordering to begin with. This removes the need for the final merge. In this approach, a "splitting vector" is employed to determine how to redistribute the data and to which node or processor a tuple should be sent. The work presented in [9] describes an implementation of parallel sorting that utilizes a splitting vector that is supplied by the user. This is similar to an already existing feature of AsterixDB where a user can provide a splitting vector as a hint to make AsterixDB repartition the data first and then sort. For previous algorithms that compute the splitting vector at runtime, there are different variations. Some of the algorithms compute what is called an "exact" splitting vector. In those algorithms, all the data is examined and processed in order to find this exact splitting vector. On the other hand, there are algorithms that compute what is known as an "approximate" splitting vector. The latter use a variety of sampling techniques to

find a splitting vector that could be considered representative of the whole data. According to the evaluation in [7], parallel sorting that uses an approximate splitting vector based on sampling outperforms parallel sorting using an exact splitting vector. Our work is similar in spirit and draws many commonalities. However, there are several points to highlight here. The first point is that in a typical ***sorting*** problem, the multiple input files are presumed to initially reside on different disks, and the sorting task is to read those files and write back to the disks a sorted result whether in one output file or multiple output files. In our work, we are instead addressing an ***order-by*** problem where the input is fed by some other operator in a general query evaluation plan, and the task is to produce a sorted stream to be fed to the next operator consuming the data. As a result, rereading the input after sampling, for example, is not an option. The second point is that we are dealing with a heterogeneous environment, "NoSQL", where a single field might have values of heterogeneous data types. Furthermore, some values may be missing from a tuple.

Chapter 3

Design and Implementation

3.1 Overview of AsterixDB

This section gives a brief background on AsterixDB and serves as an introduction to state our problem and the use case of interest. Figure 3.1 shows an overview of the AsterixDB system architecture. As mentioned before, AsterixDB runs on a cluster of commodity machines. It follows a shared-nothing model where each machine or node has its own disks and memory. In AsterixDB, each dataset is hash-partitioned on the primary key of the dataset across the cluster nodes into what we refer to as partitions, which reside on different disks. There can be one or more partitions on a disk. The cluster is managed by the cluster controller that is the query entry point and handles requests coming into AsterixDB. The node controllers in Figure 3.1 work collectively to process the requests [2].

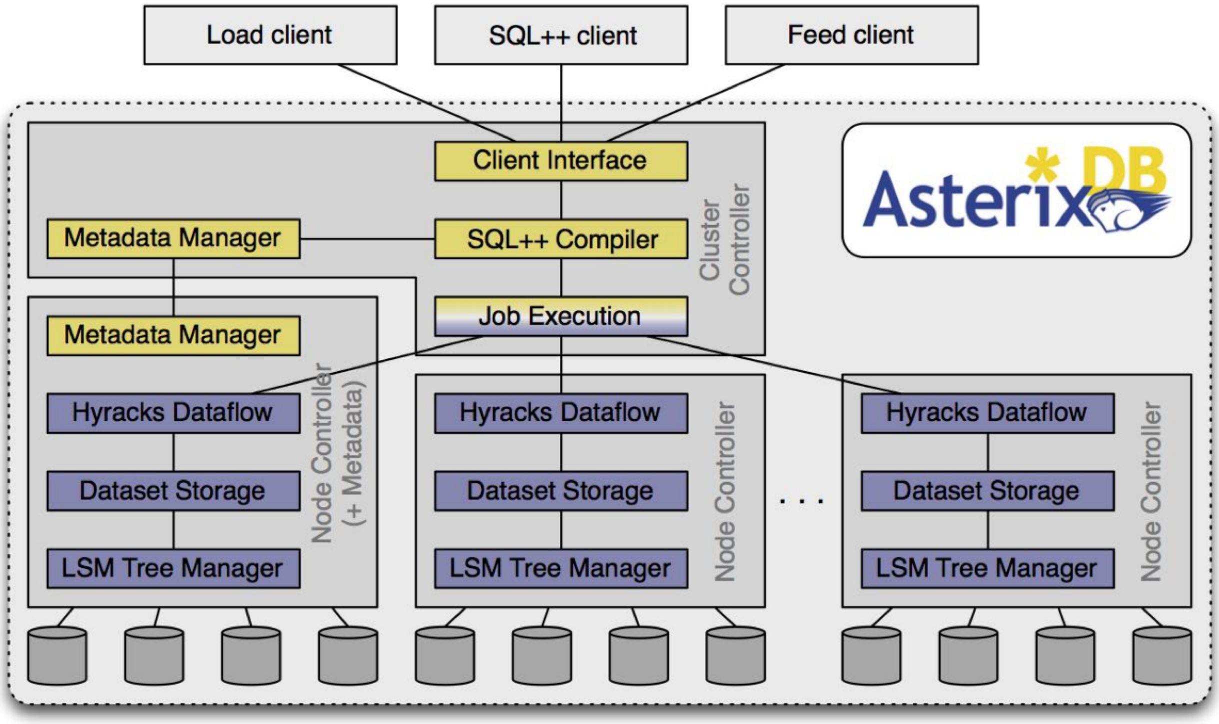

Figure 3.1: AsterixDB system architecture

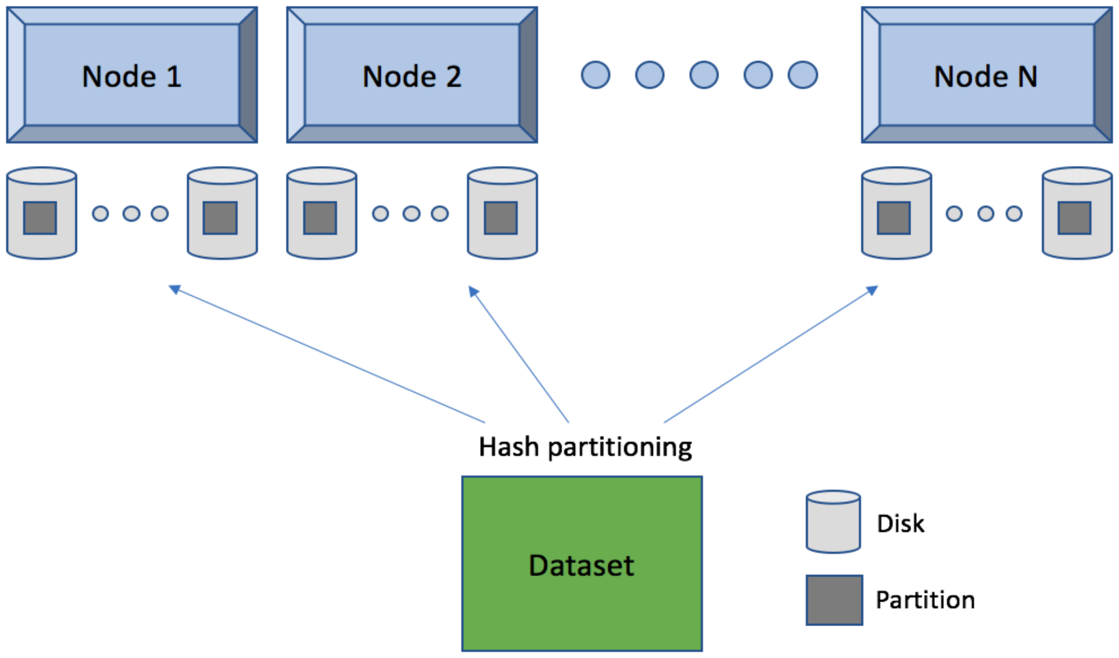

Figure 3.2: Partitioning of a dataset in AsterixDB

When a SQL++ query is submitted to AsterixDB, it goes through different stack layers for processing as illustrated in Figure 3.3. First, the cluster controller receives the query as a string and parses it into a syntax tree. Afterward, the syntax tree is translated into a logical plan for further manipulations. The logical plan goes through several optimizations and rewrites, and the output of this step is a Hyracks job specification [4, 3]. The job specification is expanded into a set of separate stages with operators having activities connected using various types of connectors. The activities themselves are organized as a DAG. Most often, stages have dependencies between them, which means some stages will not be executed until some other stage has finished. The cluster controller submits this final output to the node controllers for execution by the Hyracks runtime engine. Figure 3.4 summerizes the processing flow of a query.

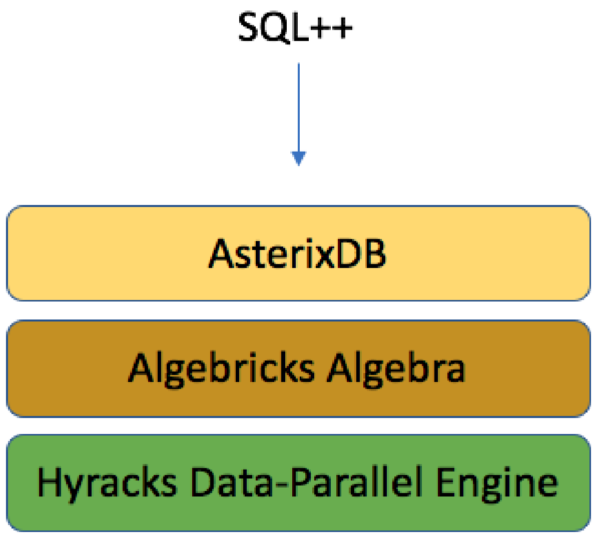

Figure 3.3: AsterixDB software stack layers

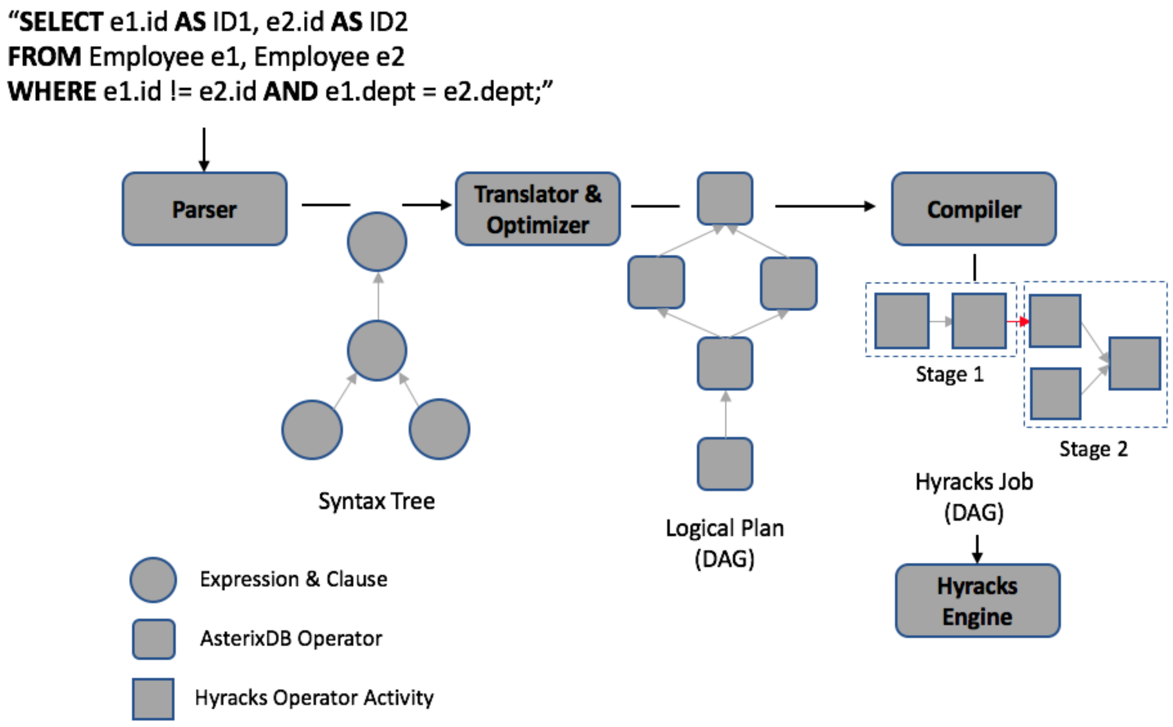

Figure 3.4: Processing flow of a query

AsterixDB follows a *push-based* architecture for query execution. Data flows through the operators (and through their activities) in units of frames. A frame is nothing but a page full of tuples. The flow of execution follows a well-defined sequence. In its simplest form, an activity opens the pipeline and starts pushing frames to the next consuming activity. It may ask the consuming activity to close the pipeline to indicate end of stream or may fail to indicate a runtime error.

