AQWA: Adaptive Query-Workload-Aware Partitioning of Big Spatial Data

Ahmed M. Aly[1], Mohamed S. Hassan[1], Ahmed R. Mahmood[1], Walid G. Aref[1],
Hazem Elmeleegy[2], Mourad Ouzzani[3]
[1]Purdue University, West Lafayette, IN
[2]Turn Inc., Redwood City, CA
[3]Qatar Computing Research Institute, Doha, Qatar

[1]{aaly, msaberab, aref, amahmoo}@cs.purdue.edu,
[2]hazem.elmeleegy@turn.com, [3]mouzzani@qf.org.qa

ABSTRACT

The unprecedented spread of location-aware devices has resulted into a plethora of location-based services in which huge amounts of spatial data need to be efficiently processed. To process large-scale data, MapReduce has become the de facto framework for large-scale computing clusters. Existing cluster-based systems for processing spatial data are oblivious to query-workload, and hence are not able to consistently provide good performance. The reason is that typical spatial query-workloads exhibit skewed access patterns, where certain spatial areas receive queries more frequently than others. To close this gap, we present AQWA, an adaptive and query-workload-aware data partitioning mechanism that minimizes the processing time of spatial queries over large-scale spatial data. AQWA does not assume prior knowledge of the query-workload. Instead, AQWA adapts to the query-workload, and in an online fashion. As queries get processed, the data partitions are incrementally updated. With extensive experiments using real spatial data of Billions of points from OpenStreetMap, and thousands of spatial range and k-nearest-neighbor queries executed over a cluster running Hadoop, we demonstrate that AQWA can achieve orders of magnitude gain in query performance compared to standard spatial partitioning structures.

1. INTRODUCTION

The ubiquity of location-aware devices, e.g., smartphones and GPS-devices, has led to a large variety of location-based services in which large amounts of geo-tagged information are created every day. Meanwhile, the MapReduce framework [10] has proven to be very successful in processing large datasets on large clusters, particularly after the massive deployments reported by companies like Facebook, Google, and Yahoo!. Moreover, tools built on top of Hadoop [34], the open-source implementation of Mapreduce, e.g., Pig [25], Hive [31], Cheetah [6], and Pigeon [12], make it easier for users to engage with Hadoop and run queries using high-level languages. However, one of the main issues with MapReduce is that executing a query usually involves scanning very large amounts of data that can lead to high response times. Not enough attention has been devoted to addressing this issue in the context of spatial data.

As noted in several research efforts, e.g., [26, 32, 8], accounting for the query-workload can achieve significant performance gains. In particular, regions of space that are queried with high frequency need to be aggressively partitioned in comparison to the other less popular regions. This fine-grained partitioning of the in-high-demand data can result in significant savings in query processing time.

Existing cluster-based systems for processing spatial data employ traditional spatial indexing methods that can effectively partition the data into multiple buckets, but that are not query-workload aware. For instance, SpatialHadoop [11, 13] supports both space-partitioning as well as data-partitioning schemes to handle spatial data on Hadoop. However, both partitioning schemes are static and do not adapt to changes in query-workload.

In this paper, we present AQWA, an adaptive and query-workload aware data partitioning mechanism that minimizes the query processing time of spatial queries over large-scale spatial data. An important characteristic of AQWA is that it does not presume any knowledge of the query-workload. Instead, AQWA can detect, in an online fashion, the pattern(s) of the query-workload, e.g., when certain spatial regions get queried more frequently than other spatial regions. Accordingly, AQWA reorganizes the data in a way that better serves the execution of the queries by minimizing the amount of data to be scanned. Furthermore, AQWA can adapt to changes in the query-workload. Instead of recreating the partitions from scratch, which is a costly operation because it requires reading and writing the entire data, AQWA incrementally updates the partitioning of the data in an online fashion.

Observe that the number of ways one can partition the underlying spatial data is large. Finding the boundaries of the partitions that would result in good performance gains for a given query-workload is challenging. The process of searching for the optimal partitioning involves excessive use of two operations: 1) Finding the number of points in a given region, and 2) Finding the number of queries that overlap a given region. A straightforward way to support these two operations is to scan the whole data (in case of Operation 1) and all queries in the workload (in case of Operation 2), which is quite costly. The reason is that: i) we are dealing with big data in which scanning the whole data is costly, and ii) the two operations are to be repeated multiple times in order to find the best partitioning. To address this challenge, AQWA employes various optimizations to support the above two operations efficiently. In particular, AQWA maintains a set of compact aggregate information about the data distribution as well as the query-workload. This aggregate information is kept in main-memory, and hence it enables AQWA to efficiently perform its adaptive repartitioning decisions.

Unlike traditional spatial index structures that can have unbounded decomposition until the finest granularity of data is reached in each split (i.e., block), AQWA tries to limit the number of partitions (i.e., files) because allowing too many small partitions can be very harmful to the overall health of a computing cluster. The metadata of the partitions is usually managed in a centralized shared resource. For instance, the name node is a centralized resource in Hadoop that manages the metadata of the files in the Hadoop Distributed File System (HDFS), and handles the file requests across the whole cluster. Hence, the name node is a critical component in Hadoop, and if overloaded with too many (small) files, it slows down the overall cluster (e.g., see [2, 18, 22, 33, 35]). Being of vital importance to a Hadoop cluster, the latest releases of Hadoop try to maintain more than one replica of the name node in order to increase the cluster availability in case of failures. However, replication of the name node does not help distribute the overhead of maintaining the data partitions, and hence in AQWA, we limit the number of data partitions in order to overcome this challenge.

AQWA employs a simple yet powerful cost function that models the cost of executing the queries and also associates with each data partition the corresponding cost. To incrementally update the data partitions and maintain a limited number of partitions, AQWA selects some partition(s) to be split (if queried with high frequency) and other partition(s) to be merged (if queried with low frequency). To efficiently select the best data partitions to split and merge, AQWA maintains dual priority queues in main-memory; one priority queue stores the cost gain corresponding to splitting a data partition (a max-heap), and the other priority queue stores the cost loss corresponding to merging data partitions (a min-heap).

AQWA is resilient to abrupt or temporary changes in the workload. Based on the cost model, an invariant relationship guards the operation of the dual priority queues to make sure that no redundant split/merge operations take place. However, when the workload permanently shifts from one hotspot area (i.e., one that receives queries more frequently, e.g., downtown area) to another hotspot area, AQWA is able to react to that change and update the partitioning accordingly. To achieve that, AQWA employs a time-fading counting mechanism that alleviates the weights corresponding to older query-workloads.

In summary, the contributions of the paper are as follows.

- We introduce AQWA, a new dynamic data partitioning scheme that is query-workload-aware. AQWA is able to: 1) partition the data without prior knowledge of the query-workload, 2) incrementally update the partitions (according to the workload) rather than rebuilding the partitions from scratch, and 3) automatically adapt to changes in the workload, e.g., when the workload shifts over time from one spatial hotspot to another hotspot.

- We present efficient algorithms for supporting two basic operations that are at the core of AQWA's partitioning mechanism, namely finding the number of points in a given region, and finding the number of queries in a workload that overlap a given region.

- We adopt a cost-based dual priority queue mechanism that manages the process of repartitioning the data by taking into consideration the cost saving/loss associated with each possible split/merge of a data partition.

- We evaluate the performance of AQWA on a Hadoop cluster by executing various query-workloads of spatial range and k-nearest-neighbor queries over 2.7 Billion points of OpenStreetMap [1] data. Experimental results demonstrate up to two orders of magnitude improvement in query performance in comparison to standard spatial data partitioning structures, e.g., uniform grid or k-d tree partitioning.

The rest of this paper proceeds as follows. Section 2 discusses the related work. Section 3 gives an overview of the problem, the proposed solution, and the cost model. Section 4 introduces AQWA along with its partition construction algorithms and its self-organizing mechanisms in response to query-workload changes. Section 5 provides an experimental study of the performance of AQWA. Section 6 includes concluding remarks.

2. RELATED WORK

Work related to AQWA can be categorized into three main categories: 1) centralized data indexing, 2) data indexing in distributed platforms, and 3) query-workload awareness in database systems.

In centralized indexing, e.g., B-tree [7], R-tree [17, 19, 4], Quadtree [30], Interval-tree [9], k-d tree [5], the goal is to split the data in a centralized index that resides in one machine. Most of the indexes in this category aim at distributing the size of the data among a set of blocks, without accounting for the query-workload. In most cases, there is no restriction on the number of blocks that can be used. Consequently, in these indexes, the structure of the index can have unbounded decomposition until the finest granularity of data is reached in each split (i.e., block). For instance, the R-tree recursively splits the data into rectangles until the number of points in each rectangle is less/greater than a certain threshold. This model of unbounded decomposition works well for any query-workload distribution; the very fine granularity of the splits enables any query to retrieve its required data by scanning minimal amount of data with very little redundancy. However, as explained in Section 1, in a typical distributed file system, e.g., HDFS, it is important to limit the number of files (i.e., partitions) because allowing too many small partitions can be very harmful to the overall health of a computing cluster (e.g., see [2, 18, 22, 33, 35]). Therefore, AQWA keeps the number of partitions below a certain limit.