3.2 High-level Architecture and Implementation

In this section, we explain the current status of sorting in AsterixDB, state the problem we are dealing with, and present our solution to it. Listing 3.1 shows a simple SQL++ query that retrieves all the tuples of the dataset "Wisconsin" ordered by the integer field "unique2". The definition of the given dataset is provided in Chapter 4 in Listing 4.1.

```
SELECT VALUE w
FROM Wisconsin w
ORDER BY w.unique2;
```

Listing 3.1: SQL++ query

3.2.1 Parallel Sort Global Merge (PSGM)

When the logical plan of the query in Listing 3.1 is compiled, it is turned into two stages that have a dependency between them. Stages are formed when there is an activity that has a blocking edge to another activity. It is at the blocking edges where stages are created and the dependency between them is established. In the case of sorting, when the first stage is executed, each partition scans its data and generates sorted runs. After that, the second stage starts, in which each partition merges their initial sorted runs down into one run. For very large inputs, this may be a multi-pass process [10]. While each partition is merging and writing their resulting single run, they push their output over the network to a single node controller, where a final merge from all the partitions occur and the final sorted result is produced. Figures 3.5 and 3.6 depict the whole process and outline the activities involved and the sequence of execution. (We omit some details that are not important to our discussion.) Note that we are dealing with an order-by problem as pointed out before. The input to the sort-related activities are not necessarily files sitting on disk, and the output may not necessarily be written back to disk. That is, the input could come from a subquery, for example, and the output might be consumed by other activities before producing the final query result. Listing 3.2 shows one such query borrowed from TCP-H (query 5) and written in SQL++.

The issue with the execution plan in Figure 3.6 lies in the final merge step [7]. It greatly impacts the performance and slows down the sort process. This final merge step is required since we started with a dataset that has been hash-partitioned, and therefore there is no

```
SELECT o1.n_name AS n_name,
  sum(o1.l_extendedprice * (1 - o1.l_discount)) AS revenue
FROM Customer c JOIN (
  SELECT l1.n_name AS n_name, l1.l_extendedprice AS l_extendedprice,
    l1.l_discount AS l_discount, l1.s_nationkey AS s_nationkey,
    o.o_custkey AS o_custkey
  FROM  Orders o JOIN (
    SELECT s1.n_name AS n_name, l.l_extendedprice AS l_extendedprice,
      l.l_discount AS l_discount, l.l_orderkey AS l_orderkey,
      s1.s_nationkey AS s_nationkey
    FROM  LineItem l JOIN (
      SELECT  n1.n_name AS n_name, s.s_suppkey AS s_suppkey,
        s.s_nationkey AS s_nationkey
      FROM Supplier s JOIN (
        SELECT n.n_name AS n_name, n.n_nationkey AS n_nationkey
        FROM Nation n JOIN Region r
        ON n.n_regionkey = r.r_regionkey AND r.r_name = 'ASIA'
      ) AS n1 ON s.s_nationkey = n1.n_nationkey
    ) AS s1 ON l.l_suppkey = s1.s_suppkey
  ) l1 ON l1.l_orderkey = o.o_orderkey
      AND o.o_orderdate >= '1994-01-01'
      AND o.o_orderdate < '1995-01-01'
) o1
ON c.c_nationkey = o1.s_nationkey AND c.c_custkey = o1.o_custkey
GROUP BY o1.n_name
ORDER BY revenue DESC;
```

Listing 3.2: SQL++ query involving “order by”

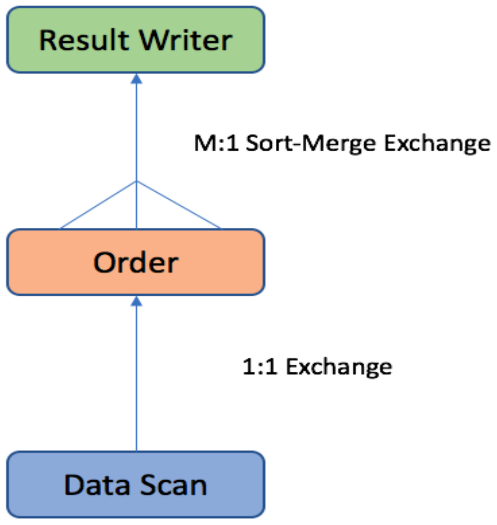

Figure 3.5: PSGM logical plan

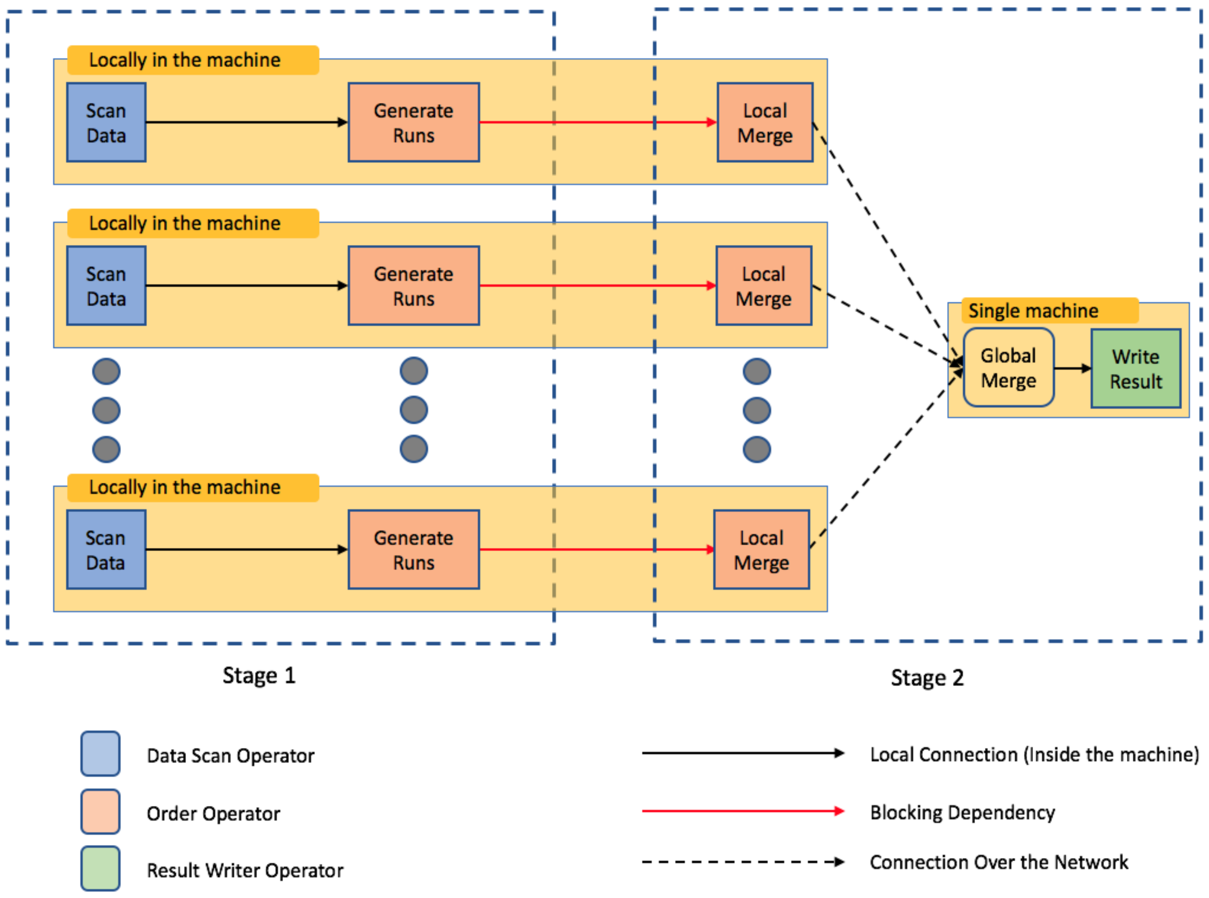

Figure 3.6: PSGM activity graph

global ordering among the initial partitions with respect to the sort keys. A solution to this is to redistribute the data first based on the sort keys before doing any sort so that there is a global partition ordering. The next subsection describes our application of this solution in more detail.

3.2.2 Repartitioning Parallel Sort (RPS)

Figures 3.7 and 3.8 show the new logical plan and the corresponding activity graph. We will explain the role of each component and provide a close-up view of the interaction between them.

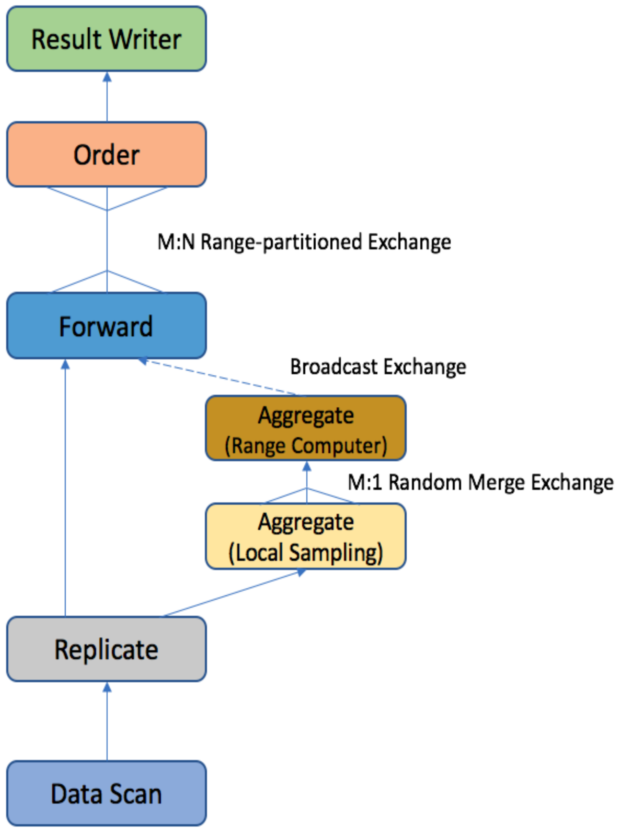
Result Writer
Order
M:N Range-partitioned Exchange
Forward
Broadcast Exchange
Aggregate (Range Computer)
M:1 Random Merge Exchange
Aggregate (Local Sampling)
Replicate
Data Scan