In the second category, distributed indexing, e.g., [11, 14, 15, 16, 21, 23, 24], the goal is to split the data in a distributed file system in a way that optimizes the distributed query processing by minimizing the I/O overhead. Unlike the centralized indexes, indexes in this category are usually geared towards fulfilling the requirements of the distributed file system, e.g., the number of splits (e.g., blocks in Hadoop) being within a certain bound. For instance, the Eagle-Eyed Elephant (E3) framework [14] avoids scans of data partitions that are irrelevant to the query at hand. However, E3 considers only one-dimensional data, and hence is not suitable for spatial two-dimensional data/queries. [11, 13] present Spatial Hadoop; a system that processes spatial two-dimensional data using two-dimensional Grids or R-Trees. A similar effort in [21] addresses how to build R-Tree-like indexes in Hadoop for spatial data. However, none of these efforts is query-workload aware. [28, 27, 20] decluster spatial data into multiple disks to achieve good load balancing in order to reduce the response time for range and partial match queries. However, their proposed data declustering schemes are not adaptive to the underlying query-workload.

In the third category, query-workload awareness in database systems, several research efforts have emphasized on the importance of taking the query-workload into consideration when designing the database and when indexing the data. [26, 8]

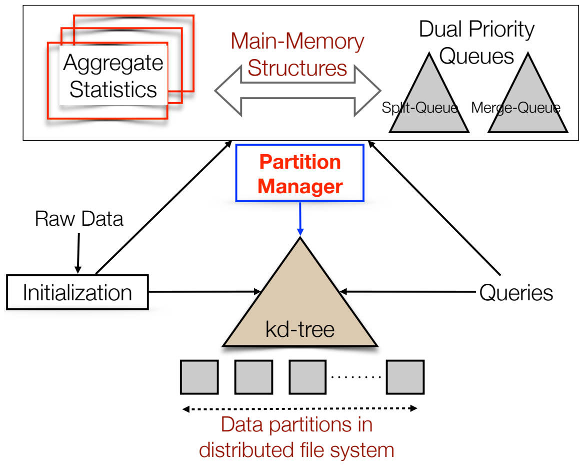


Figure 1: An overview of AQWA.

present query-workload aware data partitioning mechanisms in distributed shared-nothing platforms. However, these mechanisms are geared towards one-dimensional data and do not suit spatial two-dimensional data. [32] presents a query-workload aware indexing scheme for continuously moving objects. However, [32] assumes a centralized platform, and hence the proposed indexing scheme cannot be directly applied to distributed file systems, e.g., HDFS.

3. PRELIMINARIES

3.1 Overview

We consider range and k-nearest-neighbor queries over a set, say S, of data points in the two-dimensional space. Our goal is to partition S into a given number of partitions, say P partitions, such that the amount of data scanned by the queries is minimized, and hence the cost of executing the queries is minimized as well. The value of P is a system parameter.

Figure 1 gives an overview of AQWA that is composed of three main components: 1) a k-d tree decomposition[1] of the data, where each leaf node is a partition in the distributed file system, 2) a set of main-memory aggregates to maintain statistics about the distribution of the data and the queries, and 3) dual priority queues that maintain information about the partitions. Three main processes define the interactions among the components of AQWA, namely, Initialization, Query Execution, and Repartitioning.

- **Initialization:** During this phase, we collect statistics about the data distribution, and accordingly, create an initial partitioning layout. Thus, the initialization consists of two steps that are repeated only once:

    1. *Counting:* We collect regional (spatial) statistics about the data. In particular, we divide the space into a grid, say G, of n rows and m columns. Each grid cell, say $G[i,\ j]$, will contain the total number of points whose coordinates are inside the boundaries of $G[i,\ j]$. The grid is kept in main-memory and is used later on to find the number of points in a given region in $O(1)$.
    2. *Initial Partitioning:* Based on the counts determined in the Counting step, we identify the best partitioning layout that evenly distributes the points in a kd-tree decomposition. We create the partitions using a MapReduce job that reads the entire data and assigns each data point to its corresponding partition.

- **Query Execution:** This process selects the partitions that are relevant to, i.e., overlap, the invoked query. Then, the selected partitions are passed as input to a MapReduce job to determine the actual data points that belong to the answer of the query. Afterwards, the query is logged into the same grid that maintains the counts of points. The entries in the dual priority queues are also updated accordingly. After this update, we may (or may not) take a decision to repartition the data as we explain next.

- **Repartitioning:** Based on the history of the query-workload as well as the distribution of the data, we determine the partition(s) that, if altered (i.e., further decomposed), would result into better execution time of the queries. If we find any such partitions, we reshuffle the partitioning layout.

While the Initialization and Query Execution phases can be implemented in a straightforward way, the Repartitioning phase raises the following performance challenges:

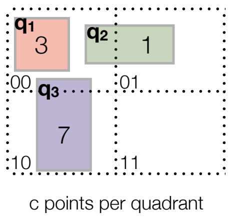
- *Overhead of Rewriting:* If the process of altering the partitioning layout reconstructs the partitions from scratch, it would be very inefficient because it will have to reread and rewrite the entire data. Hence, we propose an incremental mechanism to alter only a minimal number of partitions.

- *Efficient Search:* We repeatedly search for the best change to do in the partitioning in order to achieve good query performance. The search space is large, and hence, we need an efficient way to determine the partitions to be further split and how/where the split should take place. We present efficient techniques for supporting two basic operations, namely, finding the number of points in a given region, and finding the number of queries in a workload that overlap a given region. These two operations are at the core of the process of finding the best partitioning layout as they are repeated multiple times during the search process. In particular, we maintain main-memory aggregate statistics about the data distribution based on the Counting stage. We also maintain similar aggregate statistics about the query-workload distribution. These aggregates enable AQWA to efficiently determine the partitioning layout via main-memory lookups.

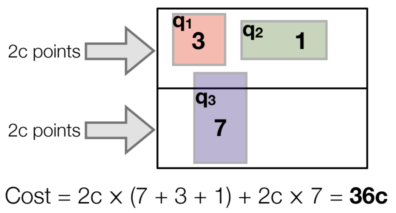
- *Workload Changes and Thrashing Avoidance:* AQWA needs to efficiently adapt to changes in the query-workload. However, we need to ensure that AQWA is resilient to sudden and temporary changes in the query-workload. AQWA should be robust to avoid unnecessary repartitioning of the data, and hence avoid thrashing, i.e., avoid the case where the same partitions get split and merged successively. At the same time, AQWA should respond to permanent changes in the query-workload, and accordingly alter the partitions.

- *Keeping a Limited Number of Partitions:* For practical considerations, it is important to limit the number of partitions because allowing too many small partitions can introduce a performance bottleneck (e.g., see [2, 18, 22, 33, 35]). Hence, we need to ensure that our incremental partitioning scheme keeps the number of partitions constant. Dual priority queues are kept in main-memory to ensure that for every partition to be split, there are two partitions to be merged, and hence the overall number of partitions remains constant.

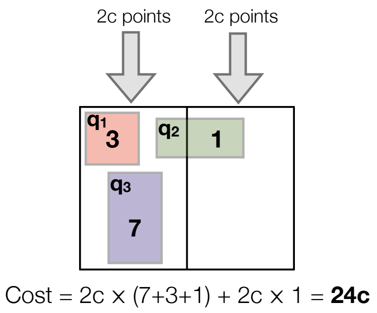
[1]The ideas presented in this paper do not assume a specific data structure and are applicable to R-Tree or quadtree decompositions.



(a) Query-workload.



(b) Configuration of horizontal partitions.



(c) Configuration of vertical partitions.

Figure 2: $P = 2$. Possible partitioning layouts and the corresponding cost (i.e., quality) according to the given query-workload.

3.2 Cost Model

In AQWA, given a query, our goal is to avoid unnecessary scans of the data. We estimate the cost of executing a query by the number of records it has to read. Given a query-workload, we estimate the cost, i.e., quality, of a partitioning layout by the number of points that the queries of the workload will have to retrieve. More formally, given a partitioning layout composed of a set of partitions, say L, the overall query execution cost can be computed as:

$$Cost(L) = \sum_{\forall p \in L} O_q(p) \times N(p), \quad (1)$$

where $O_q(p)$ is the number of queries that overlap Partition p, and $N(p)$ is the count of points in p.