Figure 3.7: RPS logical plan

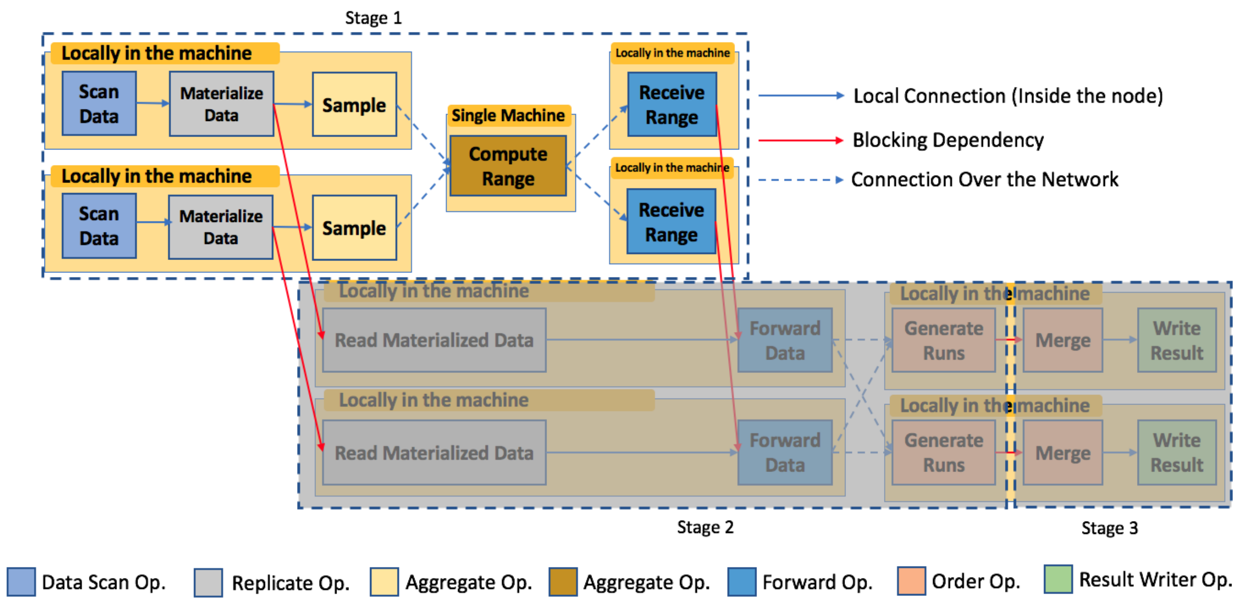

Figure 3.8: RPS activity graph: stage 1 execution

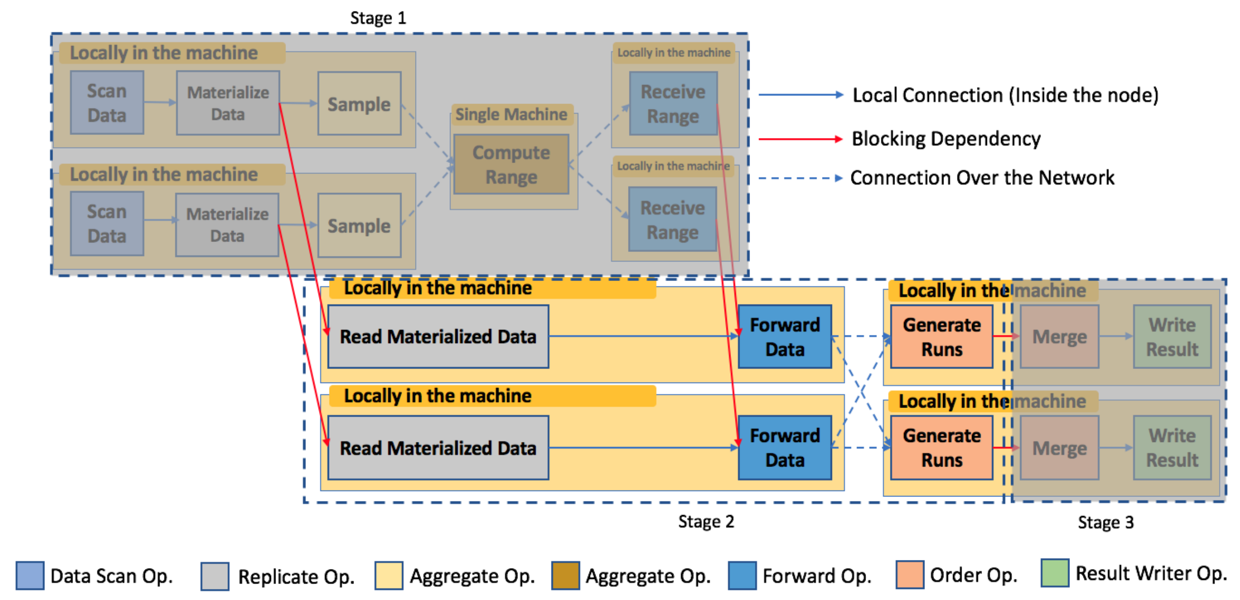

Figure 3.9: RPS activity graph: stage 2 execution

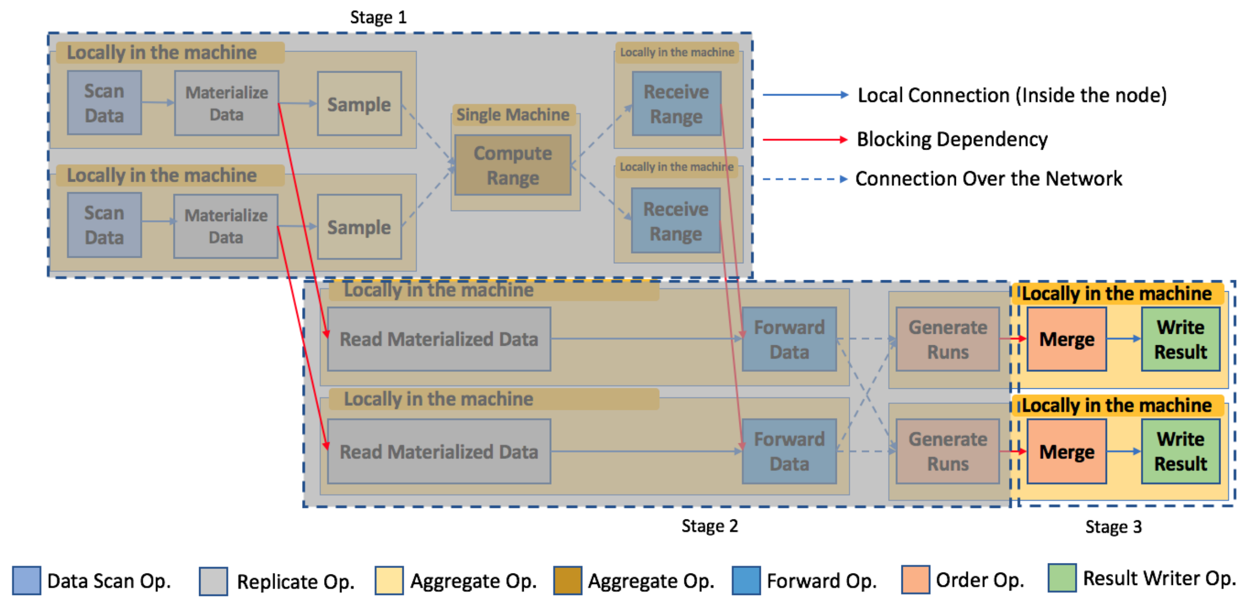

Figure 3.10: RPS activity graph: stage 3 execution

Sampling Function

One major task when seeking to redistribute and range-partition the data before doing the actual sort is determining the proper splitting vector. We employ essentially the same technique used in [7]. In our implementation, the whole process is divided into two steps, a local step and a global step. In the local step, as each partition receives input data, it picks the first 100 tuples and considers them as samples. Each partition sends its 100 samples over the network to a small global step task ("Compute Range" in Figure 3.8). The global step takes the samples from all the partitions and sorts them (ascending or descending based on the query). It then divides this set of values into equal ranges and picks k - 1 values, where k is the number of target partitions. The output of this entire process is the splitting vector or what we call in our implementation a "RangeMap" object. The splitting values in this RangeMap object are serialized and stored consecutively in a byte array. The RangeMap object also contains another integer array that specifies the offset of each splitting value. Figure 3.11 shows how it is represented.

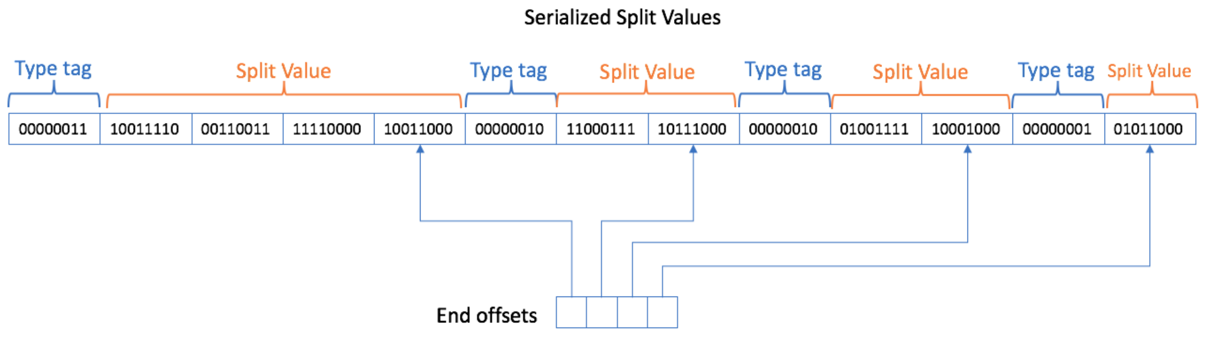

Figure 3.11: RangeMap representation

Note that the RangeMap stores the type of a split value in addition the actual value. This enables us to handle fields of heterogeneous types because AsterixDB's generic comparator has the ability to compare values of different types. It will compare the values if it finds that the types of the values match; it will promote and demote the values if the types do not match but are compatible (e.g., integer and float); it will resort to comparing the types themselves if they are not compatible. In AsterixDB, type tags are comparable for the purposes of sorting and grouping. It is worth mentioning that the resulting RangeMap may contain multiple identical split values. This indicates that such a value is so prevalent in the dataset that sampling yields several instances of it, as the RangeMap serves as a representative of the distribution of the values in a dataset. This effect manifests itself in one of our experiments in which the dataset distribution is non-uniform.

The process of combining the samples and producing the RangeMap resembles aggregation operations such as SUM or AVG. Therefore, we chose to use the aggregate operator abstraction and framework that are already available in AsterixDB and supply our sampling functions as the functions to be applied. There are several advantages to this approach besides just reusing code. One advantage is that we get the communication setup between the local step and global step for free since it is a part of the implementation of aggregate operators. All we need to do is just designate which function is the local function and which is the global one. Also, there are many rewrite and optimization rules for aggregate operators, and we get those for free as well. Finally, the global step serializes and broadcasts its computed RangeMap object to all of the participating partitions. After reading it, they can start redistributing their part of the data.

Replicate Operator

The use of AsterixDB's replicate operator in the execution plan in Figure 3.7 is important. The replicate operator's job is to take an input and route it to multiple branches. This is needed since we want to send the input frames to both the sampling function and to the partitioner that is going to partition the data in the frames. However, we cannot send the input frames to the partitioner yet since the partitioner first needs to know the splitting vector produced by the sampling branch. This is solved by enabling materialization on that specific replicate operator branch leading to the partitioner (leading to the forward data activity, in fact, and then to the partitioner), which will create a blocking edge and mark a new stage (stage 2). Hence, those replicated input frames will not be pushed until the sampling branch has finished.

Forward Operator

The forward operator is used for several purposes. First, it marks the end of stage 1 of the plan and defines proper stage dependencies. Second, upon receiving the RangeMap object from the sampling function, it communicates it (locally) to the partitioner and then starts pushing the materialized data to the partitioner. The forward operator consists of two activities. The first activity's job is to receive the RangeMap object and hand it to the next activity. We make the edge between them blocking so that the second activity forwarding the tuples waits for the sampling function to finish. Next, the second activity of the forward operator opens the pipeline and passes the RangeMap object to the partitioner. We exploit the fact that the second activity of the forward operator and the partitioner belong to the same Hyracks task. There exists a map in the Hyracks task, which is shared among these activities. We use this shared map for storing (communicating) the RangeMap object using a unique UUID that is already known beforehand by both of them. (A Hyracks task is just a thread that executes several activities that are connected directly to each other, which indicates that the connection is not over the network.) The partitioner looks up the shared map using the UUID when the pipeline is opened. After that, the forward operator starts forwarding the data. The "MtoN" partitioner receives the data and distributes it across the cluster nodes according to the RangeMap's contents. The rest of the execution is the typical sort process, but now with the final merge removed from the plan.

Range-Based Partitioner

As mentioned before, AsterixDB already supports a feature that allows a user to supply a splitting vector as a hint in the query. This will trigger another path for PSGM where the final merge is removed. Therefore, AsterixDB already has a partitioner (or connector) that can partition the data based on a range, but that range has to be known during compilation when constructing the connector. We modified this connector so that it receives this information during runtime. Now, when constructed, the range-based connector accepts a "key" that should be used to look up the splitting vector. This key is the same UUID used by the forward operator to pass the RangeMap object.

Optimization Rule

We have talked about the details of both the new logical plan and its corresponding execution plan for RPS, but we have not yet talked about how to produce it. In order for us to accomplish this, we need the query optimizer to produce our desired plan when optimizing a logical plan. It starts when parsing and translating a query. When a query is parsed and translated into a logical plan, we annotate the order-by operators so that the optimizer knows that it needs to optimize the order-by operator. This gives the user flexibility to choose between PSGM and RPS. By default, order-by operators will now be annotated to use RPS. If a user would still like to use PSGM, they can provide a hint in the query that will choose PSGM. (An example is provided in Chapter 4.) The logical plan goes through two phases of optimization, a logical optimization and a physical optimization. We added a new optimization rule to be fired during the physical optimization phase. This new rule is called "IntroduceRPSRule", which checks for several conditions first to make sure the given order-by operator is a candidate for optimization. For example, it considers only those operators that have been annotated to use RPS, and it only considers those operators that are executing in a "partitioned" mode. (In a "local" mode, a sort is only done locally to feed some other operator. In this case, the final merge step is not present in the plan, and therefore, RPS is not applicable.)

After checking that RPS applies, the optimizer modifies the logical plan to look like the one in Figure 3.7. This rule also generates the UUID used by the forward operator and the partitioner to transfer the splitting vector. A UUID, short for Universal Unique Identifier, is defined by a canonical format using hexadecimal text with inserted hyphen characters. “5a28ce1e-6a74-4201-9e8f-683256e5706f” is an example. This guarantees that if we have multiple order-by operators in one plan, each operator will have its own UUID. This resolves any conflict that might happen when passing splitting vectors through the shared map as described before. Since the rule uses aggregate operators in the sampling branch, it is oblivious to the sampling function being used. This means that we can utilize various sampling implementations without the need to modify the rule. We just need to provide the rule with the desired sampling function, and it will put it inside the aggregate operators.

The new optimization rule works hand in hand with another AsterixDB optimization rule called the “EnforceStructuralPropertiesRule”. The “IntroduceRPSRule” is fired just before the “EnforceStructuralPropertiesRule”. As the name suggests, the “EnforceStructuralPropertiesRule” is responsible for, among other things, making sure that all the connections between the different operators are properly set up as required by each operator. It introduces all kinds of connectors to satisfy the requirements of each operator. We modified the requirements of the order-by operator so that it requires the data to be range-partitioned. By doing this, this rule will automatically introduce the range-based partitioner we need and place it between the order-by operator and the forward operator as shown in Figure 3.7.

Alternatively, with small modifications, we could integrate the logic of the new optimization rule into the “EnforceStructuralPropertiesRule”. In this way, all of the plan modifications happen at the same time including inserting the replicate operator, the forward operator, the aggregate operators, and the range-based partitioner.

There are several differences, commonalities, and tradeoffs between “parallel sort global merge” (PSGM) and “repartitioning parallel sort” (RPS). Both strategies parallelize part of the sort process. However, PSGM needs one additional step to finish off its sort, while RPS requires re-distribution of data to start off. In RPS, we incur the overhead of sampling and materializing the data. There is also the potential overhead of high volume communication over the network as every partition sends data to every partition. Moreover, data skew can come into the picture with sampling and redistribution, and can degrade the performance if skew is not handled properly [8]. On the other hand, in PSGM, everything proceeds locally in each partition until the final step where the partitions need to communicate their locally sorted results to one single place. However, the overhead of this final merge can be remarkably high, which makes RPS seem a better alternative. Our performance evaluation will give more insights into these tradeoffs.

Chapter 4

Performance Evaluation

In this chapter, we share our experiments and results. We evaluate the performance of RPS under different environment and problem settings. We compare and analyze the performance of PSGM and RPS and show the improvement that we gain.

4.1 Experimental Environment and Setup

For the majority of our experiments, we used a cluster of 6 nodes. We varied the number of nodes in the cluster for some of the experiments, but the node specifications and AsterixDB instance configuration were the same for all experiments. Each node was a Dell PowerEdge 1435SC with 2 Opteron 2212HE processors with a total of 4 cores. Each node had a memory of 8GB DDR2. There were two disks attached to each node. Each disk was a 1 TB drive with 7200 RPM speed. AsterixDB was configured to use both of the I/O devices in each node and to store one partition per disk. This gave us a total of 12 partitions when we use 6 nodes. The AsterixDB sort memory budget was set to 32MB. The JVM running in each node and hosting AsterixDB was given 4GB of RAM memory. We used the same schema as the one used in the Wisconsin Benchmark [6]. We made just a small modification to generate tuples of size approximately 600 bytes [1]. Listing 4.1 shows the resulting SQL++ DDL statements.

```
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: Dataset definition for experiments

Table 4.1 shows the different sizes and cardinalities of the datasets used throughout the experiments.

Number of Tuples (Million)	Total Dataset Size
12	6 GB
36	19 GB
72	38 GB
108	57 GB
144	76 GB
180	94 GB

Table 4.1: Datasets sizes for experiments

The datasets were hash-partitioned on the primary key, *unique1*. For all our experiments, the sort key was *unique2*. Listing 4.2 contains our running queries.

```
// RPS query
SELECT VALUE w
FROM Wisconsin w
ORDER BY w.unique2;

// PSGM query
SELECT VALUE w
FROM Wisconsin w
/*+ no-rps-sort */
ORDER BY w.unique2;