Theoretically, the number of possible partitioning layouts is exponential in the total number of points because a partition can take any shape and can contain any subset of the points. For simplicity, we consider only partitioning layouts that have rectangular-shaped partitions. Our goal is to choose the cheapest partitioning layout according to the above equation. The following example illustrates the variance in the cost, i.e., quality, of the different possible partitioning layouts. Consider a small space in which points are distributed uniformly such that each quadrant has c points as shown in Figure 2(a). The quadrants are numbered: 00 for the top-left, 01 for the top-right, and so on. Consider some range queries on the given space that follow the following pattern: q_3, which is repeated 7 times, overlaps Quadrants 00 and 10, q_1, which is repeated 3 times, overlaps Quadrant 00, and q_2, which is repeated once, overlaps Quadrants 00 and 01. Assume that we want to organize the data into 2 partitions, i.e., $P = 2$. Because we consider only rectangular partitions, as Figures 2(b) and 2(c) give, there are two possible partitioning layouts: 1) partition the space horizontally, and 2) partition the space vertically. According to Equation 1, these two partitioning layouts have different costs as we explain below.

In the horizontal partitioning layout (Figure 2(b)), the upper partition, which has $2c$ points, overlaps q_1, q_2, and q_3, which are repeated 3, 1, and 7 times, respectively. Therefore, the cost corresponding to the upper partition is $2c \times (3 + 1 + 7) = 22c$. The lower partition, which also has 2 points, overlaps q_3 only. Therefore, the cost corresponding to the lower partition is $2c \times 7 = 14c$. Hence, the overall cost of the horizontal partitioning layout of Figure 2(b) is $36c$. Similarly, in the vertical partitioning layout (Figure 2(c)), the left partition, which has $2c$ points, overlaps queries q_1, q_2, and q_3, and hence the cost corresponding to the left partition is $2c \times (3 + 1 + 7) = 22c$. The right partition, which also has $2c$ points, overlaps q_1 only. Therefore, the cost corresponding to the right partition is $2c \times 1 = 2c$. Hence, the overall cost of the vertical partitioning layout of Figure 2(c) is $24c$.

The above example demonstrates that the choices of the shapes of the partitions in the partitioning layout can significantly affect the overall cost, i.e., performance, of a given query-workload.

4. AQWA

4.1 Initialization

The main goal of AQWA is to partition the data in a way that minimizes the cost according to Equation 1. Initially, i.e., before any query is executed, the number of queries that will overlap each partition is unknown. Hence, we simply assume a uniform distribution of the queries across the data. This implies that the only component of Equation 1 that matters at this initial stage is the number of points in each partition. Thus, in the initialization phase, we partition the data in a way that balances the number of points across the partitions. In particular, we apply a k-d tree decomposition [5], but with limited (i.e., constant) number of partitions.

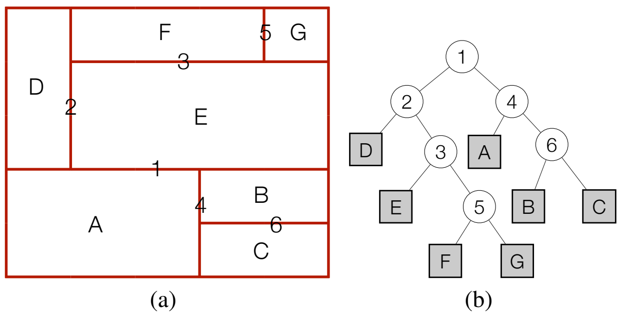
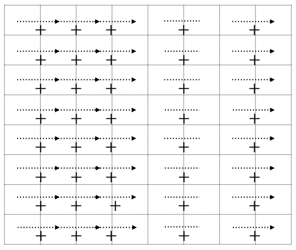
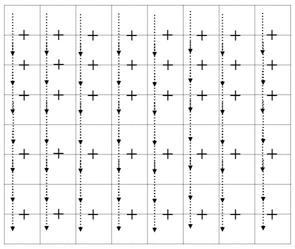
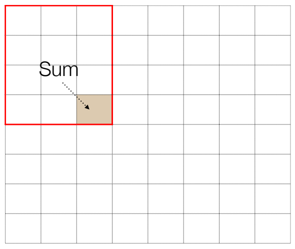


Figure 4: An example tree of $P = 7$ leaf nodes

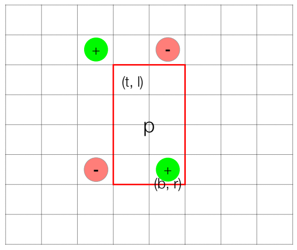
The k-d tree is a binary tree in which every non-leaf node tries to split the underlying space into two parts that have the same number of points. Only leaf nodes contain the actual data points. The splits can be horizontal or vertical and are chosen to balance the number of points across the leaf nodes. Splitting is recursively applied until the number of points in each leaf node is below a threshold. Initially, AQWA applies a similar decomposition strategy to that of the k-d tree, except that the decomposition stops when the number of partitions reaches P, i.e., when P leaf nodes are created in the tree. Figure 4 gives the initial state of an example AQWA with $P = 7$ leaf nodes along with the corresponding space partitions. Once the boundaries of each leaf node are determined, a MapReduce job creates the initial partitions, i.e., assigns each data point to its corresponding partition. In this MapReduce job, for each point,


(a) Horizontal aggregation.


(b) Vertical aggregation.



(c) Sum after aggregation.



(d) $O(1)$ lookups.

Figure 3: The aggregate counts that enable AQWA to determine the number of points in an arbitrary rectangle in $O(1)$.

say p, the key is the leaf node that encloses p, and the value is p. The mappers read different chunks of the data and then send each point to the appropriate reducer, which groups the points that belong to the same partition, and ultimately writes the corresponding partition file into HDFS.

The hierarchy of the partitioning layout, i.e., the k-d tree, is kept for processing future queries. As explained in Section 3.1, once a query, say q, is received, only the leaf nodes of the tree that overlap q are selected and passed as input to the MapReduce job corresponding to q.

Efficient Search via Aggregation

An important question to address during the initialization phase is how to split a leaf node in the tree. In other words, for a given partition, say p, what is the best horizontal or vertical line that can split p into two parts such that the number of points is the same in both parts? Furthermore, how can we determine the number of points in each split, i.e., rectangle? Because the raw data is not partitioned, one way to solve this problem is to have a complete scan of the data in order to determine the number of points in a given rectangle. Obviously, this is not practical to perform. Because we are interested in aggregates, i.e., count of points in a rectangle, scanning the individual points involves redundancy. Thus, in the initialization phase, we preprocess the raw data in a way that enables quick lookup (in $O(1)$ time) of the count corresponding to a given rectangle. In particular, we hash the points into a virtual two-dimensional grid that has a very fine granularity. The grid does not contain the data points, but rather maintains aggregate information. In particular, we divide the space into a grid, say G, of n rows and m columns. Each grid cell, say $G[i,\ j]$, initially contains the total number of points whose coordinates are inside the boundaries of $G[i,\ j]$. As explained in Section 3.1, this is achieved through a single MapReduce job that reads the entire data and determines the count for each grid cell. Afterwards, we aggregate the data corresponding to each cell in G as follows. Refer to Figure 3 for illustration. For every row in G, we scan the cells from column 0 to column m and aggregate the values as we go, i.e., $G[i,\ j] = G[i,\ j] + G[i,\ j-1]\ \forall\ j \in [2,\ m]$ (Figure 3(a)). Afterwards, we repeat the same process on the column level, i.e., $G[i,\ j] = G[i,\ j] + G[i-1,\ j]\ \forall\ i \in [2,\ n]$ (Figure 3(b)). At this moment, the value at each cell, say $G[i,\ j]$, will correspond to the total number of points in the rectangle bounded by $G[0,\ 0]$ (top-left) and $G[i,\ j]$ (bottom-right). For example, the number of points in the red rectangle of Figure 3(c) can be determined in $O(1)$ by simply retrieving the value of the shaded cell that corresponds to the bottom-right corner of the rectangle.

To compute the number of points corresponding to any given partition, i.e., rectangle, bounded by Cell $G[b,\ r]$ (bottom-right) and Cell $G[t,\ l]$ (top-left), we add/subtract the values of only four cells, i.e., perform an $O(1)$ operation in the following way. As Figure 3(d) illustrates, the number of points, say N_p, in Rectangle(b, r, t, l) is:

$$N_p(b,\ r,\ t,\ l) = G[b,\ r] - G[t-1,\ r] - G[b,\ l-1] + G[t-1,\ l-1]. \quad (2)$$

The above formulation can be explained as follows. The value at Cell $G[b,\ r]$ includes the sum of the points in the rectangle whose top-left corner is $(0,\ 0)$ and whose bottom-right corner is $(b,\ r)$. Thus, to get the sum of the number of points in Rectangle(b, r, t, l), we need to subtract from $G[b,\ r]$ the sum of the number of points corresponding to two rectangles: 1) the rectangle whose top-left corner is $(0,\ 0)$ and whose bottom-right corner is $(t-1,\ r)$, as well as 2) the rectangle whose top-left corner is $(0,\ 0)$ and whose bottom-right corner is $(b,\ l-1)$. However, if we subtract these two values, the sum corresponding to the rectangle whose top-left corner is $(0,\ 0)$ and whose bottom-right corner is $(t-1,\ l-1)$ will be subtracted twice. Hence, we add the value of $G[t-1,\ l-1]$.