// an example of a query that provides a range
SELECT VALUE w
FROM Wisconsin w
/*+ range [6000000, 12000000, 18000000]; */
ORDER BY w.unique2;
```

Listing 4.2: SQL++ queries for experiments

4.2 Results and Analysis

4.2.1 Speedup Experiment

In this experiment, we measured the speedup of both RPS and PSGM. The total dataset size was 19GB. We fixed the dataset size parameter and increased the number of partitions for each subsequent measurement. We started with a setting of 4 partitions. We then incremented it to 2x, 3x, 4x, and finally 5x. In an ideal parallel system, one would expect that if we double the hardware, the same problem can be solved in half of the original time. That is, one would expect to observe a linear speedup. In practice, if the execution is truly parallelized, it shows a trend close to the ideal situation. Indeed, the results in Figure 4.1 show that RPS exhibited this trend as expected. Of course, there are factors that affect the speedup [5]. For example, in our case, we incur some overhead due to network communications as we redistribute the tuples. There is also the sampling cost that we have to pay and the materialization cost that comes with it in our current implementation. However, there are also other factors from which we gain benefits; these are visible at the far right in the graph where the speedup is a bit higher than 5. The reason for that is that the partition size was so small that all the data fit in memory, which saved I/O. In comparison, Figure 4.1 shows that PSGM did not demonstrate signs of speedup. This is attributed to the fact that the final serial merge step dominated the execution time, and here we had to do the final serial merge in all cases. The cost is roughly the same for each case since we were merging the same amount of data. Figure 4.2 shows a performance comparison between the two approaches.

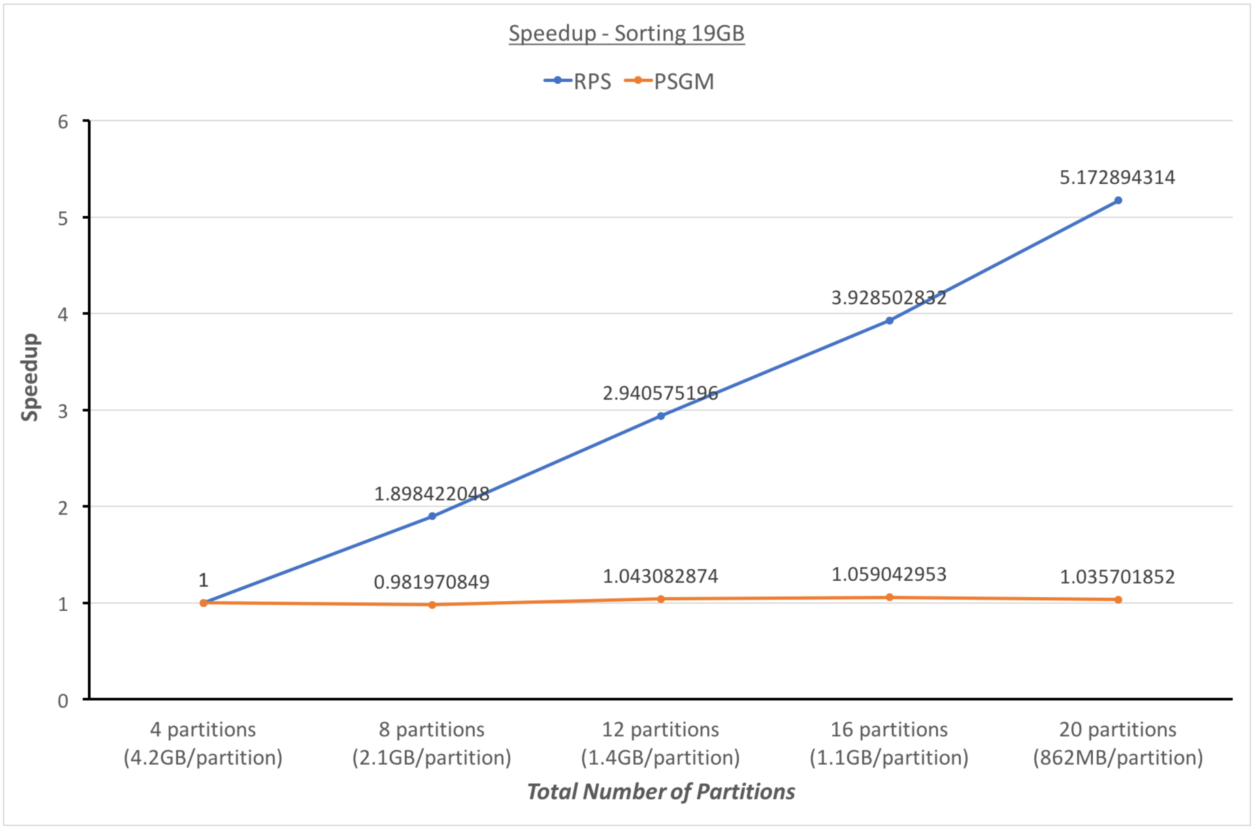

Figure 4.1: Speedup

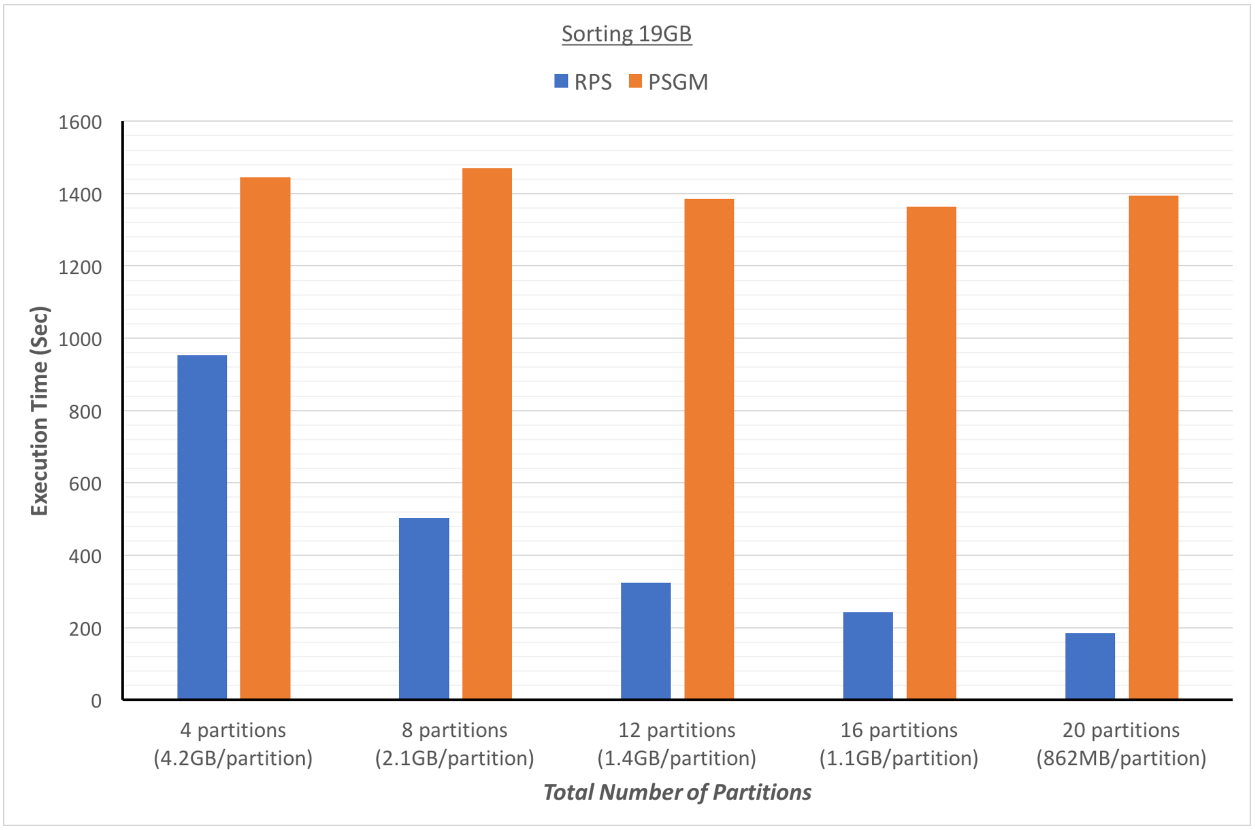

Figure 4.2: Performance comparison of different cases of the speedup experiment

4.2.2 Scaleup Experiment

Scaleup is another metric that we are interested in. In this experiment, we started with a dataset of size 19GB and a cluster 4 partitions. We changed the problem size as well as the number of partitions by the same factor. We increased both by factors of 2x, 3x, 4x, and finally 5x. In a scaleup experiment, the curve is expected to be flat with a ratio value of 1 for the old execution time and new execution time. Figure 4.3 gives us a sense that RPS scales very nicely given the factors previously mentioned in the speedup experiment. This is not the case for PSGM, however, as seen in the same figure. As we increase the problem size, the final merge will suffer no matter how many partitions we add to the cluster. This is due to the fact that there is only a single node performing the final merge. This again shows that PSGM's execution time is mostly determined by the final serial merge step.

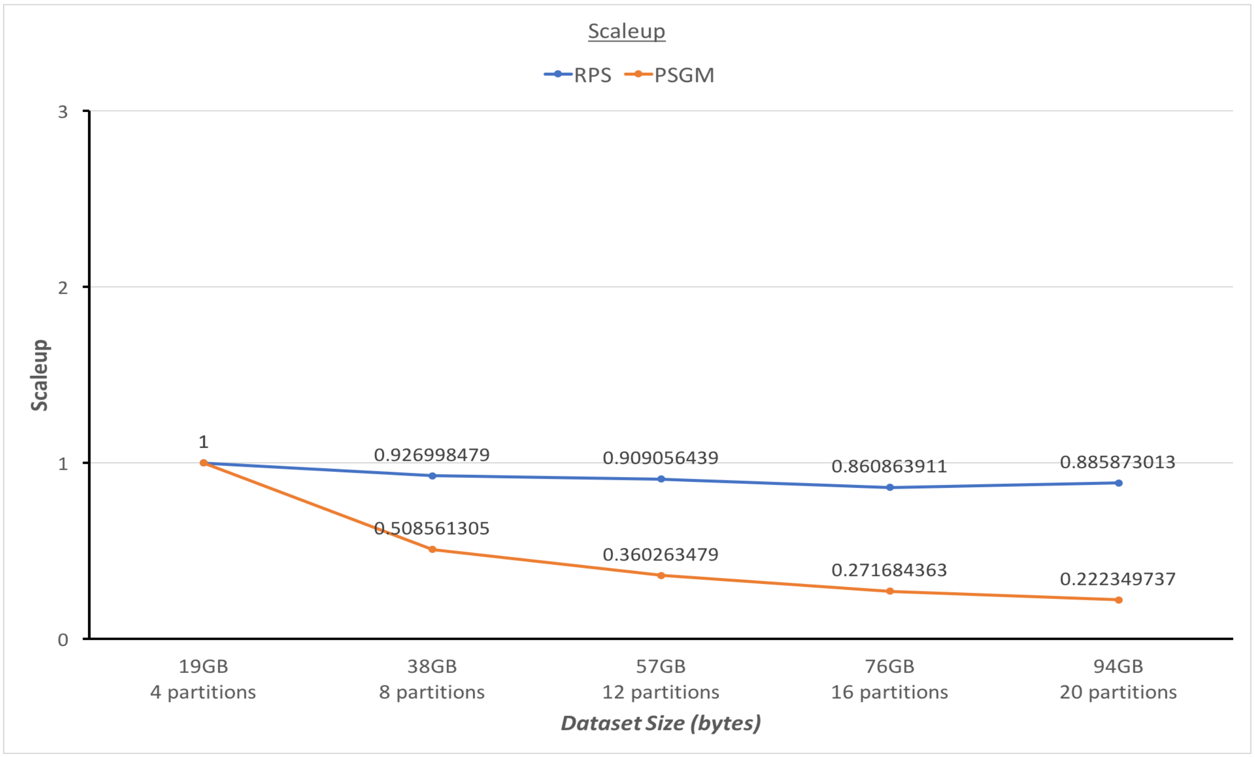

Figure 4.3: Scaleup

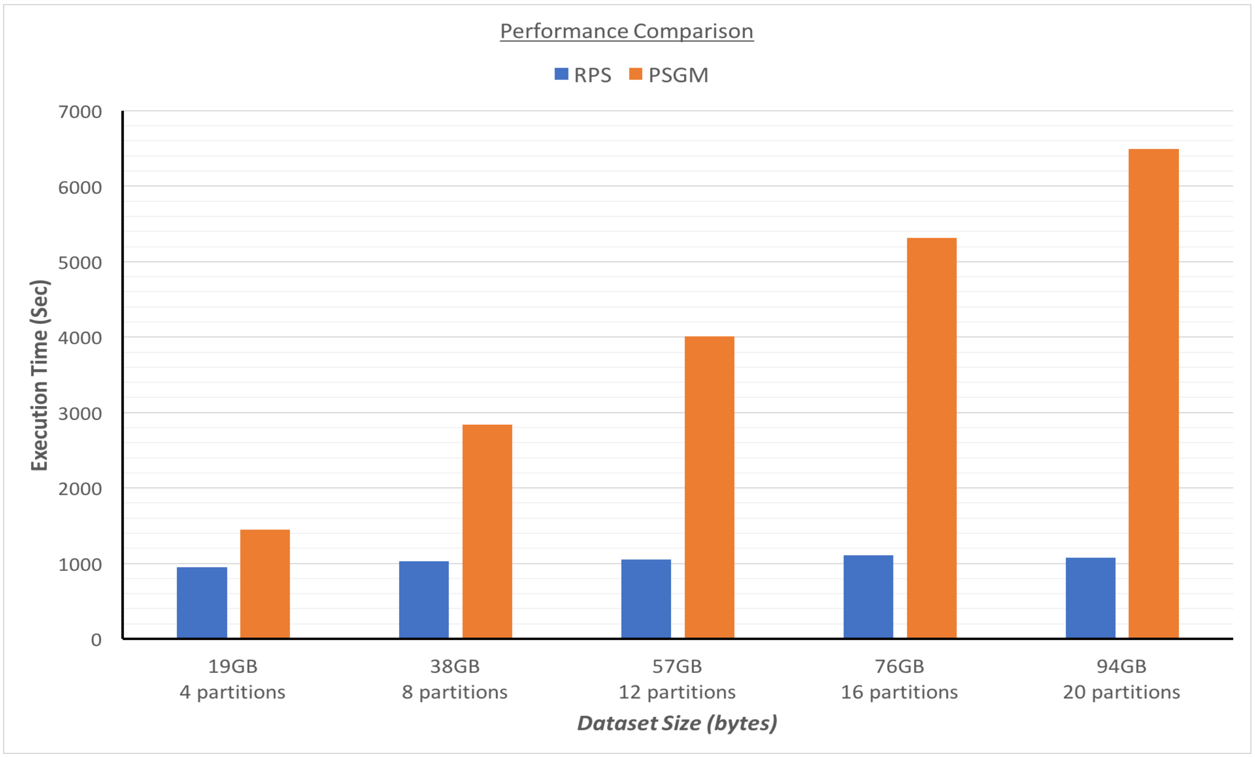

Figure 4.4: Performance comparison of different cases of the scaleup experiment

4.2.3 Improvement Experiment

In this experiment, we try to measure the execution time of both approaches and compare how they perform. The new RPS approach uses sampling to determine the splitting vector; we talked about the sampling technique in Section 3.2. Although the field values of the sort key “unique2” did not have skew, the data ended up not being redistributed evenly across the partitions with our current implementation. One partition can receive fewer or more tuples than it would if the splitting vector was exact. To explore the implications of this, we also wanted to measure the execution time when the partitioner was able to distribute the data evenly. To that end, we also ran the same query but supplied the exact splitting vector ourselves in the query as a hint. We will call this case the “balanced” RPS. We used a cluster of 6 nodes for a total of 12 partitions. Figure 4.5 shows the execution time of the three versions, namely balanced RPS, RPS, and PSGM. The results are promising and show the potential of RPS: the query execution time was significantly reduced by a factor of 3-5 fold as compared to PSGM.

4.2.4 Skewed Data Experiment

Data skew is one of the “threats” to parallelism [5]. Special handling of data skew must exist whenever partitioning comes into the picture. If not handled properly, the performance of a parallel execution of an algorithm will degrade. In this experiment, we want to see the impact of skew, as our current implementation does not handle skew as well as desired. The dataset size was 38 GB, and the cluster setup was our usual one, 6 nodes with 12 partitions. For each run of the experiment, we had a different dataset setting. The first one operated on a regular dataset with no skew. In the second run, we set 20% of the sort column values to be null to give us some skew. We further increased it to 40% and 60% in the third and fourth run, respectively. We ran the same query for both RPS and PSGM.

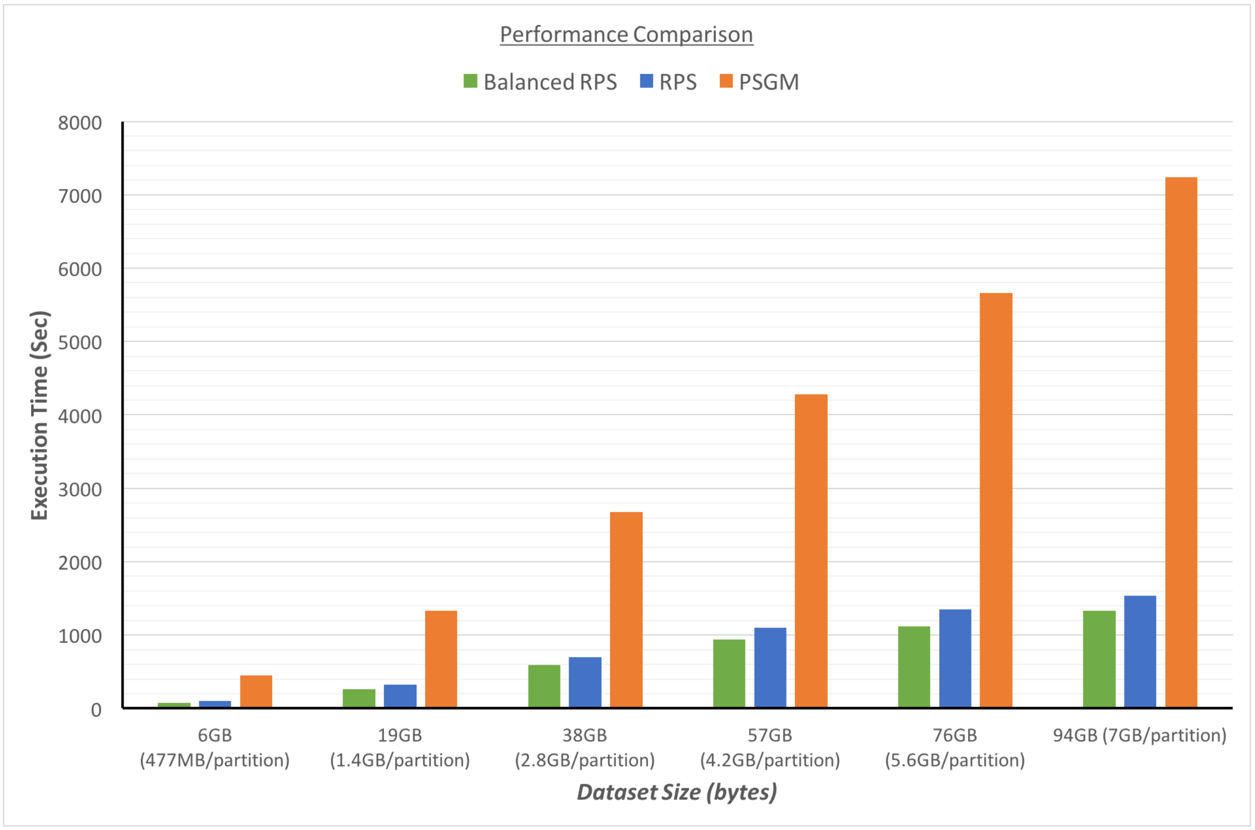

Figure 4.5: RPS and PSGM performance comparison

As shown in Figure 4.6, we clearly see that skew was taking its toll on RPS's performance. This is something we expected since the whole execution time will be determined by the overloaded nodes that receive more data than others. Figure 4.7 shows how many sorted runs each partition generated relative to each other, which gives us a sense of how skew affects RPS. For example, for the 20% null setting, we see that partitions 1 and 2 generated zero runs, which put more pressure on partition 3. This can be explained by looking at Figure 4.8, which shows the splitting vector computed for that case. First, we notice that the splitting vector contained a few identical split values, namely null. This is because null values constituted 20% of the dataset; many null values made their way to the RangeMap during sampling, which increased the likelihood that several of them were picked as split values. In this case, the first two split values in that splitting vector were null. This means that P1 (partition 1) and P2 will receive all values that are less than null (only *missing* values are less than null). Null values will end up in P3 together with other values that are less than the split value “x3”. The same phenomenon occurred, only more so, for the other two cases (40% null and 60% null). In the PSGM case, the effect did not arise since PSGM does not redistribute data. This is because PSGM starts with each partition performing a sort of its part of the data, and each partition has approximately an equal share to sort since the incoming data is hash-partitioned. The interesting point, however, is that even with skew RPS outperformed PSGM. RPS essentially degrades to PSGM as we greatly increase the degree of skew.

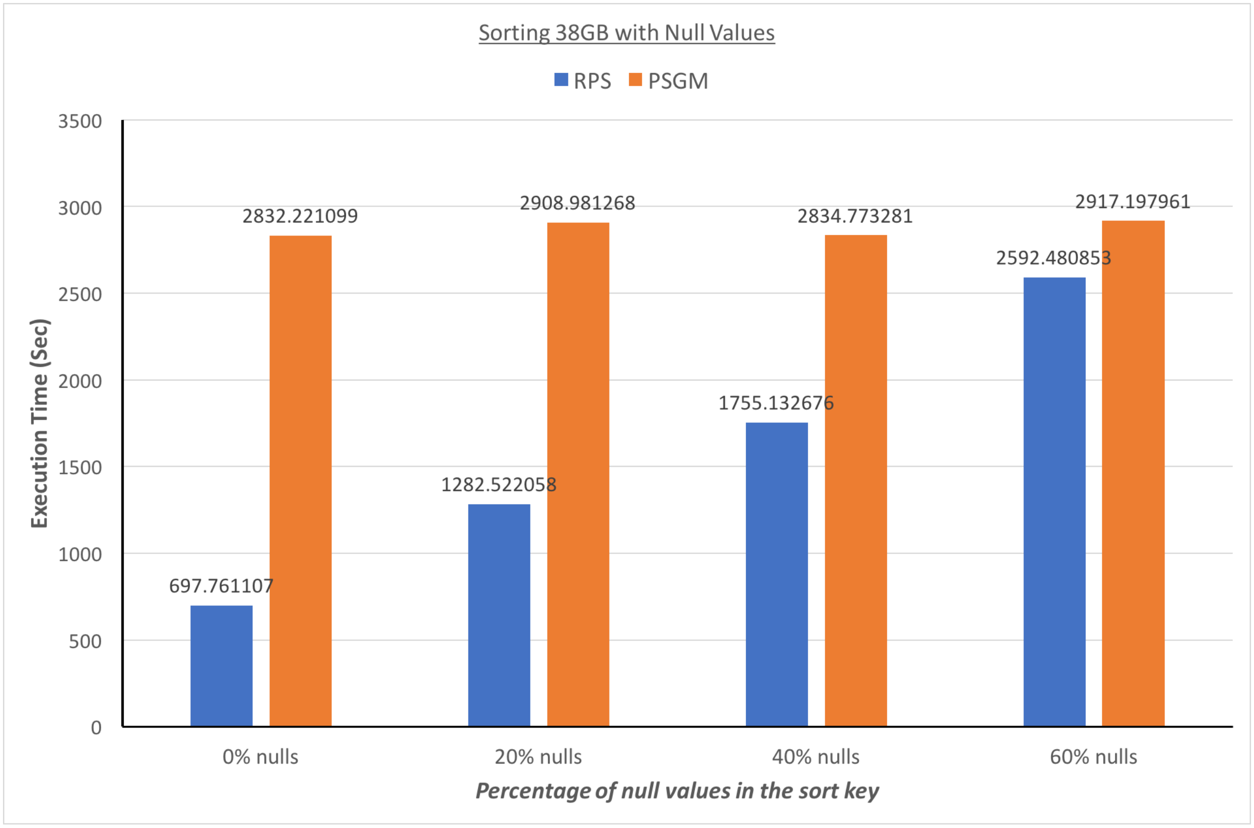

Figure 4.6: Skewed-data experiment

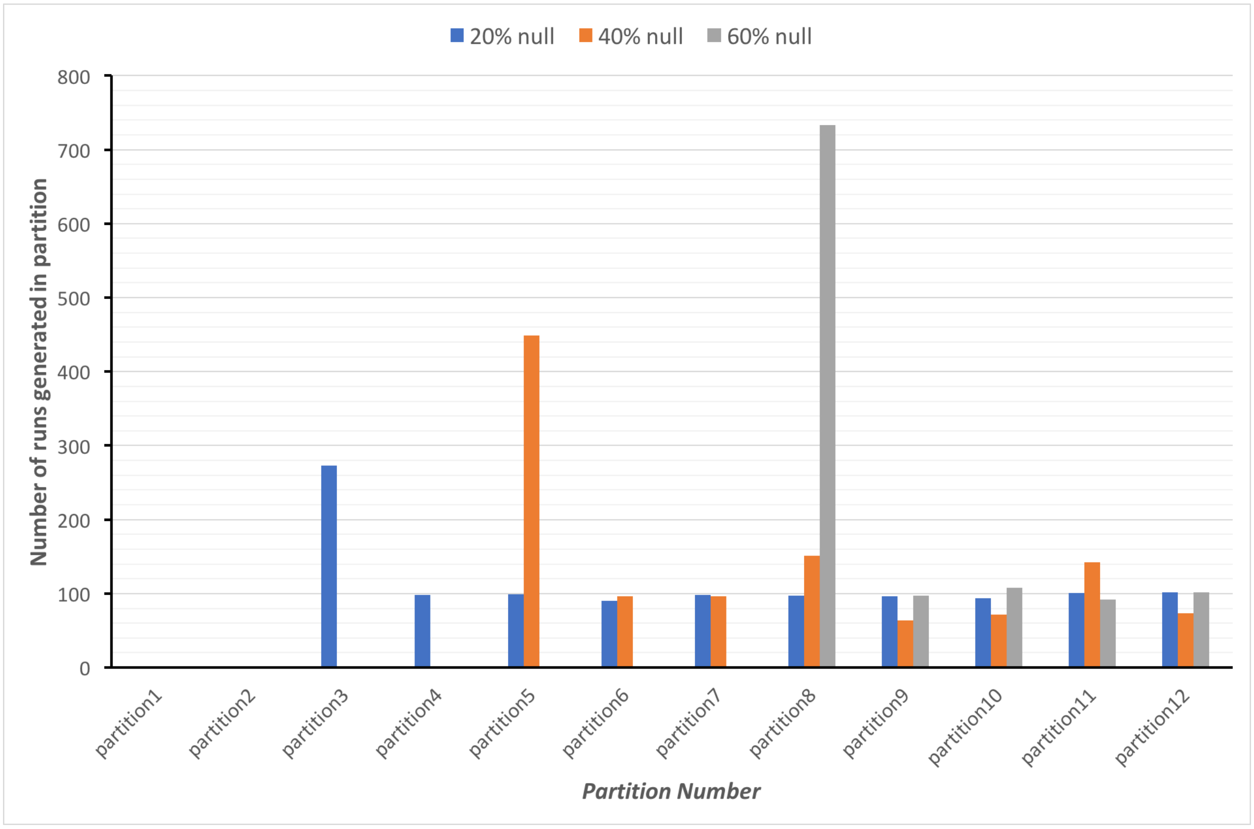

Figure 4.7: Data distribution across partitions

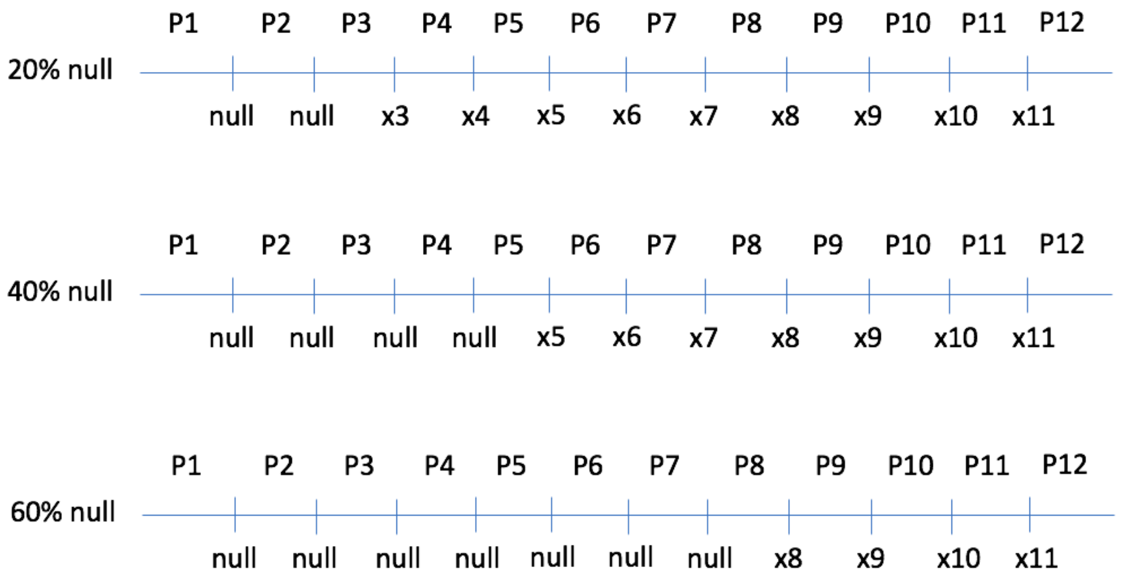
20% null
P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11 P12
null null x3 x4 x5 x6 x7 x8 x9 x10 x11
40% null
P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11 P12
null null null null x5 x6 x7 x8 x9 x10 x11
60% null