In addition to the above optimization, instead of trying all the possible horizontal and vertical lines to determine the median line that evenly splits the points in a given partition, we apply binary search on each dimension of the data. Given a rectangle of r rows and c columns, first, we try a horizontal split of the rectangle at Row $\frac{r}{2}$ and determine the number of points in the corresponding two splits (in $O(1)$ operations as described above). If the number of points in both splits is the same, we terminate, otherwise, we recursively repeat the process with the split that has higher number of points. The process may be repeated for the vertical splits if no even splitting is found for the horizontal splits. If no even splitting is found for the vertical splits, the we choose the best possible splitting amongst the vertical and horizontal splits, i.e., the split that minimizes the absolute value of the difference between the number of points in the emerging splits.

The above optimizations of grid-based pre-aggregation are essential for the efficiency of the initialization phase as well as the repartitioning phase that we describe in the next section. Without pre-aggregation, e.g., using a straightforward scan of the entire data, the partitioning would be impractical.

4.2 Query-Workload Awareness

AQWA is query-workload aware. After a query is executed, it may (or may not) trigger a change in the partitioning layout by splitting a leaf node (i.e., a partition) in the kd-tree into two nodes, and merge two nodes that share the same parent in the kd-tree into one leaf node. The decision of whether to apply such change or not depends on the cost function of Equation 1. Three factors affect this decision, namely, the cost gain that would result after splitting a partition, the cost loss that would result after merging two partitions, and the overhead of reading and writing the contents of these partitions. Below, we explain each of these factors in detail.

1. *The cost* ***reduction*** *that would result if a certain partition is further split into two splits.* Observe that a query usually *partially* overlaps few partitions. For instance, in Figure 5(a), q_1 partially overlaps partitions A, D, and E. When q_1 is executed, it reads the entire data of these overlapping partitions. However, not all the data in these overlapping partitions is relevant, i.e., there are some points that are redundantly scanned in the map phase of the MapReduce job corresponding to q_1. Thus, it would be beneficial w.r.t. q_1 to further decompose, i.e., split, Partitions A, D, and E so that the amount of irrelevant data to be scanned is minimized. For example, assume that Partitions A, E, and D contain 20, 30, and 15 points, respectively. According to Equation 1, the cost corresponding to the partitioning layout of Figure 5(a) is $20 \times 1 + 30 \times 1 + 15 \times 1 = 65$. However, if Partition E is split to Partitions E_1 and E_2, such that E_1 and E_2 have 15 points each (Figure 5(b)), the cost would drop to 50; q_1 will have to read only half of the data in Partition E (i.e., Partition E_2) instead of the entirety of Partition E. Thus, splitting a partition may lead to a decrease in the cost corresponding to a partitioning layout. More formally, assume that a partition, say p, is to be split into two Partitions, say p_1 and p_2. We estimate the decrease in cost, say C_d, associated with splitting p as follows:

$$C_d(Split,\ p,\ p_1,\ p_2) = C(p) - C(p_1) - C(p_2). \quad (3)$$

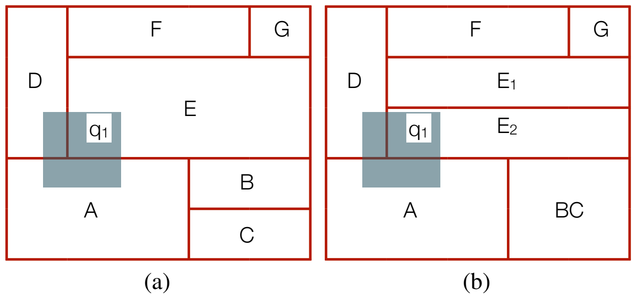


Figure 5: Incremental repartitioning of the data. Partition E is split while Partitions B and C are merged.

2. *The cost* ***increase*** *that would result if two partitions are merged.* We need to keep the number of partitions, i.e., leaf nodes in the tree, constant. Thus, if we ever split a leaf node into two leaf nodes, we increase the overall number of partitions by one. Hence, in response to this change and to keep the total number of partition constant, we need to merge two other leaf nodes in the kd-tree that have the same parent. In contrast to a split, a merge may result in a performance loss because some queries in the workload may need to read more data as partitions get merged. For example, consider the partitioning layout in Figure 6(a). q_3 reads the entire data of Partition A, and hence it is beneficial w.r.t. q_3 to split Partition A to Partitions A_1 and A_2 as illustrated in Figure 6(b). However, splitting Partition A would increase the overall number of partitions. Thus, we need to merge Partitions F and G (or similarly, merge Partitions E_1 and E_2). Merging Partitions F and G would slow down the execution of q_2 because instead of reading the small Partition G as in Figure 6(a), q_2 will have to read the entirety of the merged partition, i.e., Partition FG. Thus, merging two partitions may lead to an increase in the cost corresponding to a partitioning layout. Similarly to Equation 3, we estimate the cost loss, say C_i, associated with merging two partitions, say p_1 and p_2, into a single partition, say p as:

$$C_i(Merge,\ p_1,\ p_2,\ p) = C(p) - C(p_1) - C(p_2). \quad (4)$$

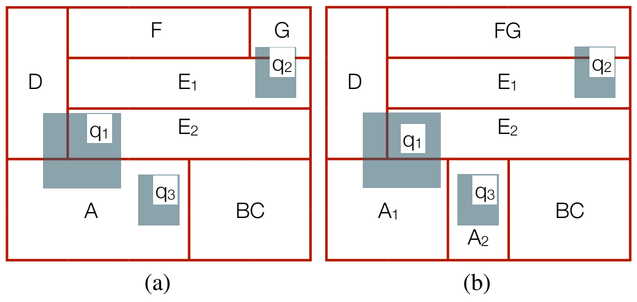


Figure 6: Incremental repartitioning of the data. Partition A is split while Partitions F and G are merged.

3. *The cost of read/write during the split and merge processes.* Should we decide to split a partition and merge two other partitions, the entire data of these three partitions will have to be read and then written in order to create the new partitions. More formally, assume that a partition, say p, is to be split. We estimate the read/write cost associated with the splitting process as:

$$C_{rw}(p) = 2 \times N(p), \quad (5)$$

where $N(p)$ is the number of points in p. Similarly, when two partitions, say p_1 and p_2, are to be merged, the read/write cost associated with the merging process is estimated as $C_{rw}(p_1, p_2) = 2 \times N(p_1) + 2 \times N(p_2)$.

According to Equations 3, 4, and 5, we decide to split a partition, say p_s, and merge two other partitions, say p_{m1} and p_{m2} if:

$$C_d(Split,\ p_s,\ p_{s1},\ p_{s2}) > [C_i(Merge, p_{m1},\ p_{m2},\ p_m) + C_{rw}(p_s) + C_{rw}(p_{m1},\ p_{m2})]. \quad (6)$$

A query may overlap more than one partition. Upon the execution of a query, say q, the cost corresponding to the partitions that overlap q changes. Also, for each of these partitions, the values of cost decrease due to split (i.e., C_d) as well as the values of cost increase due to merge (i.e., C_i) will change. Two challenges exist in this situation:

1. How can we efficiently determine the best partitions to be split and merged? We need to choose the partition that, if split, would reduce the cost the most. Similarly, we need to choose two leaf partitions that, if merged, would increase the cost the least.

2. How can we efficiently determine the best split of a partition w.r.t. the query-workload, i.e., according to Equation 1? We already address this issue in Section 4.1, but w.r.t. to the data distribution only, i.e., without considering the query-workload.

We address the above challenges in the next sections.

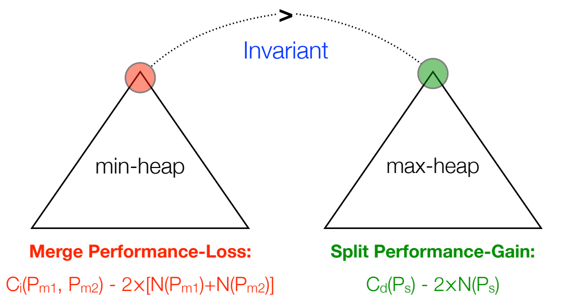

Invariant
min-heap
max-heap
Merge Performance-Loss:
$C_i(P_{m1}, P_{m2}) - 2\times[N(P_{m1})+N(P_{m2})]$
Split Performance-Gain:
$C_d(P_s) - 2\times N(P_s)$

Figure 7: Priority queues of candidate partitions.