P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11 P12
null null null null null null null x8 x9 x10 x11

Figure 4.8: Splitting vectors for each experiment setting

4.2.5 Varied Number of Samples Experiment

Recall that our sampling function takes the first 100 tuples of each partition and forms a splitting vector out of them. That number gives us a decent redistribution. In the last experiment, we are interested in seeing how even the partitioning will be as we increase the number of samples. Furthermore, we want to see how the execution time varies since taking more samples will result in a better splitting vector but at the same time increase the cost of sampling. We used a dataset of size 38 GB and a cluster of 12 partitions. We measure the evenness of a particular partitioning by taking the difference between the number of runs generated in RPS and the number of runs generated in the balanced RPS. (Balanced RPS gives an even distribution since we supply the splitting vector, and we know the exact ranges.) In the charts in Figures 4.9 through 4.14, the horizontal axis indicates the partition number. A value of "2" in the vertical axis means that the corresponding partition generated 2 more runs than it would in balanced RPS. Similarly, a value of "-3" means that the corresponding partition generated 3 fewer runs than it would in balanced RPS. (In other words, it means that the partition received either more data or less data.) This approach of measuring evenness is, of course, not accurate, but is sufficient to give us useful insights. We can see from the charts that as we increased the number of samples, the partitioning started to become more even. For example, in Figures 4.9 and 4.10, all the partitions either received more or less data than in balanced RPS. When we increased the number of samples, now several partitions received the same amount of data as in balanced RPS, and the partitioning of data became more and more even as seen in Figures 4.11 and 4.12. Finally, Figure 4.14 shows that raising the number of samples to an even higher number yielded a partitioning similar to balanced RPS. As for the impact of sampling on the query execution time of RPS, we see a tradeoff in Figure 4.15. With more samples, we get a more even partitioning, but the cost of sampling starts to kick in and increase the execution time; with fewer examples, the sampling cost is less but at the expense of a less even partitioning.

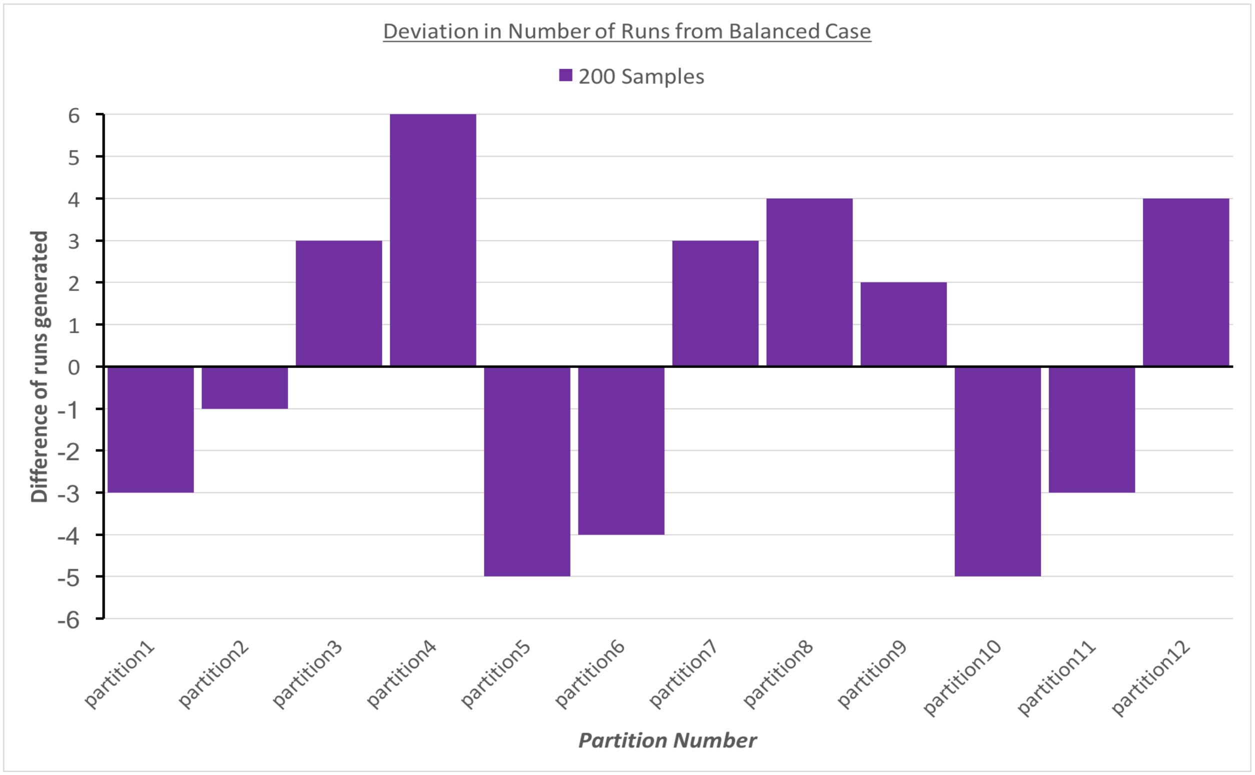

Figure 4.9: Number of runs generated by partitions taking 200 samples

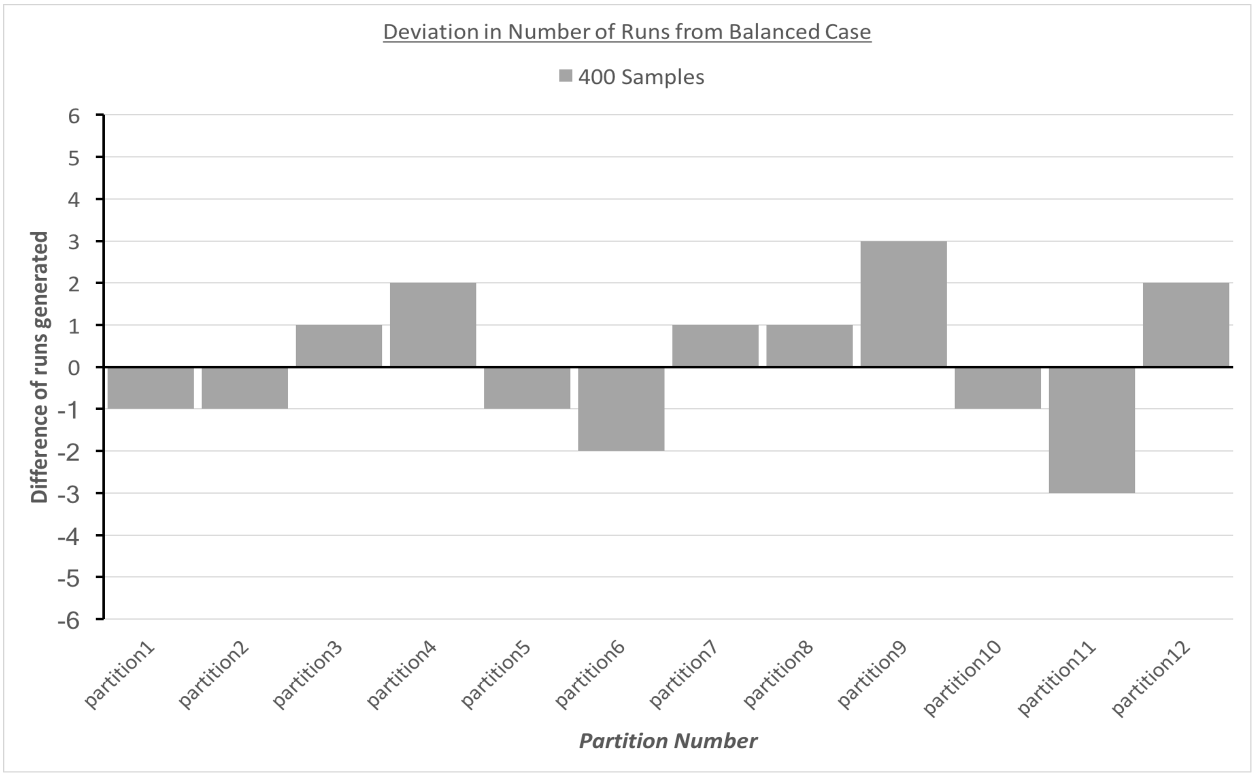

Figure 4.10: Number of runs generated by partitions taking 400 samples

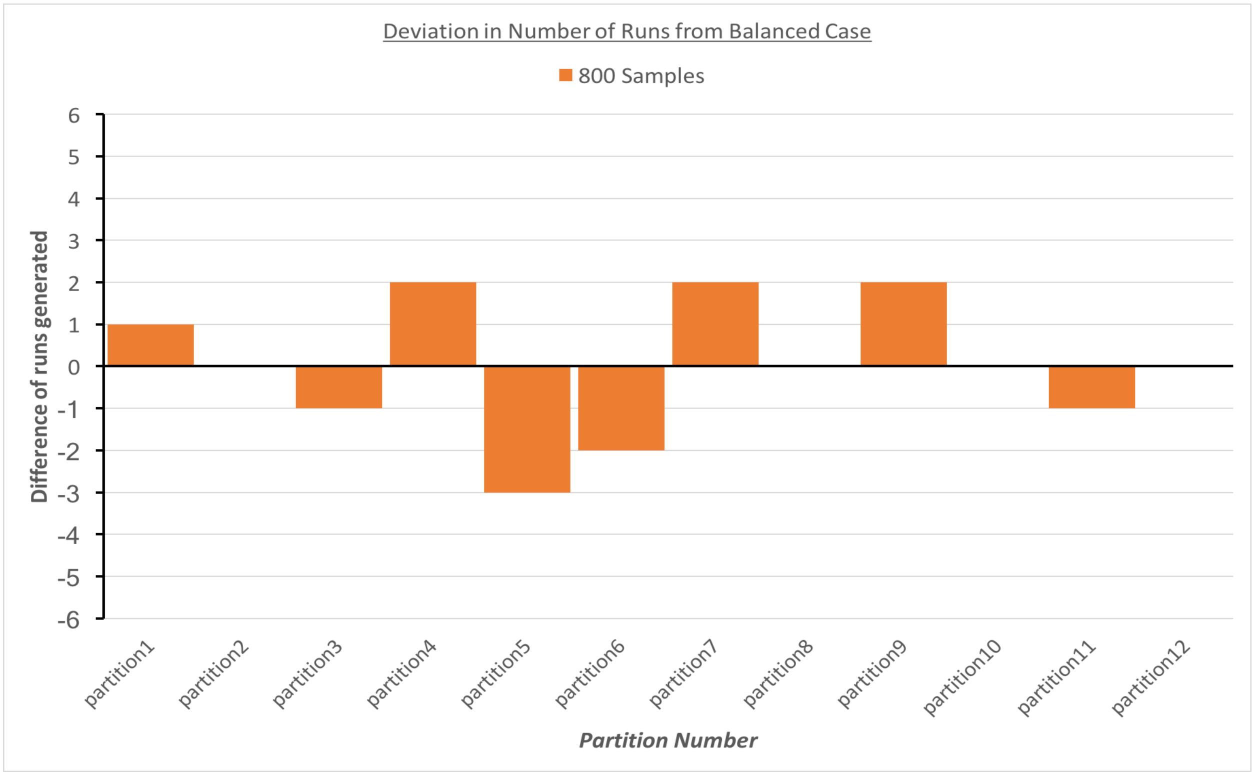

Figure 4.11: Number of runs generated by partitions taking 800 samples

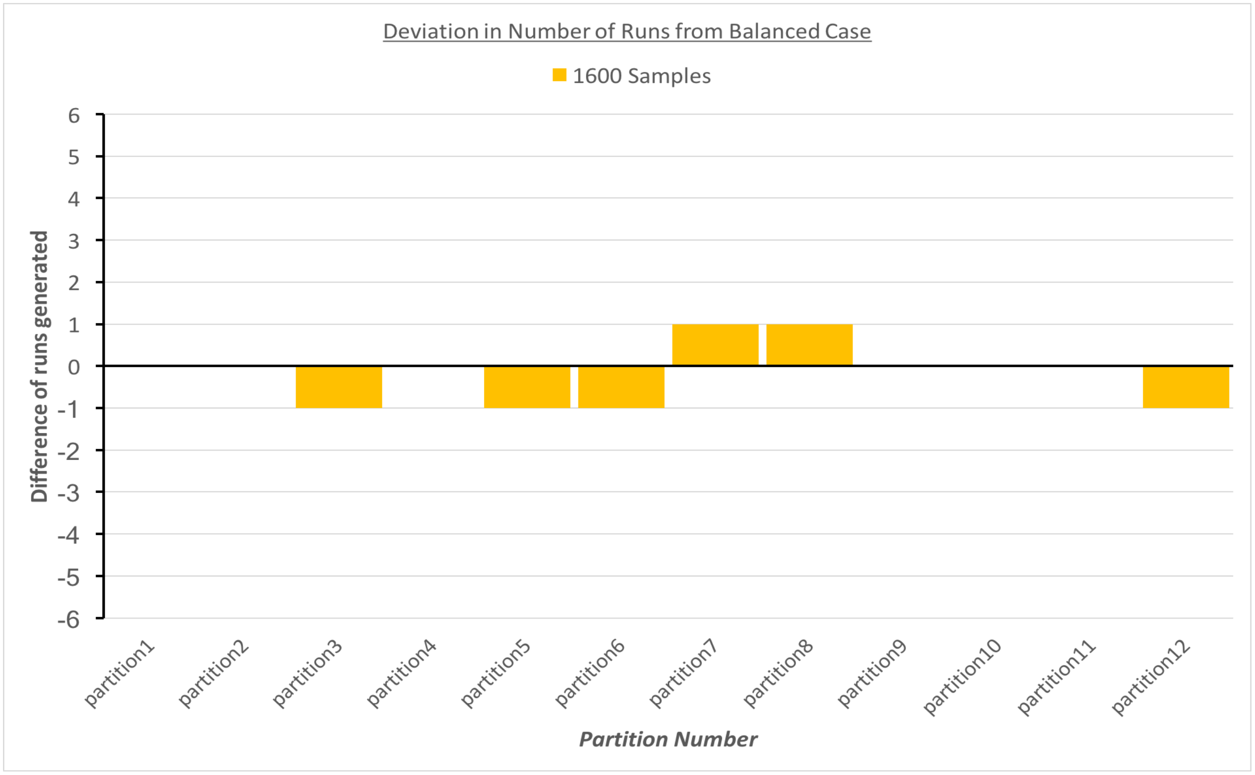

Figure 4.12: Number of runs generated by partitions taking 1,600 samples

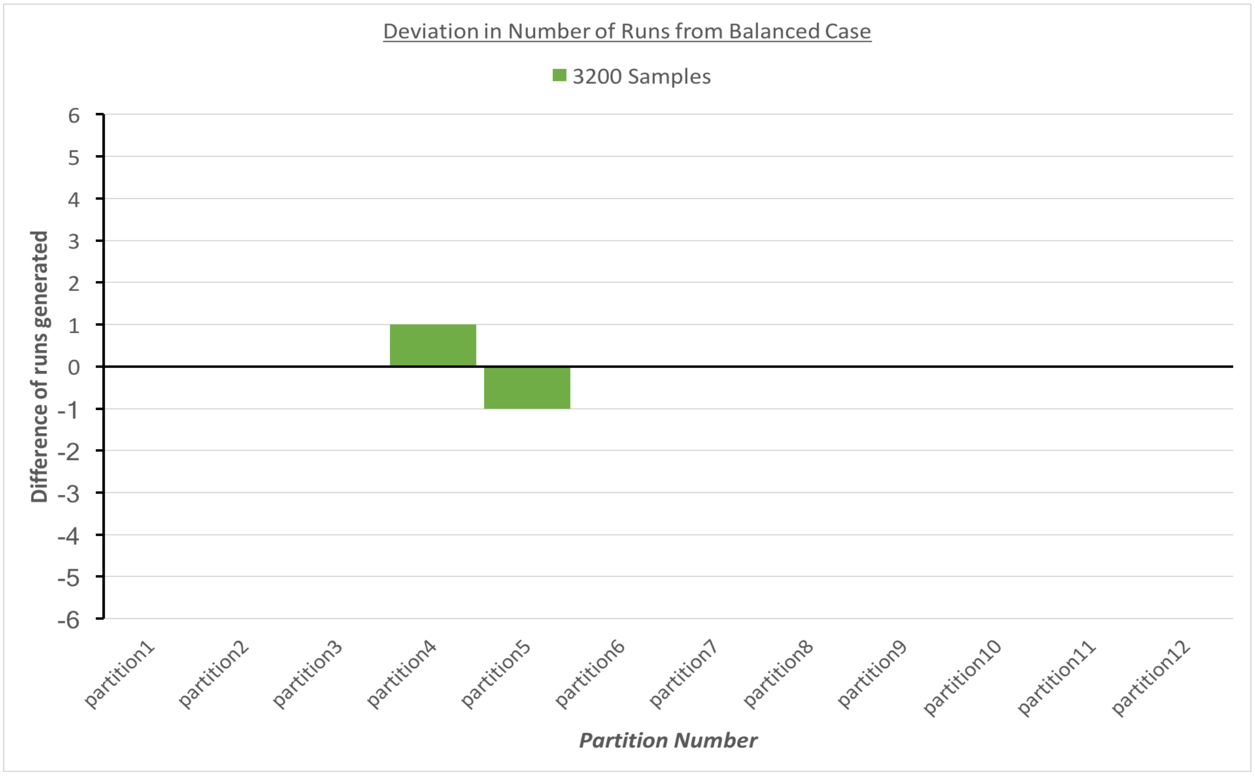

Figure 4.13: Number of runs generated by partitions taking 3,200 samples

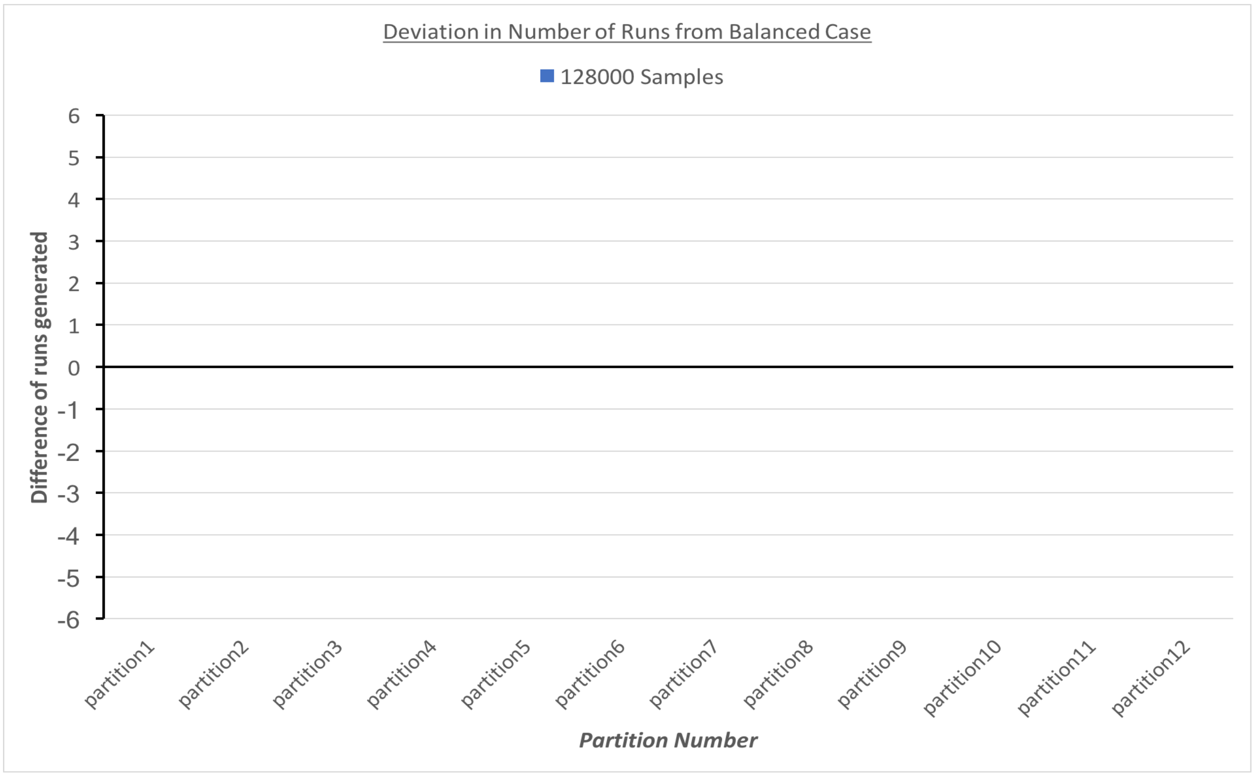

Figure 4.14: Number of runs generated by partitions taking 128K samples

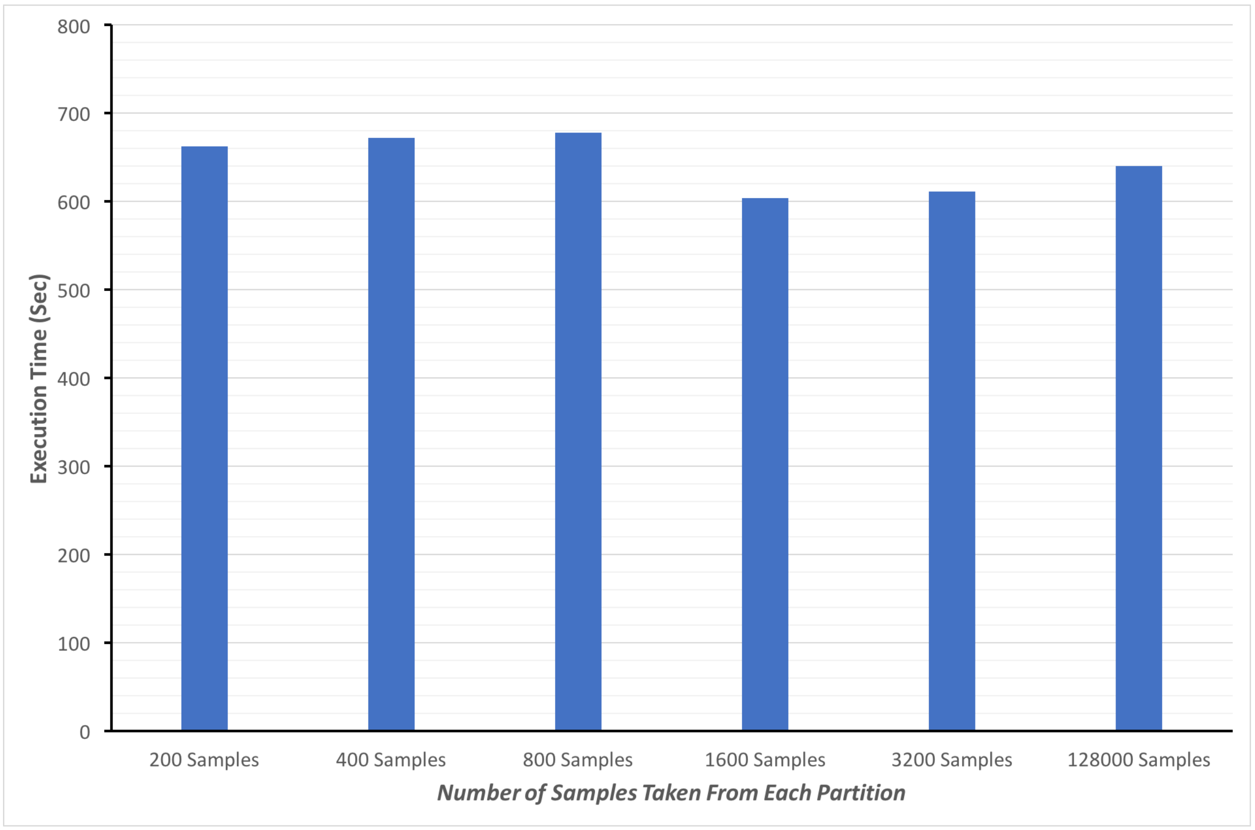

Figure 4.15: Execution time when taking different numbers of samples

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we presented an approach to optimizing external sorting in AsterixDB. We started by explaining that AsterixDB currently performs sorting in two stages. In the first stage, each partition sorts its part of the data and produces a single sorted run. Next, in the second stage, a single node merges all the runs and produces the final sorted output. We identified the issues with this approach and presented our solution, which relies on redistributing the data to eliminate the final merge step. The redistribution step employs an approximate splitting vector that is computed using sampling. We shared the results and findings from a set of experiments that we conducted to evaluate our solution. We showed that repartitioning parallel sort (RPS) has excellent potential and that it significantly outperforms the current AsterixDB parallel sort global merge (PSGM) approach.

5.2 Future Work

5.2.1 Data Skew

We have seen in one of our experiments that RPS' performance starts to degenerate when skew exists in the data. This is because our current implementation does not take any specific action in that regard. In other words, the partitioner is not "fair" when it distributes the tuples across the partitions in the face of skewed data. If a certain value exists in a dataset in large numbers, most likely the splitting vector will contain multiples of it assuming the sampling is good enough that the samples are representative of the whole dataset. One potential solution would be to tweak the partitioner and make it aware of the fact there are multiple instances of the same value in the splitting vector. We could, then, modify the partitioner so that it does not pick one specific partition and overload it. Another alternative would be to explore the option of optimal splitters described in [11].

5.2.2 Materialization Step

In our current implementation, in addition to the cost introduced by sampling, we also incur overhead caused by materializing the input tuples and reading them again. We needed that step because we scan the whole input (incoming subquery results, in the general case) even though we just pick the first 100 tuples as our samples. (We just ignore all the tuples after the first 100 as they come in.) An improvement could be to create a new multi-stage operator that would take in the first 100 tuples from each partition, save them, determine the splitting vector, and send it to the partitioner; then, it could resume processing by first forwarding the first batch of the tuples it saved before continuing to take in the rest of the input tuples and forward them. This would eliminate the need for (and cost of) the intermediate materialization.

Bibliography

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

[2] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. Tsotras, R. Vernica, J. Wen, and T. Westmann. AsterixDB: A scalable, open source BDMS. *Proc. VLDB Endow.*, 7(14):1905–1916, Oct. 2014.

[3] V. Borkar, Y. Bu, E. P. Carman, Jr., N. Onose, T. Westmann, P. Pirzadeh, M. J. Carey, and V. J. Tsotras. Algebricks: A data model-agnostic compiler backend for big data languages. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 422–433, New York, NY, USA, 2015. ACM.

[4] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1151–1162, Washington, DC, USA, 2011. IEEE Computer Society.

[5] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, June 1992.

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

[7] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *[1991] Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 280–291, Dec 1991.

[8] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 27–40, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[9] G. Graefe. Parallel external sorting in volcano. *University of Colorado - Boulder Computer Science Technical Reports*, CU-CS-459-90, 1990.

[10] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[11] K. A. Ross and J. Cieslewicz. Optimal splitters for database partitioning with size bounds. In *Proceedings of the 12th International Conference on Database Theory*, ICDT '09, pages 98–110, New York, NY, USA, 2009. ACM.