4.2.1 *The Dual Priority Queues*

To address the first challenge above, i.e., selecting the best partitions for the split and merge operations, we maintain two priority queues of candidate partitions. The first priority queue maintains the candidate partitions for the split operation, and the second priority queue maintains pairs of candidate partitions for the merge operation. We refer to the former priority queue as the split-queue and the latter as the merge-queue. Refer to Figure 7 for illustration.

On the one hand, partitions in the split-queue are decreasingly ordered in a max-heap according to the cost reduction that would result after the split operations. For each partition, say p_s, in the split-queue, we determine the best split that would maximize the cost-reduction, i.e., C_d, that corresponds to splitting p_s. We explain the process of selecting the best split in detail in the next section. Notice that for each partition, we subtract the cost of the read/write associated with the split operation from the value of the cost-reduction. Thus, the value maintained for each partition in the split-queue is $C_d - 2 \times N(p_s)$. On the other hand, pairs of partitions in the merge-queue are increasingly ordered in a min-heap according to the cost increase, i.e., C_i, that would result after the merge operations. Similarly to the values maintained in the split-queue, we subtract the cost of the read/write associated with the merge operation from the value of the cost-increase. Thus, the value maintained for each pair of partitions, say p_{m1} and p_{m2}, in the merge-queue is $C_i - 2 \times [N(p_{m1}) + N(p_{m2})]$.

After a query is executed, the overlapping partitions are determined and their corresponding values are updated in the priority queues in the following way. According to Inequality 6, if the highest cost gain due to a split, i.e., the top value of the split-queue, is greater than the lowest cost loss due to a merge, i.e., the top value of the merge-queue, we initiate a split and a merge operation. We repeat this step until the top value of the split-queue is less than the top-value of the merge-queue. In this case, we determine that further splits and merges are not expected to result in any performance gains. Thus, our incremental repartitioning mechanism tries to maintain an invariant throughout the lifetime of AQWA, which is to always make sure that the slightest increase in cost due to a merge is always greater than the highest expected gain due to a split. If this invariant is not satisfied, we reorder the partitions. Observe that the dual priority queues ensure that the number of partitions is always constant because for each split, there is a simultaneous merge.

4.2.2 *Efficient Search for the Best Split*

As illustrated in Section 3.2, for a given partition, the different choices for the position and orientation of a split can have different costs. An important question to address is how to efficiently determine, according to the cost model of Equation 3, the best split of a partition that would result in the highest cost gain. To compute

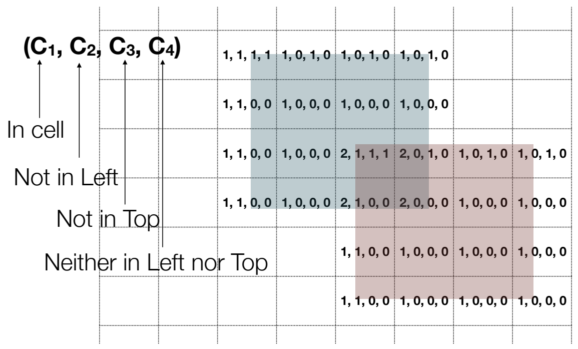


Figure 8: The four counters maintained for each grid cell.

the cost corresponding to a partition and each of its corresponding splits, Equation 1 embeds two factors that affect the cost corresponding to a partition, say p, namely, 1) the number of points in p, and 2) the number of queries that overlap p. In Section 4.1, we demonstrate how to compute the number of points in any given partition using an $O(1)$ operation. Thus, in order to efficiently compute the whole cost formula, we need an efficient way to determine the number of queries that overlap a partition.

In Section 4.1 above, we demonstrate how to maintain aggregate information for the number of points using a grid. However, extending this idea to maintain aggregate information for the number of queries is challenging because a point resides in only one grid cell, but a query may overlap more than one grid cell. Unless careful aggregation is devised, over-counting may occur. We use the same structure of the $n \times m$ grid G as in Section 4.1. At each grid cell, say $G[i,\ j]$ we maintain four additional counters, namely,

- C_1: a counter for the number of queries that overlap $G[i,\ j]$,
- C_2: a counter for the number of queries that overlap $G[i,\ j]$, but not $G[i,\ j-1]$ (not in left),
- C_3: a counter for the number of queries that overlap $G[i,\ j]$, but not $G[i-1,\ j]$ (not in top), and
- C_4: a counter for the number of queries that overlap $G[i,\ j]$, but not $G[i-1,\ j]$ or $G[i,\ j-1]$ (neither in top nor left).

Figure 8 gives an illustration of the values of the four counters that correspond to two range queries.

We aggregate the values of the above counters as follows. For C_2, for each row in the grid, we horizontally aggregate the values in each grid cell from left to right as in Figure 3(a), i.e., $G[i,\ j].C_2 = G[i,\ j].C_2 + G[i,\ j-1].C_2\ \forall\ j \in [2,\ m]$. For C_3, for each column, we vertically aggregate the values in each grid cell from top to bottom as in Figure 3(b), i.e., $G[i,\ j].C_3 = G[i,\ j].C_3 + G[i-1,\ 1].C_3\ \forall\ i \in [2,\ n]$. For C_4, we horizontally and then vertically aggregate the values in the same manner as we aggregate the number of points (see Figure 3). As queries are invoked, the aggregate values of the counters are updated according to the overlap between the invoked queries and the cells of the grid. Although the process of updating the aggregate counts is repeated per query, it does not incur overhead because the virtual grid that maintains the counts resides in main-memory, and hence the virtual grid is cheap to update.

To determine the number of queries that overlap a certain partition, say p, we perform the following four operations. 1) We determine the value of C_1 for the top-left grid cell that overlaps p. 2) We determine the aggregate value of C_2 for the top border of

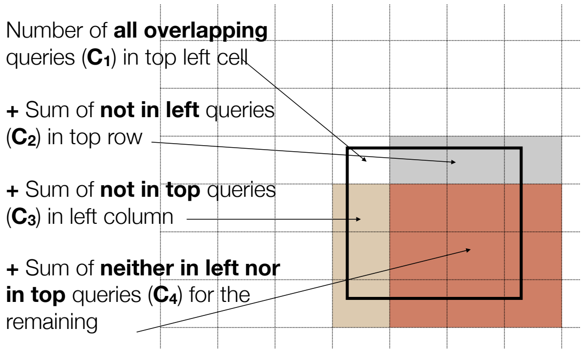

Number of **all overlapping** queries (**C₁**) in top left cell
+ Sum of **not in left** queries (**C₂**) in top row
+ Sum of **not in top** queries (**C₃**) in left column
+ Sum of **neither in left nor in top** queries (**C₄**) for the remaining

Figure 9: Determining the number of queries that overlap a partition in $O(1)$.

p except for the top-left cell, i.e., if the top border of p spans the cells from $[X,\ l]$ to $[X,\ r]$, we determine the aggregate value as $G[X,\ r].C_2 - G[X,\ l+1].C_2$. 3) We determine the aggregate value of C_3 for the left border of p except for the top-left cell, i.e., if the left border of p spans the cells from $[t,\ Y]$ to $[b,\ Y]$, we determine the aggregate value as $G[b,\ Y].C_3 - G[t+1,\ Y].C_3$. 4) We determine the aggregate value of C_4 for the remainder of p, i.e., every grid cell that overlaps p except for the left and top borders. This can be achieved using the same computation of Equation 2. The sum of the above values represents the number of queries that overlap p. Refer to Figure 9 for illustration. Observe that each of the above values is computed in $O(1)$. Thus, we can determine the number of queries that overlap a partition in an $O(1)$ computation. This results in efficient computation of the cost function and significantly improves the process of finding the best split of a partition. Given a partition, to find the best split that evenly distributes the cost between the two splits, we apply a binary search in a way that is similar to the process we discuss in Section 4.1. The main difference is that the number of queries is considered in the cost function.

4.2.3 Accounting for Changes in Query-Workload

AQWA is resistant to abrupt or temporary changes to the query-workload. Consider a scenario where a certain query-workload, say W, is consistently received. As expected, the partitions that overlap queries $\in W$ are split in a way that minimizes the execution time of these queries. Assume that a certain area, say A_t, does not belong to the areas that W hits, and that A_t temporarily receives a few queries, say W_t. Because the workload corresponding to W_t is temporary and less frequent than W, AQWA does not ruin the partitioning w.r.t. W, which is the required behaviour. In particular, if the partitions that overlap A_t make it to the top of the split-queue, the partitions that overlap W will have higher weights in the merge-queue, and hence the invariant will prevent any splitting and merge operations to occur. The reason for this robust behaviour is that the cost function gives higher weight for a partition, say p, according to the frequency of the queries that overlap p.

Time-Fading Weights

AQWA keeps the history of all the queries that have been processed. For every processed query, say q, grid cells overlapping q are determined, and the corresponding four counters are incremented for each cell (see Section 4.2.2). Although this mechanism captures the frequency of the queries, it does not differentiate between fresh queries (i.e., those that belong to the current query-workload) and relatively old queries; all queries have the same weight. This can lead to poor performance in AQWA especially when a query-workload changes. To illustrate, consider a scenario where a certain area, say A_{old}, has received queries with high frequency in the past, but the workload has permanently shifted to another area, say A_{new}. Because the data partitions corresponding to A_{old} have high weights in the merge-queue, these high weights keep the invariant between the dual priority queues for a while, and hence repartitioning with respect to A_{new} will be delayed, and hence AQWA would not recognize the new hotspot A_{new} and would not adapt to it in a timely fashion. The reason is that according to the cost model we have presented so far, an old query has the same weight as a new query.

To address this issue, we need a way to *forget* older queries, or equivalently alleviate their weight in the cost function. To achieve that, we differentiate between queries received in the last T time units that we refer to as $current$ queries, and queries received before T time units that we refer to as old queries. T is a system parameter. In particular, for each of the four counters (refer to c_1 through c_4 in Section 4.2.2) maintained at each cell in the grid, we maintain separate counts for the old queries and the $current$ queries. The count corresponding to old queries, say C_{old}, gets decaying weight by being divided by c every T time units, where $c > 1$ is a system parameter. The count corresponding to $current$ queries, say C_{new}, has no decaying weight. Every T time units, C_{new} is added to C_{old}, and then C_{new} is set to zero. At any time, the number of queries in a region is determined as $(C_{new} + C_{old})$.

Observe that every T time units, the sum $(C_{new} + C_{old})$ changes, and this can change the weights of the partitions to be split/merged. This requires revisiting each partition to determine its new weight and its new order in the dual priority queues. A straightforward approach is to update the values corresponding to all the partitions and reconstruct the dual priority queues every T time units. However, this approach can be costly because it requires massive operations to rebuild the queues. To solve this problem, we apply a lazy-update mechanism, where we process the partitions in a round-robin cycle that takes T time units to pass over all the partitions. In other words, if P is the number of partitions, we process only $\frac{P}{T}$ partitions every time unit. For each of the $\frac{P}{T}$ partitions, we recalculate the corresponding weights and reinsert these partitions into the dual priority queues. Eventually, after T time units, all the entries in the dual priority queues get updated.

4.2.4 Support for k-Nearest-Neighbor Queries

So far, we have only shown how to process spatial range queries, and how to update the partitioning accordingly. Range queries are relatively easy to process because the boundaries in which the answer of the query resides are predefined (and fixed within the query itself). Hence, given a range query, only the partitions that overlap the query can be passed as input to the MapReduce job corresponding to the query without worrying about losing the correctness of the answer of the query. In contrast, for a k-nearest-neighbor query, the boundaries that contain the answer of the query are unknown until the query is executed. Hence the partitions that are needed as input to the MapReduce job corresponding to the query are unknown. In particular, the spatial region that contains the answer of a k-nearest-neighbor query depends on the value of k, the location of the query focal point, and the distribution of the data (see [3]). To illustrate, consider the example in Figure 10. Partition p in which q_1 resides is sufficient to find q_1's k_1-nearest-neighbors. However, for $k_2 > k_1$, Partition p is not sufficient, and two further blocks (one above and one below) have to be considered. Similarly, Partition p is not sufficient to find the k-nearest-neighbors of q_2 because of the location of q_2 w.r.t. Partition p (i.e., being near one corner).

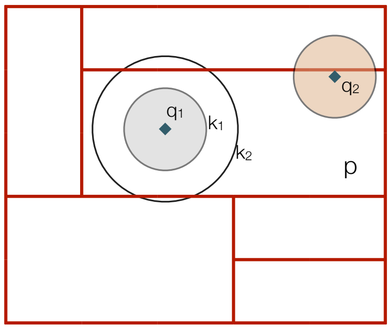


Figure 10: The partitions that contain the k-nearest-neighbors of a query point vary according to the value of k, the location of the query focal point, and the distribution of the data.

[13] tries to solve this challenge by following a three-step approach, where execution of the query starts with a MapReduce job that takes as input the partition, say p, in which the query?s focal point, say q, resides. In the second step, which is a correctness-check step, the distance, say r, between q and the k^{th} neighbor is determined, and a check is performed to make sure that the boundaries of p are within r distance from q, i.e., Partition p is sufficient to guarantee the correctness of the answer. If it is not the case that Partition p is sufficient, the third step is performed, where another MapReduce job is executed with the partitions surrounding p being added as input. The second and third steps are repeated until the correctness-check is satisfied.

A major drawback of the above solution is that it may require successive MapReduce jobs in order to answer a single query. To solve this problem, we present a more efficient approach that requires only one MapReduce job to answer a k-nearest-neighbor query. In particular, we make use of the fine-grained virtual grid that contains statistics about the data distribution. Given a k-nearest-neighbor query, we determine the grid cells that are guaranteed to contain the answer of the query using the MINDIST and MAXDIST metrics as in [29]. In particular, we scan the grid cells in increasing order of their MINDIST from the query focal point, and count the number of points in the encountered cells. Once the accumulative count reaches the value k, we mark the largest MAXDIST, say M, between the query focal point and any encountered cell. We continue scanning until the MINDIST of a scanned block is greater than M. To illustrate, consider the example in Figure 11. Given Query q, count of the number of points in the cell that contains q is determined. Assuming that this count is $> k$, the MAXDIST between q and the cell in which it is contained is determined. Cells that are within this MAXDIST are guaranteed to enclose the k-nearest-neighbors of q.

After we determine the grid cells that contain the query answer, we determine a rectangular region that bounds these cells. In some sense, we have converted the k-nearest-neighbor query into a range query, and hence our algorithms and techniques for splitting/merging and search can still handle k-nearest neighbor queries in the same way range queries are handled.

After the rectangular region that bounds the answer is determined, the partitions that overlap that region are passed as input to the MapReduce job corresponding to the query. Refer to Figure 12 for illustration. Partitions p_1, p_2, and p_3 are passed as the input for Query q_2, while Partition p_1 is passed as the input for Query q_1.

Observe that the process of determining the region that encloses the k-nearest-neighbors is efficient because it is based on counting of main-memory aggregates, and it does not require scanning

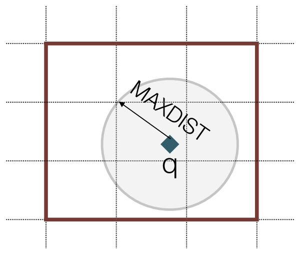


Figure 11: Finding the grid cells (with fine granularity) that are guaranteed to enclose the answer of a k-nearest-neighbor query. The rectangular region that bounds these cells maps the k-nearest-neighbor query into a range query.

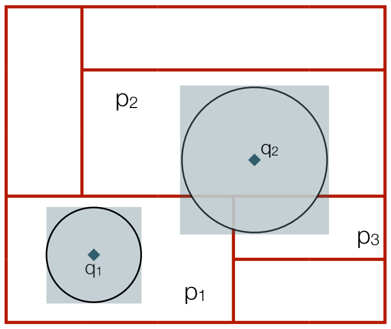


Figure 12: A k-nearest-neighbor query is treated as a range query once the rectangular bounds enclosing the answer are determined.

any data points. Moreover, because the granularity of the grid is fine (compared to that of the partitions), the determined region is compact.

4.2.5 *Concurrency Control*

As queries are received by AQWA, some partitions may need to be altered. It is possible that while a partition is being split (or similarly two partitions are being merged), a new query is received that may also trigger another change to the very same partitions being altered. Unless an appropriate concurrency control protocol is used, inconsistent partitioning will occur. To solve this problem, we use a simple locking mechanism to coordinate the incremental updates of the partitions. In particular, whenever a query, say q, triggers a change in the partitioning layout, before the partitions are updated, q tries to acquire a lock on each of the partitions to be altered. If q succeeds to acquire all the locks, i.e., no other query has a conflicting lock, then q is allowed to alter the partitions. The locks are released after the partitions are completely altered. If q cannot acquire the locks due to a concurrent query that already has one or more locks on the partitions being altered, then the decision to alter the partitions is cancelled. Observe that canceling such decision may negatively affect the quality of the partitioning, but only temporarily because for a given query-workload, queries similar to q will keep arriving afterwards and the repartitioning will eventually take place.

A similar concurrency issue arises when updating the dual priority queues. Because the dual queues reside in main-memory, updating the entries of these queues is fast (requires a few milliseconds). Hence, to avoid the case where two queries result in conflicting queue updates, we serialize the process of updating the two priority queues using a critical section.

5. EXPERIMENTS

In this section, we evaluate the performance of AQWA. We realize a cluster-based test bed in which we implement AQWA as well as the uniform grid and a static k-d tree partitioning. We choose these two structures because we want to contrast the performance of AQWA against two different extreme partitioning schemes: 1) pure spatial decomposition, i.e., when using a Grid, and 2) data decomposition, i.e., when using a k-d tree. Experiments are conducted on a 16-node cluster running Hadoop over Red Hat Enterprise Linux 6.4. Each node in the cluster has four 1TB hard drives, 8GB of RAM, and 4 cores Intel(R) Xeon(R) E31240 @3.30GHz. The nodes of the cluster are all connected with 1Gbps networking.

We use a real spatial dataset from OpenStreetMap [1]. The number of data points in the dataset is 2.7 Billion points. The format of each point is simply comma-separated longitude-latitude. We use various query-workloads that were selected to follow the spatial distribution of the data. The intuition behind selecting the query-workload is that certain areas that have high density are likely to receive more queries than less dense areas. Hence, we generated 10 different query-workloads that focus on 10 dense spatial areas. We use range and k-nearest-neighbor queries.

5.1 Initialization

In this experiment, we study the performance of the initialization phase of AQWA (Section 4.1). Figure 13 gives the execution time of the initialization phase for various values of P (the number of partitions). Observe that the initialization phase in AQWA is equivalent to that of the k-d tree partitioning. Hence, in Figure 13, the performance of the initialization phase for both AQWA and the k-d tree are contrasted against the grid.

Observe that grid partitioning requires relatively high execution time especially for low values of P. This is due to the skewness of the data distribution, which causes certain grid cells, i.e., partitions, to receive more data points than other grid cells. In the reduce phase, each reducer handles a set of grid cells and groups the corresponding points. Thus, the load across the reducers will be unbalanced. Because a MapReduce job does not terminate until the last reducer completes, the unbalanced load leads to a relatively poor performance for the grid (compared to the kd-tree). As the number of partitions increases, the load gets better balanced, but still the MapReduce job has to wait for the last reducer that has the biggest partition. In contrast, in the k-d tree partitioning, which is employed by AQWA, the initialization phase balances the data sizes across all the partitions. Consequently, the reducers have balanced load in the MapReduce job, and all of the reducers terminate almost concurrently.

Index Type	Number of Partitions (P)	Construction Time
Space Partitioning (Uniform Grid)	10	2hrs, 20min
	100	1hr, 31 min
	1000	59 min
Data Partitioning (Static k-d tree or AQWA)	10	45 min
	100	35 min
	1000	35 min

Figure 13: Performance of the initialization phase.

5.2 Query-Workload Awareness

In the following experiments, we study the query performance of AQWA. Because the goal of any partitioning scheme is to minimize the amount of data to be scanned by a query, out main performance measure is the total processing time across all mappers. We virtually split the space according to a 1000×1000 grid that represents 1000 normalized unit distance measures in each of the horizontal and vertical dimensions. This represents the search space for the partitions as well as the count statistics that we maintain. We fix the number of partitions, i.e., P, to 100 in each of the grid, static k-d tree, and AQWA. We issue range and k-nearest-neighbor queries over each of these partitioning structures. To simulate a hotspot, we concentrate the queries over one of the areas that we identified as having relatively high data density. The coordinates of the ranges of the queries are chosen at random within the hotspot, but are bounded to 100×100 unit distances of the virtual grid. The coordinates of the k-nearest-neighbor queries are also chosen at random within the hotspot, and the value of k is randomly chosen between 1 and 1000.

5.2.1 *Adaptiveness to Query-Workload*

In this experiment, we issue 100 range queries one after the other. All the queries are selected inside a single hotspot area. Figure 14 gives the performance of AQWA compared to a static grid and a static k-d tree. The x-axis represents the serial number of the query being executed. Because we are executing the queries one at a time, the x-axis can also be viewed as the timeline. As the figure demonstrates, in the early stages, AQWA has the same performance as that of the k-d tree because each of them share the same hierarchy initially. As queries get processed over time, AQWA repartitions the data and starts to have better performance than the k-d tree and the grid. Because the queries are chosen at random around the hotspot area, some queries may hit a well-decomposed partition, while others may hit a relatively big partition. Hence, the performance curve exhibits some instability at the early stages, but it eventually stabilizes after Query 50. From the figure, AQWA achieves 2 orders of magnitude gain over the static k-d tree, and one order of magnitude over the grid.

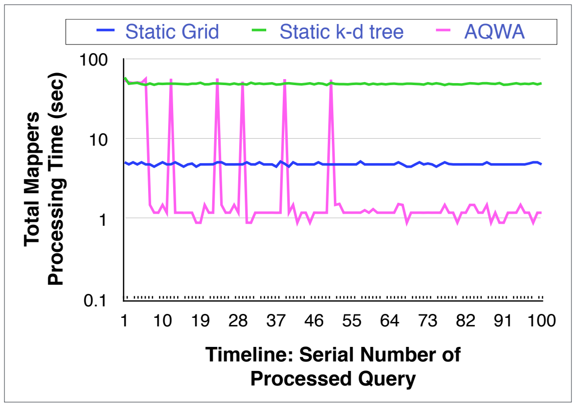


Figure 14: The performance of AQWA compared to a grid and a k-d tree for a given query-workload. As the workload gets learned, AQWA reorganizes the data leading to better performance.

Figure 15 gives the overhead of repartitioning the data. On the one hand, because the grid and the k-d tree are static, they have no repartitioning cost. On the other hand, AQWA incurs repartitioning cost, but it is amortized across subsequent query executions.

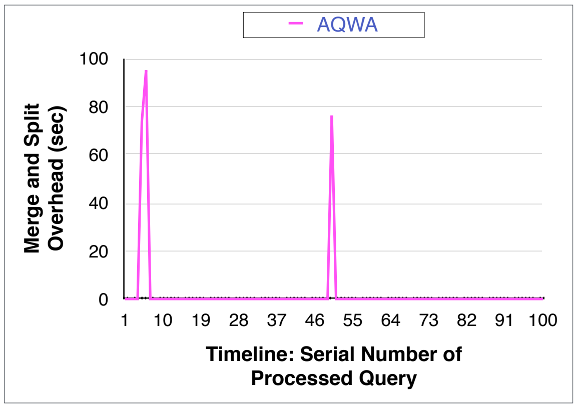


Figure 15: The overhead of repartitioning in AQWA.

For instance, Query 7 and Query 50 result in repartitioning of the data, which incurs a spike overhead of slightly less than 100 seconds. However, this repartitioning pays off for Query 51 and all the subsequent queries. Notice that the alternative would have been to perform a whole reconstruction of all the partitions, which exhibits similar construction time to that of the initialization phase, i.e., in the order of 45-60 minutes.

Although the repartitioning process incurs an overhead, it does not interfere with the execution of the other concurrent or incoming queries. Splitting a partition, say p, into two new partitions still leaves p available to other concurrent queries until the two splits get completely created. Hence, the overhead exhibited during splitting a partition does not affect the performance of the other concurrent or incoming range queries that reference p or that reference other regions of the space that are not affected by the split. The same is true for the merging process.

5.2.2 *Handling Multiple Query-Workloads*

In this set of experiments, we study the effect of having two or more query-workloads. We generate 10 different query-workloads. To have realistic workloads (where dense areas are likely to be queried with high frequency), we identify 10 hotspot areas (each of size 100×100 unit distances) that have the highest density of data. Similarly to the above experiments, for each of these hotspot areas, we generate 1000 range and k-nearest-neighbor, where the coordinates of the queries are chosen to lie within the hotspot. The coordinates of the ranges of the queries are chosen at random within the hotspot, but are bounded to 100×100 unit distances of the virtual grid. The coordinates of the k-nearest-neighbor queries are also chosen at random within the hotspot, and the value of k is randomly chosen between 1 and 1000.

In the experiments to follow, we have two modes of operation:

1. **Interleaved Execution:** In this mode, queries are executed across the hotspots simultaneously (i.e., generated in a round-robin across the hotspots). For instance, if we have 1000 queries and 2 hotspots, say h_1 and h_2, the first query will be executed at h_1, the second query will be executed at h_2, the third query will be executed at h_1, and so on.

2. **Serial Execution:** In this mode, queries are executed over one hotspot at a time. In other words, we simulate the migration of the workload from one hotspot to another. For instance, if we have 1000 queries and 2 hotspots, the first 500 queries will be executed over one hotspot, followed by the other 500 queries executed over the other hotspot.

In the first experiment, we pick two hotspots and execute 100 queries using the interleaved mode of execution. As Figure 16 demonstrates, AQWA succeeds to adapt to two simultaneous hotspots.

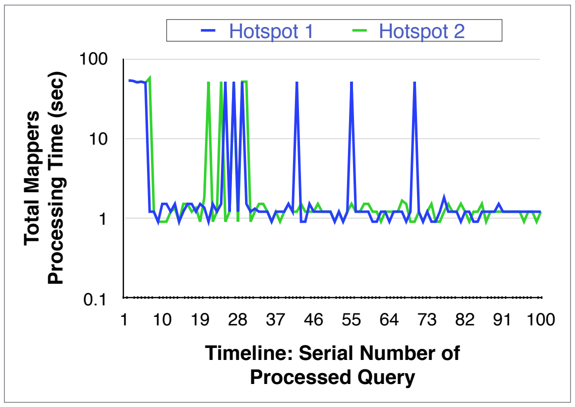


Figure 16: Interleaved execution of the queries for two hotspot areas.

In the second experiment, we pick two hotspots and execute 50 queries using the serial mode of execution. As Figure 17 demonstrates, AQWA quickly adapts to the shift in the hotspot locations.

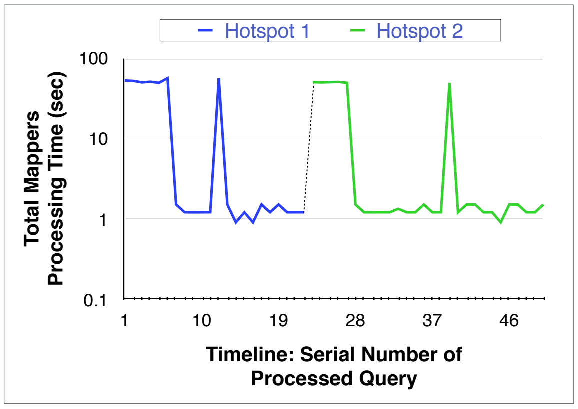


Figure 17: Serial execution of the queries for two hotspot areas.

Observe that in the second experiment, 50 can be considered as a few number of queries. As explained in Section 4.2.3, AQWA can easily adapt to the change in the workload in this case. However, when the number of executed queries gets large, AQWA may delay the repartitioning process due to the high weights that correspond to partitions in the min-heap of the dual priority queues. To study this issue, we repeat the above experiment (i.e., serial mode of execution) with 1000 queries executed at each hotspot, and also increase the number of hotspots to 9. We examine the time-fading mechanism proposed in Section 4.2.3 that assigns lower weights to older queries. We set $T = 100$ and the decay factor $c = 2$, i.e., every 100 queries, the counts are demolished by a factor of $\frac{1}{2}$. We monitor the average speedup achieved by AQWA when compared to a static kd-tree partitioning.

Figure 18 gives the performance of AQWA compared to a static kd-tree partitioning. The y-axis represents the speedup achieved by AQWA, while the x-axis represents the timeline of the serial execution of the 9 hotspots, i.e., all 1000 queries in Hotspot 1 are pro-

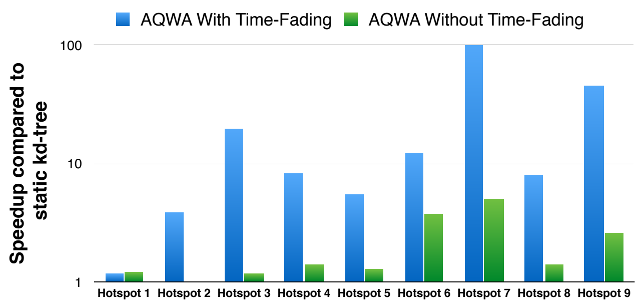


Figure 18: The effect of time-fading weights on the serial execution of queries over 9 hotspot areas (1000 queries per hotspot).

cessed, followed by all the 1000 queries in Hotspot 2, and so on. As the figure demonstrates, the time-fading mechanism achieves orders of magnitude gain in performance. Without time-fading, AQWA fails to achieve good performance gains when the workload shifts from one hotspot to another. For instance, for Hotspot 2, the weights of the partitions that result from Hotspot 1 prevent AQWA from performing any repartitioning. In contrast, with time-fading, the effect of older queries (from Hotspot 1) is alleviated, and AQWA is able to appropriately update the partitions.

6. CONCLUDING REMARKS

AQWA is an adaptive query-workload-aware partitioning mechanism for large-scale spatial data. Without prior knowledge of the query-workload, AQWA can incrementally update the data partitioning layout in a way that minimizes the execution time of spatial queries. AQWA has the ability to react to permanent changes changes in the query-workload, and incrementally update the data partitions accordingly. AQWA employs a time-fading cost model that captures both the data distribution and the frequency of the queries. AQWA maintains a set of main-memory structures, namely a grid and dual priority queues, to efficiently manage the process of repartitioning the data. Experimental results that are based on real spatial data and various query-workloads demonstrate that AQWA outperforms the standard spatial partitioning mechanisms by up to two orders of magnitude in terms of query performance.

7. REFERENCES

[1] OpenStreetMap bulk gps point data. `http://blog.osmfoundation.org/2012/04/01/bulk-gps-point-data/`.
[2] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. H. Saltz. Hadoop-gis: A high performance spatial data warehousing system over mapreduce. *PVLDB*, 6(11):1009–1020, 2013.
[3] A. M. Aly, W. G. Aref, and M. Ouzzani. Cost estimation of spatial k-nearest-neighbor operators. In *EDBT (to appear)*, 2015.
[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference*, pages 322–331, 1990.
[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.
[6] S. Chen. Cheetah: A high performance, custom data warehouse on top of mapreduce. *PVLDB*, 3(2):1459–1468, 2010.
[7] D. Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.
[8] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010.
[9] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 978-3-540-77973-5.
[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.
[11] A. Eldawy and M. F. Mokbel. A demonstration of spatialhadoop: An efficient mapreduce framework for spatial data. *PVLDB*, 6(12):1230–1233, 2013.
[12] A. Eldawy and M. F. Mokbel. Pigeon: A spatial mapreduce language. In *ICDE*, pages 1242–1245, 2014.
[13] A. Eldawy and M. F. Mokbel. SpatialHadoop: A mapreduce framework for spatial data. In *ICDE (to appear)*, 2015.
[14] M. Y. Eltabakh, F. Özcan, Y. Sismanis, P. J. Haas, H. Pirahesh, and J. Vondrák. Eagle-eyed elephant: split-oriented indexing in hadoop. In *EDBT*, pages 89–100, 2013.
[15] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. Cohadoop: Flexible data placement and its exploitation in hadoop. *PVLDB*, 4(9):575–585, 2011.
[16] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-oriented storage techniques for mapreduce. *Proc. VLDB Endow.*, 4(7):419–429, Apr. 2011.
[17] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.
[18] L. Jiang, B. Li, and M. Song. The optimization of HDFS based on small files. In *IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*, pages 912–915, 2010.
[19] H.-P. Kriegel, P. Kunath, and M. Renz. R*-tree. In *Encyclopedia of GIS*, pages 987–992. 2008.
[20] J. Li, K. Liu, S. Lin, and K. A. S. Abdel-Ghaffar. Algebraic quasi-cyclic LDPC codes: Construction, low error-floor, large girth and a reduced-complexity decoding scheme. *IEEE Transactions on Communications*, 62(8):2626–2637, 2014.
[21] H. Liao, J. Han, and J. Fang. Multi-dimensional index on hadoop distributed file system. In *NAS*, pages 240–249, 2010.
[22] G. Mackey, S. Sehrish, and J. Wang. Improving metadata management for small files in HDFS. In *IEEE International Conference on Cluster Computing*, pages 1–4, 2009.
[23] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.
[24] G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *VLDB*, pages 476–487, 1998.
[25] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.
[26] A. Pavlo, C. Curino, and S. B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, pages 61–72, 2012.
[27] S. Prabhakar, K. A. S. Abdel-Ghaffar, D. Agrawal, and A. El Abbadi. Cyclic allocation of two-dimensional data. In *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*, pages 94–101, 1998.
[28] S. Prabhakar, D. Agrawal, and A. El Abbadi. Efficient disk allocation for fast similarity searching. In *SPAA*, pages 78–87, 1998.
[29] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD Conference*, pages 71–79, 1995.
[30] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., 2006.
[31] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.
[32] K. Tzoumas, M. L. Yiu, and C. S. Jensen. Workload-aware indexing of continuously moving objects. *PVLDB*, 2(1):1186–1197, 2009.
[33] C. Vorapongkitipun and N. Nupairoj. Improving performance of small-file accessing in hadoop. In *International Joint Conference on Computer Science and Software Engineering JCSSE*, pages 200–205, 2014.
[34] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.
[35] S. Zhang, L. Miao, D. Zhang, and Y. Wang. A strategy to deal with mass small files in hdfs. In *International Conference on Intelligent Human-Machine Systems and Cybernetics IHMSC*, volume 1, pages 331–334, 2014.